VECPAR 98

3rd internacional meeting on
vector and parallel processing

1998
June, 21 - 23

Faculdade de Engenharia
da Universidade do Porto

Proceedings
Part II *(June 22)*

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	10 August 1998	Conference Proceedings

4. TITLE AND SUBTITLE

VECPAR 98 3rd International Meeting on Vector and Parallel Processing

5. FUNDING NUMBERS

F6170898W0009

6. AUTHOR(S)

Conference Committee

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Faculdade de Engenharia da Universidade do Porto
Seccao dos Bragas
Porto Codex 4099
Portugal

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0200

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

CSP 98-1006

11. SUPPLEMENTARY NOTES

Consists of three volumes.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

A

13. ABSTRACT (Maximum 200 words)

The Final Proceedings for VECPAR 98 3rd International Meeting on Vector and Parallel Processing, 21 June 1998 - 23 June 1998

This is an interdisciplinary conference. Topics include parallel and distributed computing, image processing and synthesis, real-time and embedded systems.

14. SUBJECT TERMS

Computers, Signal Processing, Mathematics, Modelling & Simulation

15. NUMBER OF PAGES

1088

16. PRICE CODE

N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

VECPAR'98

3rd International Meeting on Vector and Parallel Processing

1998, June 21-23

Conference Proceedings
Part II
(Monday, June 22)

FEUP
Faculdade de Engenharia
da Universidade do Porto

Table of Contents

Partitioning Regular Domains on Modern Parallel Computers

Manuel Prieto-Matías, Ignacio Martín-Llorente and Francisco Tirado-Fernández

Departamento de Arquitectura de Computadores y Automatica
Facultad de Ciencias Fisicas
Universidad Complutense
28040 Madrid, Spain
mpmatias@eucmos.sim.ucm.es, {llorente,ptirado}@eucmax.sim.ucm.es

Abstract. It has become apparent in recent years that the performance of current high performance computers, from powerful workstations to massively parallel processors, is strongly dependent on the behaviour of the memory hierarchy. In fact, it does not only affect the computation time but the time consumed in performing communications. In this research, the impact of the memory hierarchy usage on the partitioning of multidimensional regular domain problems is studied. We use as an example the numerical solution of a three-dimensional partial differential equation in a regular mesh, by means of a multigrid-like iterative method. Experimental results contradict the traditional regular partitioning techniques on some present parallel computers like the Cray T3E or the SGI Origin 2000: a linear decomposition is more efficient than a three dimensional one due to the better exploitation of the spatial data locality. For similar reasons, computation-communication overlapping increases also execution time.

1. Introduction

The performance of current parallel computers, composed of up to hundreds of superscalar commodity microprocessors, presents an increasing dependence on the effective usage of their hierarchical memory structures. Indeed, the maximum performance that can be obtained in current microprocessors is limited by the memory access. The peak performance of the microprocessors has increased by a factor of 4-5 every 3 years by exploiting the increasing integration density, reducing the clock cycle, and by implementing architectural techniques to take advantage of the multiple levels of parallelism. However, the memory access time has been reduced by a factor of just 1.5-2 over the same period. Thus, the latency of memory access in terms of processor performance grows by a factor of 2-3 every three years. This situation seems likely to continue over the next few years and it has been suggested that such

trends may result in a "memory wall" in which application performance is entirely dominated by memory access time [1][2].

The common technique to bridge this gap and hide the problem is by using a hierarchical memory structure with large and fast cache memories close to the processor. As a result, the memory structure has a strong impact on the design and development of a code, and the programs must exhibit spatial and temporal locality to make efficient use of the cache memory and so keep the processor busy. The effectiveness of data locality has been well demonstrated in the LAPACK project, and major research has just begun to develop cache-friendly iterative methods [3] [4]. However, to the best of the authors' knowledge, the impact of the memory hierarchy usage on the partitioning has not previously been studied.

In this research, we have studied applications where the main computational portion of the program belongs to a class of kernels known as stencils. A stencil is a matrix computation in which groups of neighbouring data elements are combined to calculate a new value. This type of computation is common in image processing, geometric modelling and solving partial differential equations by means of finite difference or finite volume. The simplest approach to parallelizing these kinds of regular applications distributes the data among the processes, and each process runs essentially the same program on its share of the data. For three-dimensional applications, decompositions in the x, y, and/or z dimensions are possible.

During the last decade, a d-dimensional mesh of processors has been considered as the best partitioning to split a d-dimensional regular domain because in this way the interconnection network is more efficiently exploited [5][6]. Furthermore, communication-computation overlapping techniques are performed to keep the processor busy and so improve the parallel efficiency. However, our results show that in modern parallel computers it is more important to make effective use of the local memory hierarchy than to reduce the overheads due to network delay cost. The interconnection systems have also taken advantage of the increasing integration density offered by the integrated circuit processing technology and the effective bandwidth and latency are now hundreds of times faster than ten years ago.

This paper is organised as follows. In Section 2 we describe the sample code that has been used in our research. The effect of spatial locality on message sending is described in Section 3. Based on this analysis, the choice of an optimal partition is presented in Section 4. The influence of overlapping computations with communications is presented in Section 5. The paper ends with some conclusions to guide the partitioning of regular applications in current parallel computers.

2. Sample Code.

In this research, we are only interested in a qualitative description of the most important aspects that affect the performance, and that should be considered for making informed design decisions. As a sample problem, we have studied the numerical solution of a time-dependent partial differential equation, the three-dimensional Bose-Einstein equation [7], in a regular mesh subject to Dirichlet

boundary conditions. The problem is to describe the evolution of a physical field (a complex function) given an initial condition. An implicit finite difference method has been used to carry-out the simulation, and the systems of equations are solved by means of a multigrid-like iterative method [8]. The execution times that we present in this paper are the result of a single time step simulations using only one multigrid iteration.

Like other regular applications, the parallel program execution is a sequence of computation and communication steps. The subdomains of every processor are independently computed and then, a communication between neighbouring logical processors updates the boundaries of these subdomains.

The code used in this study parallelizes well for a number of reasons. The discretization is regular, and the same operations are applied at each grid point, even though the evolution of the system is non-linear. Thus, the problem can be statically load-balanced at the start of the code.

3. Spatial Locality Impact on Message Sending.

Message sending between two tasks located on different processors can be divided into three phases: two of them are where the processors interface with the communication system (the send and receive overhead phases), and a network delay phase, where the data is transmitted between the physical processors. Details of what the system does during these phases varies. Typically, however, during the send overhead phase the message is copied into a system-controlled message buffering area, and control information is appended to the message. In the same way, on the receiving process, the message is copied from a system-controlled buffering area into user-controlled memory (receive overhead is usually larger than send overhead):

In several out-of-date parallel computers, like the Thinking Machines CM5, the Parsys Supernode 1000 or the Meiko CS-2, the most important component was the network delay [9]. However, in current machines like the Cray T3E or the SGI Origin 2000, as the interconnection networks increase their bandwidth, the send and receive overheads are becoming important. The factors determining these overheads are different in each system, but they are mainly due to uncached operation, misses and synchronisation instructions, generally considered to be infrequent events and therefore a low priority for architectural optimisations of commodity microprocessors. The use of these components allows a rapidly increasing performance and excellent price performance, but microprocessors are designed for workstations and modestly parallel servers. A large-scale multiprocessor creates a foreign environment into which they are ill- equipped to fit. For example, the memory interfaces are cache line based, making references to single words (corresponding to strided or scatter/gather references in a vector machine) inefficient [10]. Therefore, the cost of communication depends not only on the amount of communication but also on how it is structured to interact with the architecture (mainly the spatial data locality).

3.1 The Cray T3E Message Passing Performance

The T3E used in this study had 32 DEC Alpha 21164 running at 300 MHz at the beginning of our research, and has recently been upgraded with 450 MHz processors. Like the T3D, The T3E contains no board-level cache, but the Alpha 21164 has two levels of caching on-chip: 8 KB first-level instructions and data caches, and a unified, 3-way associative, 96-Kbyte write-back second-level cache. The local memory is distributed across eight banks, and its bandwidth is enhanced by a set of hardware stream buffers. These buffers, which exploit spatial locality alone, can take the place of a large board-level cache, which is designed to exploit both spatial and temporal locality. Each node augments the memory interface of the processor with 640 (512 user and 128 system) external registers (E-registers). They serve as the interface for message sending; packets are transmitted by first assembling them in an aligned block of 8 E-registers.

The processors are connected via a 3D torus with an inter-processor communication bandwidth of 480 Mbytes/sec. Using MPI, however, the effective bandwidth is smaller due to overhead associated with buffering and with deadlock detection. The library message passing mechanism uses the E-registers to implement transfers, directly from memory to memory. Data does not cross the processor bus; it flows from memory into E-registers and out to memory again in the receiving processor. E-registers enhance performance when no locality is available by allowing the on-chip caches to be bypassed. However, if the data to be loaded were in the data cache, then accessing that data via E-registers would be sub-optimal because the cache-backmap would first have to flush the data from data cache to memory [9][10][11].

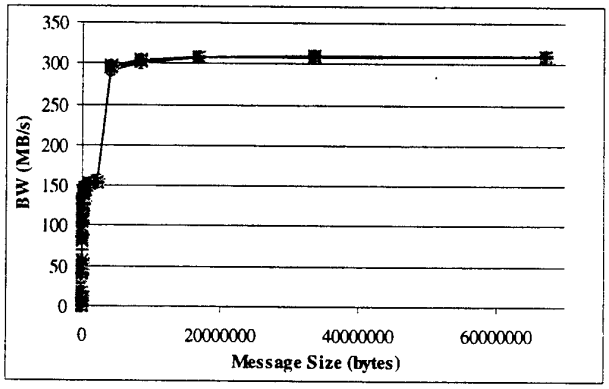


Fig. 1. CRAY T3E message passing performance for contiguous data. The network distance between the processors involved in the communication varies.

Figure 1 shows the measured one-way communication bandwidth for different message sizes using MPI. The test program uses all of the 28 processors available in the system. There is always the same sender processor and one receiver processor that

varies. The sender initiates an immediate send followed by an immediate receive, then it waits until both the send and the receive have been completed. The receiver begins by starting an immediate receive operation, then waits until it is finished. It replies with another message using a send/wait combination. Because this operation is repeated many times, if all the data fits into the cache then, except for the first echo, the required data will be found in the cache. But, on the CRAY T3E, the `suppress` directive [12] can be used to invalidate the entire cache and so, it forces all entities in the cache to be read from memory. The measures demonstrate that there is no difference between close and distant processors in the CRAY T3E.

Figure 2 shows the impact of the spatial data locality. We use also the simple echo test, but we modify the data locality by means of different strides between successive elements of the message. The stride is the number of double precision data between successive elements of the message, so stride-1 represents contiguous data. We use MPI datatypes (`MPI_Type_vector`) instead of the `MPI_Pack` / `MPI_Unpack` routines, because they may allow certain performance optimisations. However, we must be careful because the use of certain MPI datatypes can dramatically slow down communication performance, e.g., the `MPI_Type_hvector` type in the T3E implementation. We send buffers that are 8-byte aligned because the T3E copies non-aligned data slowly. This is automatic for the usual case of sending double precision data. Due to memory constraints the larger message is limited to 32Kbytes, although it is not big enough to obtain the asymptotic bandwidth for the stride-1 case, these sizes are similar to the messages used in our application program.

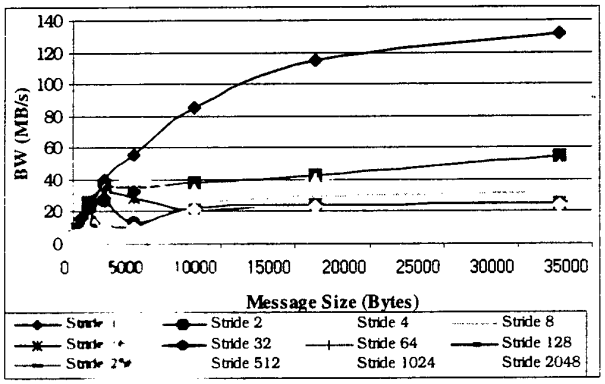


Fig. 2. CRAY T3E message passing performance using non-contiguous data

It is interesting to note that almost the same effective bandwidth is obtained for strides between 16 and 512 double precision data. For 32 KB messages, stride-1 bandwidth is around 5 times better than stride-16. Beyond Stride-1024 this difference grows, being stride-1 10 times better than stride-2048.

3.2 SGI Origin 2000 Message Passing Performance

We repeated these tests in a SGI Origin 2000. The Origin is a distributed shared-memory system with a hypercube network in which each processing node contains two processors, a portion of the shared memory, a directory for cache coherence, and interfaces to I/O devices and other system nodes. The system used in this study has the MIPS R10000 running at 195 MHz. Each processor has a 32 Kbyte two-way set-associative primary data cache and a 4-Mbyte two-way set-associative secondary data cache. One important difference between this system and the T3E is that it caches remote data, while the T3E does not. The memory bandwidth per node is 780 Mbytes/sec. Latencies to the memory modules of the Origin 2000 system depend on the network distance from the issuing processor to the destination memory node. Accesses to local memory take 80 clock cycles (CC) (400 ns), while latencies to remote nodes are the local memory time plus 22 CC (110 ns) for each network router, plus a one-time penalty of 33 CC for a remote access. On a 32-processor machine, the maximum distance covers 4 routers, so that the longest memory access is about 201 CC (1005 ns) [13][14][15].

However, as in the CRAY T3E, using MPI, the time required to send a message from one processor to another is almost independent of both processor locations. We have measured erratic differences of around 7%.

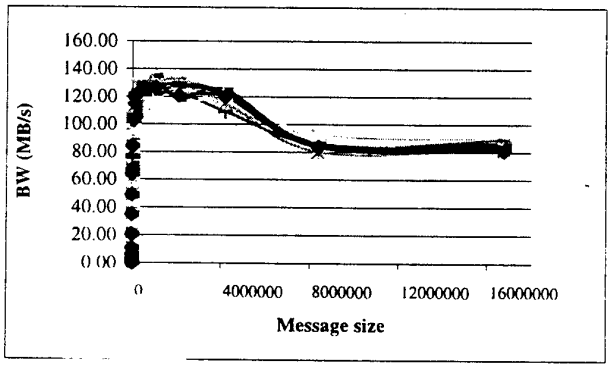


Fig. 3. SGI Origin 2000 message passing performance for contiguous data. The network distance between the processors involved in the communication varies

It is interesting to note that the measured bandwidth slows down when the message sizes are larger than the second level cache (4 MB). Figure 4 shows the impact of the spatial data locality; the legend on the right is the number of double precision data between successive elements. To avoid temporal locality effects we build and free the message every echo operation

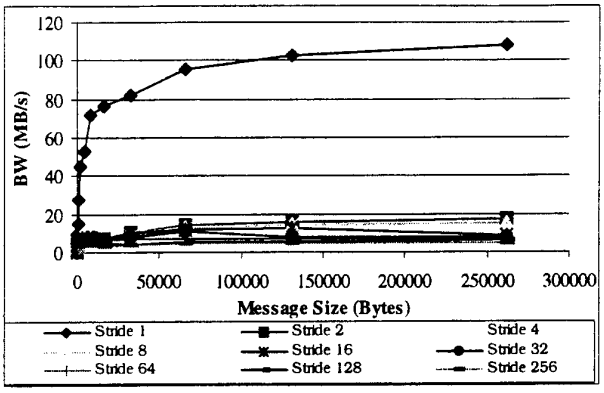


Fig. 4. SGI Origin 2000 message passing performance using non-contiguous data .

For non-contiguous data, the reduction in the effective bandwidth is even greater than in the T3E case. For 256 KB messages, stride-1 bandwidth is around 6.3 times better than stride-2. This difference grows with the stride, being 23 times for stride-256. The memory interface of the Origin is cache line based, making references to single data more inefficient than in the Cray T3E. Moreover, the current MPI implementation on the Origin 2000 requires one extra buffer copy.

3.4 Experimental Results in Our Sample Code

Although the communication pattern that we found in our application program is not a one-way transfer, but a message exchange between neighbouring logical processors, we notice the impact of the spatial locality as well. In this data exchange, advantage can be taken of bi-directional links, and a greater bandwidth can be obtained than is possible with the echo test. The code was written in C, so a three dimensional domain is stored in a row-ordered (x,y,z)-array. It can be distributed across a 1D mesh of processors following three possible partitionings: x-direction, y-direction and z-direction. The x and y-direction partitioning were found to be more efficient, because the message data exhibits a better spatial locality. X and Y boundaries are stride-1 data, except strides between different Z-columns (two complex data, i.e. four doubles, for X-partitioning and this quantity plus two times the number of elements in a x-plane for Y-partitioning). A message using Z-partitioning has a stride 2 times the number of elements in dimension z (all the elements are double precision complex data). Figures 5 and 6 show the experimental results from the CRAY T3E and the SGI Origin 2000 respectively. Due to main memory capacity, the SGI allows larger simulations.

X-partitioning is found to be 2 times better than Z-partitioning for the 128-element simulation on the two different configurations of the CRAY T3E. Although message-passing bandwidth is very important, we should also note that this difference is not only a message passing effect. X and Y-partitioning more efficiency exploit stream

buffers because they maximise inner loop iterations [11]. By means of the MPP Apprentice performance tool we have found that the time spent in the initiation of message sending is 5 times larger in the Z-partitioning simulations. This fact fits in with what we measure in the echo test.

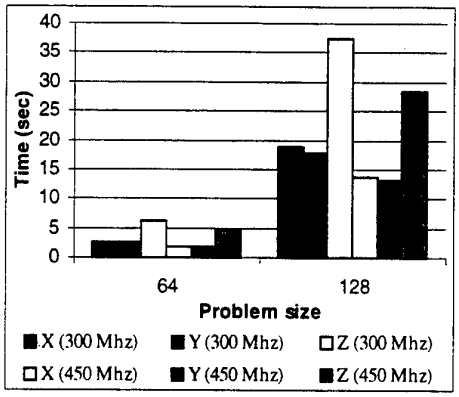


Fig. 5. Different linear partitioning of our sample application using sixteen processor in the CRAY T3E. The problem size is the number of cells in each dimension for the finest grid in the multigrid hierarchy.

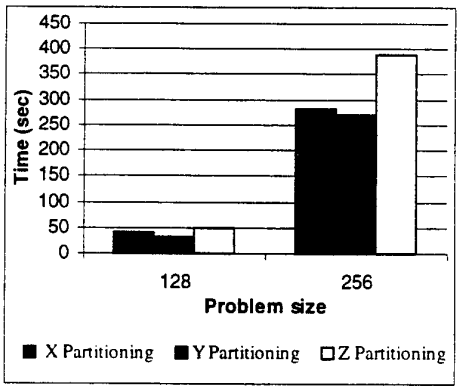


Fig. 6. Different linear partitioning of our sample application using 32 processors in the SGI Origin 2000. The problem size is the number of cells in each dimension for the finest grid in the multigrid hierarchy.

Equivalent differences in the Origin 2000 are important, but lower than the T3E ones. For the 128-element problem, X partitioning is only 1.2 times better. For the 256 one, it grows to 1.4. The large second-level cache of this system, which allows the best exploitation of the temporal locality, influences these results [16].

Using 2D and 3D decompositions, we notice the same effects. Z-plane boundaries slow down the performance of the application because they are discontinuous in memory. Therefore, as figure 7 show, a 2D decomposition using a 4x4x1 array of abstract processors (4 processors in the x and y dimensions and no decomposition in the z direction) is better than 4x1x4 and 1x4x4 topologies (the differences are around 15 % in the Cray T3E). In the same way, a 3D decomposition using a 4x2x2 array is better than a 2x2x4 one.

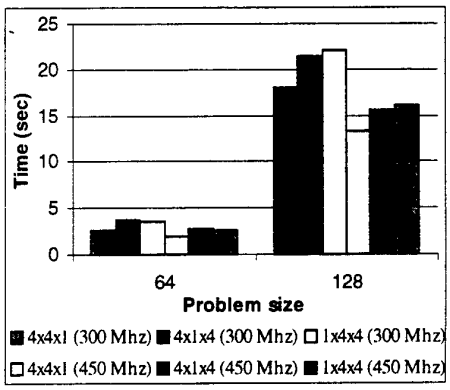


Fig. 7. Different 2D decompositions of our sample application using 16 processors in the CRAY T3E. The problem size is the number of cells in each dimension for the finest grid in the multigrid hierarchy.

4. Partitioning for Performance

Over the last decade the partitioning has been focused on reducing communications that are inherent to the parallel program. As is well known, for a d-dimensional problem, the communication requirements for a process grow proportionally to the size of the boundaries, while computations grow proportionally to the size of its entire partition. The communication to computation ratio is thus a perimeter-to-surface area ratio in a two-dimensional problem, and similarly, a surface area to volume ratio in three-dimensions. So, the three dimensional decomposition leads to a lower inherent communication-to-computation ratio.

Moreover, as we have experimentally proved in the previous section, the time required for sending a message from one processor to another is independent of both processor locations. Therefore, there is no sense in talking about physical neighbours, and the mapping of the logical processors over the physical ones is not very important, as far as communication locality is concerned.

Therefore, these ideas suggest a general rule: Higher-dimensional decompositions tend to be more efficient than lower-dimensional decompositions [5][8].

However, as we discussed in the previous section, the communication cost is also a function of the spatial data locality. Therefore, a trade-off between the improvement of the message data locality and the efficient exploitation of the interconnection network exists.

The following figures compare the different decompositions for our sample application in the Cray T3E. In the larger problem using 8 processors, and for the new processor, the best 1D-decomposition achieves improvements of 6.5% and 14,5% over the best 2D and 3D-decompositions respectively. These differences have grown by 2% and 10 % compared to the old 300 MHz configuration. In the 16-processor simulation the differences are lower (only 2.2 % and 7%) for the same problem size because the local matrices are smaller too.

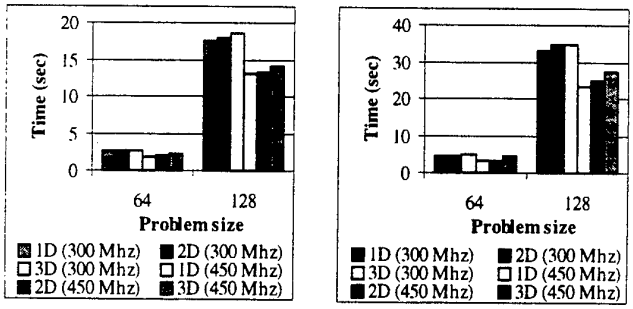


Fig. 8. Different decompositions for our sample program in the CRAY T3E using 16 (on the left) and 8 processors (on the right). The problem size is the number of cells in each dimension for the finest grid in the multigrid hierarchy.

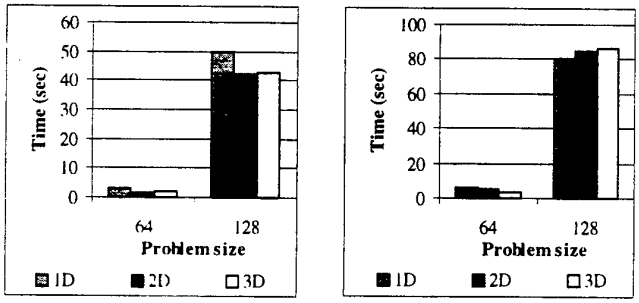


Fig. 9. Different decompositions for our sample program in the SGI Origin 2000 using 16 (on the left) and 8 processors (on the right). The problem size is the number of cells in each dimension for the finest grid in the multigrid hierarchy.

In the SGI Origin 2000. we have obtained lower differences. Using 8 processors, the best choice is also a linear decomposition, but it is only 5% and 7% better than the 2D and 3D decompositions. However, for the 16-processor simulation, the 2D decomposition is 15 % and 1% better than the 1D and 3D decompositions. The large

second-level cache of this system is again the reason of these results. Cray T3E is more sensitive to spatial data locality than the SGI because its performance depends significantly on the effective use of the stream buffers system.

Therefore, in both multiprocessors, it is more important to make effective use of the local memory hierarchy than to reduce the overheads due to network delay cost. So, the best performance is usually obtained by means of a simple linear decomposition.

We should also note that, although we have considered execution time as the performance metric, there are many aspects to the evaluation of a parallel program. A lower-dimensional partitioning program is easier to code, so if we consider implementation cost, a one-dimensional partitioning is also the best choice. Besides, it allows the implementation of fast sequential algorithms in the non-partitioned directions [17].

In a workstation cluster a linear data distribution is also the best because the fewer the number of neighbours, the fewer the number of messages to be sent. Therefore, a one-dimensional decomposition reduces TCP/IP overheads as well [18]. So, if we consider portability, a one-dimensional partitioning is also the best choice.

5. Computation – Communication Overlapping.

A typical approach for dealing with the communication cost due to the transit latency, the bandwidth-related cost, and contention, is to hide it by overlapping this part of the communication with other useful work. The results in the previous sections have been obtained without overlapping, but these types of algorithms can be structured so that every process request for remote data is interleaved explicitly with local computation. For this purpose, it is necessary to deal with the boundaries before the inner domain. In this way, it is possible to initiate an immediate send operation before the point where it naturally appears in the program and the message may reach the receiver before it is actually needed. Thus, the receive operation does not stall waiting for the message to arrive; it will copy the data straight away from an incoming buffer into the application address space. Therefore, instead of using the simple pattern:

1- Exchange artificial Boundary:
    Send boundaries to neighbours
    Receive artificial boundaries from neighbours
2- Update local domain using artificial boundaries

we must use:

1- Update boundaries
2- Send boundaries to neighbours
3- Update local domain using artificial boundaries
4- Receive artificial boundaries from neighbours

In order to evaluate the benefits and limitations of this new approach, we will assume that message initiation and reception costs are the same in the two structures, so the execution time can be estimated as:

$$T_{without\_overlapping} = T_{local} + T_{com\_overhead} + T_{com} . \quad (1)$$

$$T_{overlapping} = T_{boundaries} + T_{com\_overhead} + \max(T_{inner}, T_{com}) . \quad (2)$$

Tlocal is the time spent in the local domain update, Tinner is the cost of inner domain actualisation, Tboundaries is the time required for updating the boundaries, Tcom_overhead is the send and receive overheads (it is important to recall that these overheads incurred on the processors cannot be hidden) and Tcom is the network delay. For a real problem, Tcom is lower than Tinner. Therefore, the overlapping pattern is better than the simple approach while:

$$T_{boundaries} + T_{inner} < T_{local} + T_{com} . \quad (3)$$

Tlocal can be divided in a Tinner and a Tboundaries_2, so the last inequality can be simplified to:

$$T_{boundaries} - T_{boundaries\_2} < T_{com} . \quad (4)$$

This latter boundary actualisation time is different from the previous one. Usually, the cost of updating the boundaries in the non-overlapping approach (they are updated together with the inner local domain) is lower than in the overlapping pattern due to the better exploitation of the memory hierarchy.

The overlapping approach has been successfully used in old parallel computers like the Parys Supernode SN 1000, where the network bandwidth-related cost is very important. In workstations clusters, the benefits are even greater because the network is usually a non-private resource [18]. However, as we have discussed in the previous sections, in the current generation of parallel computers Tcom is not so important. Therefore, the increase due to the boundary actualisation may be greater than the reduction obtained by way of the overlapping.

We have verified these ideas with our test program. Figure 10 illustrates both patterns using a linear decomposition. In the CRAY T3E the non-overlapping approach performance is 7.3% higher than the overlap pattern for the 16-processor simulation (for the larger problem size with the 450 MHz processor) and 5% higher using 8 processors. These differences have grown compared to the old configuration where the differences are 6.4% and 4% respectively. Using 2D and 3D decompositions we have obtained the similar differences [16].

In the SGI, the differences are similar. In the 32 processor-simulation, using a linear decomposition, the difference for the larger problem is 7.5 % [16].

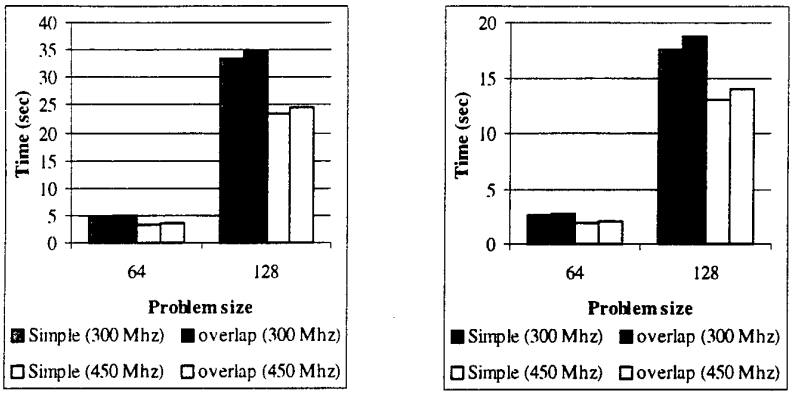


Fig. 10. Overlapping versus non-overlapping approach on the Cray T3E using 8 (on the left) and 16 processors (on the right). The problem size is the number of cells on each dimension for the finest grid in the multigrid hierarchy.

6. Conclusions

We have shown how the optimal data partitioning of regular domains is a trade off between the improvement of the message data locality and the computation/communication ratio. In older parallel computers the performance depends mainly on the efficient exploitation of the interconnection network. However, the performance obtained on current parallel computers, based on the replication of commodity microprocessors, present a growing dependence on the efficient use of the memory hierarchy.

The main conclusions of the paper can be summarized in the following points, that contradict to a certain extent the traditional wisdom on data partitioning: (1) the partioning of the domain must avoid boundaries with poor data locality due to the reduction in the effective bandwidth, (2) 1D partitioning is becoming more efficient than higher dimension partitioning (Moreover, it is easier to code, more suitable to include fast sequential algorithms in non-partitioned directions and more portable), and (3) communication/computation overlapping does not reduce execution time. These conclusions have been verified by experimental results on two microprocessor based computers: the Cray T3E and the SGI Origin 2000.

Acknowledgements

This work has been supported by the Spanish research grants TIC 96-1071 and TIC IN96-0510, the Human Mobility Network CHRX-CT94-0459 and the Access to

Large-Scale Facilities (LSF) Activity of the European Community's Training and Mobility of Researchers (TMR) Programme.

We would like to thank Ciemat, the Department of Computer Architecture at Malaga University and C4(Centre de Computatió i Comunicacions de Catalunya) for providing access to the parallel computers that have been used in this research.

References

[1] W. A Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," Comp. Arch. News, Assoc. for Computing Mach., March, 1995.

[2] A. Saulsbury, F. Pong, A. Nowatzyk."*Missing the Memory Wall: The Case for Processor/Memory Integration*". In Proceeding of ISCA'96. May 1996.

[3] C. C. Douglas, "*Caching in with multigrid algorithms: Problems in two dimensions*" Parallel Algorithms and Applications, (1996), pp. 195 - 204.

[4] L. Stals and U. Rude. "*Techniques for improving the data locality of iterative methods*". Tech. Report MRR97-038, School of Math. Sc. of the Australian National University, 1997.

[5] Ian T. Foster. "Designing and building parallel programs. Concepts and tools for parallel software engineering", Addison-Wesley Publishing Company 1995.

[6] I. M. Llorente, F. Tirado y L. Vázquez, "*Some Aspects about the Scalability of Scientific Applications on Parallel Computers*", Parallel Computing, Vol. 22, pp. 1169-1195, 1997

[7] V. M. Pérez-García, et al. "*Low Energy Excitations of a Bose-Einstein Condensate*", Physical Review Letters, Vol 77, pp. 5320-5323, 1996

[8] I. M. Llorente y F. Tirado, "*Relationships between Efficiency an Execution Time of Full Multigrid Methods on Parallel Computers*", IEEE Trans. on Parallel and Distributed Systems, Vol. 8, N° 6, 1997

[9] David Culler, Jaswinder Pal Singh, Annop Gupta. Preliminary draft of Parallel Computer Architecture. A hardware /software approach. Morgan-Kaufmann Publishers 1997.

[10] S. L. Scott. "*Synchronization and Communication in the T3E Multiprocessor*", Proceeding of the ASPLOS VII, October 1996.

[11] E. Anderson, J. Brooks, C. Grassl, S. Scott. "Performance of the CRAY T3E Multiprocessor". In Proceeding of SC97, November 1997.

[12] Cray C/C++ Reference Manual, SR-2179 3.0.

[13] J. Laudon and D. Lenoski. "*The SGI Origin: A ccNUMA Highly Scalable Server*". In Proceeding of ISCA'97.May 1997.

[14] H. J. Wassermann, O. M. Lubeck, F. Bassetti. "Performance Evaluation of the SGI Origin 2000: A Memory-Centric Characterization of LANL ASCI Applications". In Proceeding of the SC97, November 1997.

[15] Silicon Graphics Inc., Origin Servers, Technical Report, April 1997.

[16] M. P. Matías, D. Espadas, I. M. Llorente, F. Tirado, "*Experimental results of different partitionings of a regular domain on the Cray T3E, the SGI Origin 2000 and the IBM SP2*", Tech. Report 98-001, Dept. of Computer Architecture at Complutense University, Madrid, Spain, 1998.

[17] C. C. Douglas, S. Malhotra, and M. H. Schultz, "*Transpose free alternating direction smoothers for serial and parallel methods*", MGNET at http://www.mgnet.org/. 1997.

[18] I. M. Llorente, J. C. Fabero, F. Tirado, A. Bautista."*Distributed Parallel Computers versus PVM on a Workstation Cluster in the Simulation of Time Dependent PDE* ". In Proceeding of 3rd Euromicro Workshop on Parallel and Distributed Processing, Italy 1995, pp. 20-26.

A performance analysis of the SGI Origin2000

Aad J. van der Steen[1] and Ruud van der Pas[2]

[1] Computational Physics, Utrecht University
P.O. Box 80195, 3508 TD Utrecht
The Netherlands
steen@phys.uu.nl
[2] Ruud van der Pas, European HPC Team
Silicon Graphics
Veldzigt 2a, 3454 PW De Meern
De Meern, The Netherlands
ruud@demeern.sgi.com

Abstract. In this paper we present the results of benchmark experiments carried out on a Silicon Graphics Origin2000. We used the three modules of the EuroBen Benchmark ([1]) to assess the performance of a single node, as a shared memory system, and as a distributed memory system. Where the situation calls for it, we compare the results with those obtained on a Cray T3E and an IBM SP2. The results obtained from this benchmark give a good impression of what performances can be attained on the Origin2000 under what circumstances and expose the weak and strong points of the system.

Keywords: Performance analysis, High-performance computers, Programming models.

1 Introduction

The Silicon Graphics Origin2000 has been introduced in the last quarter of 1996. Since then a considerable amount of these systems have been installed, ranging from 4-128 processors per system. The Origin2000 machine has a rather complicated architecture and, like most high-performance computers, shows a wide range of performance levels depending on memory access patterns, loop content, fitness for and grain size of parallelism, etc. It was our intention to make a *performance profile* of the Origin2000 which will allow to obtain a fair estimate of the performance under a variety of realistic operating circumstances. At the same time, architectural bottlenecks can be identified. This may be valuable for future system development and will in the end be of benefit for end users.

To assess the performance of the Origin2000 we used the EuroBen Benchmark, version 3.2 ([1]). This benchmark was initially designed for testing shared-memory MIMD systems. However, for a limited number of important cases also message-passing codes have been developed.

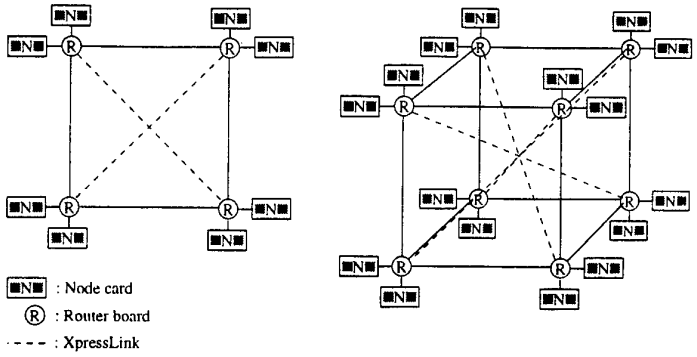


Fig. 1. *Configurations of Origin2000 systems with 16 and 32 processors.*

This paper has the following structure: first the Origin2000 and the EuroBen Benchmark are briefly described, next we present the most relevant results of our benchmark study and we conclude with a summary and issues that might be addressed in further research.

2 The Origin2000 system

The Origin2000 is a cache coherent, logically shared, physically distributed memory system with 4–128 MIPS R10000 RISC processors. The features of the processors are extensively described in [2, 3]. These include out-of-order execution of instructions and prefetching of operands in order to hide data-access latency.

The system as we have benchmarked contained 195 MHz processors with a theoretical peak performance of 390 Mflop/s. The processors have 32 KB, two-way set-associative primary instruction and data caches and a combined secondary instruction and data cache of 4 MB. In parallel processing the caches of the processors involved are kept coherent via a directory memory, see [2]. The memory of the total system was in our case 16 GB.

Two processors are mounted on a node card together with a local part of the memory and a *HUB chip*, an ASIC which connects all components on the node card with each other. In addition, the HUB chip also connects the node card to the other node cards and the I/O facilities of the system. The raw bandwidth of the connections on the node card and between node cards is 780 MB/s, see [4]. However, the two processors have to share this bandwidth when accessing data from memory. For the actual point-to-point bandwidth between processors on the user level Silicon Graphics quotes a bandwidth of 150 MB/s. This is due to various overheads and the cache-coherency that is enforced by the system.

Node cards are, via their HUB chip, connected by routers to the rest of the system. The interconnection of the routers has a hypercube topology. However, for up to 32 processors so-called XpressLinks can be added to reduce the system diameter Ω to 3. Figure 1 shows some system configurations.

Silicon Graphics provides auto-parallelising compilers that attempt to spread the content of loops evenly over the processors. In addition, the user may add parallelisation directives in various styles. Next to SGI-proprietary, also ANSI X3H5 recommended ([5]) and OpenMP ([6]) directives are accepted. Also distributed memory message passing libraries are available. Apart from the SGI/Cray-style `shmem` library, MPI ([7]) and PVM ([8]) are supported. An HPF compiler ([9]) for the Origin2000 is distributed by the Portland Group.

3 The EuroBen Benchmark

To get a complete insight in the behaviour of the machine one has to investigate the single-node performance, the shared-memory parallelisation capabilities, and the possible (dis)advantages of using the system as distributed memory system. The EuroBen Benchmark has been build in a hierarchical way to extract the necessary information and to build the performance profile from programs in three modules of increasing complexity:

- The first module contains programs that identify the machine parameters that govern upper and lower bounds of the performance.
- The second module contains simple but basic algorithms: full and sparse linear systems solvers, FFTs, random number generation, etc.
- The third module places the algorithms in a compact application setting and applies them in various PDE and ODE problem implementations. In addition, linear and non-linear least-squares problems and some I/O-bound problems are considered.

For a full description of the benchmark one is referred to [1].

3.1 Testing circumstances

The full benchmark applied on single nodes, together with the parallel execution of relevant programs from the benchmark both with a shared-memory and a distributed-memory message-passing programming model gives a sufficient insight in the machine behaviour to enable reasonable performance estimates in many circumstances. For the shared-memory programming model we used both the SGI-proprietary as well as the ANSI X3H5 directives, for the message-passing programs MPI was used. Moreover, features like Inter Procedural Analysis and the quality of the numerical libraries provided by Silicon Graphics have been assessed to complete the profile of the machine. Where relevant, to compare and contrast the distributed memory results we also have done similar tests on two other widely available DM-MIMD systems, a Cray T3E Classic and a IBM SP. In addition some results from a Hitachi SR2201 were used.

We had the following testing circumstances for the systems quoted in this paper:

- **Origin2000** The FORTRAN 77 MIPSPro compiler, version 7.20, compiler options `-O3 -64 -OPT:IEEE:arithmetic=3:roundoff=3`, Operating System IRIX 6.4 02121744. For the hardware specifications seen section 2.

- **IBM RS6000/SP** We used IBM RS6000/SP Thinnodes with 160MHz P2SC processors and 512 MB memory per node. The Fortran 90 compiler was xlf, version 4.1, compiler options were `-O3 -qarch=pwr2`, Operating System AIX, version 2.4 002006959400.
- **Cray T3E Classic** We used 300 MHz DEC Alpha 21164 processors with 128 MB memory per node. The Fortran 90 compiler was CF90, version 3.0.1.3, compiler options were `-O3 -dp`, Operating System UNICOS/mk, version 2.0.2.19.
- **Hitachi SR2201** We used 200 MHz PA-RISC 720 processors with 256 MB of memory per node. The Fortran 90 compiler was OFORT90, version V02-05-/A, compiler option was `-O3`, Operating System HI-UX/MPP, version SR220001 02-02 0.

In all cases we used the system clock with resolutions ranging from 0.5–15 μs. We took care to use timing measurement intervals of at least a few hundred ms to exclude measuring artefacts, repeating measurements where necessary.

4 Benchmark results

From each of the three benchmark modules we present some representative results as the complete discussion of all results is far to extensive for this paper. One is referred to the report [3] for a comprehensive presentation. The report is downloadable from: `http://www.phys.uu.nl/~steen/euroben/reports/` as a compressed PostScript file.

4.1 Module 1 results

Program `mod1ac` measures the speed of a number of important basic operations as a function of the array length. With the bandwidth to the CPU known we should be able to assess whether the code generated by the compiler is optimal. In Table 1 we list the single-node speeds for these operations with stride 1 access to the operands as found for operation from the level 1 and level 2 cache.

Program `mod1ac` obtains which the speeds of the operations with stride 1, 3, and 4 memory access. Moreover, also the speeds of the same operations is measured when accessing the operands via an index vector. Non-unit stride access turns out to have quite little influence on the performance. Indirect indexed operations incur a loss of roughly 30% in speed due to address operations. So, we present only the stride-1 values. The first and fourth column show the maximum observed performance, $r_{\max}$, when accessed from the primary and secondary cache, respectively. As the secondary cache is quite large (4 MB), a relatively small proportion of data references will have to be to the main memory.

The dependency of the execution time of the array length can be modelled with considerable precision by a linear model $t(n) = a + bn$ where a is the latency and b is the time per operation per element. These parameters are given as the third and second column entries of Table 1. It enables us to draw definite conclusions about the optimality of the generated code for the operations considered.

	Operation	L-1 cache r_{max} Mflop/s	L-1 cache Cycles per op/element	L-1 cache Latency cycles	L-2 cache r_{max} Mflop/s
1	Broadcast	195.60	1	23	61.92
2	Copy	95.29	2	15	43.65
3	Addition	64.66	3	21	34.36
4	Subtraction	64.48	3	18	34.57
5	Multiplication	64.45	3	18	34.52
6	Division	9.23	21	0	9.22
7	Dotproduct	194.46	2	14	137.61
8	$x = x + \alpha y$	128.92	3	19	69.13
9	$z = x + \alpha y$	128.62	3	17	66.99
10	$y = x_1 x_2 + x_3 x_4$	107.39	6	23	56.97
11	1st order recurs.	96.39	2	23	46.04
12	2nd order recurs.	96.69	4	22	80.31
13	2nd difference	242.31	2.5	36	132.54
14	9th Degr. Polynomial	376.92	9	31	351.17

Table 1. r_{max}, *the number of cycles per operation per element, and the latency values for the primary cache operations on a single processor of the Origin2000. Only results of the first 14 of kernels are shown. The operations all have unit stride access. The operation latency from secondary cache is completely hidden by the data access.*

The dyadic operations addition, subtraction, and multiplication operate at $1/6^{th}$ of the Theoretical Peak Performance, 390 Mflop/s, when accessed from the primary cache as the total operation takes 3 cycles. With an ideal bandwidth situation, transferring two operands to the relevant functional unit and shipping one result back per clock cycle, the performance should approximately be half the Theoretical Peak Performance. One can conclude that only one 8-byte data item can be transferred per cycle. This is in agreement with the bandwidth quoted by the vendor. The dotproduct and the **daxpy** operation (kernel 7 and 8) also show speeds that closely agree with this bandwidth with computational intensities of 1 and 2/3, respectively ([10]). It shows that, at least for these simple operations, the compiler is able to generate optimal code given the limited bandwidth of one operand/cycle. With a high reuse of operands, like the evaluation of a 9^{th}-degree polynomial and a computational intensity of 9, a large fraction of the Theoretical Peak Performance can be obtained: kernel 14 shows a performance of 96% of the Theoretical Peak Performance.

Shared-memory parallel performance of program `modlac` Ideally, the simple, vector-oriented operations in program `modlac` should speed up almost linearly with the number of processors when executed in parallel. There are two effects that will decrease the potential speedup: the parallelisation overhead inherent in the distribution of the data and the synchronisation of the multiple processes and, secondly, the slowdown per processor when the array length per processor decreases because of the latency of the operation. In Figure 2 the speeds on 1, 8, and 32 processors is displayed for the first 14 kernels of program `modlac`.

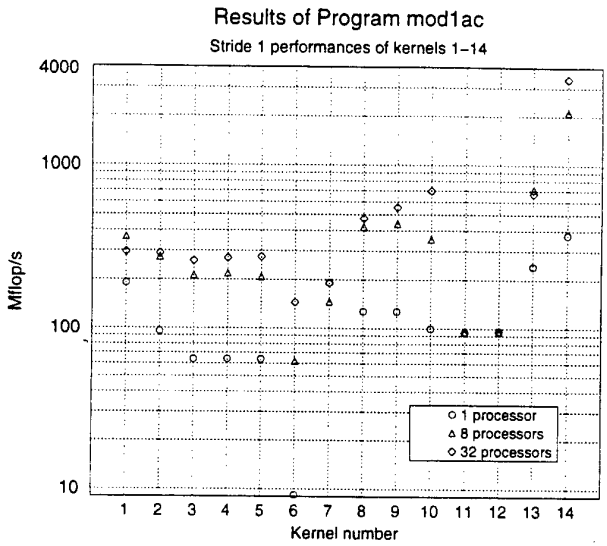


Fig. 2. *Speeds in Mflop/s of the first 14 kernels of program* `mod1ac` *on 1, 8, and 32 processors.*

The FORTRAN compiler uses heuristics to determine whether the computational content of a loop is sufficient to warrant parallel processing. If not, the loop is executed sequentially. When recurrences are detected, the loop is also executed sequentially. This is the case with kernels 11 and 12 representing first and second order recurrences, respectively. All other kernels but one are executed in parallel. For all these kernel there turns out at least some benefit in parallel execution. The exception is the dotproduct that shows a lower performance on 8 processors in parallel and is executed sequentally on 32 processors. It shows that the heuristics used to determine a sufficient amount of parallelism basically are correct in that the parallel execution is not slower than the sequential one.

In many cases, however, the speedup is not very high. The inherently slow division (kernel 6) and kernel 14, the evaluation of a 9^{th}-degree polynomial, which have both a large computational content benefit the most while a kernel like the `daxpy` operation (kernel 8) show a speedup of only 12% from 8 to 32 processors. Here also the latency of the operation plays a rôle: the array length on 32 processors is only 31 elements. With this array length the speed per processor is already 15% lower than $r_{\max}$.

In summary one can conclude that the computational content of a loop should preferably not be below 10 flops to attain a sizable speedup at 32 processors.

Distributed-memory parallel dotproduct From Figure 2 it was clear that the use of the shared-memory programming model is not suited for parallel execution of the dotproduct. We also executed the dotproduct with a distributed-memory programming mmodel using MPI. Three implementations were considered: a "naive" implementation, in which all partial sums are sent to a root processor which also distributes the global sum back directly to all other proces-

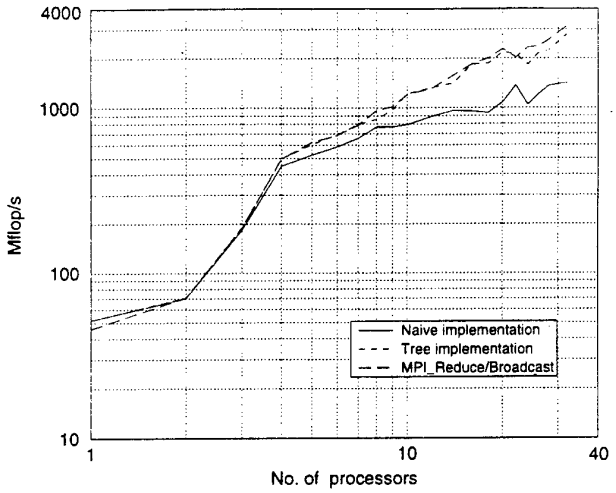


Fig. 3. *Performance in Mflop/s of the three distributed-memory dotproduct implementation on 1-32 processors.*

sors, a FORTRAN-implemented tree algorithm for gathering the partial sums and broadcasting the global sum, and an implementation based on `MPI_Reduce` and `MPI_Broadcast`. The last implementation contains MPI functions that should be optimised by the vendor and perform at least as good as the FORTRAN-implemented tree algorithm. Figure 3 shows the result of this distributed-memory dotproduct.

The first observation that can be made is that the FORTRAN-based tree implementation and the `MPI_Reduce/Broadcast` implementation indeed are quite close in performance. So, `MPI_Reduce` and `MPI_Broadcast` are optimised communication functions. Both perform considerably better than the naive implementation, especially for a larger number of processors. The second observation is that the distributed-memory version of the dotproduct scales well with the number of processors: at 32 processors a speed of 3167 Mflop/s is attained: about 100 Mflop/s, including the time lost in communication. So, the distributed-memory version is preferable by far over the shared-memory version from a performance point of view.

Point-to-point communication The program `mod1h` measures bandwidth and latency between two processors using the MPI library functions `MPI_Send` and `MPI_Receive` with message lengths varying from 40–10,000,000 bytes. This covers the full range of possibilities: communication from the primary cache, from the secondary cache, and from the main memory. The interprocessor communication speed with point-to-point communication is not negligible in comparison with the speed between the local memory and the CPUs. Therefore, it is useful to consider this full range as it may affect the communication patterns one wants to use.

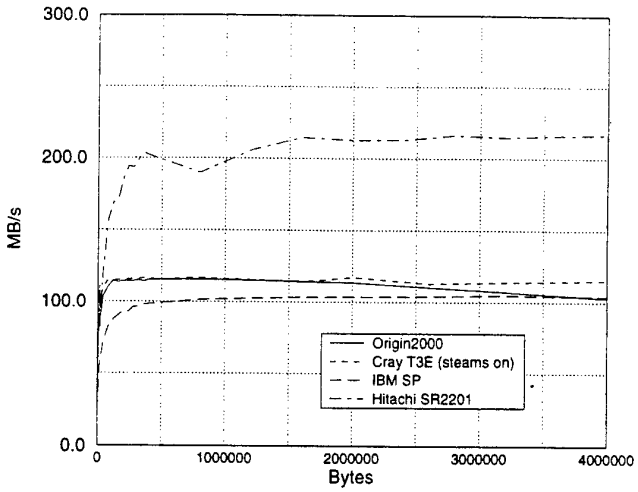


Fig. 4. *Graph of bandwidths in point-to-point message passing using* `MPI_Send` *and* `MPI_Recieve`. *Results for the Origin2000, the SGI/Cray T3E-Classic, and the IBM SP are shown. On the T3E the stream buffers were on.*

The same program has also been run on a Cray T3E Classic, an IBM SP2 and a Hitachi SR2201. As the cache sizes of these systems are different, one might expect to see different behaviour for these systems as indeed is the case. This is, however, not only due to the different access speed in the memory hierarchies. In MPI the strategy in `MPI_Send` of buffering messages, or not, is left to the implementator. As it may be assumed that different implementation decisions have been made for different machines, observed differences in bandwidth may originate from differences in local access times, another message buffer strategy or both. Therefore, the best decision seems to be to give the bandwidth as a function of the message length and the latency as derived from very short messages (e.g., up to 400 bytes). For these short messages one may assume that no auxiliary buffering is required and one may obtain a fair idea of the latency as experienced through the software. In addition, this information is important because of the frequency that messages of only one data item are exchanged which enables an estimate for the slow-down caused by such messages. The bandwidth versus the message length is shown in Figure 4.

Note that the bandwidth of the Origin2000 is decreasing from about 115 MB/s for sufficiently long messages up to 2 MB to 102 MB/s at 4 MB. As already mentioned in section 2, the bandwidth available at the application level is 150 MB/s, so the bandwidth found reasonably matches this figure. For messages longer than 4 MB the bandwidth even drops to about 78 MB/s. We do not observe this behaviour on the other three systems. We ascribe the decreasing bandwidth on the Origin to the fact that buffer copies above 4 MB do not fit in

System	Bandwidth Mbyte/s	Latency μs
SGI Origin2000	115.75	14.6
SGI/Cray T3E-Classic	117.30	22.3
IBM SP	104.85	34.7
Hitachi SR2201	216.69	29.7

Table 2. *Maximum bandwidths and latencies for the Origin2000, the SGI/Cray T3E-Classic, the IBM SP, and the Hitachi SR2201.*

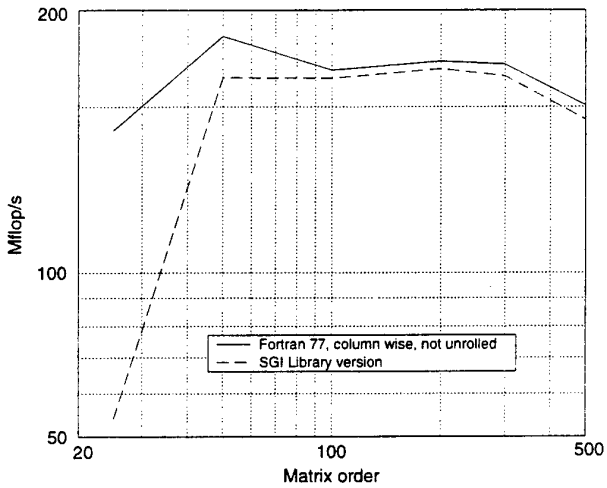


Fig. 5. *Performance for $y = Ax$. Only the fastest* FORTRAN 77 *and the SGI library routine are shown.*

the secondary cache anymore and therefore the memory must be accessed. The less than ideal MPI implementation might be at the base of this effect. In table 2 we summarise the maximal bandwidths and latencies for the four systems.

4.2 Module 2 results

Of module 2 we present two programs. Program `mod2a`, which measures the speed of a matrix-vector multiplication and `mod2e` which solves a large sparse eigen value problem system. For the discussion of all programs of module 2 one is referred to [3].

`mod2a`, single-node In In the single-node version problem sizes of $n = 25$, 50, 100, 200, 300, and 500 are considered for each of five implementations. For the sake of clearness, we show only the fastest of the FORTRAN 77 implementations together with the result of the library version of the BLAS 2 routine `dgemv` in Figure 5. The implementations actually used are a dotproduct, or row-wise

Order	Not unrolled Row-wise Mflop/s	4×unrolled Row-wise Mflop/s	Not unrolled Column-wise Mflop/s	4×unrolled Column-wise Mflop/s	Librar y versio n Mflop/s
25	135.8	78.2	181.3	130.4	53. 9
50	173.3	102.3	269.0	166.5	226. 2
100	167.7	123.7	233.4	169.5	225. 6
200	184.2	138.4	242.3	184.4	234. 5
300	186.9	138.1	239.0	187.6	227. 6
500	187.1	77.3	201.4	181.2	189. 5

Table 3. *Performances on the Origin2000 for $y = Ax$. Four different* FO RTRAN 77 *implementations and the SGI libary version are shown.*

implementation, a **daxpy** or column-wise implementation and the four times unrolled versions of these two methods. On many systems the unrolled versions perform better than their not unrolled equivalents. This is, however, not the case on the Origin. The reason is that the FORTRAN 77 compiler itself already unrolls loops where possible and this is certainly so for the simple inner loops used in the various not unrolled implementations. For the implementations where a hand unrolling is done the compiler is not able to generate code of comparable quality and the performance of the unrolled versions lag behind as shown in Table 3. So, a fairly obvious hand optimisation does not work out very well here. The lesson could be *not* to do these kind of optimisations on the Origin to give the compiler a better chance for automatic optimisation. One of the objectives of program `mod2a` is to make users aware of such facts.

Note that in the column-wise version, using **daxpy** operations a speed is attained that is twice as high as found with program `mod1ac` for kernel 8 (see Table 1). Within the context of a matrix-vector multiplication with the **daxpy** as an inner loop, the compiler is able to overlap two succesive iterations of the inner loop, thus winning a factor of 2 in speed.

`mod2a`, parallel versions Of `mod2a` also a shared-memory and a distributed-memory version were executed to assess the potential benefit of the parallelisation in both programming models. In Figure 6 the results for the two implementations is shown. It is clear from the Figure that the distributed-memory version is much faster than its shared-memory counterpart: 7.3 vs. 2.7 Gflop/s on 32 processors. In the distributed-memory implementation the data distribution is such that no data have to be communicated between the processors. In this situation the distributed-memory is preferable. However, when the transposed matrix-vector product is performed, all-to-all communication is required. The overhead in sending messages turns out to be so high in this case that the shared-memory version is now faster then the distributed-memory version: 2.5 vs. 0.15 Gflop/s on 32 processors.

Program `mod2e` In program `mod2e` the 10 smallest eigenvalues of penta-diagonal, symmetric systems with matrix orders n =100,...,10000 are computed by a generalised Lanczos iteration scheme. In Figure 7 we show the speed per iteration for the range of system orders both without and with interprocedural analysis.

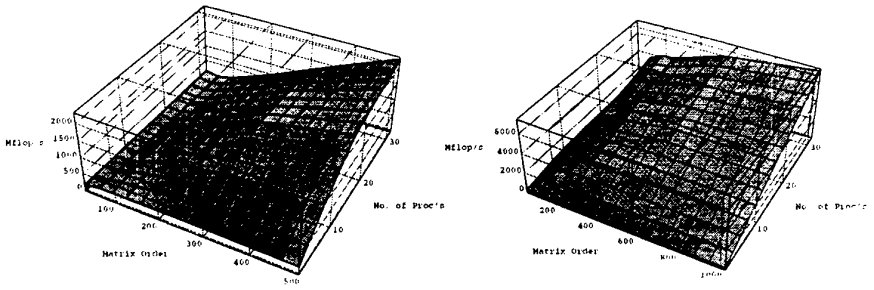


Fig. 6. *Performance surface of a parallel shared-memory implementation (left) and a distributed memory implementation (right) of a matrix-vector product.*

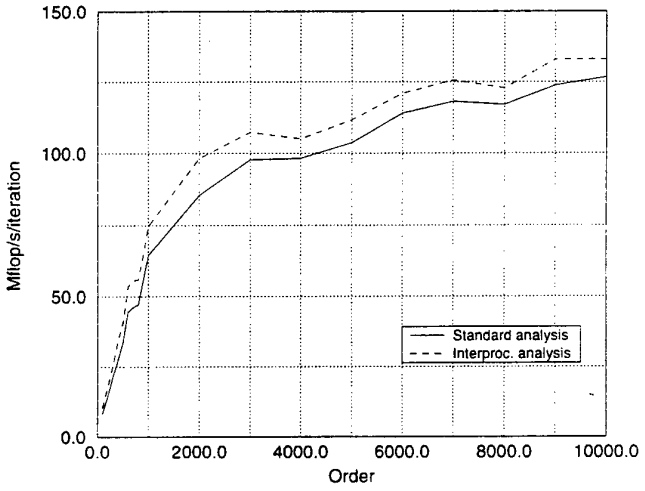


Fig. 7. *Performance per iteration for sparse eigenvalue computation without and with interprocedural analysis*

Figure 7 shows that the interprocedural analysis results in a small but consistently better performance over the whole problem range. The difference becomes slightly larger for larger problem size because in this case the floating-point operations in the generalised Lanczos routine more strongly dominate the computation.

The floating-point operations on the diagonals of the matrices are typical vector operations as were measured in program `mod1ac` and therefore the kernels from `mod1ac` should predict the speed of the Lanczos routine to a reasonable extent. The mix of floating-point operations as measured in `mod1ac` was as follows:

Dotproduct	34.3%
Kernel 10	25.7%
axpy	22.9%
Dyadic mult.	17.1%

The weighted average of the peak speeds of these operations in the primary cache is 134.5 Mflop/s. From Figure 7 we see that with interprocedural analysis the speed for the largest problem is 133 Mflop/s and without interprocedural analysis we find 127 Mflop/s. This is in excellent agreement with the speeds found for the kernels of `mod1ac`. This consistency shows that in the right context the prediction of the performance from kernel speeds might help to understand the observed performance. The right context is important though, as was demonstrated with program `mod2a`.

In the present form program `mod2e` is badly suited for parallelisation. Therefore no parallel results are presented.

4.3 Module 3 results

In module 3 various programs are considered that represent important classes of applications. The programs have been tailored in the sense that only the essential floating-point parts have been retained as this is our main concern. However, the first two programs in this module are designed to test important I/O patterns to obtain an idea of the I/O capabilities of the systems considered. Again, we do not discuss the full range of programs in this module. See [3] for the complete results.

Most of the programs in this module have a complexity that makes it difficult to estimate their Mflop-rate. So, mainly execution times are reported. In addition, only one of the programs was amenable for parallelisation (program `mod3h`). On the other hand, many module 3 programs have a complexity that made it worthwhile to subject them to interprocedural analysis.

To place the results in context, we added timings of two other systems: the T3E-Classic and the IBM RS/6000 SP.

PDE programs In module 3 three implementations of Elliptic/Parabolic PDE solvers are included. programs `mod3c` a Multigrid solver, `mod3g` a Fast Elliptic solver, and `mod3h` a Block Relaxation solver, respectively. They all solve the same model problem a Laplace equation on the unit square. They differ vastly in their solution speed for this particular problem but each method has its own virtues that make them more or less complementary. The execution times are given in Table 4. As can be seen from the Table, a single node of the the T3E is consistently slower than those of the IBM SP and the Origin2000. Note that only in program `mod3c` the IBM SP is significantly faster than the Origin2000, although the theoretical peak performance is much higher: 640 vs. 390 Mflop/s. Furthermore, it turns out that interprocedural analysis gives a very slight advantage over the normal analysis. In general, for the programs of this module the effects of interprocedural analysis were not large.

ODE program In program `mod3f` the problem of gas diffusion into a porous medium is considered. In this program two gases with different diffusion coef-

System	mod3c seconds	mod3g seconds	mod3h seconds
Cray T3E-Classic	2.424	0.114	10.083
IBM RS/6000 SP	0.970	0.083	3.670
SGI Origin2000	1.486	0.065	2.366
SGI Origin2000 Interproc. analysis	—	0.062	—

Table 4. *Execution times for three PDE solvers on the Cray T3E-Classic, the IBM RS/6000 SP, and the Origin2000.*

System	Execution time seconds
Cray T3E-600	16.003
IBM RS/6000 SP	8.5646
SGI Origin2000	8.7060
SGI Origin2000 Interproc. analysis	7.6141

Table 5. *Performances in seconds in program* `mod3f` *for various systems (single-node performance).*

ficients are modeled. The implementation is such that a time sequence of stiff two-point boundary-value ODEs is solved. The timing results for the program are displayed in Table 5. Table 5 shows the same general pattern as was found for the PDEs: the T3E is notably slower than the other two machines while the IBM SP is only marginally faster than the Origin2000 with standard code analysis, notwithstanding its higher Theoretical Peak Performance. With interprocedural analysis, the Origin2000 is about 15% faster than with standard code analysis.

5 Summary and future work

The amount of information from our experiments has been vast and, although we have discussed them to a fair extent, we are sure that a more extensive analysis would still bring up new points in the interpretation. It would almost certainly also would give grounds for new experiments. In this study we also have refrained from hand-optimisation: we just let the compiler do the work with the appropriate complier options. Other subjects not considered but probably important are: the explicit placement of data on the Origin2000 system and the migration of data by the operating system to the processor that most uses them. On the other hand, a number of useful conclusions can be drawn from this study of which we list the main ones below:

- In many cases a large proportion of the Theoretical Peak Performance can be attained when operating from the primary cache. The performance with access from the secondary is generally 2–3 times slower, except for the division operation.

- The experiments in program `mod1ac` showed that one 8-byte operand can be loaded or stored from/to the primary cache. From the secondary cache this is about one operand per two cycles.
- When automatic parallelisation is applied, the default choices whether or not to parallelise a certain loop seem to be adequate in most cases we observed.
- The point-to-point bandwidth measured with MPI is about 110 MB/s, about 70% of the bandwidth of 150 MB/s quoted by SGI.
- The automatic shared-memory parallelisation of codes generates a non-negligible parallelisation overhead as shown by program `mod2a`, a matrix-vector multplication. Compared with the distributed-memory version it gives a large performance loss. On the other hand, as soon as also messages must be exchanged, the shared-memory implementation is clearly faster than the MPI version. The similar phenomenon was observed in the FFT program `mod2f`. Communication timings suggest that MPI implementation we used in the present tests is not optimal.
- In the rather small programs of module 3 interprocedural analysis generally had a quite modest influence on the execution time (5–15% decrease).

Acknowledgments

We would like to thank Silicon Graphics Inc. for making their Origin2000 system at the Advanced Technology Center in Cortaillod, Switzerland, available to us to conduct the experiments described in this report.

References

1. A.J. van der Steen, *The benchmark of the EuroBen Group*, Parallel Computing. **17**, (1991) 1211-1221.
2. Silicon Graphics Inc., *Origin Servers*, Technical Report, April 1997.
3. A.J. van der Steen, R. van der Pas, *Benchmarking the Silicon Graphics Origin2000 system*, Technical Report WFI-98-2, Utrecht University, May 1998.
4. M. Galles, *Spider: A High-Speed Network Interconnect*, IEEE Micro, **17**, 1, (1997).
5. ANSI Standard Commitee X3H5, *Fortran language binding*, X3H5 Document Number X3H5/91-0023 Revision B, 1992.
6. `http://www.openmp.org/`
7. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, *MPI: The Complete Reference*, MIT Press, Boston, 1996.
8. A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jaing, and V. Sunderam, *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Boston, 1994.
9. High Performance Fortran Forum, *High Performance Fortran Language Specification*, Scientific Programming, **2**, 13, (1993) 1–170.
10. R.W. Hockney, $f_{1/2}$: *A parameter to characterize memory and communication bottlenecks*, Parallel Computing, **10** (1989) 277-286.

Parallel Computing over the Internet with Java

Hernâni Pedroso, Luís M. Silva, Víctor Batista, Paulo Martins, Guilherme Soares and Telmo Menezes

Departamento de Engenharia Informática
Universidade de Coimbra – POLO II
Vila Franca – 3030 Coimbra
PORTUGAL
{luis,hernani}@dei.uc.pt

Abstract. JET is a parallel library implemented with Java for parallel computing over the Internet. The JET library is oriented to long-running Master/Worker applications with a coarse-grain task distribution. The computation is performed by Java applets that are downloaded through a Web page. The paper describes some internals of JET and its mechanisms to provide support for fault-tolerance, interoperability with PVM/MPI and the use of statistics. The paper includes some performance figures that were taken with simple benchmarks and more complex applications.

1. Introduction

In the last years we have seen an extraordinary increase in the number of machines that are connected to the Internet, this is estimated to continue with an exponential growth. According to a survey accomplished by Network Wizards [NetWizards] in January 1998, 29.6 millions hosts were connected to the Internet (against 16 million in January 1997). This mass of processors connected together represent a very significant processing power, with a performance level of a Petaflop (10^{15}).

In a large percentage of their time, workstation machines and personal computers are only used to small iterative tasks, such as reading mail or editing files. As was remarked in [Schrage92] workstations remain idle in about 90% of their time.

The idea of using this spare computational power in computers that are connected to the Internet seems to be quite promising and is getting an enthusiastic acceptance within the high-performance computing community. Two main things are required:

- appropriate applications, that take a long time to execute and have low communication requirements;
- an effective infrastructure to support the execution of massively parallel applications in hundreds or thousands of computers geographically dispersed through the Internet;

The main challenge of JET is to provide such infrastructure. It was implemented in Java [JavaSoft] to provide the portability of code, to solve the problem of heterogeneity of systems and to allow the easy distribution of code through the machines that want to volunteer their CPU spare cycles for solving a massively parallel application.

Applications that are good candidate programs to the JET parallel machine should divide the problem into small tasks to be executed by different processors distributed over the Internet. Those applications should be coarse-grained, take a long time to execute, do not require ultimate performance and should tolerate, in some extent, the low latency of the network. There are some quite important applications from the field of cryptography and mathematics that can be effectively executed with JET.

2. JET Architecture

The applications that can be executed with JET follow the Master/Worker paradigm. There is a process, the Master, which is responsible for the decomposition of the problem into small and independent tasks. The tasks are distributed among the worker processes, which are executing a simple cycle: receive a task, compute it and send back the result. The results are gathered by the Master process, which merges them to construct the final solution. Since every task is independent from each other, there is no need for communication between the worker processes.

JET is non-intrusive to the machines that access any Web page: only those users that are willing to volunteer their CPU time will have an applet working on their computer contributing for a JET computation. The users that wish to volunteer to a JET computation have to access to a Web page using a Java-enabled browser and follow a Web link. The downloaded Web page has an inlaid Java applet (Worker applet) which will indicate the status of the computation and communicates with the JET Master.

The security features of Java only allow the applets to communicate with the machine from where they where downloaded. Hence, the Master process has to be executing in the same machine where the http daemon is executing. It has a well-known port to all the Workers. The communication between Workers and Master is done through UDP sockets. Although the UDP protocol does not guarantee the delivery of messages, it provides a higher scalability and consumes fewer resources than TCP sockets. The communication layer of JET implements a reliable service that assumes sequenced and error-free message delivery.

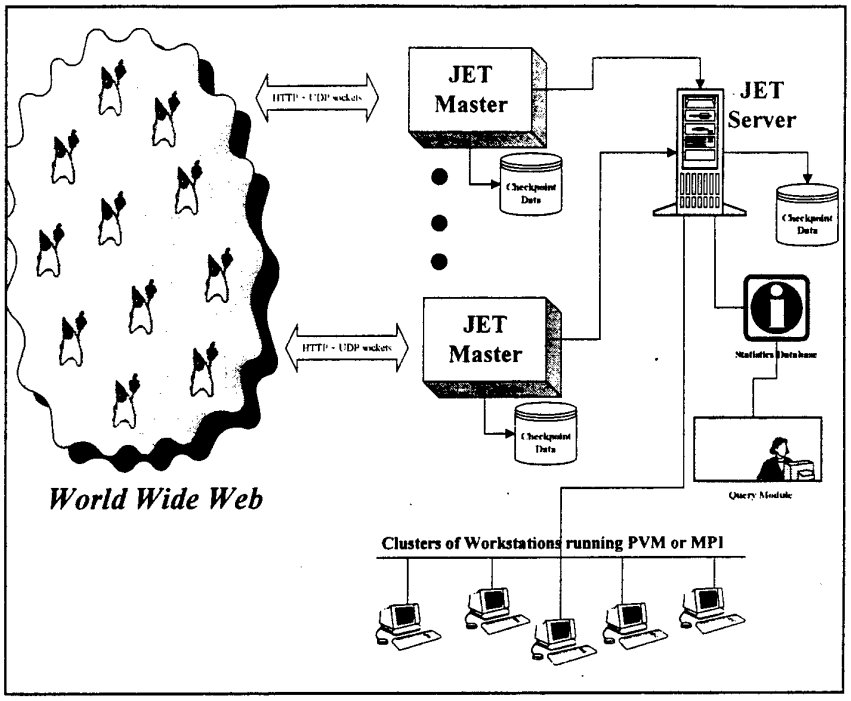


Fig. 1. The structure of the JET virtual computation.

The JET library as a server checkpointing mechanism to assure the continuity of the application when there is a failure or a preventive shutdown of the JET Server. The critical state of the application is saved periodically in stable storage in some portable format that follows its resumption later in the same or different machine.

To tolerate the loss of the stateless worker applets the JET library maintains a task-reconfiguration scheme. The library keeps the jobs that have been sent to each worker applet. If one applet fails or withdraws from the virtual machine, the only part of the computation that is affected is the task it was being executed. Re-allocating that task to another worker would reproduce the lost work without changing the ultimate outcome of the computation. However, for those applications with very long-running tasks it is important to save intermediate states of the task execution in the worker applets.

Implementing client checkpointing is not trivial in a Java applet since it cannot write to the local disk. Thereby, the only way he had to implement the client checkpointing was to send the checkpoint data over a socket stream to the associated JET Master. When a Worker applets withdraws from the virtual machine the last checkpoint of its task is distributed to another worker.

The JET machine needs to motivate the Web surfers to participate in the computation, and even on interesting applications is necessary to increase their

enthusiasm. The JET Server gathers information about the computation done by each volunteer and creates a statistics module with several rankings. The statistical information, organized by several categories (e.g. users, countries, operating systems, processors and browsers) ranks, is published on the Web. The users are also able to create teams. These rankings create a healthy competition between users and keeps their interest to participate in the computation.

JET is not restricted to Web-based computation. The use of some existing parallel libraries and computer resources is also be possible. The basic idea is to allow existing clusters of machines running PVM or MPI to inter-operate with JET computations.

To achieve this we have used two Java bindings developed in our research group for Windows versions of the MPI (WMPI) [WMPI] and PVM (WPVM) [Alves95] libraries. The big master process of the PVM/MPI cluster only needs to create an instance of a class that implements a *bridge* between the cluster and the JET Master. The jobs are fetched by this object and placed in an internal buffer of the PVM/MPI big master, which is responsible to distribute them among the workers of the cluster. The results are gathered by the big master of the cluster and passed to the *bridge* object to be sent to the JET Master.

3. Performance Results

In this section, some performance results of JET are presented. These measurements were taken in a heterogeneous environment of NT and Solaris Workstations. The workers were running on 6 PentiumPro-based machines, all of them running at 200 MHz, with the NT Workstation operating system. Two of those machines are dual-processor; hence, in overall the performance results were taken with 8 processors. The Master process was running on a Sun Ultra-Sparc machine running Solaris V4.0. The machines were connected through a non-dedicated 10 Mbit/sec Ethernet network. The Worker applets were executed through the Netscape Communicator 4.0; the Master process was executed with JDK 1.1.

3.1 Simple Benchmarks

The relative speedup of the NQUEENS application with 14, 15 and 16 queens is presented in Figure 2. In this example, the speed up was calculated with the parallel version of the algorithm running on one processor. The achieved results are quite good: with 8 processors the speed up was 7.66, 7.36 and 7.24 with 14, 15 and 16 queens respectively. The reason why the speedup decreases with the increase of the number of queens is due to small differences of performance of the processors that are more visible with larger jobs. Hence, the time that the JET machine has to wait for the last job increases with the size of the jobs. Although the task distribution of JET has intrinsic load-balancing behavior, they can not tolerate these fine-grain differences.

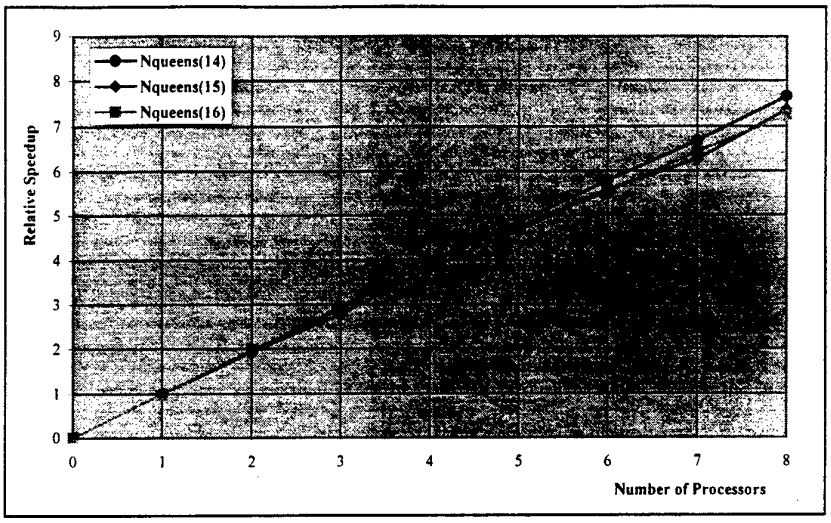


Fig. 2. Relative speedup of NQUEENS (14, 15 and 16 queens).

The EP-NAS application, which makes part of the NAS benchmark suite [Bailey93], was also used as benchmark. Due to the temporary unavailability of the dual-Pentium machines, the results were taken in just four processors. The speedup presented in Figure 3 was calculated with a serial Java version of the program.

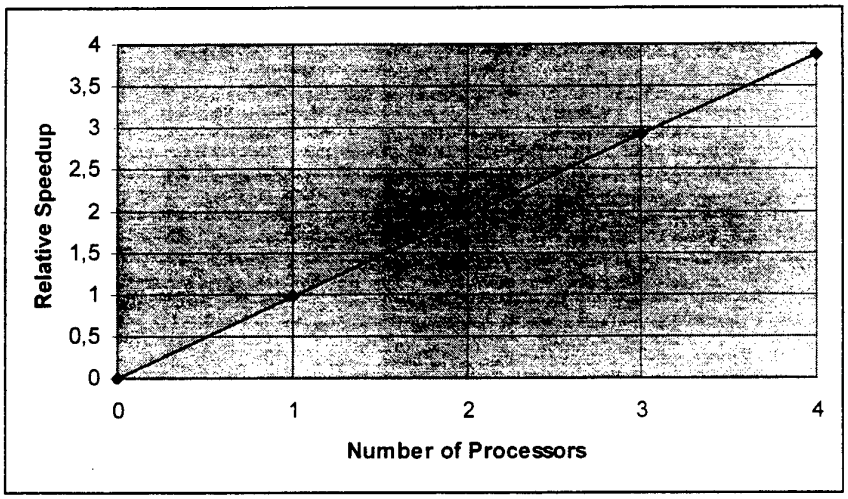


Fig. 3. Relative Speedup of EP-NAS.

Although EP-NAS problem has a significant amount of floating-point calculations the performance of Java, and therefore JET performance, was not affected since the speedup once again is quite good: 3.87 with 4 processors.

Although the speedup results are always dependent from the characteristics of each application, these results show that JET does not degrade the performance with the Increasing number of processors.

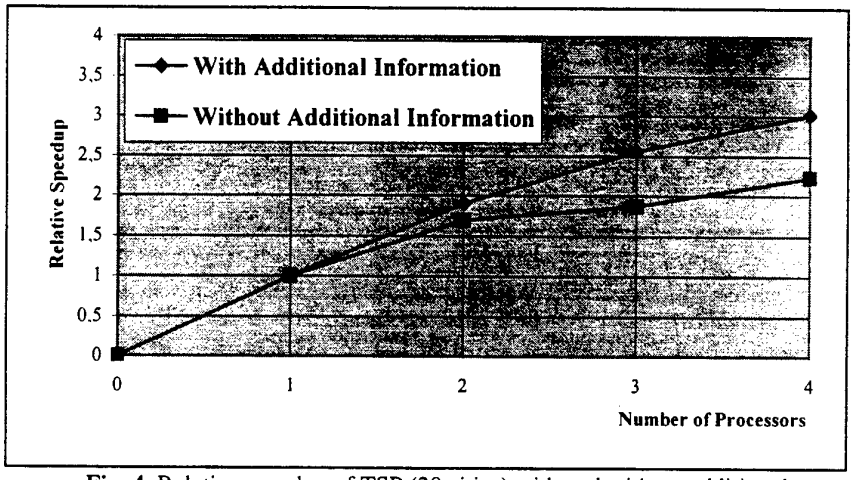


Fig. 4. Relative speedup of TSP (20 cities) with and without additional information.

The next experiment was made with an application that has different characteristics from the last two. In the Travel Salesman Problem (TSP), additional information was passed asynchronously to the Workers, which is enabled by the JET library. Each Worker is informed if a shorter path (new minimum path) was found by another Worker every time a result with a new minimum arrives to the Master. A version without this capability (in this case each worker only knows its minimum) also was implemented. Figure 4 presents the relative speedup achieved by the two versions when searching on a 20 cities map.

The application, due to its intrinsic characteristics, does not scale as well as the previous examples. The version that does not use the JET library capability of pass information additional information to the workers does not scale so well when compared with the other one.

3.2 Complex Applications

Besides these simple benchmark applications, a few more complex applications were ported to JET: a program to find Mersenne Primes [Mersenne] and a RC5 (64-bit key) encryption algorithm [Rivest95] crack application.

The RC5 encryption attack is an example of a embarrassingly parallel application. The jobs are a set of keys to be tested, by using them to decrypt the message and test is if it is the correct one. The result only has to indicate if the correct key was in the tested set and the correct key. The key-space to be searched is enormous and a concerted world effort [Bovine] is on the way to crack this code. The JET computation is a candidate to join this effort and use Web-based computation to help finding the correct key.

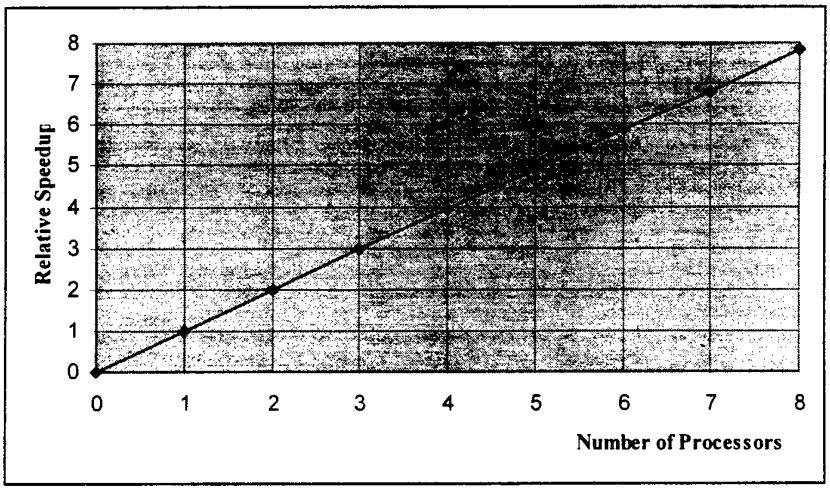


Fig. 5. Speedup of the RC5 64-bit encryption attack application.

Figure 5 shows the speedup achieved by JET when computing this application. The speedup was calculated with a serial Java version of the application.

The Merssene Primes search application was tested with two versions, the difference between these versions is the order by which the numbers are searched. The version which starts from the higher number has a better speedup (Figure 6). This fact occurs due to the better task distribution achieved by JET. The size of the jobs grows exponentially with the increase of the number to be searched. If the biggest task is the last to be assigned, then all the other processes will stall waiting for that task to be ended. However, if the largest task is the first one, all the other processes will be working (on other tasks). At the final of the computation, the tasks are so small that the time to wait for the end of the last job is very small.

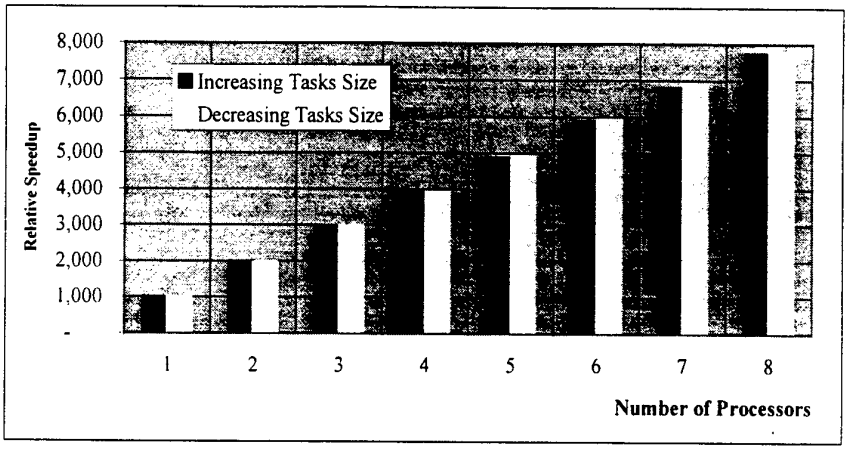


Fig. 6. Speedup of the Mersenne Primes search application.

Figure 6 presents the speedup of the Mersenne Primes search, relative to a serial Java version of the application. As it can be seen the version with decreasing tasks size scales better, this shows the importance of a correct task distribution.

4. Related Work

In the past years several projects have confirmed the ability of the Internet for massively parallel computing. In [Silverman91] was presented an example of massively distributed computing over the Internet. It used 400 machines that were located at research institutes of three different continents. The problem was the factorization of a 100-bits integer used by the RSA cryptographic algorithm. Each site has received by electronic mail a set of polynomials to independently work with. It took 275 MIP-Years to perform one of the computations. The project has been active [RSAFact] since then and the factoring of 130-bits number was successfully solved in November 1996. To solve this problem a collection of CGI scripts were used to automate and coordinate the flow of tasks within the distributed network of Web sieving clients.

Another representative example is the Gordon Bell Prize of 1992 big winner: a collection of 192 heterogeneous machines scattered around the United States was used to solve a simulation of polymer chains [Karp93]. The outstanding

[1] One MIP-Year is referred as the amount of work performed by 1-MIP machine running for one year.

price/performance ratio achieved granted the prize to the project. In [Nieplocha96] is presented another project which had used four supercomputers located in geographically dispersed computing centers of the United States connected together to compute a molecular simulation program. The speedups achieved were quite good. Another interesting example was presented in [Strumpen93]. This paper describes the use of 800 workstations to solve a problem that involved molecular sequence analysis. The machines were dispersed through 31 different local area networks and 5 continents.

More recently, there were other remarkable examples of Internet parallel computing. For instance, in February of 1997 a team of researchers using 3500 computers spread across Europe was able to crack a RSA code of 48 bits in less than two weeks [Lash97].

In January 27th of 1998 a Californian 19 year-old student found the 37th Mersenne Prime (the world's largest known prime) on behalf of the GIMPS project (Great Internet Mersenne Prime Search) [GIMPS]. The computation comprised about 4000 users that volunteer their machines to that computation and the lucky man was Roland Clarkson, that have contributed with his 200 MHz Pentium computer for 46 days, in part-time, to prove the number prime.

Finally, in October 19th of 1997, it was announced that one of the largest distributed-computing effort ever seen, involving tens of thousands of computers connected to the Internet: the Bovine cooperative effort [Bovine] decrypted a message encoded with RSA Labs' 56-bit RC5 encryption algorithm. The search took 250 days of massive Internet computing: the medium computational power was equivalent to 14,685 Intel Pentium Pro 200 processors. This time the lucky man that found the right key was Peter Stuer from Belgium.

All these examples demonstrate that the use of worldwide-distributed computing resources is feasible to perform large computations.

In the latest years, the exploitation of geographically distributed machines for parallel computing has become a clear trend. A considerable number of project have been proposed: Globe [Steen95], Legion [Grimmshaw96], Globus [Foster96], Atlas [Baldeschweiler96], ParaWeb [Brecht96], Popcorn [Camiel96], Charlotte [Baratloo96], DAMPP [Vanhelsuwe97], IceT [Gray97], Javelin [Cappelo97], JavaParty [Philippsen97], Albatross [Bal97], among others.

Some of these projects were also developed in Java: Javelin, Popcorn, DAMPP, Charlotte, JavaParty, Atlas, ParaWeb, IceT and Albatross. Most of these systems lack some support of fault-tolerance, scalability, support for interoperability with other existing tools and a module of statistics to motivate Internet users to participate in Web-based computations.

Although there are some differences between these projects and JET, all of them try to prove the idea that Java can be used for parallel computing over the Internet. It would be interesting that some standardization protocols could be developed to allow the cooperative execution of JET and any of these Java-based parallel tools. This way the number of machines working out on the same global computation could be extended.

5. Final Conclusions

JET can be a massively parallel machine. It may compromise several hundreds of machines connected to the Internet. Each machine that takes part on a JET computation is absolutely ubiquitous: it just requires a Java-enabled browser. The user can volunteer his CPU spare cycles just by clicking in some URL of a Web page. A Java applet is downloaded to that machine and executes some independent tasks of a number-crunching application. JET is a really inexpensive parallel computing platform: it is based on the idea of "scavenging" the idle CPU cycles of machines that are connected to the Internet, reusing the existing computing facilities.

Some built-in features provide support for fault-tolerance on the JET computation, interoperability with PVM and MPI libraries and the usage of statistics to keep the motivation and enthusiasm of the user volunteers.

The first performance results of JET with simple benchmarks were very promising. When complex application were ported to JET the results achieved have confirmed the ability of JET to be used for massively parallel computing over the Internet.

References

[Alves95] A.Alves, L.M.Silva, J.Carreira, J.G.Silva, *"WPVM: Parallel Computing for the People"*, Proc. of HPCN'95, High Performance Computational and Networking Europe, May 1995, Milano, Italy, Lecture Notes in Computer Science 918, pp. 582-587

[Bailey93] D.Bailey, E.Barszcz, L.Dagum, H.Simon. *"NAS Parallel Benchmark Results"*, IEEE Parallel and Distributed Technology, pp. 43-51, February 1993

[Bal97] H.Bal, "Albatross – Wide Area Cluster Computing", `http://www.cs.vu.nl/albatross`

[Baldeschwieler96] J.Baldeschwieler, R.Blumofe, E.Brewer. *"ATLAS: An Infrastrusture for Global Computing"*, Proc. of HPCN'95, High Performance Computing and Networking Europe, Lecture Notes in Computer Science 918, pp. 582-587, Milano, Italy, May 1995

[Baratloo96] A.Baratloo, M.Karaul, Z.Kedem, P.Wyckoff. *"Charlotte: Metacomputing on the Web"*, Proc. ISCA Int. Conf. on Parallel and Distributed Computing, PDCS'96, Dijon, France, pp.181-188, Sept. 1996

[Bovine] Distributed.net, `http://www.distributed.net/`

[Brecht96] T.Brecht, H.Sandhu, M.Shan, J.Talbot. *"ParaWeb: Towards World-Wide Supercomputing"*, Proc. 7th ACM SIGOPS European Workshop on System Support for Worldwide Applications, 1996

[Cappelo97] P.Cappelo, B.Christiansen, M.F.ionescu, M.Neary, K.Schauser, D.Wu. *"Javelin: Internet-based Parallel Computing using Java"*, ACM 1997 Workshop on Java for Science and Engineering Computation, Las Vegas, USA, June 1997

[Camiel96] N.Camiel, S.London, N.Nisan, O.Regev. *"The Popcorn Project: Distributed Computation over the Internet in Java"*, Proc. 6th International World Wide Web Conference, April 1997

[Foster96] I.Foster, S.Tuecke. *"Enabling Technologies for web-Based Ubiquitous Supercomputing"*, Proc. 5th IEEE int. Symposium on High-Performance Distributed Computing, HPDC-5, Syracuse, USA, pp. 112-119, August 1996

[GIMPS] `http://www.utm.edu/research/primes/notes/2976221/`

[Gray97] P.A.Gray, V.Sunderam, *"IceT: Distributed Computing and Java"*, 1997 ACM Workshop on Java for Science and Engineering Computation, Las Vegas, Nevada, June 1997

[Grimshaw96] A.Grimshaw, W.Wulf. *"Legion - A view from 50,000 Feet"*, Proc. 5th IEEE Int. Symposium on High-Performance Distributed Computing, HPDC-5, Syracuse, USA, pp. 89-99, August 1996

[JavaSoft] JavaSoft Homepage, `http://www.javasoft.com/`

[Karp93] A.H.Karp, K.Miura. *"1992 Gordon Bell Prize Winners"*, IEEE Computer, pp 77-82, January 1993

[Lash97] A.Lash. *"48-bit crypto latest to crack"*. CNET: The Computer Network, February 1997, `http://www.news.com/News/Item/0,4,7849,4000.html`

[Mersenne] Mersenne Primes: History, Theorems and Lists, `http://www.utm.edu/research/primes/mersenne.shtml`

[NetWizards] Network Wizards, `http://www.nw.com/`

[Nieplocha96] J.Nieplocha, R.J.Harrison. *"Shared Memory NUMA Programming on I-Way"*, Proc. 5th IEEE Int. Symposium on High-Performance Distributed Computing, HPDC-5, Syracuse, USA, pp432-441, August 1996

[Philippsen97] M.Philippsen, M.Zenger, "JavaParty – Transparent Remote Objects in Java", 1997 ACM Workshop on Java for Science and Engineering Computation, Las Vegas, Nevada, June 1997

[Rivest95] Rivest Ronald L. *"The RC5 Encryption Algorithm"* Proceedings of the Second International Workshop on Fast Software Encryption, Leuven, Belgium, pages 86-96, Springer-Verlag, January 1995

[RSAFact] RSA Factoring-By-Web Project, NPAC Syracuse, USA, `http://www.npac.syr.edu/factoring.html`

[Schrage92] M.Schrage. *"Piranha processing - utilizing your down time"*, HPCwire (Electronic Newsletter), August 1992

[Silverman91] R.D.Silverman. *"Massively Distributed Computing and Factoring Large Integers"*, Communications of the ACM, Vol. 34, No 11, pp. 95-103, November 1991

[Steen95] M.V.Streen, P.Homburg, L.V.Doorn, A.S.Tanenbaum. *"Towards Object-Based Wide Area Distributed Systems"*, Proc. Int. Workshop on Object Orientation in Operating Systems, Lund, Sweden, pp. 224-227, August 1995

[Strumpen93] V.Strumpen. *"Parallel Molecular Sequence Analysis on Workstations in the Internet"*, Technical Report 93-28, Department of Computer Science, University of Zurich, 1993

[WMPI] Windows Message Passing Interface, `http://dsg.dei.uc.pt/wmpi/`

[Vanhelsuwe97] L.Vanhelsuwe. *"Create your own Supercomputer with Java"*, Javaworld, January 1997, `http://www.javaworld.com/`

The Parallel Problems Server: A Client-Server Model for Interactive Large Scale Scientific Computation

Parry Husbands[1] and Charles Isbell[2]

[1] Laboratory for Computer Science, MIT, Cambridge MA 02139 USA
parry@supertech.lcs.mit.edu
[2] Artificial Intelligence Laboratory, MIT, Cambridge MA 02139 USA
isbell@ai.mit.edu

Abstract. Applying fast scientific computing algorithms to large problems presents a difficult engineering problem. We describe a novel architecture for addressing this problem that uses a robust client-server model for interactive large-scale linear algebra computation.
We discuss competing approaches and demonstrate the relative strengths of our approach. By way of example, we describe MITMatlab, a powerful transparent client interface to the linear algebra server. With MITMatlab, it is now straightforward to implement full-blown algorithms intended to work on very large problems while still using the powerful interactive and visualization tools that Matlab provides. We also examine the efficiency of our model by timing selected operations and comparing them to commonly used approaches.

1 Introduction

We describe a novel architecture for a "linear algebra server" that operates on very large matrices. Matrices are created by the server and distributed across many machines or processors. Operations take place automatically in parallel. The server includes a general communication interface to clients and is extensible via a robust package system.

We are motivated by three observations. First, many widely-used algorithms in machine learning, differential equations, simulation, etc. can be realized as operations on matrices. Second, it is vital to be able to test new ideas quickly in an interactive setting. Finally, algorithms that appear promising on small data sets can fail on large problems and it would be helpful to have a tool that easily enables experimentation on large problems.

Common approaches suffer from several difficulties. Interactive prototyping environments such as Mathematica, Maple, Octave, and Matlab exist; however, they often fail to work well on large problems. Linear algebra libraries designed to work on large problems abound; however, they involve steep learning curves. Further they are typically not interactive, requiring that applications be written in a compiled language, such as C++ or Fortran. This is a burden for users who

simply want a library's functionality and for programmers who wish to extend it.

We address these problems directly. Like standard libraries, our system encapsulates basic functionality; however, by modeling the system as a server, we allow for on-the-fly interaction with arbitrary user interfaces. Further, the server is a self-contained application, so we are able to extend it at run-time.

In this paper, we show that our model opens several possibilities. We briefly describe standard approaches in Section 2 before describing the Parallel Problems Server itself in Section 3. We detail its architecture, focusing on its extensibility. Section 4 describes MITMatlab, a system that enables users to compute interactively with very large data sets directly from within Matlab. We then report on the results of some performance experiments in Section 5. Finally, we conclude, discussing further extensions to the system.

2 Standard Approaches

2.1 Linear Algebra Libraries

For many compute-intensive tasks, the best way to maximize performance is to use a library. For example, optimized versions of LAPACK [1] exist that outperform similar code written in a high-level programming language (thanks primarily to native implementations of the BLAS). For distributed memory architectures, vendor-optimized libraries (e.g. Sun's S3L and IBM's ESSL) coexist with public domain offerings such as ScaLAPACK [5], PARPACK [11] and Petsc [4][9].

Each of these libraries has its own idiosyncratic interface and assumptions about the types and distributions of data allowed. It is often a major programming effort to incorporate library routines into an application.

2.2 Interactive Systems

The power of prototyping systems like Maple, Matlab, Mathematica and Octave is that they are interactive. It is straightforward for both seasoned programmers and relatively naive users to develop algorithms and to visualize results from such algorithms. Unfortunately, while these tools work well for small problems, they are often inadequate for production-level data.

There have been many attempts to extend prototyping tools in order to make them work in parallel with large data sets. Here, we focus on systems that add parallel features to Matlab, a widely-used scientific computing tool.

Both MultiMatlab from Cornell University [13] and the Parallel Toolbox for Matlab from Wake Forest University [10], make it possible to manage Matlab processes on different machines. Matlab is extended to include *send*, *receive* and *collective* operations so that separate Matlab processes can communicate. In short, these approaches implement traditional message passing with Matlab as the implementation language.

fashion and managed among worker processes, which may live on different machines. Currently we support row and column distributed dense arrays, column distributed sparse arrays, and replicated arrays in single precision. Communication and synchronization among the workers is accomplished using the MPI [8] message passing library. This is a standard library available on a wide range of platforms; it is currently the most portable way to develop applications on distributed memory computers.

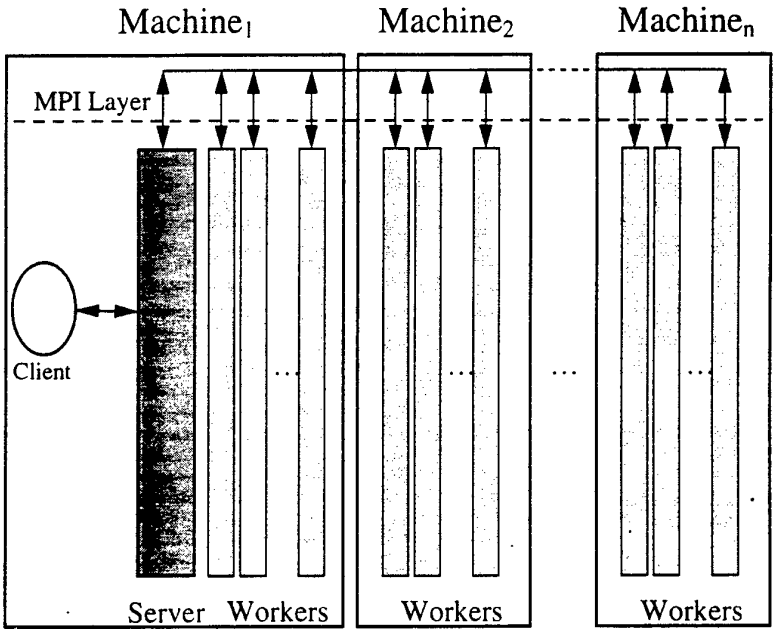


Fig. 1. The General Organization of the Parallel Problems Server. The server process provides an interface to any client that implements its communication protocol.

3.2 Communication and Extensibility

We use the client-server model in two ways. First, there is a protocol for communicating with clients. Just as importantly, there is a separate plug-in architecture that allows for straightforward run-time extensibility of the PPServer.

The Client Interface While we believe that servers are crucial, they remain only academic oddities without useful clients. HTTP servers are useful but they are much more useful when powerful browsers exist. Therefore, it is important

Compilers for Matlab are also an active area. Both the CONLAB system from the University of Umeå [7] and the FALCON environment from the University of Illinois at Urbana-Champaign [3][12] translate Matlab-like languages into intermediate languages for which high performance compilers exist. For example, FALCON compiles Matlab to Fortran 90 and pC++. Sophisticated analyses of the Matlab source are performed so that efficient target code is generated.

Both of these approaches have merits; however, it is our claim that they do not adequately address the issues we have raised. The former approach is too involved for the naive user and the latter approach sacrifices direct interaction with the computation and includes an edit-compile-run cycle that increases development time.

3 The Parallel Problems Server

The Parallel Problems Server (PPServer) combines many aspects of the approaches we have described so far. Like standard linear algebra packages, the PPServer neatly encapsulates basic functionality; however, because it is a server with a general communication protocol, interaction with arbitrary programs (with their own user interfaces) is possible. Also, the server implements a robust protocol for accessing compiled libraries. Thus, extending the functionality of the PPServer is a simple, modular task.

3.1 The Client-Server Model

The client-server model is ubiquitous. There are HTTP servers that allow access to data via the World Wide Web and database servers that admit access to specially indexed data. Because these servers implement robust protocols for communicating the information they provide, it is possible to build useful clients, such as web browsers.

We believe that this model is also a useful one for scientific computation. First, there is no need to force a client to operate in parallel by endowing it with communication primitives; rather, such communication remains implicit. As a result, the user is not responsible for managing data among various processes. The user simply issues the client's standard commands; these are then transparently executed on multiple machines.

Secondly, there is no need to use the client as the computational engine. While this has the possible short-term disadvantage of the server's functionality being different than the client's, we gain extremely high performance. We are free to use the fastest distributed memory implementations of the algorithms that we need. Furthermore, we are not required to use the client's data representation. For example, Matlab uses double precision numbers. For the very large operations that concern us, it often preferable to use single precision, gaining significant time and space advantages when accuracy is not a concern.

A high-level view of our implementation of the PPServer is shown in Figure 1. Clients make requests of the server. Data are created in a distributed

that the client interface be simple to use but powerful enough to allow for arbitrary operations.

The PPServer uses standard Unix sockets for client communication. The protocol is straightforward. A client sends a request, consisting of a command and arguments. A command is a string, naming a function. Functions may request data or the loading, saving, or creating of data. Furthermore, they may require that specific operations be performed on already existing data or that library extensions to be included with the server. Arguments are lists of characters, integers and real numbers. Once a command has been completed, it is acknowledged with a message from the server that includes any errors and returned values.

A C++ library (and source) is provided that implements this protocol, including automatic conversion between standard C/C++-style data types and a form suitable for transmission to/from the server. Clients need only provide a suitable wrapper for these functions.

The Server Interface The PPServer is extensible (see Figure 2). It includes a robust function interface using C++ objects. New functions are defined using this interface. These new functions are compiled into dynamically loadable libraries, dubbed "packages" and loaded on demand. Each package is its own name space, so new functions can be loaded "on top" of others, hiding functions of the same name in other packages. Like the PPServer itself, package functions use MPI. These functions enjoy access to the basic functionality of the Server, including direct access to data and the ability to execute all the same commands that are available to clients, including those in other packages.

Figure 3 shows the code for a sample package. It contains one function **sumall** that sums the elements of a distributed matrix. This example shows the mechanisms for extracting input arguments, accessing the elements of the matrix, and returning results to the client. With only a handful of exceptions, all current server functionality is written in this way.

We have used the PPServer as the core of several applications, implementing packages that provide access to ARPACK, SCALAPACK and S3L, Sun's optimized version of SCALAPACK. The functions in the packages are merely short wrappers for the underlying functions provided by the libraries.

Portability The use of standard C++ and MPI has allowed us to develop a system that is highly portable. Although the PPServer was originally developed on a network of symmetric multiprocessors from Sun Microsystems, we have been able to port it to a cluster of SMPs from Digital Equipment Corporation with minimal effort. We are currently working on a port to Pentium-driven Linux systems.

3.3 Other Client-Server Models

There have been previous library systems that implement a similar model. Both RCS [2] and Netsolve [6] act as fast back-ends for slower clients. In their model,

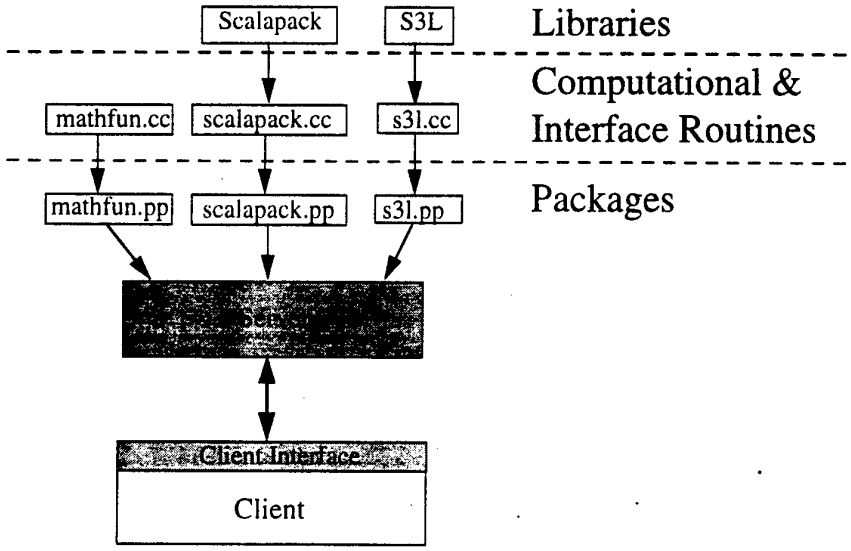


Fig. 2. Extending the PPServer. A client communicates with the PPServer using a simple command-argument protocol. The Server itself uses a "package" mechanism to implement all but its most basic functions. New functionality can be added to the PPServer and managed in a reasonable way. (S3L is Sun's optimized version of some ScaLAPACK routines)

```
void sumall(PPServer &theServer, PPArgList &inArgs, PPArgList &outArgs)
{
  // Get the matrix identifier that was passed in
  PPMatrixID srcID=*(inArgs[0]);

  // Make sure that we're passing in a dense matrix
  if(!theServer.isDense(srcID)) {
    // Return the corresponding error
    outArgs.addError(BADINPUTARGS,"Expecting a Dense Matrix");
    outArgs.add(0);
    return;
  }

  // Get a pointer to the actual matrix
  PPDenseMatrix *src = (PPDenseMatrix *) theServer.getData(srcID);
  float sum=0, answer;

  // Find the local sum of all of the elements
  for(int i=0;i < src->numRows();i++)
    for(int j=0;j < src->numCols();j++)
      sum+=src->get(i,j);

  // Add the local sums to find the global sum
  MPI_AllReduce(&sum,&answer,1,MPI_FLOAT,MPI_SUM,MPI_COMM_WORLD);

  // Return an error code
  outArgs.addNoError();

  // Return the result to the client
  outArgs.add(answer);
}

// Register this function to the server
extern "C" PPError ppinitialize(PPServer &theServer);
PPError ppinitialize(PPServer &theServer)
{
  theServer.addPPFunction("sumall",sumall);
  return(NOERR);
}
```

Fig. 3. A Sample Server Extension. This code is essentially complete other than a few header files

clients issue requests, arguments are communicated to the remote machine and results sent back. Clients have been developed for Netsolve using both Matlab and Java.

Our approach to this problem is different in many respects. Our clients are not responsible for storing the data to be computed on. Generally, data is created and stored on the server itself: clients receive only a "handle" to this data (see Figure 4 for an example). This means that there is no cost for sending and receiving large datasets to and from the computational server. Further, this approach allows computation on data sets too large for the client itself to even store.

We also support transparent access to server data from clients. As we shall see below, given a sufficiently powerful client, PPServer variables can be created remotely but still be treated like local variables.

Both Netsolve and RCS assume that the routines that perform needed computation have already been written. Through our package system we support on-the-fly creation of parallel functions. Thus, the server is a meeting place for both data and algorithms.

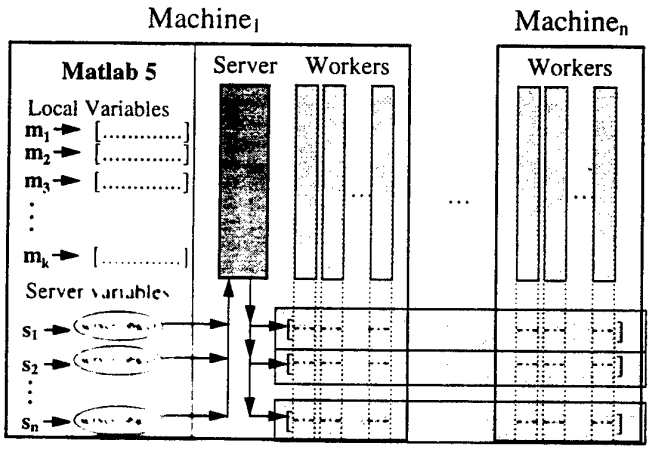


Fig. 4. MITMatlab Variables. Use of the PPServer by Matlab is almost completely transparent. PPServer variables remain tied to the server itself while Matlab receives "handles" to the data. Using Matlab scripts and Matlab's object and typing mechanisms, functions using PPServer variables invoke PPServer commands implicitly.

4 MITMatlab

Using the client interface, we have implemented a Matlab front end, called MITMatlab. At present, we can process gigabyte-sized sparse and dense matrices "within" Matlab, admitting many of Matlab's operations transparently (see Figure 5). By using a client as the user interface, we take advantage of whatever interactive mechanisms are available to it. In Matlab's case, we inherit a host of parsing capabilities, a scripting language and a host of powerful visualization tools.

For example, we have implemented BRAZIL, a text retrieval system for large databases. BRAZIL can process queries on a million documents comprised of hundreds of thousands of different words. Because of Matlab's scripting capabilities, little functionality had to be added to the server directly; rather, most of BRAZIL was "written" in Matlab.

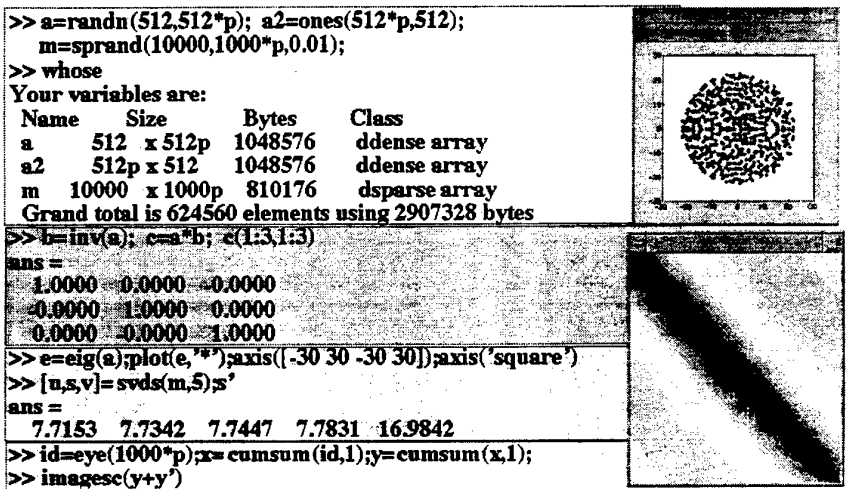


Fig. 5. A Screen Dump of a Partial MITMatlab Session. Large matrices are created on the PPServer through special constructors. Multiplication and other matrix operations proceed normally.

5 Performance

In this section we present results demonstrating the performance of the PPServer. We begin with experiments comparing the efficiency of individual operations in Matlab with the same operations using MITMatlab. We conclude with a case study of a computation that requires more than a single individual operation. We

compare the performance impact of implementing a short program in Matlab, directly on the PPServer, and using optimized Fortran.

5.1 Individual Operations

We categorize individual operations into two broad classes, according to the amount of computation that is performed relative to the overhead involved in communicating with the PPServer. For *fine grained* operations, most of the time is spent communicating with the server. A typical fine grained task would involve accessing or setting an individual element of a matrix. *Coarse grained* operations include functions such as matrix multiplication, singular value decompositions, and eigenvalue computations where the majority of the time is spent computing instead of communicating input and output arguments with the server.

Below we assess MITMatlab's performance on both kinds of operations. Experiments were performed on a network of Digital AlphaServer 4/4100s connected with Memory Channel.

Fine Grained Operations These operations are understandably slow. For example, in order to access an individual element, the client sends a message to the server specifying the matrix and location, the server locates the desired element among its worker processes, and then finally sends the result back to the client.

MITMatlab cannot compete with the local function calls that Matlab uses for these operations. For example, accessing an element in Matlab only takes 139 microseconds on average, while on a request from the server such can take 2.8 milliseconds. This result can be entirely explained by the overhead involved in communicating with the server; a simple "ping" operation where MITMatlab asks the PPServer for nothing more than an empty reply takes 2 milliseconds.

Coarse Grained Operations For coarse grained operations, the overhead of client/server communication is only a small fraction of the computation to be performed.

Table 1 shows the performance of dense matrix multiplication using Matlab and MITMatlab. Large performance gains result from the parallelism obtained by using the server; however, even in the case where the server is only using a single processor, it gains significantly over Matlab. This is due in part because the PPServer can use an optimized version of the BLAS. This illustrates one of the advantages of our model. We can use the fastest operations available on a given platform.

Using PARPACK, MITMatlab also shows superior performance in computing singular value decompositions on sparse matrices (see Table 2).

It is worth noting that Matlab's operations were performed in double precision while the PPServer's used single precision. While this clearly has an effect on performance, we do not believe that it can account for the great performance difference between the two systems.

Table 1. Matrix multiplication performance of the MITMatlab on p processors. Time are in seconds. Here "$p = 3 + 3$" means 6 processors divided between two machines.

	Matrix Size N		
	1Kx1K	2Kx2K	4Kx4K
Matlab	41.1	267.1	2814.9
MITMatlab			
with $p = 1$	5.5	45.1	357.9
$p = 2$	2.8	21.5	175.6
$p = 4$	3.9	12.9	94.7
$p = 3 + 3$	1.4	14.4	64.5

Table 2. SVD performance of MITMatlab on p processors using PARPACK. These tests found the first 5 singular triplets of a random 10K by 10K sparse matrix with approximately 1, 2, and 4 million nonzero elements. Matlab failed to complete the computation in a reasonable amount of time. Times are in seconds.

Processors	Nonzeros		
used	1M	2M	4M
2	136.8	169.2	433.5
4	88.8	91.9	241.0
3 + 3	75.2	78.8	168.6

Discussion These results make it clear what types of tasks are best performed on the server. Computations that can be described as a series of coarse grained operations on large matrices fare very well. By contrast, those that use many fine grained operations may be slower than Matlab. Such tasks should be recoded to use coarse grained operations if possible, or incorporated directly into the server via the package system. Note that on many tasks that involve computation on large matrices, fine grained operations occupy a very small amount of time and so the advantages that we gain using the server are not lost.

5.2 Executing Programs

Figure 6 shows the Matlab function that we used for this experiment. It performs a matrix-vector multiplication and a vector addition in a loop. Table 3 shows the results when the function is executed: 1) in Matlab, 2) in Matlab with server operations, 3) directly on the server through a package, and 4) in Fortran. Experiments were performed using a Sun E5000 with 8 processors. The Fortran code used Sun's optimized version of LAPACK.

The native Fortran version is the fastest; however, the PPServer package version did not incur a substantial performance penalty. The interpreted MITMatlab version, while still faster than the pure Matlab version, was predictably slower than the two compiled versions. It had to manage the temporary variables that were created in the loop and incurred a little overhead for every server

function called. We believe that this small cost is well worth the advantages we obtain in ease of implementation (a simple Matlab script) and interactivity.

```
A=rand(3000,3000);
x0=rand(3000,1);
Q=rand(3000,9);
n=10;

function X=testfun(A,x0,Q,n)

X(:,1)=x0;
for i=1:n-1
  X(:,i+1)=A*X(:,i)+Q(:,i);
end
```

Fig. 6. Matlab code for the program test. The Matlab version that used server operations included some garbage collection primitives in the loop.

Table 3. The performance of the various implementations of the program test. Although Matlab takes some advantage of multiple processors in the SMP we list it in the $p = 1$ row.

Processors Used	Time (sec)			
	Fortran	Server Package	Matlab with Server	Matlab
1	3.07			49.93
2	1.61	1.92	2.43	
4	0.90	1.02	1.49	
6	0.62	0.78	1.26	
8	0.55	0.67	1.84	

6 Conclusions

Applying fast scientific computing algorithms to large everyday problems represents a major engineering effort. We believe that a client-server architecture provides a robust approach that makes this problem much more manageable.

We have shown that we can create tools that allow easy interactive access to large matrices. With MITMatlab, researchers can use Matlab as more than a prototyping engine restricted to toy problems. It is now possible to implement full-blown algorithms intended to work on very large problems without sacrificing interactive power. MITMatlab has been used successfully in a graduate course

in parallel scientific computing. Students have implemented algorithms from areas including genetic algorithms and computer graphics. Packages encapsulating various machine learning techniques, including gradient-based search methods, have been incorporated as well.

Work on the PPServer continues. Naturally, we intend to incorporate more standard libraries as packages. We also intend to implement out-of-core algorithms for extremely large problems, as well implement interfaces to other clients, such as Java-enabled browsers. Finally, we wish to use the PPServer as real tool for understanding the role of interactivity in supercomputing.

Acknowledgments Parry Husbands is supported by a fellowship from Sun Microsystems. Charles Isbell is supported by a fellowship from AT&T Labs/Research. Most of this research was performed on clusters of SMPs provided by Sun Microsystems and Digital Corp.

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Criz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. Siam Publications, Philadelphia, 1995.
2. P. Arbenz, W. Gander, and M. Oettli. The Remote Computation System. Technical Report 245, ETH Zurich, 1996.
3. Falcon Group at the University of Illinois at Urbana-Champaign. The Falcon Project. http://www.csrd.uiuc.edu/falcon/falcon.html.
4. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*. Birkhauser Press, 1997.
5. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhilon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. Scalapack Users' Guide. http://www.netlib.org/scalapack/slug/scalapack_slug.html, May 1997.
6. Henri Casanova and Jack Dongarra. Netsolve: A Network Server for Solving Computational Science Problems. In *Proceedings of SuperComputing 1996*, 1996.
7. Peter Drakenberg, Peter Jacobson, and Bo Kagstrom. A CONLAB Compiler for a Distributed Memory Multicomputer. In *Proceedings of the Sixth SIAM Conference on Parallel Processing from Scientific Computing*, volume 2, pages 814–821. Society for Industrial and Applied Mathematics, 1993.
8. William Gropp, Ewing Lusk, and Anthong Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.
9. PETSc Group. PETSc - the Portable, Extensible Toolkit for Scientific Computation. http://www.mcs.anl.gov/home/gropp/petsc.html.
10. J. Hollingsworth, K. Liu, and P. Pauca. *Parallel Toolbox for MATLAB PT v. 1.00: Manual and Reference Pages*. Wake Forest University, 1996.
11. K. J. Maschhoff and D. C. Sorensen. A Portable Implementation of ARPACK for Distributed Memory Parallel Computers. In *Preliminary Proceedings of the Copper Mountain Conference on Iterative Methods*, 1996.

12. L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. Falcon: An Environment for the Development of Scientific Libraries and Applications. In *Proceedings of KBUP'95 - First International Workshop on Knowledge-Based Systems for the (re)Use of Program Libraries*, November 1995.
13. Anne E. Trefethen, Vijay S. Menon, Chi-Chao Chang, Gregorz J. Czajkowski, Chris Myers, and Lloyd N. Trefethen. MultiMATLAB: MATLAB on Multiple Processors. http://www.cs.cornell.edu/Info/People/lnt/multimatlab.html, 1996.

A Thread-level Distributed Debugger

João Lourenço José C. Cunha

{jml,jcc}@di.fct.unl.pt
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Portugal

Abstract. *In order to address the diversity of existing parallel programming models, it is important to provide development environments that can be incrementally extended with new services. Concerning the debugging of process-based models, we have previously designed and implemented a basic interface that can be accessed by other tools as well as by debugging modules associated with high-level programming languages.*
In this paper we describe our work towards the support of further debugging functionalities for parallel and distributed programs, by discussing a model to support thread-based debugging services. We then show how those services are supported on top of a distributed monitoring and control software architecture.

1 Introduction

In order to ease the task of parallel and distributed application development, a debugging service must support the following aspects:

1. Inspection and control of the computation state;
2. Tool interfacing;
3. Heterogeneity.

There are several difficulties regarding the development of debugging services. On one hand, due to the large diversity of programming and computational models, it is not possible to define a universal debugging interface that can meet the requirements of all such models. On the other hand, there is an increasing number of applications which are composed of multiple separate components, each based on its own computational model, be it sequential or parallel.

So aspect (1) depends on each specific computational model, e.g. process-based, object-based, multi-threaded, as well as the underlying programming paradigm, e.g. imperative or declarative. At a basic level, as far as parallel and distributed debugging is concerned, the following entities should be modeled: processes, threads, and their interactions. Efforts such as the one from the HPDF initiative [BFP97] are currently trying to establish a standard interface for the most common basic debugging functionalities, that can hopefully improve the current situation.

Aspects (1) and (2) were addressed in our previous work [CL97,KCD+97,LCK+97], when we have developed a distributed process-level debugger (**DDBG**) for *C/PVM* programs. The **DDBG** debugger was integrated in a parallel software engineering environment within the scope of an European project [S+94].

In both of the above situations, a debugging service must be able to handle the requirements of very distinct models, and this can be achieved through heterogeneous debuggers (aspect (3) above).

We have recently implemented a process-level debugging interface on top of a very flexible monitoring and control software architecture (**DAMS**) [CLV+98]. One important aspect of this architecture is that it can be easily extended with new services and functionalities, such as for debugging, profiling, and distributed resource management. This allows an incremental development of tools and their experimentation with rapid prototyping.

In this paper we extend such debugging functionalities with a thread-based service, and show how it is implemented on top of the mentioned architecture.

The organization of the paper is as follows. Section 2 discusses process and thread-based debugging services, and Sec. 3 discusses implementations on top of the DAMS architecture. Then we discuss related work and present some conclusions.

2 Process and Thread-oriented Debugging Services

In order to provide debugging functionalities for process- and thread-based models, we must identify the basic concepts and mechanisms supporting inspection and control of the computation state. We define a model that is intended to be neutral concerning the diversity of semantics of existing process and thread-based models.

2.1 The Components of the Model

The model defines the following basic entities:

- *Processes.* A process is a passive entity, a kind of "capsule" supporting contexts for the concurrent execution of multiple threads. A context is defined by a non-empty set of cells. A process is completely specified by four types of "contexts":
  - *Process Memory Context.* It corresponds to the process address space which is represented by a set of values of accessible memory cells. Code, data and stack regions are mapped onto such memory cells.
  - *Process Synchronization Context.* It contains cells representing synchronization variables such as locks and mutexes, as well as condition variables. Of course they are also mapped onto memory cells but we prefer to separate them for greater clarity of the model.
  - *Process Communication Context.* It is represented by the values of the input/output ports and the communication channels (such as message queues). Such communication channels and input/output ports can also be modeled by associated memory cells, but they are explicitly identified here, because they describe the process interaction with its outside environment.
  - *Process Execution Context.* This is defined by the set of threads that execute within the scope of the process. Each such thread has a precise logical specification in terms of specific contexts, as described below. Additionally, each process has an associated Scheduling Context which describes the status of the physical processor scheduling for all threads (this is not further detailed in this paper).
- *Threads.* A thread is an active entity which executes some code within the contexts defined by its enclosed process. It is specified by two types of "contexts":
  - *Thread Memory Context.* It is defined by the set of values of the memory cells containing the code, the data, and the stack regions that were specified for each thread. Of course, both the code and data regions are shared by all threads in a single process, unlike the stack regions which must be kept private.
  - *Thread Execution Context.* This is defined by the status of the Virtual Processor that is associated with each thread in order to model its logical behavior. The status of a virtual processor is defined by the set of values of the processor registers, and by a logical state, a cell containing one of the values T_Running, T_Blocked, T_Stopped, T_Terminated.

The thread logical state transition diagram presented in Fig. 1 identifies the possible state transitions allowed to a thread, identifying at the same time some of the debugging functionalities that trigger each state transition. Associated with each transition in the state diagram there is a set of labels naming the possible causes of the transition. Their name suggest the associated functionality. Labels between angle braces, such as `<T_Exit>`, define actions resulting of the thread execution and generated internally or by the system. Other labels, such as `T_step`, identify transitions forced by an external agent, such as the debugger.

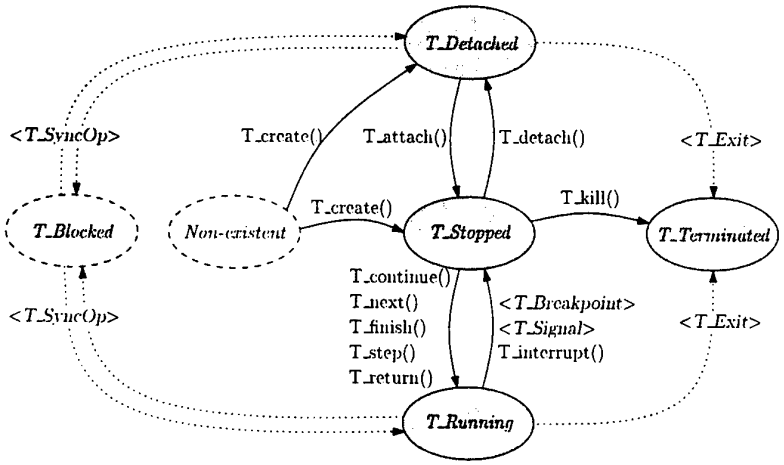


Fig. 1. The thread state transition diagram

- T_DETACHED. The thread is running free and it is not under control of the debugger.
- T_RUNNING. The thread is running under control of the debugger.
- T_STOPPED. The thread is stopped as a result of a debugger command or due to the occurrence of some exception.
- T_BLOCKED. The thread has invoked a blocking call and is temporarily blocked until that request is satisfied.
- T_TERMINATED. The thread has terminated due to a debugger or system command, or because it has reached its exit point.

2.2 Events

Using such a model, we are able to precisely identify the events which are relevant to describe and control a concurrent computation with multiple processes and threads.

In the following we briefly illustrate how this model can help in the process of precisely specifying the operational semantics of debugging primitives in terms of events.

Generally, given a specific Context (as previously defined above) an event is defined by a modification in a single value of a cell contained in that context. This corresponds to the basic notion that an event describes a transition from one state to another state.

Process-level Events These events describe all modifications made to any of the contexts defined in the process (Memory, Synchronization, Communication, and Execution). For example, events are triggered by modification of global process variables, by modifications of the state of a mutex, by the arrival of a message, or by the creation or destruction of a thread in a process.

Thread-level Events These events describe the modifications in the thread Memory and Execution contexts. For example, the modification of a local thread variable, or a physical processor register. Thread-level events are also triggered by any change in the logical state of its virtual processor.

2.3 Actions

An action is responsible for the state modification that triggers each event. We identify two classes of actions:

- *Internal Actions.* They are enforced by the virtual processor associated with a given thread in a process. The sequence of all pairs (Internal_action, Generated_Event) that are produced during thread execution, precisely specify the computation path followed by the thread. Such internal actions may correspond to physical processor instructions or to higher level instructions, for example C code statements.
- *External Actions.* They are enforced by external controller entities such as the debugger, acting upon the contexts defined within a process. The sequence of all pairs (External_action, Generated_Event) gives the history of a debugging session.

2.4 The Debugging Activity

Debugging functionalities fall into two categories: inspection commands, and controlling commands. On the other hand, they can refer to individual processes or threads. They can also refer to process interactions or thread interactions. The core of the debugging activity amounts to observe and/or enforce well-defined sequences of events so that deviations from the program specification can be localized and corrected. Our model provides a foundation to develop a mechanism that controls the detection and registering of events. Basicly event detection can be enabled for a well-defined class of action/event pairs. For example:

- Detect events in a given process/thread, associated with its Memory context, which were generated by internal actions only. It is possible to detect events associated with a given memory cell.
- Detect events in a given process, associated with its Synchronization context, and generated by internal actions of a given thread.
- Detect events in a given process/thread, associated with the logical state of its Execution context, and were generated by external actions.

In general it is possible to selectively enable/disable event detection for specific types through the specification of the following elements:

- Which class of action triggers the event (External, Internal).
- Which entity should be monitored (Process, Thread, Context, Cell).

Well-known debugging primitives can easily be represented in terms of this model. For example, concerning threads, a command "set_var()" of a local variable in a given thread would generate an event related to the Thread Memory Context. A command "set_breakpoint()" in a given thread would relate to the Thread Execution Context. Concerning processes, a command "kill_thread()" would relate to the Process Execution Context (and also to the Thread Execution Context because it also changes the thread state).

By monitoring the occurrence of events of a certain type, it is possible to construct event histories that contribute to a better understanding of the concurrent computation. For example, in order to implement a deterministic replay facility concerning process interactions only (i.e. message exchange), one needs to enable the detection of events related to the Process Communication Context. A replay facility for thread interactions internal to a single process depends on the enabling of events related to Process Memory and Process Synchronization Contexts.

2.5 Asynchronous Event Notification

Several types of debugging commands provide an immediate response, e.g. as in a "set_var()" or "set_breakpoint()", which give a success or failure indication, and possibly return some result (e.g. a breakpoint identification).

Other types of debugging commands typically act upon Thread Execution Contexts in such a way that it is not possible to obtain immediate meaningful imformation, besides knowing that the command was successfully applied. For example, commands like "continue()" or "next()" immediately originate a logical state transition in a thread (from T_STOPPED to T_RUNNING), but it might take an unpredictable amount of time for the thread to reach a point that should be inspected during debugging, e.g. to reach a breakpoint. In general, the debugger interface or the application that is invoking debugging commands should not be forced to wait until the desired event is reached. Instead, an asynchronous event notification mechanism must be provided by the debugging interface, allowing a thread to explicitly register its interest in receiving *event notifications* through the declaration of an event handler.

This declaration is achieved by calling the service

```
T_sethandler (process_thread_list, event_type, handler)
```

which defines the function `handler` as an handler of events of the given type (according to the previous subsection) which are originated from any of the processes or threads from `process_thread_list`. Multiple threads in the same or different processes can register handlers for a specific type of events. If such event occurs, a notification is sent to all the registered threads. When a thread receives an event, its current execution is suspended while the associated handler function is executed.

This event mechanism is also used to support tool synchronization and coordination in an integrated software development environment where multiple tools (for debugging, testing, visualization, etc.) concurrently observe and control the evolution of a computation. This coordination is achieved by having some tools, e.g. a graphical user-interface or a thread-based visualization tool, registering handlers related to the occurrence of some types of events, that may be originated by internal and/or external actions (e.g. setting breakpoints). On event occurrence, such tools can react and update the graphical view that is being presented to the user, consistently with the evolution of the computation and with the actions triggered by the debugging tool.

2.6 Summary on the Debugging Functions

In this section we have discussed how an event-based model can provide the foundation to develop process and thread based debugging services. In this paper we have not presented the interface of process and thread debugging primitives. Our goal is to be able to support distinct and evolving interface primitives so that our debugging framework can be used to support experimentation and building of prototypes.

3 A Process- and Thread-oriented Debugging Tool

In this section we discuss implementation issues, including the support for multiple connections from concurrent client tools, as well as the infrastructure for implementing the debugging functionalities that we have outlined in the previous section.

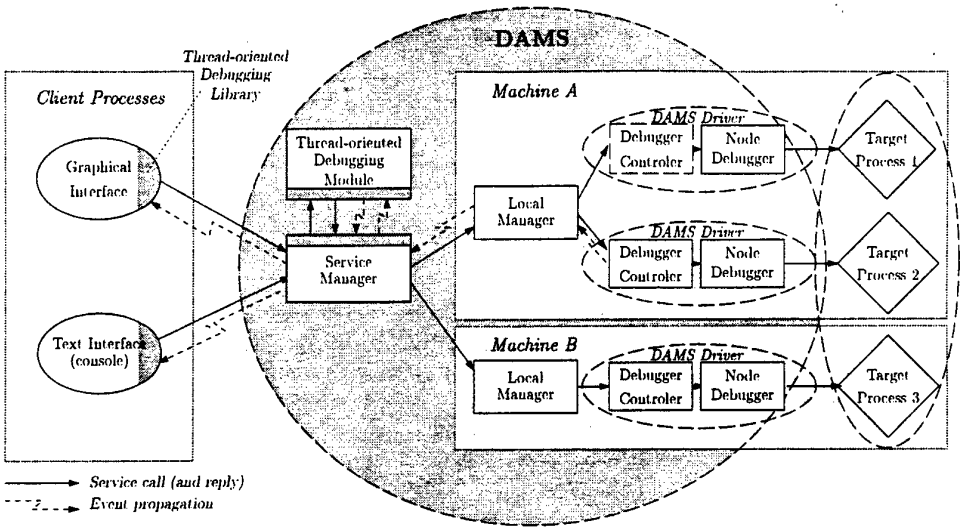


Fig. 2. The **TDBG** architecture

3.1 The DAMS system

The **DAMS** (*Distributed Application Monitoring System*) system provides the basic layer to support the incremental development of parallel and distributed monitoring and control services, such as debugging, profiling and resource management. Its design and implementation are neutral regarding the programming and computational models of the target application.

The processes related to **DAMS** can be classified in one of three classes (see Fig. 2):

- *Target application processes*. The set of processes that will be controlled/monitored by the **DAMS** system.
- *Client application processes*. The set of independent client tools, that may operate concurrently over the *Target application processes* by issuing requests to the **DAMS** system, through a service interface library.
- *The **DAMS** processes*. The set of internal processes that implement the **DAMS** system and its services. This set includes:
  - *System processes*. This includes a single Service Manager and several Local Managers, one per physical node of the target architecture. These processes manage the internal **DAMS** resources and provide an architecture independent communication layer that allows the *Client application processes* to control and inspect the evolution of the *Target application processes*.
  - *Service processes*. Each class of service (*e.g.* debugging, resource management, profiling) requires a **DAMS** configuration which includes a set of specific components: one Service Module, to handle the *Client application* service requests and their high-level system-independent parts; and a set of *Driver processes*, usually one per process of the *Target Application*, to implement the low-level system-dependent control and monitoring aspects.

The most important aspects of **DAMS** are: its extensibility; its neutrality concerning the target application models; its builtin support for multiple concurrent connections from client tools; and its functionalities for tool coordination and synchronization using events.

3.2 The TDBG tool

In [CLV+98] we have described the implementation of **PDBG**, a process-level debugger as a **DAMS** service. Here we describe how thread-level debugging (the **TDBG** debugger) is implemented as a service on top of the **DAMS** system by the provision of an adequate set of *Service processes*.

For a better understanding of how the **TDBG** components interact, we present an example, which also refers to Fig. 2. There are three *Target application processes*; two *Client applications*: the Graphical Interface and the Text Interface; and, for simplicity, the pictured **DAMS** configuration is providing the **TDBG** service only.

Let us consider that a client application (*e.g.* a Text Interface) issues a debugging command by calling a debugging library function, that sets a breakpoint in a given line of a given thread *e.g.* `set_break(t 12345, l 98)`. This function establishes the communication with the Service Manager, which identifies the type of requested service (related to debugging), and forwards it to the appropriate component: the Debugging Module.

The Debugging Module parses the received data, identifies the type of request, and then sends the (possibly) transformed request to the Debugging Driver. The **DAMS** system internally manages the routing tables to assure that the request reaches the desired Debugging Driver which is associated with the identified target process.

In order to allow easy plug-in of existing commercial or public-domain Node Debuggers, the Debugging Driver includes a front-end process, called a Controller, which is responsible for all interactions with the actual Debugger. The Controller acts as a kind of "user", as far as the Debugger is concerned[1].

After parsing the data that was sent by the Debugging Module, the Controller identifies the target process, and issues an adequate sequence of commands conforming to the existing Debugger interface *e.g.* `select_thread 12345`, `break_line 98`. The Controller waits for the completion of each command before issuing the next one. When the sequence is terminated, the results of the command, *e.g.* `local_brkpt_id=2`, are sent back to the Debugging Module.

The Debugging Module parses the received data, and does the necessary post-processing, for example converting a local breakpoint identifier into a global breakpoint identifier, *e.g.* `global_brkpt_id=14`. Afterwards, it sends the results back to the *Client process* in the form of return values of the invoked library call.

3.3 Summary on DAMS and TDBG

By describing how the interfacing between the client tools and the **TDBG** debugger is done, we have illustrated the great flexiblility of the **DAMS** architecture in order to support extended functionalities. Namely, it is possible to integrate multiple heterogeneous target debuggers, for processes and threads, in a single **DAMS** configuration.

4 Related Work

There are many current efforts on the field of parallel and distributed debugging (with and without thread's support) and related topics [LWSB97,Zho94,MB94,Lum95,XWZS96,PHK91,DJ88,HS88]. Because we cannot cover them all here, we have chosen two related approaches that are briefly presented and compared with our own approach. The first one concerns the specification of debugging functionalities and the second concerns a distributed design supported by an existing tool.

4.1 The HPDF (*proposed*) Standard

The High-Performance Debugging Forum (**HPDF**) [BFP97] is a collaborative effort between researchers and industry, aiming to define a standard for parallel debuggers. As of Version 1 of the standard, a command based (non-graphical) interface has been prepared, specifying either syntax and semantics of the proposed services. The definition of graphical interfacing and complex I/O operations are still under work.

According to **HPDF**, a parallel debugger is either a *thread-oriented debugger*, a *process-oriented debugger* or a *hybrid debugger*, and sets of required and recommended services have been defined for each of them. Our design can easily accommodate most of the **HPDF** proposed functionalities for hybrid debuggers.

In this regard, the tool integration features of **TDBG**, presenting an unified event-based model for the *internal* and *external* actions, is a distinct contribution to the integration of parallel debuggers in more complete and complex program development environments [KCD+97,LCK+97].

4.2 The p2d2 Distributed Debugger

The **p2d2** distributed debugger [Hoo96] is a *process-oriented debugger*. It uses a client-server approach, with a well defined interface, promoting portability by isolating the system dependent code into a debugger server. There is an user-interface capable of displaying and controlling many processes, individually or associated in groups. The GNU gdb is used as a Node Debugger, and a call-back method supports asynchronous interactions between gdb and the user-interface.

The distinctive feature of our approach (*i.e.* **TDBG+DAMS**) is to support multiple concurrent client tools and to offer the necessary mechanisms to implement client tool coordination.

[1] From an implementation point of view, the existing Node Debugger must provide an interface library to be accessed by the Controller front-end.

5 Conclusions and Ongoing Work

In this paper we have discussed a model to support the development of process and thread debugging functionalities, and their implementation as services of the **DAMS** distributed architecture. This work is part of our experimentation towards the incremental building of tool support services for parallel and distributed processing.

There is a prototype of **DAMS** running on our Ethernet LAN with Linux/PC's nodes, and a cluster of FDDI-interconnected Alpha processors under OSF/1. A process-level debugger (**PDBG**) runs as a **DAMS** service, and uses the GNU gdb as the target debugger. The efficieny of this prototype suffers because gdb is very *heavy*.

This prototype is being extended to implement **TDBG** which provides a thread-based debugging service according to the description in Sec. 3.2. A different Node Debugger is used, namely **SmartGDB** [Hal92], which is a *thread-oriented debugger*, extending GNU gdb with **TCL** scripting capabilities and debugging support for user-level threads.

An ongoing related project focus on the integration of **TDBG** and a visualization tool for thread-based programs. In this project we are experimenting with the **TDBG** tool integration and coordination support mechanisms.

Acknowledgements

Thanks are due to João Vieira, Bruno Moscão e Daniel Pereira for their work in the **DAMS** system.

The work reported in this paper was partially supported by the Portuguese CIÊNCIA Programme, by the PRAXIS XXI SETNA-ParComp (Contract 2/2.1/TIT/1557/95), and by the DEC EERP PADIPRO (Contract P-005).

References

[BFP97] J. Brown, J. Francioni, and C. Pancake. White paper on formation of the high performance debugging forum. Available in "http://www.ptools.org/hpdf/meetings/mar97/whitepaper.html", February 1997.

[CL97] J. Cunha and J. Lourenço. An Experiment in Tool Integration: the DDBG Parallel and Distributed Debugger. *Journal of Systems Architecture, 2nd Special Issue on Tools and Environments for Parallel Processing, Elsevier Science*, 1997.

[CLV+98] J. C. Cunha, J. Lourenço, J. Vieira, B. Moscão, and D. Pereira. A framework to support parallel and distributed debugging. In *Proceedings of the International Conference on High-Performance Computing and Networking (HPCN'98), Springer, LNCS vol. 1401*, pages 708–717, Amsterdam, The Netherlands, April 1998. Springer.

[DJ88] Thomas W. Doeppner, Jr. and David D. Johnson. A multi-thread debugger. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 295–297, Madison WI, May 1988. [Extended abstract].

[Hal92] Sudhir Halbhavi. Thread debugger—implementation and integration with the SmartGDB debugging paradigm. Master's thesis, University of Mysore, India, 1992.

[Hoo96] Robert Hood. The p2d2 project: Building a portable distributed debugger. In *Proceedings of the 2nd Symposium on Parallel and Distributed Tools (SPDT'96)*, Philadelphia PA, USA, 1996. ACM Press.

[HS88] Gil Hansen and Andy Sheppard. Debugging multithreaded programs. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 310–312, Madison WI, May 1988. [Extended abstract].

[KCD+97] P. Kacsuk, J. C. Cunha, G. Dózsa, J. Lourenço, T. Fadgyas, and T. Antão. A graphical development and debugging environment for parallel programs. *Parallel Computing, Elsevier Science*, 22(1997):1747–1770, 1997.

[LCK+97] J. Lourenço, J. C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewsk. An integrated testing and debugging environment for parallel and distributed programs. In *Proceedings of the 23rd Euromicro Conference (EUROMICRO'97)*, pages 291–298, Budapeste, Hungary, September 1997. IEEE Computer Society Press.

[Lum95] Steve S. Lumetta. Mantis: A debugger for the split-C language. Technical Report CSD-95-865, University of California, Berkeley, March 1995.

[LWSB97] T. Ludwig, R. Wismuller, V. Sunderam, and A. Bode. OMIS — On-Line Monitoring Interface Specification (Version 2.0). Technical report, Lehrstuhl fur Informatik, Technical University of Munich (LRR-TUM), Munich, Germany, July 1997.

[MB94] John May and Francine Berman. Designing a parallel debugger for portability. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing*, pages 909–915, Los Alamitos, CA, USA, April 1994. IEEE Computer Society Press.

[PHK91] M. Krish Ponamgi, Wenwey Hseush, and Gail E. Kaiser. Debugging multithreaded programs with MPD. *IEEE Software*, 8(3):37–43, May 1991.

[S+94] S.Winter et al. Software Engineering for Parallel Processing, copernicus programme. Progress report 1, University of Westminster, London, UK, October 1994.

[XWZS96] Jianxin Xiong, Dingxing Wang, Weimin Zheng, and Meiming Shen. BUSTER: an integrated debugger for PVM. In IEEE, editor, *Proceedings of 1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing, ICA PP '96*, pages 11–13, Singapore, June 1996. IEEE Computer Society Press.

[Zho94] W. Zhou. A layered distributed program debugger. In *Symposium on Parallel and Distributed Systems (SPDP '93)*, pages 665–668, Los Alamitos, Ca., USA, December 1994. IEEE Computer Society Press.

New Access Order to Reduce Inter-Vector-Conflicts

A. M. del Corral & J. M. Llaberia
Department d'Arquitectura de Computadors.
Universitat Politecnica de Catalunya. Barcelona (Spain)
e-mail: anna@ac.upc.es

Abstract. In vector processors, when several vector streams concurrently access the memory system, references of different vectors can interfere in the access to the memory modules, causing module conflicts. Besides, in a memory system where several modules are mapped in every bus, delays due to bus conflicts are added to module conflict delays. This paper proposes an access order to the vector elements that avoids conflicts when the concurrent access corresponds to vectors of a subfamily, and the request rate to the memory modules is less than or equal to the service rate. For other cases of concurrent access, the proposal dramatically reduces conflicts.

1 Introduction

In vector processors, the ideal execution of a memory vector instruction would permit to obtain a datum at every cycle after an initial latency. As, in general the memory module reservation time is much longer than the processor cycle time, the memory system usually consists of multiple memory modules with independent access paths.

Usually, vector processors have more than one port to the memory subsystem to allow several memory vector instructions to proceed concurrently. Under these conditions, conflicts appear in the access to the memory modules when two or more references are simultaneously issued to the same module. Besides, a reference to a busy module also causes a memory module conflict.

In vector processors with several paths to the memory system, or in multi-vector processors, another factor that affects the performance of the memory system is the interconnection network between processors and memory modules. In the design of some memory systems, the decision of reducing the number of independent access paths to the memory modules (several modules are mapped on every bus) [2][6], implies a reduction in its economic cost. However, this solution implies assuming the presence of conflicts in the access to the interconnection network, as well as the memory module conflicts mentioned above. Both type of conflicts appear even in the specially common case of several one-strided vector streams concurrent access. The main effect of the conflicts is the starvation of the functional units, with the subsequent loss of performance.

Memory vector instructions with regular access patterns generate periodical conflicts as these kind of instructions generate periodical streams of references (vector streams with a constant stride). In the context of this paper, our interest is the reduction, and the elimination when possible, of the memory conflicts (interconnection and memory module conflicts) caused by concurrent constant-strided vector streams.

Several kind of methods have been proposed to reduce the number of cycles lost due to memory conflicts. Some authors propose to accurately place in memory the vectors to

be concurrently accessed [10][14][17]. This technique implies that patterns must be known in compilation time, and, the access to a vector stream in different context of a program could decrease its effectiveness. Other authors propose the use of buffers in the memory modules [17] or in the interconnection network [19]. Buffers allow the requesting processor to keep sending requests without waiting, but this technique requires labelling the memory references to allow their reordering before being used by the processor; the cost of the interconnection network increases as the tag must be sent along with the request [17]. In addition, buffers do not directly solve the problem of the convergence to a single port of the requests in the return network [21].

Our proposal consists of a new access order to the vector stream elements. In parallel with our work, other authors have studied this kind of solution [15]. This new order working with a new arbitration algorithm will help concurrent vector streams perform their memory request with no conflicts or less number of conflicts than the *classical access* implies.

One of the cases for which our proposal completely avoids conflicts is the very common case of the concurrent access of several one-strided vector streams. J. Fu and J.H. Patel in [7] show that between 7% and 54% of the vector streams in four programs of the Perfect Club benchmark set [1] (ADM, ARC2D, BDNA and DYESM) access the memory with stride 1.

Section 2 outlines the architecture model, on which the present study is based, and the characterization of the interleaving mapping and vector access functions. The interaction between vector streams in a complex memory system is studied in Section 3. Section 4 presents the proposed access order to the memory modules and presents its hardware support. Finally, Section 5 deals with the comparison between the proposal and the method used in a classical system, like CRAY X-MP.

2 Architecture

The memory architecture presented in Fig. 1 is an example of the complex memory system, similar to the one used in the CRAY X-MP [2].

The memory subsystem consists of $M = 2^m$ memory modules (memory cycle, $n_c = 2^c$ clock cycles), connected to $P = \lfloor M/n_c \rfloor$ memory ports through an interconnection network. To reduce the number of access paths to the memory subsystem the memory modules are distributed into SC sections. A memory module request occupies the section path where the module is located during one cycle. It is supposed that $SC = 2^{sc}$, and the number of memory modules is a multiple of SC.

In each cycle, every port requests an element of a vector stream except when a conflict appears in the interconnection network or in a memory module. In case of conflict, only one vector stream obtains the access and the other requests must wait; a priority rule must determine which port will be able to proceed. In the present paper, we use the arbitration implemented in the CRAY X-MP [2], to measure the performance of the *classical access* (Definition 5) and in the examples of concurrent access when another algorithm is not specified. This arbitration gives priority to the vector stream with the lower 2^s stride factor; for ports with same parity of strides, the priority is fixed.

The memory is organized as an interleaved address mapping model (*section* = A_i *mod SC, memory module* = A_i *mod M, offset* = $\lfloor A_i/M \rfloor$). The interleaving function which maps the address into memory modules has a period of $P=M$.

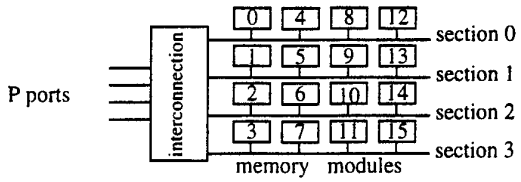


Fig. 1. Complex Memory System.

The following definitions will help the reader to follow the method.

Definition 1: A *vector stream* $A = (A_0, S, VL)$ is the set of references to memory modules $\{A_i | A_i = A_0 + i \times S, 0 \le i < VL\}$, where A_0 is the address of the first reference, S (*stride*) is the distance between two consecutive references and VL is the vector length, or number of references. If the length is not relevant a stream is specified as $A = (A_0, S)$.

Vector streams can be classified into different families according to their stride.

Definition 2: A *stride family* (F_s) is the set of vector streams with strides $S = \sigma \times 2^s$, where σ is an odd factor [9].

A vector stream with a stride $S = \sigma \times 2^s$ references $P_s = M / gcd(M, 2^s)$ memory modules periodically, and the period is P_s.

Definition 3: The memory module set (*MMS*) of the vector stream $A = (A_0, S)$ is the set of all the memory modules accessed by the vector stream $A=(A_0,S,P_s)$. $MMS = \{m_i | m_i=(A_0+i \times S) mod M, 0 \le i < P_s\}$.

Definition 4: A *stride subfamily* ($SF_s^{m_0}$) is the set of vector streams of a family that reference the same set of memory modules.

To give some examples, the family F_0 (odd-strided vector streams) only has one subfamily SF_0^0, and the family F_1 (even-strided vector streams) has two subfamilies, SF_1^0 references the even memory modules, and SF_1^1 references the odd modules.

Definition 5: *Classical access* is the access order that uses the recurrence $A_{i+1} = A_i + S$ (S=*Stride*) to compute vector stream addresses.

Since the vector length is usually greater than the vector register length, the compiler is required to transform the code using strip-mining. Under this condition, a great proportion of memory accesses from vector streams are issued by vector instructions *load* and *store*, which are of a fixed length equal to the vector register length. Let us assume that, in order to simplify the explanation of the proposed method, the vector stream length ($VL = 2^{vl}$) is a multiple of the vector register length $MVL = 2^{mvl}$ which is assumed to be a multiple of the number of memory modules $M = 2^m$.

3 Characterization of the Conflicts

Only in the case that the memory request rate imposed by concurrent vector streams is equal to or less than the memory module response rate, the concurrent access can be conflict-free. When the request rate is equal to the response rate, it is said that the memory system (or similarly, the memory modules) works tight, and when the request rate is less than the response rate, the memory system works loose.

To obtain a conflict-free access, not only the system must work loose or tight, in addition, the concurrent access of the vector streams must fulfil two conditions:

- consecutive references to a memory module must be distanced at least n_c cycles (to avoid memory module conflicts).

- since memory modules share sections, only a few sets of concurrent memory module references are correct (to avoid section conflicts).

To analyse the effect of the first condition, we first study a memory system that can only present conflicts in the access to the modules, not in the interconnection network. Then, we extend the study to a complex memory system to discuss the second condition.

Simple Memory System

A simple memory system has an independent access path from every port to every memory module, thereby its interconnection network does not present conflicts. In a system like that, the concurrent *classical access* of vector streams that have the same stride has a conflict-free steady state when the request rate they imply is less than or equal to memory modules response rate (the system works loose or tight) [16][17].

Fig. 2 presents the concurrent *classical access* of four one-strided vector streams in a memory system with 16 memory modules and an n_c of 4 cycles. Vector streams start their concurrent access in different memory modules. In the figure, it is possible to observe for every cycle the memory module that begins to be occupied by every vector stream (the module remains occupied during latency cycles). A delay due to a memory module conflict is depicted in black.

Cycles		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Modules	A	0				1	2	3		4	5	6	7	8	9	10	11	12
	B	1	2	3		4	5	6	7	8	9	10	11	12	13	14	15	0
	C	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4
	D	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8

Fig. 2. 16-way interleaved memory with $n_c = 4$. Conflicts with the *classical access*.

This concurrent access presents conflicts at the very beginning, but the steady state, that starts at cycle 8, is conflict-free. At the steady state, four sets of concurrent memory module references ({0, 4, 8, 12}, {1, 5, 9, 13}, {2, 6, 10, 14} and {3, 7, 11, 15}) are periodically repeated every n_c cycles, thereby, consecutive references to the same memory module are distanced n_c cycles. The periodicity of these four sets (called *CMR -Concurrent memory Module References-* from now on) can be guaranteed because vector streams reference the memory modules with the same order.

R. Raghavan and J.P. Hayes stated with theorem 6 of [17] the conditions the concurrent vector streams must fulfil to obtain a conflict-free *classical access* in a simple memory system. These conditions can be fulfilled only by vector streams that belong to the same subfamily. All the combinations of vector streams that have the same stride have a conflict-free access whenever the system works loose or tight. The concurrent *classical access* of vector streams of different subfamilies is always conflictive (corollary 3 of [4]).

Complex Memory System

Combinations of vector streams that obtain a conflict-free access in a simple memory system, may not have a good behaviour in a complex memory system. The sets *CMR* that are suitable in a simple memory system may not be appropriated in a system where several memory modules are mapped in the same section. As an example, none of the *CMR* of the concurrent access of Fig. 2 are appropriated in a complex memory system

where the 16 memory modules are interleavedly mapped in 4 sections (Fig. 1): all the memory modules of every *CMR* are mapped in the same section, then, they can not be concurrently accessed.

Fig. 3 shows the conflictive *classical access* of four one-strided vector streams in the system of Fig. 1. The delay due to a section conflict is represented in light grey, and a memory module conflict is depicted in black; a section is locked during one cycle in the access to a memory module. In this concurrent access, conflicts are linked and periodically repeated: a section conflict causes a memory module conflict which also causes a section conflict, and so on.

	Cycles	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Sections	A	0	1	2	3	0	0	1	2	3	0	0	1	2	3	0	0	1
	B		0	1	2	3		0	1	2	3		0	1	2	3		0
	C			0	1	2	3		0	1	2	3		0	1	2	3	
	D				0	1	2	3		0	1	2	3		0	1	2	3
Modules	A	0	1	2	3		4	5	6	7		8	9	10	11		12	13
	B		4	5	6	7		8	9	10	11		12	13	14	15		0
	C			8	9	10	11		12	13	14	15		0	1	2	3	
	D				12	13	14	15		0	1	2	3		4	5	6	7

Fig. 3. 16-way interleaved memory system with n_c = 4 and *SC*=4. Conflicts with the *classical access*.

T. Cheung and J.E. Smith characterize in [2] the linked conflicts that appear in the concurrent *classical access* of two one-strided vector streams and use the term complex linked conflict (complex conflict) when three or more vector streams interfere with each other in a less precise way. Authors prove that the steady-state linked conflicts and complex conflicts reduce the effective bandwidth.

Authors of [2] show that in the concurrent *classical access* of three one-strided vector streams (the system works loose), in 34% of the cases (combinations of initial memory modules) linked conflicts appear, in 7% of the cases complex conflicts are generated, and performance can be degraded by 20%.

To solve these conflicts, W. Oed and O. Lange conclude in [16] that n_c and *SC* must be coprime (theorem 9). A solution with a prime *SC* is proposed in [15]. In [2], authors give some alternatives to avoid linked conflicts, i.e. a solution with odd values of n_c. For all the proposals, if vector streams have different strides conflicts persists and, in any case, complex conflicts do not disappear.

Tab. 1 shows the asymptotic number of operations per cycle[1] (R_∞) the *classical access* obtains in average for four types of combinations of vector streams, in a simple memory system (*M*=16 and n_c=4) and in the corresponding complex system (*M*=16, n_c=4 and *SC*=4). The concurrent accesses simulated are all the combinations of four, three and two odd strided vector streams, two odd strided with one even strided vector streams, and two even strided vector streams. For the simple memory system, the average R_∞ for the *classical access* is far away from the ideal, even for combinations for

1. $R_\infty = ops \times r_\infty \times t_c$, where t_c is the processor cycle time, *ops* is the number of concurrent vector streams, and r_∞ is the asymptotic performance [12].

which the system works very loose and vector streams belong to the same subfamily. Comparing the results for both memory systems, it can be easily concluded that in a complex memory system, the results are worst because of interferences in the interconnection network.

Tab. 1. R_∞ for the *classical access* and Ideal.

Combinations of Strides		*Complex Mem. Syst.* $M=16\ n_c=4\ SC=4$		*Simple Mem. Syst.* $M=16\ n_c=4$	
Odd	*Even*	R_∞ *Classical*	R_∞ *Ideal*	R_∞ *Classical*	R_∞ *Ideal*
4	*0*	*1.57*	*4*	*1.86*	*4*
3	*0*	*1.51*	*3*	*1.66*	*3*
2	*0*	*1.35*	*2*	*1.38*	*2*
2	*1*	*1.39*	*3*	*1.60*	*3*
0	*2*	*1.05*	*2*	*1.27*	*2*

The next section presents an access method that completely avoids conflicts in the concurrent access of vector streams of the same subfamily when the system works loose or tight. This method also dramatically reduces conflicts for other cases of concurrent access. The name of the proposal is *Skewed Sequence of memory Modules* (*SSM*).

4 Proposal *SSM*

To reduce the number of memory module conflicts, we propose that concurrent vector streams reference the memory modules with the same order. All the vector streams of a subfamily reference the same set of P_s memory modules ($P_s = M/\gcd(M,2^s)$), but with the *classical access*, the order every vector uses to access them depends on the σ-factor of the stride. We propose to construct a σ-independent access order, then all the vector streams of a subfamily will reference the P_s modules with the same order.

To avoid section conflicts, this σ-independent access order must be constructed considering that the resulting *CMR* sets must comprise memory modules mapped in different sections.

This new sequence of memory modules will be called *SSM* (*Skewed Sequence of memory Modules*) Fig. 4 shows the *SSM* proposed for different subfamilies in a memory system that has M= 16, n_c=4 and SC=4 (Fig. 1). For every sequence *SSM* it is also shown the sequence of sections referenced and the corresponding *CMR*.

Subfamily SF^0_0 (odd strides, odd modules) - *CMR = {{0,7,10,13}, {1,4,11,14}, {2,5,8,15}, {3,6,9,12}}*

SSM	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***10***	***11***	***8***	***9***	***13***	***14***	***15***	***12***
sections	0	1	2	3	0	1	2	3	2	3	0	1	1	2	3	0
	subperiod 0				***subperiod 1***				***subperiod 2***				***subperiod 3***			

Subfamily SF^0_1 (even strides, even modules)
CMR = {{0,10}, {2,8}, {4,14}, {6,12}}

SSM	***0***	***2***	***4***	***6***	***10***	***8***	***14***	***12***
sections	0	2	0	2	2	0	2	0
	subperiod 0				***subperiod 1***			

Subfamily SF^0_1 (even strides, odd modules)
CMR = {{1,11}, {5,15}, {7,13}, {3,9}}

SSM	***1***	***3***	***7***	***5***	***11***	***9***	***13***	***15***
sections	1	3	3	1	3	1	1	
	subperiod 0				***subperiod 1***			

Fig. 4. 16-way interleaved memory with n_c = 4 and SC=4. *SSM* for several subfamilies.

Each one of the *SSM* we propose has n_c *CMR* sets of P_s/n_c memory modules. In consequence, P_s/n_c concurrent vector streams of a subfamily can concurrently reference memory modules of different sections, avoiding section conflicts. Besides, module conflicts are also avoided as consecutive references to a *CMR* are distanced n_c cycles.

Fig. 5 shows the conflict-free access of four odd-strided vector streams in the system of Fig. 1, when the corresponding *SSM* is used. This *SSM* has n_c=4 *CMR* sets with P_s/n_c =16/4 modules, so four odd-strided vector streams could have a conflict-free access.

	Cycles	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Sections	***A***	1	2	3	0	0	1	2	3	3	0	1	2	2	3	0	1	1
	B	2	3	0	1	1	2	3	0	0	1	2	3	3	0	1	2	2
	C	3	0	1	2	2	3	0	1	1	2	3	0	0	1	2	3	3
	D	0	1	2	3	3	0	1	2	2	3	0	1	1	2	3	0	0
Modules	***A***	**13**	**14**	**15**	**12**	**0**	**1**	**2**	**3**	**7**	**4**	**5**	**6**	**10**	**11**	**8**	**9**	**13**
	B	**10**	**11**	**8**	**9**	**13**	**14**	**15**	**12**	**0**	**1**	**2**	**3**	**7**	**4**	**5**	**6**	**10**
	C	**7**	**4**	**5**	**6**	**10**	**11**	**8**	**9**	**13**	**14**	**15**	**12**	**0**	**1**	**2**	**3**	**7**
	D	**0**	**1**	**2**	**3**	**7**	**4**	**5**	**6**	**10**	**11**	**8**	**9**	**13**	**14**	**15**	**12**	**0**

Fig. 5. 16-way interleaved memory system with n_c = 4 and *SC*=4. Conflict-free concurrent access of four odd-strided vector streams using *SSM*.

Vector streams of Fig. 5 start their concurrent access in correct memory modules (same *CMR*), so the concurrent access synchronizes from the beginning. When the start addresses do not correspond to a *CMR*, an arbitration algorithm is necessary. Section 4.2 presents a dynamic arbitration that forces vector streams to concurrently access memory modules of the appropriate *CMR* [3].

4.1 *Skewed Sequence of memory Modules - SSM*

The new sequence of memory modules is called "*Skewed*" as the *SSM* we define for every subfamily is the result of applying a skew function to the subfamily *MMS* lexicographically ordered.

Definition 6: For a vector stream $A = (A0, S, P_s)$, of the subfamily $SF_s^{M_0}$, ($M_0 = A_0 \bmod \gcd(M,2^s)$), we call *Skewed Sequence of memory Modules* (*SSM*) to the sequence determined by the expression:

$$k = f(m_i) = ((m_i + \lfloor m_i/n_c \rfloor) \bmod n_c + \lfloor m_i/n_c \rfloor \times n_c)/\gcd(M,2^s),$$

where k is the position that the memory module m_i ($0 \le m_i < M$) occupies in the sequence and m_i belongs to the vector stream *MMS*.

The function $f'(m_i)$, that gives the memory module from a position in the sequence (reverse function of $f(m_i)$), will permit to generate the *SSM* sequence. We express $f'(m_i)$ as an algorithm, but before presenting it, we will make some considerations (Fig. 6 helps to follow the explanation):

- The first module a vector references with the *SSM* is $M_0 = A_0 \bmod \gcd(M,2^s)$.
- Every set of $\lceil n_c/\gcd(M,2^s) \rceil$ consecutive memory modules of the *MMS* lexicographically ordered suffers a skew. We call *GS* to every one of these sets, and in a *SSM* there are $(M/n_c)/\lceil \gcd(M,2^s)/n_c \rceil$ *GS* sets.

- The same skew is applied to gcd$(M,2^s)$ consecutive *GS* sets, but the first skew is applied to at most gcd$(M,2^s)$ consecutive *GS*. If M_0 is not the memory module 0, only the gcd$(M,2^s)$-M_0 first *GS* sets suffer the same skew.

To give an example of the former considerations, ina system with M=16, $n_c = 4$ and *SC*=4, the *SSM* of the subfamily SF_1^I has $(M/n_c)/\lceil \gcd(M,2^s)/n_c \rceil = 4$ *GS* sets. The first skew is applied only to gcd$(M,2^s)$-M_0=1 *GS* as M_0 is the memory module 1, but the second skew is applied to gcd$(M,2^s)$=2 consecutive *GS*.

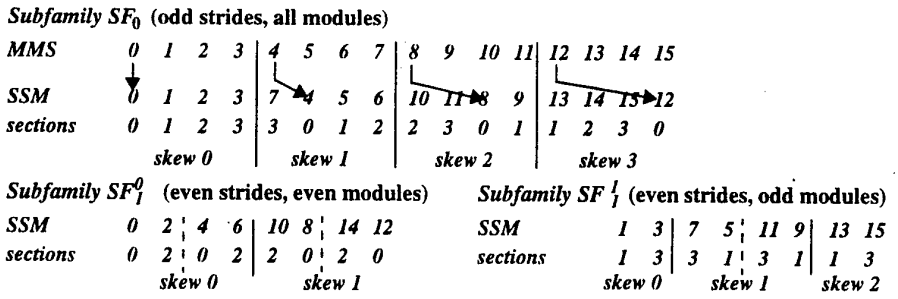


Fig. 6. 16-way interleaved memory system with $n_c = 4$ and *SC*=4. *SSM* construction for several subfamilies.

The algorithm used to generate the *SSM* sequence for any subfamily is:

$M_0 = A_0 \bmod \gcd(M,2^s)$
control = M_0
skew = 0
for NGS = M_0 / n_c *to* M/n_c -1 *step* $\lceil \gcd(M,2^s)/n_c \rceil$
  for I = $M_0 \bmod n_c$ *to* n_c-1 *step* $\gcd(M,2^s)$
    module = $((I - \text{skew} \times \gcd(M,2^s)) \bmod n_c + \text{NGS} \times n_c) \bmod M$
  endfor
  control = $(\text{control} + \lceil \gcd(M,2^s)/n_c \rceil) \bmod \gcd(M,2^s)$
  if (control = 0) then skew = skew + 1
endfor

In the algorithm, NGS controls the generation of the memory module references for every *GS* set. The variable I controls the generation of the memory module references within a *GS* set. Control controls the skew changes after the generation of gcd$(M,2^s)$ consecutive *GS*. If M_0 is not the memory module 0, only the gcd$(M,2^s)$-M_0 first sets GS suffer the same skew.

4.2 Arbitration algorithm

An arbitration algorithm is needed in order to synchronize vector streams to reach a conflict-free steady-state phase, or to dramatically reduce inter-conflicts, for any combination of initial memory modules.

The *SSM* sequences can be divided in P_s/n_c *subperiods* of n_c memory modules. In Fig. 4, we can observe that each *subperiod* of a *SSM* references the sections following a predetermined order which is different for every *subperiod*. Thus, in the concurrent access of P_s/n_c vector streams of a subfamily, we obtain a conflict-free access if we overlap different *subperiods* (different sections are simultaneously referenced as in the

example of Fig. 5 with family F_0). The main idea is that, in every cycle concurrent vector streams reference memory modules of a different *subperiod*, and these different *subperiods* must be aligned.

The arbitration algorithm controls the *subperiod* changes between vector streams; when all *subperiod* changes have been detected for all the vector streams, *subperiods* are assigned using a fixed priority. The *subperiod* change is detected by computing the expression $subperiod=\lfloor m_i/(n_t \times \gcd(M,2^s))\rfloor mod SC$ ($m_i = A_i^{SSR} mod M$) for two consecutive memory module requests of a vector stream (the current and the previous).

4.3 *Skew Sequence of memory References - SSR*

The *SSM* is the order in which memory modules must be referenced periodically, then vector stream memory references must be generated to periodically access the modules with this new order.

Definition 7: For a vector stream $A = (A0, S)$ of the subfamily SF_s^{M0} ($M_0 = A_0 \bmod \gcd(M,2^s)$), the *Skewed Sequence of References* (*SSR*) is the sequence of memory references that permits to reference the memory modules following the *SSM* periodically.

The algorithm that generates *SSR* is a modification of the algorithm that generates *SSM*. The following definition will help designing the algorithm.

Definition 8: The *order number* (*ON*) of a vector stream element, is the position on which its address is generated using the *classical access*, $0 \le ON < VL$.

The address of an element of a vector stream $A = (A_0, S)$, can be computed using its order number as $Addr = A_0 + ON \times S$. With the *classical access*, addresses of elements with consecutive *ON* are consecutively generated ($ON_{i+1} = ON_i + 1$). This is not the case with the *SSR*, but, if we know how to generate the sequence of order numbers that fulfil *SSM*, we will be able to generate *SSR*.

First, we suppose *SSR* is P_s references long, then we extend the study to any length.

P_s references long (one Period)

Vector elements placed in memory modules adjacent in the *MMS* lexicographically ordered have *order numbers* separated by a constant distance, C_s [3]. Then, we can compute the *ON* of a vector element from the *ON* of any other vector element if we know the distance between the memory modules[1] where they both are placed: $ON_j = ON_i + K \times C_s$, where K is the distance.

To compute the sequence of *order numbers* the *SSM* implies, the K we can use can be the distance between the memory module to be referenced and the first memory module referenced using the *SSM* that is $M_0 = A_0 \bmod \gcd(M,2^s)$. Then, we must use the *order number* of the first vector stream element referenced using the *SSR*, *NO0*, that can be easily computed. In this case, the order number of a vector element placed in the memory module m_j is:

$$ON_j = NO0 + ((m_j - M_0) mod\ M / gcd(M,2^s)) \times C_s.$$

Any Length (any number of Periods)

As the distance between memory modules can be computed within a period, the former recurrence actually gives the *order number* relative to a period (*ONR*). To extend the

1. distance is the number of memory modules between them in the *MMS* lexicographically ordered.

computation of the *order number* to any number of periods, we can consider that every period has a *base order number* (*BN*), to be added to the *ONR* to obtain the *ON*. From period to period this *BN* must be increased in P_s units.

The next algorithm is based in the algorithm proposed in Section 4.1, adding the computation of the *order number* and the loop that controls the period. The bold lines are the ones added.

$M_0 = A_0 \bmod \gcd(M,2^s)$
BN = 0
for* Q = 0 *to $\lceil VL/P_s \rceil$ **-1**
control = M_0
skew = 0
for NGS = M_0/n_c *to* M/n_c -1 *step* $\lceil \gcd(M,2^s)/n_c \rceil$
for I = $M_0 \bmod n_c$ *to* n_c-1 *step* $\gcd(M,2^s)$
module=((I-skew×$\gcd(M,2^s)$)mod n_c+NGS×n_c) mod M
K = ((module - M_0) mod M)/$\gcd(M,2^s)$
ONR = (ON0 + K × C_s) mod P_s
$Addr^{SSR}$ = A_0 + (BN + ONR) × S
endfor
control = (control + $\lceil \gcd(M,2^s)/n_c \rceil$) mod $\gcd(M,2^s)$
if (control = 0) *then* skew = skew + 1
endfor
BN = BN + P_s
endfor

As a synopsis, the recurrences that compute the vector memory references are:

$$A_i^{SSR} = A_0 + \text{Base\_Addr} + A_i^r \text{ and } A_i^r = (A_j^r + K \times C_s \times S) \bmod (P_s \times S)$$

where A_i^r is the vector element address relative to a period, A_i^{SSR} is its absolute address, K is the distance between memory modules where A_i^r and A_j^r are placed, and Base_Addr is the base address of a period $(BN \times S)$.

4.4 Hardware Support to Reduce Conflicts

To design the hardware that computes the *SSR*, we must rewrite the algorithm to make it easier to implement. There are two issues that must be solved: the presence of a multiplier and a modulo operation in the critical path of the address computation (every iteration).

To avoid the use of a multiplier, the relative addresses A_i^r are computed using the relative addresses A_j^r, so only two precomputed products $K \times C_s \times S \bmod (P_s \times S)$ must be used (K=1 and K=n_c). This implies using three registers to store different previous values A_j^r.

The modulo operation (mod $(P_s \times S)$) can be performed by subtracting $P_s \times S$ if necessary, as demonstrated in [5]. In fact, the two values, $A_j^r + K \times C_s \times S$ and $A_j^r + K \times C_s \times S - P_s \times S$, are computed in parallel, and the selection between them is performed by a signal that comes from the vector register index computation [5]. This signal indicates if $ON_j^r + K \times C_s \geq P_s$, easier to compute as P_s is a power of two number.

Fig. 7 shows a hardware design of the data-path. The hardware cost is moderate, two adders in the critical path and a CSA, and it is not more complex than that needed by other solutions [8][18] proposed to reduce the average memory latency time in vector processors.

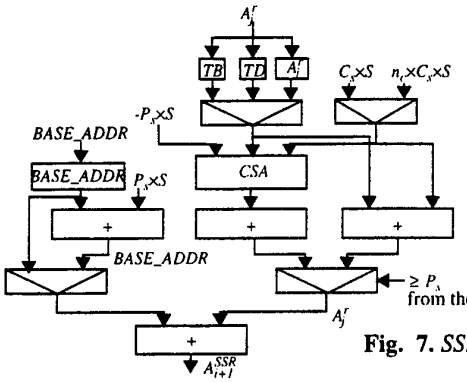


Fig. 7. *SSR* generation Hardware.

The rate at which a memory request can be issued is limited by the rate at which additions can be performed. The design can be pipelined to obtain a reduction of the cycle time (this would be also needed in the *classical access*). The additional hardware introduces a initial delay of a few cycles in the memory path. The number of clock cycles needed to access the memory is of the order of *14 + MVL* for the CRAY X-MP, *17 + MVL* for the CRAY Y-MP and *23 + MVL* for the C90 [20]. However, as the processor speed continues to increase faster than the memory speed, an extra initial delay of some cycles introduced by the hardware proposed is acceptable.

The number of parameters to be calculated is comparable to the number needed for other proposals [8][18][22], and most of them can be determined by the compiler.

The hardware needed to access the vector registers is similar to the hardware shown at Fig. 7 but simpler.

The cost of the hardware components can be considered a minor part of the cost of the memory subsystem. Additionally, in contrast with other solutions, which include a significant number of buffers to eliminate the effect of unsuitable temporal distributions [8][18], this proposal does not need buffers.

5 New method performance

In this section we present the advantages of the method proposed in this paper. Tab. 2 shows the comparison between the *SSM* and the *classical access* in a memory system with *M=16* memory modules, interleavedly mapped in *SC=4* sections, with an n_c=4.

Some considerations about the simulations:

a) We obtain the value R_∞ for the concurrent access, using the *classical access* and the proposal, of all the possible combinations of four, three or two vector streams of the families F_0 and SF_1^0.

b) All the combinations of vector streams whose concurrent access has been simulated have a non void intersection of *MMS* sets.

c) The parameter we use to perform the comparison is the increment in performance (IR_∞) implied by the proposal, and it is computed as $IR_\infty = ((R_{\infty SSM} - R_{\infty classical})/R_{\infty classical}) \times 100$[11].

d) The results presented under the name R_∞ are harmonic means of the asymptotic number of operations per cycle that the *classical access* and the *SSM* obtain for combinations of vector streams we group in types.

Tab. 2 presents the R_∞ for several types of vector stream combinations the *classical access* and the *SSM* obtain in a 16-way interleaved memory system with n_c = 4 and *SC*=4. The table also shows the maximum number of operations per cycle (IR_∞ *Ideal*) that could be ideally obtained for every combination in the supposed memory system. The increment in performance the *SSM* implies is presented in the column labelled as IR_∞.

Tab. 2. 16-way interleaved memory system with n_c = 4 and *SC*=4. R_∞ and IR_∞ for *SSM*.

	STRIDE		R_∞	R_∞	R_∞	IR_∞
	Odd	Even	Ideal	Classical	SSR	SSR
*	4	0	4	1.57	3.95	152%
*	3	0	3	1.51	2.98	97%
*	2	0	2	1.35	1.99	47%
	2	1	3	1.39	1.99	43%
	1	1	2	1.18	1.33	13%
	1	2	2.4	1.19	1.99	67%
	2	2	2.67	1.34	2.65	98%
*	0	2	2	1.05	1.99	90%
	0	3	2	1.03	1.50	46%
	0	4	2	1.04	1.99	91%

In the table, the types with an asterisk ('*') correspond to combinations of vector streams of the same subfamily that make the system work loose or tight. For these types the R_∞ the *SSM* obtains is almost R_∞ *Ideal*, and the IR_∞ is very important, between 47% and 152%. For the other types, the IR_∞ is also important, between 13% and 98%.

Fig. 8.a presents the IR_∞ the use of the *SSM* implies in function of the σ–factor of the stride, in the concurrent access of: four vector streams of the family F_0 (dark bars), four vector streams of the subfamily SF_I^0 (medium grey bars), and two vector streams of F_0 with two vector streams of SF_I^0 (light bars). For every case we grouped combinations that have four (bars labelled as "four"), three, two o zero (bars labelled as "zero") vector stream with the same σ-factor.

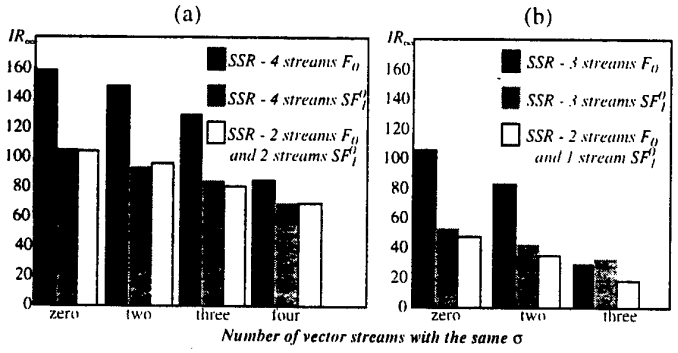


Fig. 8. 16-way interleaved memory system with n_c = 4 and *SC*=4. IR_∞ for *SSM*, in the concurrent access of four (a) or three vector streams (b).

When four F_0 vector streams (odd-strides, dark bars) access the memory system, the memory works tight, but the concurrent access with the *SSM* is conflict-free and IR_∞ is substantial, between 85% and 159%. Even when all the concurrent vector streams have the same σ-factor (same stride), *SSM* overworks the *classical access*, as this access does not avoids section conflicts.

For combinations of four SF_1^0 vector streams (even-strides, medium grey bars) the concurrent access with the *SSM* is not conflict-free as there are more than P_s/n_c (=8/4=2) concurrent vector streams, but the IR_∞ is important, between 69% and 105%.

When in the concurrent access there are two F_0 vector streams and two SF_1^0 vector streams the concurrent access with the *SSM* is not conflict-free as there are vector streams of different subfamilies but the IR_∞ is important, it ranges from 69% and 104%

Fig. 8.b presents the IR_∞ the *SSM* represents in function of the σ–factor, in the concurrent access of: three vector streams of the family F_0 (dark bars), three vector streams of the subfamily SF_1^0 (medium grey bars), and two vector streams of F_0 with one vector stream of SF_1^0 (light bars). For every case we grouped combinations that have three (bars labelled as "three"), two or zero (bars labelled as "zero") vector stream with the same σ-factor in the stride. For these cases, the IR_∞ the *SSM* obtains is lower than in the case of four vector streams, as the *classical access* finds the system working looser and, in consequence, there are less conflicts or they have less effect.

Vectors and matrices are the most common data structures in vector processors. In Fortran, the most frequent accesses to matrices are made by columns, rows and diagonals, that correspond to the strides 1, n and $n+1$ respectively, where n is the column length, which is dependent on the problem size that varies widely. Present compilation technology detects if n is even, then the matrix size can be increased in one row (odd stride), and the number of referenced memory modules is M. Thus, in row-major and column-major accesses the use of *SSM* performs equally well, and there are no conflicts. When n is even and there is no possibility of increasing the number of rows, the *SSM* reduces the number of conflicts.

6 Conclusions

The interferences between concurrent vector streams accessing the memory system of a vector or multivector processor cause conflicts in the memory that reduce the processor efficiency.

The present paper has proposed a σ-independent access order to the vector stream elements (*SSM*), for which all the vector streams of a subfamily reference the memory modules with the same order. The use of the *SSM* associated with the proposed arbitration algorithm, avoids conflicts when the concurrent access correspond to vector streams of the same subfamily and the system works loose or tight. The proposal significantly reduces conflicts for other types of concurrent accesses.

The hardware solution that generates the *SSM* and the hardware used to access the vector registers have a moderate cost.

The simulations confirmed that the proposal can achieve the maximum number of operations per cycle, and the results showed that the *SSM* always outperforms the *classical access*, with performance increments between 13% and 152% for combinations of even and odd strided vector streams. In the interesting case of the concurrent access of 4 one-strided vector streams the increment in performance is 85%.

Acknowledgments

Work supported by the Ministry of Education of Spain, contract TIC-95-429.

References

1. M. Berry et al. Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. Int. Journal for Supercomputer Applications. 1989.
2. T. Cheung and J.E. Smith.. A Simulation Study of the CRAY X-MP Memory System. IEEE Transactions on Computers. Vol. C-35, no 7, october 1980.
3. A.M. del Corral and J.M. Llaberia. Reduce Conflicts between Vector Streams in Complex Memory Systems. CEPBA Report. DAC-UPC Report. June 1994.
4. A.M. del Corral and J.M. Llaberia. Eliminating Conflicts between Vector Streams in Interleaved Memory Systems. CEPBA Report. DAC-UPC Report. August 1995.
5. A. M. del Corral and J. M. Llaberia. Avoiding Inter-Vector-Conflicts in Complex Memory Systems. CEPBA Report, DAC-UPC Report. March 1996.
6. U. Detert and G. Hofemann. CRAY X-MP and Y-MP memory performance. Parallel Computing, North-Holland, n0 17, 1991.
7. J.W.C. FU and J.H.Patel. Memory Reference Behavior of Compiler Optimized Programs on High Speed Architectures. International Conference on Parallel Processing, Vol II. 1993.
8. D.T. Harper III and J.R. Jump. Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme.IEEE Transactions on Computers, Vol. C-36, no 12, december 1987.
9. D.T. Harper III and D.A. Linebarger. Conflict-free Vector Access Using a Dynamic Storage Scheme. IEEE Transactions on Computers, Vol. C-40. no 3, march 1991.
10. D.T. Harper III and J.R. Jump. Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme. IEEE Transactions on Computers, Vol. C-36, no 12, december 1987.
11. J.L. Hennessy and D.A. Patterson. Computer Architecture. A Quantitative Approach. Morgan Kaufmann Publishers, inc. 1990.
12. R.W. Hockney and C.R. Jesshope. *Parallel Computers 2*, Adam Hilger. 1988.
13. K. Kitai, T. Isobre, T. Sakakibara, S. Yazawa, Y. Tamaki, T. Tanaka and K. Ishii. Distributed Storage Control Unit for the Hitachi S-3800 Multivector Supercomputer. Int. Conference on Supercomputing, 1994.
14. L. Kurian. B. Choi, P.T. Hulina and L.D. Coroor. Module Partitioning and Interleaved Data Placement Schemes to Reduce Conflicts in Interleaved Memories. International Conference on Parallel Processing, Vol. 1. 1994.
15. D.L. Lee. Prime-way Interleaved Memory. International Conference on Parallel Processing. Vol. I. 1993.
16. W. Oed and O. Lange. On the Effective Bandwidth of Interleaved Memories in Vector Processor Systems. IEEE Transactions on Computers, Vol. C-34, *no 10*. October 1985.
17. R. Raghavan and J. Hayes. Reducing Interference Among Vector Accesses in Interleaved Memories. IEEE Transactions on Computers, Vol. 42, n.4. April 1993.
18. R.Raghavan and J.P.Hayes. On Randomly Interleaved Memories. Proceedings of the Supercomputing'90. November 1990.
19. J.E. Smith y W.R. Taylor. Accurate Modelling of Interconnection Networks. Int. Conference on Supercomputing, pp. 264 - 273. 1991.
20. J.E. Smith, W.C. Hsu and C. Hsiung. Future General Purpose Supercomputer Architectures. Proc. Supercomputing'90. 1990.
21. J. Torrellas and Z. Zhang. The Performance of Cedar Multistage Switching Network. Proceeding of the Supercomputing'94. November 1994.
22. M. Valero, T. Lang. J.M. Llaberia. M. Peiron. E. Ayguade y J.J. Navarro. Increasing the Number of Strides for Conflict-free Vector Access. Int. Symp. on Comp. Architecture. 1992.

Multilevel Mesh Partitioning for Aspect Ratio

C. Walshaw[1], M. Cross[1], R. Diekmann[2], and F. Schlimbach[2]

[1] School of Computing and Mathematical Sciences, The University of Greenwich, London, SE18 6PF, UK. {C.Walshaw, M.Cross}@gre.ac.uk

[2] Department of Computer Science, University of Paderborn, Fürstenallee 11, D-33102 Paderborn, Germany. {diek, schlimbo}@uni-paderborn.de

Abstract. Multilevel algorithms are a successful class of optimisation techniques which address the mesh partitioning problem. They usually combine a graph contraction algorithm together with a local optimisation method which refines the partition at each graph level. To date these algorithms have been used almost exclusively to minimise the cut-edge weight, however it has been shown that for certain classes of solution algorithm, the convergence of the solver is strongly influenced by the subdomain aspect ratio. In this paper therefore, we modify the multilevel algorithms in order to optimise a cost function based on aspect ratio. Several variants of the algorithms are tested and shown to provide excellent results.

1 Introduction

The need for mesh partitioning arises naturally in many finite element (FE) and finite volume (FV) applications. Meshes composed of elements such as triangles or tetrahedra are often better suited than regularly structured grids for representing completely general geometries and resolving wide variations in behaviour via variable mesh densities. Meanwhile, the modelling of complex behaviour patterns means that the problems are often too large to fit onto serial computers, either because of memory limitations or computational demands, or both. Distributing the mesh across a parallel computer so that the computational load is evenly balanced and the data locality maximised is known as mesh partitioning. It is well known that this problem is NP-complete, so in recent years much attention has been focused on developing suitable heuristics, and some powerful methods, many based on a graph corresponding to the communication requirements of the mesh, have been devised, e.g. [12].

A particularly popular and successful class of algorithms which address this mesh partitioning problem are known as multilevel algorithms. They usually combine a graph contraction algorithm which creates a series of progressively smaller and coarser graphs together with a local optimisation method which, starting with the coarsest graph, refines the partition at each graph level. These algorithms have been used almost exclusively to minimise the cut-edge weight, a cost which approximates the total communications volume in the underlying solver. This is an important goal in any parallel application, to minimise the communications overhead, however, it has been shown, [18], that for certain classes of solution algorithm, the convergence of the solver is actually heavily influenced by the shape or aspect ratio (AR) of the subdomains. In this paper therefore, we modify the multilevel algorithms (the matching and local optimisation) in order to optimise a cost function based on AR. We also abstract the process of modification in order to suggest how the multilevel strategy can be modified into a generic technique which can optimise arbitrary cost functions.

1.1 Domain decomposition preconditioners and aspect ratio

To motivate the need for aspect ratio we consider the requirements of a class of solution techniques. A natural *parallel* solution strategy for the underlying problem is to use an iterative solver such as the conjugate gradient (CG) algorithm together with domain decomposition (DD) preconditioning, e.g. [2]. DD methods take advantage of the partition of the mesh into subdomains by imposing artificial boundary conditions on the subdomain boundaries and solving the original problem on these subdomains, [4]. The subdomain solutions are independent of each other, and thus can be determined in parallel without any communication between processors. In a second step, an 'interface' problem is solved on the inner boundaries which depends on the jump of the subdomain solutions over the boundaries. This interface problem gives new conditions on the inner boundaries for the next step of subdomain solution. Adding the results of the third step to the first gives the new conjugate search direction in the CG algorithm.

The time needed by such a preconditioned CG solver is determined by two factors, the maximum time needed by any of the subdomain solutions and the number of iterations of the global CG. Both are at least partially determined by the shape of the subdomains. Whilst an algorithm such as the multigrid method as the solver on the subdomains is relatively robust against shape, the number of global iterations are heavily influenced by the AR of subdomains, [17]. Essentially, the subdomains can be viewed as elements of the interface problem, [7, 8], and just as with the normal finite element method, where the condition of the matrix system is determined by the AR of elements, the condition of the preconditioning matrix is here dependent on the AR of subdomains.

1.2 Overview

Below, in Section 2, we introduce the mesh partitioning problem and establish some terminology. We then discuss the mesh partitioning problem as applied to AR optimisation and describe how the graph needs to be modified to carry this out. Next, in Section 3, we describe the multilevel paradigm and present and compare three possible matching algorithms which take account of AR. In Section 4 we then describe a Kernighan-Lin (KL) type iterative local optimisation algorithm and describe two possible modifications which aim to optimise AR. Finally in Section 5 we compare the results with a cut edge partitioner, suggest how the multilevel strategy can be modified into a generic technique and present some ideas for further investigation.

The principal innovations described in this paper are:

- In §2.2 we describe how the graph can be modified to take AR into account.
- In §3.2 we describe three matching algorithms based on AR.
- In §4.3 we describe two ways of using the cost function to optimise for AR.
- In §4.4 we describe how the bucket sort can be modified to take into account non-integer gains.

2 The mesh partitioning problem

To define the mesh partitioning problem, let $G = G(V, E)$ be an undirected graph of vertices V, with edges E which represent the data dependencies in the mesh. We assume that both vertices and edges can be weighted (with positive integer values) and that $|v|$ denotes the weight of a vertex v and similarly for edges and sets of vertices and edges. Given that the mesh needs to be distributed to P processors, define a partition π to be a

mapping of V into P disjoint subdomains S_p such that $\bigcup_p S_p = V$. To evenly balance the load, the optimal subdomain weight is given by $\overline{S} := \lceil |V|/P \rceil$ (where the ceiling function $\lceil x \rceil$ returns the smallest integer $\geq x$) and the *imbalance* is then defined as the maximum subdomain weight divided by the optimal (since the computational speed of the underlying application is determined by the most heavily weighted processor).

The definition of the mesh-partitioning problem is to find a partition which evenly balances the load or vertex weight in each subdomain whilst minimising some cost function Γ. Typically this cost function is simply the total weight of cut edges, but in this paper we describe a cost function based on AR. A more precise definition of the mesh-partitioning problem is therefore to find π such that $S_p \leq \overline{S}$ and such that Γ is minimised.

2.1 The aspect ratio and cost function

We seek to modify the methods by optimising the partition on the basis of AR rather than cut-edge weight. In order to do this it is necessary to define a cost function which we seek to minimise and a logical choice would be $\max_p \mathrm{AR}(S_p)$, where $\mathrm{AR}(S_p)$ is the AR of the subdomain S_p. However maximum functions are notoriously difficult to optimise (indeed it is for this reason that most mesh partitioning algorithms attempt to minimise the total cut-edge weight rather than the maximum between any two subdomains) and so instead we choose to minimise the average AR

$$\Gamma_{AR} = \sum_p \frac{\mathrm{AR}(S_p)}{P}. \tag{1}$$

There are several definitions of AR, however, and for example, for a given polygon S, a typical definition, [15], is the ratio of the largest circle which can be contained entirely within S (inscribed circle) to the smallest circle which entirely contains S (circumcircle). However these circles are not easy to calculate for arbitrary polygons and in an optimisation code where ARs may need to be calculated very frequently, we do not believe this to be a practical metric. It may also fail to express certain irregularities of shape. A careful discussion of the relative merits of different ways of measuring AR may be found in [16] and for the purposes of this paper we follow the ideas therein and define the AR of a given shape by measuring the ratio of its perimeter length (surface area in 3d) over that of some ideal shape with identical area (volume in 3d).

Suppose then that in 2d the ideal shape is chosen to be a square. Given a polygon S with area ΩS and perimeter length ∂S, the ideal perimeter length (the perimeter length of a square with area ΩS) is $4\sqrt{\Omega S}$ and so the AR is defined as $\partial S/4\sqrt{\Omega S}$. Alternatively, if the ideal shape is chosen to be a circle then the same argument gives the AR of $\partial S/2\sqrt{\pi \Omega S}$. In fact, given the definition of the cost function (1) it can be seen that these two definitions will produce the same optimisation problem (and hence the same results) with the cost just modified by a constant C (where $C = 1/4$ for the square and $1/2\sqrt{\pi}$ for circle). These definitions of AR are easily extendible to 3d and given a polyhedron S with volume ΩS and surface area ∂S, the AR can be calculated as $C\partial S/(\Omega S)^{2/3}$, where $C = 1/4$ if the cube is chosen as the optimal shape and $C = 1/(4\pi)^{1/3}3^{2/3}$ for the sphere. Note that henceforth, in order to talk in general terms for both 2d & 3d, given an object S we shall use the terms ∂S or *surface* for the surface area (3d) or perimeter length (2d) of the object and ΩS or *volume* for the volume (3d) or area (2d).

Of the above definitions of AR we choose to use the square/cube based formulae for two reasons: firstly because we are attempting to partition a mesh into interlocking subdomains (and circles/spheres are not known for their interlocking qualities) and secondly because it gives a convenient formula for the cost function of:

$$\Gamma_{\text{template}} = \frac{1}{C} \sum_p \frac{\partial S_p}{(\Omega S_p)^{\frac{d-1}{d}}} \tag{2}$$

where $C = 2dP$ and d (= 2 or 3) is the dimension of the mesh. We refer to this cost function as Γ_{template} or Γ_t because of the way it tries to match shapes to chosen templates.

In fact, it will turn out (see for example §3.2) that even this function may be too complex for certain optimisation needs and we can define a simpler one by assuming that all subdomains have approximately the same volume, $\Omega S_p \approx \Omega M/P$, where ΩM is the total volume of the mesh. This assumption may not necessarily be true, but it is likely to be true locally (see §4.5). We can then approximate (2) by

$$\Gamma_{\text{template}} \approx \frac{1}{C'} \sum_p \partial S_p \tag{3}$$

where $C' = 2dP^{\frac{1}{d}}(\Omega M)^{\frac{d-1}{d}}$. This can be simplified still further by noting that the surface of each subdomain S_p consists of two components, the *exterior* surface, $\partial^e S_p$, where the surface of the subdomain coincides with the surface of the mesh ∂M, and the *interior* surface, $\partial^i S_p$, where S_p is adjacent to other subdomains and the surface cuts through the mesh. Thus we can break the $\sum_p \partial S_p$ term in (3) into two parts $\sum_p \partial^i S_p$ and $\sum_p \partial^e S_p$ and simplify (3) further by noting that $\sum_p \partial^e S_p$ is just ∂M, the exterior surface of the mesh M. This then gives us a second cost function to optimise:

$$\Gamma_{\text{surface}} = \frac{1}{K_1} \sum_p \partial^i S_p + K_2 \tag{4}$$

where $K_1 = 2dP^{\frac{1}{d}}(\Omega M)^{\frac{d-1}{d}}$ and $K_2 = \partial M/K_1$. We refer to this cost function as Γ_{surface} or Γ_s because it is just concerned with optimising surfaces.

2.2 Modifying the graph

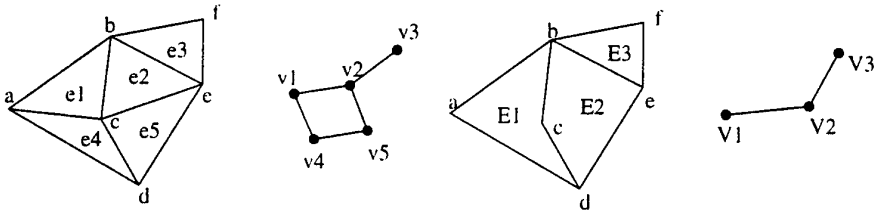


Fig. 1. Left to right: a simple mesh (a), its dual (b), the same mesh with combined elements (c) and its dual (d)

To use these cost functions in a graph-partitioning context, we must add some additional qualities to the graph. Figure 1 shows a very simple mesh (1a) and its dual graph (1b). Each element of the mesh corresponds to a vertex in the graph. The vertices of the graph can be weighted as is usual (to carry out load-balancing) but in addition, vertices store the volume and total surface of their corresponding element (e.g. $\Omega v_1 = \Omega e_1$ and $\partial v_1 = \partial e_1$). We also weight the edges of the graph with the size of the surface they correspond to. Thus, in Figure 1, if $D(b,c)$ refers to the distance between points b and c, then the weight of edge (v_1, v_2) is set to $D(b,c)$. In this way, for vertices v_i corresponding to elements which have no exterior surface, the sum of their edge weights is equivalent to their surface ($\partial v_i = \sum_E |(v_i, v_j)|$). Thus for vertex v_2, $\partial v_2 = \partial e_2 = D(b,c) + D(c,e) + D(e,b) = |(v_2, v_1)| + |(v_2, v_3)| + |(v_2, v_5)|$.

When it comes to combining elements together, either into subdomains, or for the multilevel matching (§3) these properties, volume and surface can be easily combined. Thus in Figure 1c where $E_1 = e_1 + e_4$, $E_2 = e_3 + e_5$ and $E_3 = e_3$ we see that volumes can be directly summed, for example $\Omega V_1 = \Omega E_1 = \Omega e_1 + \Omega e_4 = \Omega v_1 + \Omega v_4$, as can edge weights, e.g. $|(V_1, V_2)| = D(b,c) + D(c,d) = |(v_1, v_2)| + |(v_4, v_5)|$. The surface of a combined object S is the sum of the surfaces of its constituent parts less twice the interior surface, e.g. $\partial V_1 = \partial E_1 = \partial e_1 + \partial e_4 - 2 \times D(a,c) = \partial v_1 + \partial v_1 - 2|(v_1, v_4)|$. These properties are very similar to properties in conventional graph algorithms, where the volume combines in the same way as weight and surfaces combine as the sum of edge weights (although including an additional term which expresses the exterior surface ∂^e). The edge weights function identically.

Note that with these modifications to the graph, it can be seen that if we optimise using the Γ_s cost function (4), the AR mesh partitioning problem is identical to the cut-edge weight mesh partitioning problem with a special edge weighting. However, the inclusion of non integer edge weights does have an effect on the some of the techniques that can be used (e.g. see §4.4).

2.3 Testing the algorithms

Table 1. Test meshes

mesh	no. vertices	no. edges	type	aspect ratio	mesh grading
uk	4824	6837	2d triangles	3.39	7.98e+02
t60k	60005	89440	2d triangles	1.60	2.00e+00
dime20	224843	336024	2d triangles	1.87	3.70e+03
cs4	22499	43858	3d tetrahedra	1.07	9.64e+01
mesh100	103081	200976	3d tetrahedra	1.63	2.45e+02
cyl3	232362	457853	3d tetrahedra	1.28	8.42e+00

Throughout this paper we compare the effectiveness of different approaches using a set of test meshes. The algorithms have been implemented within the framework of JOSTLE, a mesh partitioning software tool developed at the University of Greenwich and freely available for academic and research purposes under a licensing agreement (available from `http://www.gre.ac.uk/~c.walshaw/jostle`). The experiments were carried out on a DEC Alpha with a 466 MHz CPU and 1 Gbyte of memory. Due to space considerations we only include 6 test meshes but they have been chosen to be a representative sample of medium to large scale real-life problems and include both 2d and 3d examples. Table 1 gives a list of the meshes and their sizes in terms of the number of vertices and edges. The table also shows the aspect ratio of each entire mesh and the mesh grading, which here we define as the maximum surface of any element over the minimum surface, and these two figures give a guide as to how difficult the optimisation

may be. For example, 'uk' is simply a triangulation of the British mainland and hence has a very intricate boundary and therefore a high aspect ratio. Meanwhile, 'dime20' which has a moderate aspect ratio, has been very heavily refined in parts and thus has a high mesh grading – the largest element has a surface around 3,700 times larger than that of the smallest.

Table 2. Final results using template cost matching and surface gain/template cost optimisation

	$P = 16$			$P = 32$			$P = 64$			$P = 128$		
mesh	Γ_t	$\|E_c\|$	t_s	Γ_t	$\|E_c\|$	t_s	Γ_t	$\|E_c\|$	t_s	Γ_t	$\|E_c\|$	t_s
uk	1.48	206	0.12	1.31	331	0.12	1.23	543	0.22	1.25	917	0.50
t60k	1.16	1003	1.63	1.10	1547	2.07	1.11	2437	2.33	1.11	3647	2.65
dime20	1.22	1623	5.78	1.20	2868	5.17	1.15	4406	5.70	1.12	6620	7.57
cs4	1.22	2727	0.85	1.22	3738	0.90	1.23	5066	1.12	1.23	6747	1.60
mesh100	1.25	5950	3.20	1.24	8752	3.53	1.26	12467	4.13	1.28	17346	5.13
cyl3	1.21	11141	10.05	1.21	15944	10.77	1.23	22378	13.02	1.22	29719	13.18

Table 2 shows the results of the final combination of algorithms – TCM (see §3.2) and SGTC (see §4.3) – which were chosen as a benchmark for the other combinations. For the 4 different values of P (the number of subdomains), the table shows the average aspect ratio as given by Γ_t, the edge cut $|E_c|$ (that is the number of cut edges, not the weight of cut edges weighted by surface size) and the time in seconds, t_s, to partition the mesh. Notice that with the exception of the 'uk' mesh, all partitions have average aspect ratios of less than 1.30 which is well within the target range suggested in [6]. Indeed for the 'uk' mesh it is no surprise that the results are not optimal because the subdomains inherit some of the poor AR from the original mesh (which has an AR of 3.39) and it is only when the mesh is split into small enough pieces, $P = 64$ or 128, that the optimisation succeeds in ameliorating this effect. Intuitively this also gives a hint as to why DD methods are a very successful technique as a solver.

3 The multilevel paradigm

In recent years it has been recognised that an effective way of both speeding up partition refinement and, perhaps more importantly giving it a global perspective is to use multilevel techniques. The idea is to match pairs of vertices to form *clusters*, use the clusters to define a new graph and recursively iterate this procedure until the graph size falls below some threshold. The coarsest graph is then partitioned and the partition is successively optimised on all the graphs starting with the coarsest and ending with the original. This sequence of contraction followed by repeated expansion/optimisation loops is known as the multilevel paradigm and has been successfully developed as a strategy for overcoming the localised nature of the KL (and other) optimisation algorithms. The multilevel idea was first proposed by Barnard & Simon, [1], as a method of speeding up spectral bisection and improved by Hendrickson & Leland, [11], who generalised it to encompass local refinement algorithms. Several algorithms for carrying out the matching have been devised by Karypis & Kumar, [13], while Walshaw & Cross describe a method for utilising imbalance in the coarsest graphs to enhance the final partition quality, [19].

3.1 Implementation

Graph contraction. To create a coarser graph $G_{l+1}(V_{l+1}, E_{l+1})$ from $G_l(V_l, E_l)$ we use a variant of the edge contraction algorithm proposed by Hendrickson & Leland,

[11]. The idea is to find a maximal independent subset of graph edges, or a *matching* of vertices, and then collapse them. The set is independent because no two edges in the set are incident on the same vertex (so no two edges in the set are adjacent), and maximal because no more edges can be added to the set without breaking the independence criterion. Having found such a set, each selected edge is collapsed and the vertices, $u_1, u_2 \in V_l$ say, at either end of it are merged to form a new vertex $v \in V_{l+1}$ with weight $|v| = |u_1| + |u_2|$.

The initial partition. Having constructed the series of graphs until the number of vertices in the coarsest graph is smaller than some threshold, the normal practice of the multilevel strategy is to carry out an initial partition. Here, following the idea of Gupta, [10], we contract until the number of vertices in the coarsest graph is the same as the number of subdomains, P, and then simply assign vertex i to subdomain S_i. Unlike Gupta, however, we do not carry out repeated expansion/contraction cycles of the coarsest graphs to find a well balanced initial partition but instead, since our optimisation algorithm incorporates balancing, we commence on the expansion/optimisation sequence immediately.

Partition expansion. Having optimised the partition on a graph G_l, the partition must be interpolated onto its parent G_{l-1}. The interpolation itself is a trivial matter; if a vertex $v \in V_l$ is in subdomain S_p then the matched pair of vertices that it represents, $v_1, v_2 \in V_{l-1}$, will be in S_p.

3.2 Incorporating aspect ratio

The matching part of the multilevel strategy can be easily modified in several ways to take into account AR and in each case the vertices are visited (at most once) using a randomly ordered linked list. Each vertex is then matched with an unmatched neighbour using the chosen matching algorithm and it and its match removed from the list. Vertices with no unmatched neighbours remain unmatched and are also removed. In addition to **Random Matching (RM)**, [12], where vertices are matched with random neighbours, we propose and have tested 3 matching algorithms:

Surface Matching (SM). As we have seen in §2.2, the AR partitioning problem can be approximated by the cut-edge weight problem using (4), the Γ_s cost function, and so the simplest matching is to use the Heavy Edge approach of Karypis & Kumar, [13], where the vertex matches across the heaviest edge to any of its unmatched neighbours. This is the same as matching across the largest surface (since here edge weights represent surfaces) and we refer to this as *surface matching*.

Template Cost Matching (TCM). A second approach follows the ideas of Bouhmala, [3], and matches with the neighbour which minimises the cost function. In this case, the chosen vertex matches with the unmatched neighbour which gives the resulting element the best aspect ratio. Using the Γ_t cost function, we refer to this as *template cost matching*.

Surface Cost Matching (SCM). This is the same idea as TCM only using the Γ_s cost function, (4), which is faster to calculate.

3.3 Results for different matching functions

In Tables 3, 4 & 5 we compare the results in Table 2, where TCM was used, with RM, SM & SCM respectively. In all cases the SGTC optimisation algorithm (see §4.3) was used. For each value of P, the first column shows the average AR, Γ_t of the partitioning. The second column for each value of P then compares results with those in Table 2 using the

metric $\frac{\Gamma(\mathrm{RM})-1}{\Gamma(\mathrm{TCM})-1}$ for RM, etc. Thus a figure > 1 means that RM has produced worse results than TCM. These comparisons are then averaged and so it can be seen, e.g. for $P = 16$ that RM produces results 24% (1.24) worse on average than TCM. Indeed the average quality of partitions produced by RM was 30% worse than TCM. This is not altogether surprising since the AR of elements in the coarsest graph could be very poor if the matching takes no account of it, and hence the optimisation has to work with badly shaped elements.

Table 3. Random matching results compared with template cost matching

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	Γ_t	$\frac{\Gamma(\mathrm{RM})-1}{\Gamma(\mathrm{TCM})-1}$	Γ_t	$\frac{\Gamma(\mathrm{RM})-1}{\Gamma(\mathrm{TCM})-1}$	Γ_t	$\frac{\Gamma(\mathrm{RM})-1}{\Gamma(\mathrm{TCM})-1}$	Γ_t	$\frac{\Gamma(\mathrm{RM})-1}{\Gamma(\mathrm{TCM})-1}$
uk	1.50	1.04	1.38	1.25	1.25	1.06	1.23	0.91
t60k	1.20	1.28	1.16	1.59	1.17	1.53	1.17	1.54
dime20	1.30	1.37	1.31	1.57	1.27	1.79	1.23	1.89
cs4	1.29	1.31	1.27	1.21	1.30	1.30	1.26	1.15
mesh100	1.31	1.24	1.29	1.24	1.31	1.19	1.32	1.15
cyl3	1.25	1.19	1.25	1.19	1.26	1.15	1.27	1.22
Average		1.24		1.34		1.34		1.31

When it comes to comparing TCM with SM & SCM (Tables 4 & 5) there is actually very little difference; SM is about 3.5% worse and SCM only about 1.5%. This suggests that the multilevel strategy is relatively robust to the matching algorithm *provided* the AR is taken into account in some way.

Table 4. Surface matching results compared with template cost matching

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	Γ_t	$\frac{\Gamma(\mathrm{SM})-1}{\Gamma(\mathrm{TCM})-1}$	Γ_t	$\frac{\Gamma(\mathrm{SM})-1}{\Gamma(\mathrm{TCM})-1}$	Γ_t	$\frac{\Gamma(\mathrm{SM})-1}{\Gamma(\mathrm{TCM})-1}$	Γ_t	$\frac{\Gamma(\mathrm{SM})-1}{\Gamma(\mathrm{TCM})-1}$
uk	1.54	1.13	1.34	1.11	1.24	1.01	1.28	1.10
t60k	1.14	0.87	1.11	1.05	1.12	1.10	1.12	1.08
dime20	1.26	1.18	1.24	1.23	1.15	1.00	1.13	1.04
cs4	1.22	0.97	1.24	1.08	1.24	1.04	1.23	1.00
mesh100	1.20	0.78	1.24	1.03	1.27	1.04	1.26	0.94
cyl3	1.19	0.93	1.21	1.02	1.24	1.05	1.24	1.08
Average		0.98		1.08		1.04		1.04

Table 5. Surface cost matching results compared with template cost matching

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	Γ_t	$\frac{\Gamma(\mathrm{SCM})-1}{\Gamma(\mathrm{TCM})-1}$	Γ_t	$\frac{\Gamma(\mathrm{SCM})-1}{\Gamma(\mathrm{TCM})-1}$	Γ_t	$\frac{\Gamma(\mathrm{SCM})-1}{\Gamma(\mathrm{TCM})-1}$	Γ_t	$\frac{\Gamma(\mathrm{SCM})-1}{\Gamma(\mathrm{TCM})-1}$
uk	1.47	0.99	1.31	1.00	1.27	1.14	1.25	0.98
t60k	1.11	0.69	1.10	0.99	1.14	1.23	1.13	1.14
dime20	1.23	1.06	1.18	0.91	1.14	0.93	1.13	1.02
cs4	1.23	1.04	1.23	1.04	1.24	1.03	1.23	1.00
mesh100	1.23	0.91	1.25	1.07	1.25	0.99	1.27	0.97
cyl3	1.22	1.06	1.23	1.10	1.23	1.02	1.24	1.06
Average		0.96		1.02		1.05		1.03

We are not primarily concerned with partitioning times here, but for the record, RM was about 0.5% slower than TCM (although this is well within the limits of noise). This is because the optimisation stage took considerably longer (although the matching was

much faster than TCM). SM & SCM were 3.3% & 1.8% faster respectively than TCM. Overall this suggests that TCM is the algorithm of choice although there is little benefit over SM & SCM.

4 The Kernighan-Lin optimisation algorithm

In this section we discuss the key features of an optimisation algorithm, fully described in [19] and then in §4.3 describe how it can be modified to optimise for AR. It is a Kernighan-Lin (KL) type algorithm incorporating a hill-climbing mechanism to enable it to escape from local minima. The algorithm uses bucket sorting (§4.4), the linear time complexity improvement of Fiduccia & Mattheyses, [9], and is a partition optimisation formulation; in other words it optimises a partition of P subdomains rather than a bisection.

4.1 The gain function

A key concept in the method is the idea of *gain*. The gain $g(v,q)$ of a vertex v in subdomain S_p can be calculated for every other subdomain, S_q, $q \neq p$, and expresses how much the cost of a given partition would be improved were v to migrate to S_q. Thus, if π denotes the current partition and π' the partition if v migrates to S_q then for a cost function Γ, the gain $g(v,q) = \Gamma(\pi') - \Gamma(\pi)$. Assuming the migration of v only affects the cost of S_p and S_q (as is true for Γ_t and Γ_s) then we get

$$g(v,q) = \mathrm{AR}(S_q + v) - \mathrm{AR}(S_q) + \mathrm{AR}(S_p - v) - \mathrm{AR}(S_p). \tag{5}$$

For Γ_t this gives an expression which cannot be further simplified, however, for Γ_s, since

$$\begin{aligned}\mathrm{AR}(S_q + v) - \mathrm{AR}(S_q) &= \frac{1}{K_1}\left\{\partial^i(S_q + v) - \partial^i S_q\right\} \\ &= \frac{1}{K_1}\left\{\partial^i S_q + \partial^i v - 2|(S_q, v)| - \partial^i S_q\right\} \\ &= \frac{1}{K_1}\left\{\partial^i v - 2|(S_q, v)|\right\}\end{aligned}$$

(where $|(S_q, v)|$ denotes the sum of edge weights between S_q and v), we get

$$g(v,q) = \frac{2}{K_1}\left\{|(S_p, v)| - |(S_q, v)|\right\} \tag{6}$$

Notice in particular that $g(v,q)$ is the same as the cut-edge weight gain function and that it is entirely localised, i.e. the gain of a vertex only depends on the length of its boundaries with a subdomain and not on any intrinsic qualities of the subdomain which could be changed by non-local migration.

4.2 The iterative optimisation algorithm

The serial optimisation algorithm, as is typical for KL type algorithms, has inner and outer iterative loops with the outer loop terminating when no migration takes place during an inner loop. The optimisation uses two bucket sorting structures or bucket trees

(see below, §4.4) and is initialised by calculating the gain for all border vertices and inserting them into one of the bucket trees. These vertices will subsequently be referred to as *candidate* vertices and the tree containing them as the *candidate tree*.

The inner loop proceeds by examining candidate vertices, highest gain first (by always picking vertices from the highest ranked bucket), testing whether the vertex is acceptable for migration and then transferring it to the other bucket tree (the tree of *examined* vertices). This inner loop terminates when the candidate tree is empty although it may terminate early if the partition cost (i.e. the number of cut edges) rises too far above the cost of the best partition found so far. Once the inner loop has terminated any vertices remaining in the candidate tree are transferred to the examined tree and finally pointers to the two trees are swapped ready for the next pass through the inner loop.

The algorithm also uses a KL type hill-climbing strategy; in other words vertex migration from subdomain to subdomain can be *accepted* even if it degrades the partition quality and later, based on the subsequent evolution of the partition, either rejected or *confirmed*. During each pass through the inner loop, a record of the optimal partition achieved by migration within that loop is maintained together with a list of vertices which have migrated since that value was attained. If subsequent migration finds a 'better' partition then the migration is *confirmed* and the list is reset. Once the inner loop is terminated, any vertices remaining in the list (vertices whose migration has not been confirmed) are migrated back to the subdomains they came from when the optimal cost was attained.

The algorithm, together with conditions for vertex migration acceptance and confirmation is fully described in [19].

4.3 Incorporating aspect ratio: localisation

One of the advantages of using cut-edge weight as a cost function is its localised nature. When a graph vertex migrates from one subdomain to another, only the gains of adjacent vertices are affected. In contrast, when using the graph to optimise AR, if a vertex v migrates from S_p to S_q, the volume and surface of both subdomains will change. This in turn means that, when using the template cost function (2), the gain of all border vertices both within and abutting subdomains S_p and S_q will change. Strictly speaking, all these gains should be adjusted with the huge disadvantage that this may involve thousands of floating point operations and hence be prohibitively expensive. As an alternative, therefore, we propose two localised variants:

Surface Gain/Surface Cost (SGSC). The simplest way to localise the updating of the gains is to make the assumption in §2.1 that the subdomains all have approximately equal volume and to use the surface cost function Γ_s from (4). As mentioned in §2.2 the problem immediately reduces to the cut-edge weight problem, albeit with non-integer edge weights, and from (6) only the gains of the vertices adjacent to the migrating vertex will need updating. However, if this assumption is not true, it is not clear how well Γ_s will optimise the AR and below we provide some experimental results.

Surface Gain/Template Cost (SGTC). The second method we propose for localising the updates of gain relies on the observation that the gain is simply used as a method of rating the elements so that the algorithm always visits those with highest gain first (using the bucket sort). It is not clear how crucial this rating is to the success of the algorithm and indeed Karypis & Kumar demonstrated that (at least when optimising for cut-edge weight) almost as good results can be achieved by simply visiting the vertices in random order, [14]. We therefore propose approximating the gain with the surface cost function Γ_s from (4) to rate the elements and store them in the bucket tree structure, but

using the template cost function Γ_t from (2) to assess the change in cost when actually migrating an element. This localises the gain function.

4.4 Incorporating aspect ratio: bucket sorting with non-integer gains

The bucket sort is an essential tool for the efficient and rapid sorting and adjustment of vertices by their gain. The concept was first suggested by Fiduccia & Mattheyses in [9] and the idea is that all vertices of a given gain g are placed together in a 'bucket' which is ranked g. Finding a vertex with maximum gain then simply consists of finding the (non-empty) bucket with the highest rank and picking a vertex from it. If the vertex is subsequently migrated from one subdomain to another then the gains of any affected vertices have to be adjusted and the list of vertices which are candidates for migration resorted by gain. Using a bucket sort for this operation simply requires recalculating the gains and transferring the affected vertices to the appropriate buckets. If a bucket sort were not used and, say, the vertices were simply stored in a list in gain order, then the entire list would require resorting (or at least merge-sorting with the sorted list of adjusted vertices), an essentially $O(N)$ operation for every migration.

The implementation of the bucket sort is fully described in [19]. It includes a ranking for prioritising vertices for migration which incorporates their weight as well as their gain. The non-empty buckets are stored in a binary-tree to save excessive memory use (since we do not know *a priori* how many buckets will be needed) and this structure is referred to above as a bucket tree.

The only difficulty in adapting this procedure to AR optimisation is that with non-integer edge weight, the gains are also real non-integer numbers. This is not a major problem in itself as we can just give buckets an interval of gains rather than a single integer, i.e. the bucket ranked 1 could contain any vertex with gain in the interval $[1.0, 2.0)$. However, if using the surface gain function, the issue of scaling then arises since for a mesh entirely contained within the unit square/cube, all the vertices are likely to end up in one of two buckets (dependent only on whether they have positive or negative gains). Fortunately, if using Γ_s as a gain function, as in SGSC and SGTC, we can easily calculate the maximum possible gain. This would occur if the vertex with the largest surface, $v \in S_p$ say, were entirely surrounded by neighbours in S_q. The maximum possible gain is then $2 \max_{v \in V} \partial v$ (strictly speaking $2 \max_{v \in V} \partial^i v$) and similarly the minimum gain is $-2 \max_{v \in V} \partial v$. This means we can easily choose the number of buckets and scale the gain accordingly. A problem still arises for meshes with a high grading because many of the elements will have an insignificant surface area compared to the maximum. However the experiments carried out here all used a scaling which allowed a maximum of 100 buckets and we have tested the algorithm with up to 10,000 buckets without significant penalty in terms either memory or run-time.

4.5 Results for different optimisation functions

Table 6 compares SGSC against the SGTC results in Table 2. Both set of results use template cost matching (TCM). The table is in the same form as those in §3.3 and shows that there is on average only a tiny difference between the two (SGTC is 0.5% better than SGSC) and again, with the exception of the 'uk' mesh for $P = 16$ & 32, all results have an average AR of less than 1.30. This implication of this table is that the assumption made in §2.1, that all subdomains have approximately the same volume, is reasonably good. However this assumption is not necessarily true, because for example, for $P = 128$, the 'dime20' mesh, with its high grading, has a ratio of $\max \Omega S_p / \min \Omega S_p =$

2723. A possible explanation is that although the assumption is false globally, it is true locally, since the mesh density does not change too gradually (as should be the case with most meshes generated by adaptive refinement) and so the volume of each subdomain is approximately equal to that of its neighbours.

Table 6. Surface gain/surface cost optimisation compared with surface gain/template cost

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	Γ_t	$\frac{\Gamma(\text{SGSC})-1}{\Gamma(\text{SGTC})-1}$	Γ_t	$\frac{\Gamma(\text{SGSC})-1}{\Gamma(\text{SGTC})-1}$	Γ_t	$\frac{\Gamma(\text{SGSC})-1}{\Gamma(\text{SGTC})-1}$	Γ_t	$\frac{\Gamma(\text{SGSC})-1}{\Gamma(\text{SGTC})-1}$
uk	1.49	1.02	1.32	1.05	1.24	1.02	1.23	0.92
t60k	1.15	0.95	1.10	0.96	1.12	1.07	1.12	1.11
dime20	1.23	1.03	1.17	0.86	1.15	0.98	1.11	0.91
cs4	1.20	0.90	1.23	1.05	1.24	1.03	1.22	0.97
mesh100	1.24	0.95	1.26	1.10	1.27	1.06	1.27	0.97
cyl3	1.23	1.10	1.22	1.08	1.24	1.06	1.22	1.00
Average		0.99		1.01		1.04		0.98

Again we are not not primarily concerned with partitioning times, but it was surprising to see that SGSC was an average 30% slower than SGTC. A possible explanation is that although the cost function Γ_s is a good approximation, Γ_t is a more global function and so the optimisation converges more quickly.

5 Discussion

5.1 Comparison with cut-edge weight partitioning

In Table 7 we compare AR as produced by the edge cut partitioner (EC) described in [19] with the results in Table 2. On average AR partitioning produces results which are 16% better than those of the edge cut partitioner (as could be expected). However, for the mesh 'cs4' EC partitioning is consistently better and this is a subject for further investigation.

Table 7. AR results for the edge cut partitioner compared with the AR partitioner

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	Γ_t	$\frac{\Gamma(\text{EC})-1}{\Gamma(\text{AR})-1}$	Γ_t	$\frac{\Gamma(\text{EC})-1}{\Gamma(\text{AR})-1}$	Γ_t	$\frac{\Gamma(\text{EC})-1}{\Gamma(\text{AR})-1}$	Γ_t	$\frac{\Gamma(\text{EC})-1}{\Gamma(\text{AR})-1}$
uk	1.52	1.08	1.33	1.07	1.26	1.09	1.28	1.14
t60k	1.19	1.18	1.[illegible]	1.76	1.17	1.47	1.17	1.55
dime20	1.32	1.45	1.[illegible]	1.34	1.25	1.65	1.21	1.72
cs4	1.19	0.86	1.[illegible]	0.93	1.20	0.87	1.21	0.92
mesh100	1.22	0.89	1.[illegible]	0.91	1.26	1.03	1.24	0.86
cyl3	1.22	1.05	1.[illegible]	1.09	1.23	1.00	1.23	1.02
Average		1.09		1.18		1.19		1.20

Meanwhile in Table 8 we compare the edge cut produced by the EC partitioner with that of the AR partitioner. Again as expected, EC partitioning produces the best results (about 11% better than AR). In terms of time, the EC partitioner is about 26% faster than AR on average. Again this is no surprise since the AR partitioning involves floating point operations (assessing cost and combining elements) while EC partitioning only requires integer operations.

Table 8. $|E_c|$ results for the edge cut partitioner compared with the AR partitioner

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	$\|E_c\|$	$\frac{\|E_c\|(\text{RM})}{\|E_c\|(\text{AR})}$	$\|E_c\|$	$\frac{\|E_c\|(\text{RM})}{\|E_c\|(\text{AR})}$	$\|E_c\|$	$\frac{\|E_c\|(\text{RM})}{\|E_c\|(\text{AR})}$	$\|E_c\|$	$\frac{\|E_c\|(\text{RM})}{\|E_c\|(\text{AR})}$
uk	189	0.92	290	0.88	478	0.88	845	0.92
t60k	974	0.97	1588	1.03	2440	1.00	3646	1.00
dime20	1326	0.82	2294	0.80	3637	0.83	5497	0.83
cs4	2343	0.86	3351	0.90	4534	0.89	6101	0.90
mesh100	4577	0.77	7109	0.81	10740	0.86	14313	0.83
cyl3	10458	0.94	14986	0.94	20765	0.93	27869	0.94
Average		0.88		0.89		0.90		0.90

5.2 Generic multilevel mesh partitioning

In this paper we have adapted a mesh partitioning technique originally designed to solve the edge cut partitioning problem to a different cost function. The question then arises, is the multilevel strategy an appropriate technique for solving partitioning problems (or indeed other optimisation problems) with different cost functions? Clearly this is an impossible question to answer in general but a few pertinent remarks can be made:

- For the AR based cost functions at least, the method seems relatively sensitive to whether the cost is included in the matching. This suggests that, if possible, a generic multilevel partitioner should use the cost function to minimise the cost of the matchings. Note, however, that this may not be possible as a cost function which, say, measured the cost of a mapping onto a particular processor topology would be unable to function since at the matching stage no partition, and hence no mapping exists.
- The optimisation relies, for efficiency at least, on having a local gain function in order that the migration of a vertex does not involve an $O(N)$ update. Here we were able to localise the cost function by making a simple approximation to give a local gain function, however, it is not clear that this is always possible.
- The bucket sort is reasonably simple to convert to non-integer gains, however this relies on being able to estimate the maximum gain. If this is not possible it may not be easy to generate a good scaling which separates vertices of different gains into different buckets.

5.3 Conclusion and future research

We have shown that the multilevel strategy can be modified to optimise for aspect ratio. To fully validate the method, however, we need to demonstrate that the measure of aspect ratio used here does indeed provide the benefits for DD preconditioners that the theoretical results suggest. It is also desirable to measure the correlation between aspect ratio and convergence in the solver.

Also, although parallel implementations of the multilevel strategy do exist, e.g. [20], it is not clear how well AR optimisation, with its more global cost function, will work in parallel and this is another direction for future research. Some related work already exists in the context of a parallel dynamic adaptive mesh environment, [5, 6, 16], but these are not multilevel methods and it was necessary to use a combination of several complex cost functions in order to achieve reasonable results so the question arises whether multilevel techniques can help to overcome this.

References

1. S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice & Experience*, 6(2):101–117, 1994.
2. S. Blazy, W. Borchers, and U. Dralle. Parallelization methods for a characteristic's pressure correction scheme. In E. H. Hirschel, editor, *Flow Simulation with High Performance Computers II, Notes on Numerical Fluid Mechanics*, 1995.
3. N. Bouhmala. *Partitioning of Unstructured Meshes for Parallel Processing*. PhD thesis, Inst. d'Informatique, Univ. Neuchatel, 1998.
4. J. H. Bramble, J. E. Pasciac, and A. H. Schatz. The Construction of Preconditioners for Elliptic Problems by Substructuring I+II. *Math. Comp.*, 47+49, 1986+87.
5. R. Diekmann, B. Meyer, and B. Monien. Parallel Decomposition of Unstructured FEM-Meshes. *Concurrency: Practice & Experience*, 10(1):53–72, 1998.
6. R. Diekmann, F. Schlimbach, and C. Walshaw. Quality Balancing for Parallel Adaptive FEM. To appear in Proc. Irregular '98.
7. C. Farhat, N. Maman, and G. Brown. Mesh Partitioning for Implicit Computations via Domain Decomposition. *Int. J. Num. Meth. Engng.*, 38:989–1000, 1995.
8. C. Farhat, J. Mandel, and F. X. Roux. Optimal convergence properties of the FETI domain decomposition method. *Comp. Meth. Appl. Mech. Engrg.*, 115:367–388, 1994.
9. C. M. Fiduccia and R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, IEEE, Piscataway, NJ, 1982.
10. A. Gupta. Fast and effective algorithms for graph partitioning and sparse matrix reordering. *IBM Journal of Research and Development*, 41(1/2):171–183, 1996.
11. B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. Tech. Rep. SAND 93-1301, Sandia National Labs, Albuquerque, NM, 1993.
12. B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proc. Supercomputing '95*, 1995.
13. G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. TR 95-035, Dept. Comp. Sci., Univ. Minnesota, Minneapolis, MN 55455, 1995.
14. G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. TR 95-064, Dept. Comp. Sci., Univ. Minnesota, Minneapolis, MN 55455, 1995.
15. S. A. Mitchell and S. A. Vasavis. Quality Mesh Generation in Three Dimensions. In *Proc. ACM Conf. Comp Geometry*, pages 212–221, 1992.
16. F. Schlimbach. *Load Balancing Heuristics Optimising Subdomain Shapes for Adaptive Finite Element Simulations*. Diploma Thesis, Dept. Math. Comp. Sci., Univ. Paderborn, 1998.
17. D. Vanderstraeten, C. Farhat, P. S. Chen, R. Keunings, and O. Zone. A Retrofit Based Methodology for the Fast Generation and Optimization of Large-Scale Mesh Partitions: Beyond the Minimum Interface Size Criterion. *Comp. Meth. Appl. Mech. Engrg.*, 133:25–45, 1996.
18. D. Vanderstraeten, R. Keunings, and C. Farhat. Beyond Conventional Mesh Partitioning Algorithms and the Minimum Edge Cut Criterion: Impact on Realistic Applications. In D. Bailey *et al*, editor, *Parallel Processing for Scientific Computing*, pages 611–614. SIAM, 1995.
19. C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. Tech. Rep. 98/IM/35, Univ. Greenwich, London SE18 6PF, UK, March 1998.
20. C. Walshaw, M. Cross, and M. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Par. Dist. Comput.*, 47(2):102–108, 1997.

Visualization of HPF data mappings and of their communication cost

Christian Lefebvre * and Jean-Luc Dekeyser

Laboratoire d'Informatique Fondamentale de Lille Université des Sciences et Technologies de Lille, France

Abstract. HPF-BUILDER graphical environment provides an interactive and visual solution to edit and visualize HPF data mapping directives. It frees the HPF programmers of all the syntactic constraints. General and detailled visualizations give complete information about data distribution along the grids of processors.
Compare several mappings implies to evaluate some statistics about load distribution and communications. This paper introduces an evolution of HPF-BUILDER which produces such statistics, and provides a graphical way to visualize them.

1 Introduction

With the emergence of parallel and massively parallel machines and of clusters of communicating computers, where the memory is physically distributed on a large number of processors, new parallel programming techniques have appeared.

With data parallel model, the program is replicated over all the processors, and vectors or matrices are distributed across them, parallel operations being processed simultaneously by each processor.

Data parallelism is well suited in the domain of scientific computing: algorithms have to manage with large regular data structures (vector, matrix), and the same treatment has to be achieved onto each item of the structures.

The expression of parallelism at the data level has the advantage of maintaining a single control flow. A data parallel algorithm consists of a sequence of elementary instructions applied to scalar or parallel data.

As FORTRAN is the standard language for scientific computing, FORTRAN 90, a data parallel extension, has been developed. It allows programmers to benefit of the data parallel model without having to rewrite their codes in a completly new language.

FORTRAN 90 promotes arrays as global parallel entities. It supports array expressions and proposes restructuring operations onto them (gather, scatter, reductions ...).

The compilation for distributed memory machines relies on the notion of data distribution by the use of mapping directives. These directives specify sets of elementary data that should be allocated on the same processor. HPF (High

* tel.: +33-3 20 43 47 30, fax.: +33-3 20 43 65 66, e-mail:lefebvre@lifl.fr

Performance FORTRAN) [6,7] is an example of this approach and seems to be becoming the most popular language for data parallel scientific programming.

A distributed data parallel algorithm designer usually starts from a FORTRAN 90 code and inserts HPF directives respecting the HPF syntactic rules. The FORTRAN 90 parts express the data parallel algorithm itself and the HPF directives ensure the mapping of the data without semantic contribution. The effects of these directives are essential in balancing between the parallel processing and communications. The programmer has to insert by hand all these mapping directives. Therefore the scientific programmer must learn a third generation dialect of FORTRAN to take advantage of parallel machines.

Furthermore, the programmer have to evaluate himself the accuracy of his mappings.

Like FORTRAN 90, HPF supports regular data structures (multi-dimensional arrays). Furthermore, HPF provides a geometrical support to express the distribution of data among grids of abstract processors.

The expression of parallelism at the data level allows the programmer to have a visual perception of the distribution of data in space (at least for 1, 2 and 3-dimensional arrays and grids). Often programmers use papers and colour pencils to draw and improve their mapping before translating the drawing into HPF directives.

The first goal of the HPF-BUILDER project[5] is to provide a tool to help the programmer at this level. It proposes to replace the paper and pencils by a screen and a mouse. Then it automatically generates the HPF directives from the drawing.

HPF-BUILDER graphical environment frees the programmer from all the syntactic constraints due to the data mapping. Furthermore, it verifies the coherence of mappings and avoids errors (like shifts in indices) during the phase of translation from drawing to HPF code.

HPF-BUILDER respects the hierarchical HPF programming model (arrays aligned together, or with templates, and distribution of them into virtual processor grids). For each level, HPF-BUILDER provides a graphical interactive editor. In a WYSIWYG way each editor is able to generate the appropriate directives according to the data manipulation of the programmer.

2 The hierarchical HPF programming model

A complete use of HPF directives respects a three level hierarchical approach (see figure 1).

For each operation in the code involving data parallel handling, remote accesses imply communications. In order to minimize this overcost, programmers need to specify how each part of arrays has to be placed relatively to other ones. HPF alignment directives implement these specifications.

The second level is the **template**, with which arrays are aligned.

The third level, the **processors**, defines multidimensionnal grids of abstract processors into which the templates are distributed.

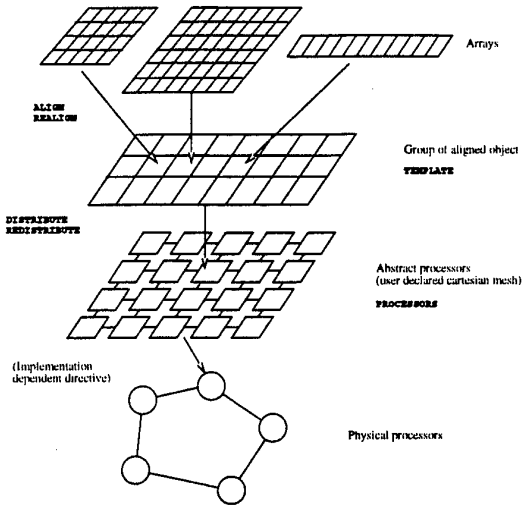


Fig. 1. Hierarchical HPF programming model

It is the compiler charge (eventually helped by compiler specific directives) to decide which physical computation node will correspond to a given processors item.

This construction ensures a progressive refinement of the data mapping on the physical processors. In this way the programmer is able to group in the same **`template`** all the arrays that interact. It avoids a number of levels due to array with array alignments.

This three level hierarchy is the more complete use of HPF directives. HPF directives as alignment between arrays, or distribution of arrays directly onto processors can bypass the template definition.

All of these directives are supported by HPF-BUILDER.

3 Graphical interfaces and HPF

To replace papers and pencils, a graphical editor has to provide several features:

- a display of the source code and/or a summary of its syntactic architecture (modules, subroutines, array declarations ...),
- a global view of the hierarchical HPF construction,
- a general visualization of each directive,
- a detailed visualization, with the possibility of tracing the mapping of each item of objects,
- a WYSIWYG editing of mapping HPF directives,
- a graphical tool to visualize and modify existing directives,

- the automatic generation of the HPF directives,
- the interpretation of directives to help the programmer in evaluating the quality of mappings (array load balancing on virtual processors, evaluation of the redistribution and realignment cost in term of communications ...).

A few visual tools already exist to help the HPF programmers. Some of them are limited to visualization, they do not help with directive editing.
It's the case of Annai/DDV[4], developped at CSSE/NEC, which allows to visualize distributed data. It is integrated into a debugger, which implies to execute the code. Its goal is more to look at data values than at their mappings.

Often, such tools need to execute the code to process effectively the data mapping.
For example, DAQV[11] or Prism[12], allows to trace communications at runtime, and to generate accurate statistics, but the user has to execute heavy codes with large amount of data, for each mapping he wants to test.

We prefer to evaluate the mapping during the editing phase.

Another limitation we want to avoid is to be dedicated to a particular compiler, as GDDT[8] does into the Vienna Fortran environment. It is well suited to visualize mapping onto physical processors, and to generate real communication statistics, but it limits the user to a particular kind of targets.

Lastly, our goal is to work only on mapping, and not on the code production. We don't wish a complete visual programming solution, like HELP-DRAW [1], where the user programs everything from scratch to get automaticaly a HPF code.

4 HPF-Builder

HPF-BUILDER is built according to the HPF programming model. For each level of the data mapping hierarchical representation a graphical editor is defined to visualize and modify in a WYSIWYG way the corresponding HPF directives. This procures a step by step transformation from a FORTRAN 90 code towards an HPF version. The data parallel algorithm expressed in FORTRAN 90 is never modified. The HPF transformations concern exclusively the data mapping.

For each level, we present the corresponding editor with its main specifications.

4.1 Example program

This matrix/vector product example is used along this paper to describe the step by step transformation:

```
integer :: NCol, NLine
parameter (NCol=20)
 ...
subroutine MV(M,V,R, NLine)
  integer :: NLine
  real, dimension(NLine,NCol), intent(in):: M
  real, dimension(NCol), intent(in)      :: V
  real, dimension(NLine), intent(out)    :: R

  R(1:NLine)= 0.0
  do k = 1,NCol
    forall(i= 1:NLine)
      R(i)=R(i) + V(k)*M(i,k)
    end forall
  end do
end subroutine
```

To generate an efficient `V(k)*M(i,k)` product, we must align together the parts of `M` and `V` that interact. That means, each item `V(k)` must be aligned with `M(i,k)` for each `i`. So, `V` items must be replicated along columns of `M`.

In the same way, the sum implies to replicate `R` along the lines of `M`.

The processor grid used is a 2D mesh. The matrix `M` is arbitrarily distributed *Cyclic*, *Block* on this grid: Implicit communications will be produced by the compiler to update the `R` values replicated on the second dimension.

Finally, we obtain this HPF code:

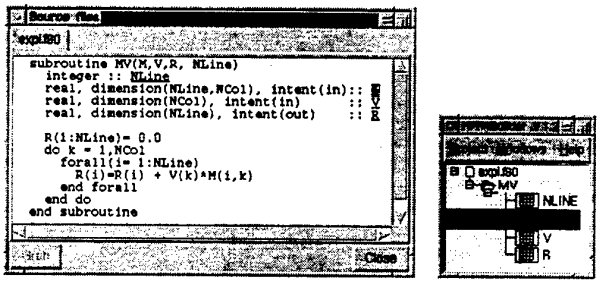
```
subroutine MV(M,V,R, NLine)
  integer :: NLine
  real, dimension(NLine,NCol), intent(in):: M
  real, dimension(NCol), intent(in)      :: V
  real, dimension(NLine), intent(out)    :: R
!HPF$ PROCESSORS MyProc(NUMBER_OF_PROCESSORS(1), &
!HPF$   NUMBER_OF_PROCESSORS(2))
!HPF$ DISTRIBUTE M(Cyclic, Block) ONTO MyProc

!HPF$ ALIGN V(:) WITH M(*,:)
!HPF$ ALIGN R(:) WITH M(:,*)

  R(1:NLine)= 0.0
  do k = 1,NCol
    forall(i= 1:NLine)
      R(i)=R(i) + V(k)*M(i,k)
    end forall
  end do
end subroutine
```

4.2 Source editing and parsing

The first phase of HPF-BUILDER concerns the analysis of the source file. A modified version of Cocktail HPF parser[9] is used. It supports FORTRAN 90 and almost all the data mapping directives of HPF. From both FORTRAN 90 and HPF code, HPF-BUILDER is able to build the syntactic tree of array and HPF directive declarations and the hierarchical skeleton of the program.



(a) Source (b) Tree

Fig. 2. Source and syntactic tree at begining

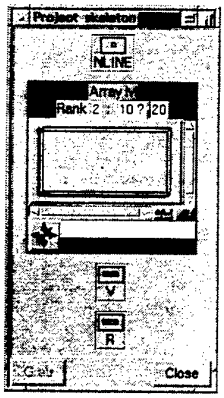


Fig. 3. skeleton

At this step, HPF-BUILDER presents:

- A full screen editor opened in the *source* window (2(a)). The content of this editor is updated with any interactive graphical manipulation. Underlined pieces of text indicate selectable objects.
- The syntactic tree summary, in the *tree* window (2(b)).
- array and variable declarations, represented by icons in the *skeleton* window (3).

The main window for visualization and edition is the skeleton. The other windows add informations in the syntactic structure of the program, and reflect automaticaly any modification made by the user.

Clicking an entry anywhere selects it in the three windows. Several different objects can be opened at a time. This lets the user see details about several objects at a time.

In the skeleton window, selection changes the icon in a subwindow (array `M` in figure 3) which presents some

details about the object: Its name, rank and size, and a wire representation in which directives will be displayed.

We can see in the subwindow of array **M** that an interprocedural analysis is performed to find the value of the constant **NCol**. On the other hand, as **NLine** as an unknown value at parsing time, a default value of 10 (marked by a "?") is taken (the user can specify other values to test different cases).

4.3 Processors and distributions

In the skeleton, clicking on an subwindow opens a menu from which new directives can be created. A drag'n drop to another icon specifies the second entry to set an alignment or distribution. A creation menu allows to create new templates and processors.

HPF imposes some restrictions about alignments and distributions. For example, an already distributed object can't be realigned.

To avoid the user to create such an invalid directive, the creation menu is adapted for each object. For a distributed object, the "realign" entry is disabled.

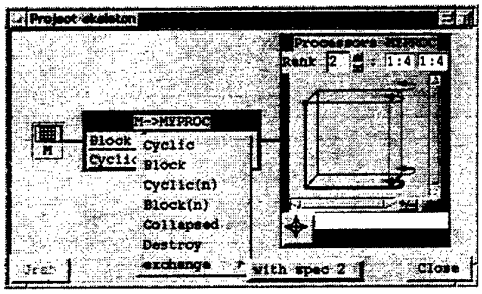


Fig. 4. Distribution specification

Furthermore, HPF imposes that processors size matches the number of physical processors. The intrisic **number_of_processors** returns this number.

As HPF-BUILDER is not dedicated to a given target computer, a configuration option defines this value. Therefore, the parser is able to evaluate this function call as a constant value.

Thus, HPF-BUILDER let the user create graphically the global structure of its HPF skeleton, and verifies their coherency.

Once the two dimensionnal processor mesh **MyProc** is created, a distribution directive can be setted between **M** and **MyProc** (figure 4).

Into the editing window associated with this directive, a block, cyclic, or collapsed distribution can be specified for each dimension of the distributee.

Then, into the wire representation of the processors, the projection of **M** is drawn. It shows cyclic distribution by an arrow ended by a small loop. A dashed line is added for cyclic(k) and block(k) specifications.

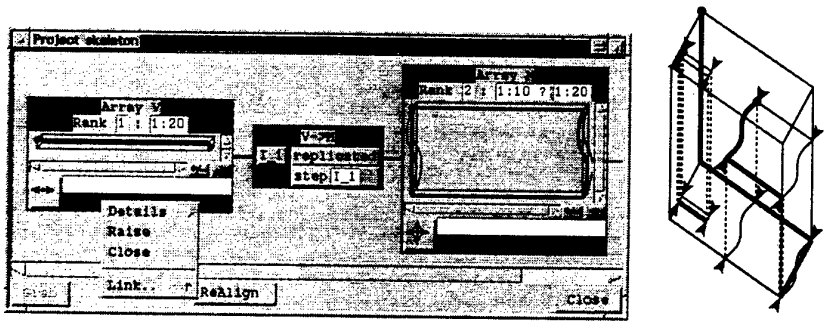
Beside each distribution specification, a formula describe in detail the distribution. In the example the expression $(2 \times 3) + (2 \times 2)$ specifies that the 2 first lines contains 3 lines of the template, and the 2 last contains 2 lines. This describes a cyclic distribution which does 2 loops and a half.

On the other dimension, the block distribution cuts the template in 4 blocks of 5 columns.

These two parts of the visualization let the user see a global draft of its distribution, and a more detailled aspect of the processor load.

4.4 Alignments

As for the distribution, a drag'n drop between V and M let specify an alignement directive (see the creation menu in figure 5(a)).



(a) 2D

(b) 3D

Fig. 5. alignments visualization and edition

In the same way, R will be replicated along the lines of M.

By default, direct alignment is chosen, after what selecting the alignment icon allows to change its specifications: Here, we modify the direction where V must be aligned. and then we apply the replicate action in the other direction.

These specifications are displayed in the alignment selection (central selection of figure 5(a))

Now, in the wire representation of the array M, the image of V is drawn. It follows the columns of M and its replication is shown by a curve along the lines (right selection in figure 5(a)).

Collapsing is shown by a double arrow, and stepped alignment by a dashed line (figure 5(b)).

This wire representation let the user see globally where its data are aligned. Replications and steps appear clearly, following the geometrical aspect of HPF.

Visualization subwindows can be resized and zoomed in or out, therefore, the size of the objects is not a limit.

4.5 Detailled visualization

Once all of the directives are defined, one wants to know if data are really mapped as he was thinking. For that, HPF-BUILDER uses a zoom effect to watch exactly what parts of data are mapped onto a given processor.

The small compas under each edition subwindow, let the user move a cursor along the objects. Its projection into and from its upper and lower subwindows is drawn. Therefore, the user can see where a given item is projected and what parts of other objects are projected into it. Beside this compas, a label indicates exact coordinates of the cursor and of its projection. Thus, when data are very large, the draft gives a graphical information, and this label gives numeric values.

In figure 6, `V(10)` (the cube in the upper left selection) is mapped onto all the 10th column of `M` (bar in the center selection), itself distributed into the second column of `MYPROC` (the column of the right hand selection).

To see the processors load, the zoom effect can be used in the other sense: We can see that `P(1,2)` (upper left cube in right hand selection) contains lines 1 to 3 and columns 2, 6 ... 18 of `M` (bars in the central selection), items 2, 6, 10 ... 18 of `V`, and items 1 to 3 of `R`

So, when $i = 1$ and $k = 10$, the instruction `R(1)=R(1)+V(10)*M(1,10)` will find all its operands onto the same processor `MYPROC(1,2)`.

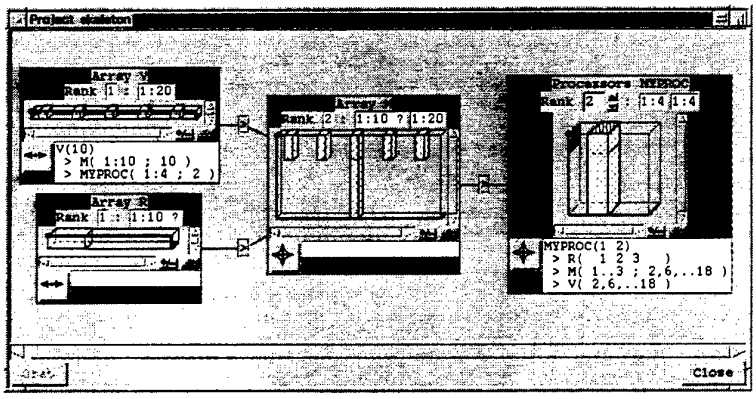


Fig. 6. Detailled visualization

Now, the user can change distribution specifications. HPF-BUILDER automatically update all the visual perception of this code. Programmer can concludes distribution don't change the locality of interacting items of `M`, `V` and `R`.

After that, other experimentations using realignment directives could produce less implicit communications due to replications.

This example shows how, even without execution and without knowing all variables values and physical distributions, HPF-BUILDER can help the user in choosing *a priori* a better data mapping.

5 Communication visual predictions

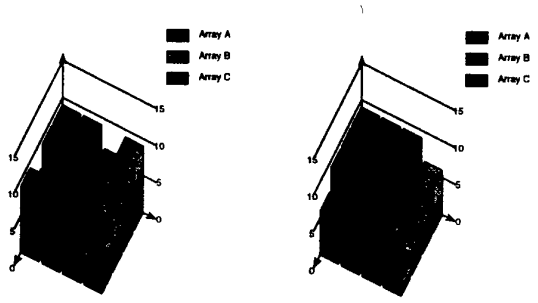
Once the programmer created its mapping, efficiency have to be demonstrated. This is achieved by using tools to visualize data distribution load and communication cost predictions.

The following instruction, with cyclic or block distribution, is taken as an example:

```
forall(i=2:size(A,1), j=2:size(a,2))
  A(i,j)= B(i)+A(i-1,j-1)
```

These predictions can be classified in several parts:

- The amount of data stored on each virtual processor. In order to know the efficiency of data distribution, a simple histogram with a bar per processor is used (figure 7). Clicking one of these bars can display a list with details of data stored on it (like in the zoom effect described in 4.5).

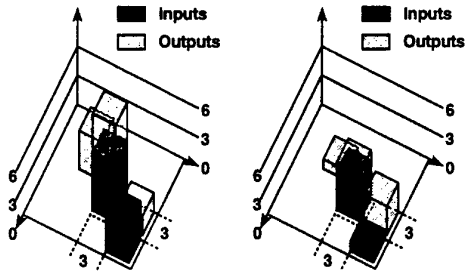


Cyclic distribution *Block* distribution

Fig. 7. load histogram example

- For a given instruction, the number of operations needed onto each virtual processor. This is equivalent to the number of LHS data onto each processor (assuming the owner computing rule). Thus, the visualization is the same.
- The number of data movements implied by an instruction, for a given processor, in input (respectively output). From the instruction to be executed onto a given processor, we can deduce which data have to be read (written). Then, we can obtain histograms showing where these data come from (go to).

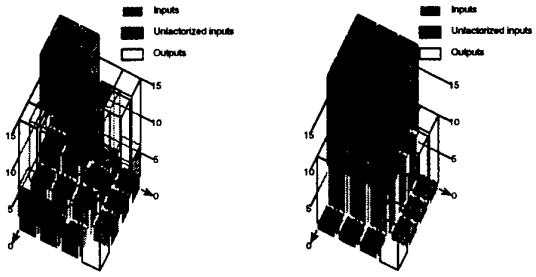
Figure 8 shows data movements around processors $(3,3)$. Movements come from white blocks and go to black ones. In the example, the processor reads data from $(2,2)$ and $(3,4)$ and sends other ones to $(4,4)$.



Cyclic distribution *Block* distribution

Fig. 8. data movements from and to one processor

- The total number of movements implied by an instruction. This means to iterate the previous results in one graph for all the processors (figure 9). Here again, the user can click a bar to see details about data origins and destinations.



Cyclic distribution *Block* distribution

Fig. 9. global data movements

The same visualization tool can be extended for a loop nest, or a block of instructions. The computation can be iterated for each instruction of a block, and then iterated for each loop of the nest. The CPU time could become huge according to the number of abstract processors.

Implementation

All the informations needed to evaluate the distribution load are the same as the ones needed to accurately visualize the mappings. During the visualization, the user can specify default values for variables and runtime data. Thus, the load distribution is calculable at visualization time, without execution or even compilation of the program.

Evaluation of communication costs uses the same methods than HPF compilers: for each processor, we have to identify which data has to been sent to (received from) every other processors.

A solution currently studied in [2] consists in identify how communications are computed into the code generated by a compiler as Adaptor.

For large data or grids of processors, the calculation time could become huge (too huge for interactive evaluation).

Assuming the owner compute rule, any given instruction implies data movements for each remote access. Their number can be interpreted as an enumeration of common points between two sets of positions. These calculations may not need to enumerate all data movements. It is possible to eval them with symbolic methods, to obtain formulas depending of variables and runtime data. While the user sets this values, it is possible to visualize data movements without having to compute everything from scratch. Furthermore, this method is independent of the size of data. First results were obtained in [3] for a global communication cost evaluation.

6 Conclusion

Data parallel programming is still a difficult art. Scientific programmers have expended a lot of efforts in learning vector programming. Now they have to learn a third generation dialect of FORTRAN to map their data onto distributed memory machines. To succeed in this task, they need some tools to help them to manage their data distributions. HPF-BUILDER is a first step in Computer Assisted High Performance Programming. The automatic insertion of HPF directives in a FORTRAN 90 code frees the programmer from the new syntactic constraints.

Optimization of both load distribution and communication overhead is a key element for parallel programming. The extension of HPF-BUILDER presented in this paper gives more informations to guide the programmer during this development phase in HPF.

Visualization of distribution and prediction of communication costs lead the user to refine his HPF directives during the editing phase.

HPF-BUILDER is a good plateform into which such tools can be plugged in.

The user still decides if a solution is better than another one. Future works should include more complex evaluation methods to guide the user to better mappings.

More target specific informations like computation/communication overlapping, netword capabilities, cache effects ... may be taken into account in these methods.

The last version of HPF-BUILDER is always available on the Web[10]. All are welcomed to use it and report all comments on improving the functionalities of this tool.

References

1. A. Benalia, J.-L. Dekeyser, and P. Marquet. HelpDraw graphical environment: A step beyond data parallel programming languages. In *Fifth Int'l Conf. on Human-Computer Interaction*, pages 591–596, Orlando, FL, Aug. 1993. Elsevier Science Publishers.
2. P. Boulet, J.-L. Dekeyser, C. Lefebvre, and D. Ruckebusch. Communication pre-visualization. In *HPF Second User Group Meeting*, Porto, Portugal, June 1998.
3. P. Boulet and X. Redon. communication pre-evaluation in HPF. In *EuroPar'98*, SouthHampton, UK, Sept. 1998.
4. K. M. Decker and B. J. Wylie. Software tools for scalable multi-level application engineering. In *Workshop on Environments and Tools For Parallel Scientific Computing*, Aug. 1996.
5. J.-L. Dekeyser and C. Lefebvre. Hpf-builder: A visual environment to transform fortran 90 codes to hpf. *International Journal of Supercomputing Applications and High Performance Computing*, 11(2):95–102, Summer 1997.
6. H. P. F. Forum. High Performance Fortran language specification, version 1.0. Rice University, Houston, TX, May 1993.
7. H. P. F. Forum. High Performance Fortran language specification, version 2.0. Rice University, Houston, TX, Jan. 1997.
8. R. K. S. Grabner and J. Volkert. Graphical support for data distribution in spmd parallelization environments. In *Proc. IEEE 2nd International Conference on ALgorithms and Parallel Processing, Singapore*, 1996.
9. http://www.gmd.de/SCAI/lab/adaptor/cocktail.html. Cocktail compiler toolbox.
10. http://www.lifl.fr/west/hpf-builder. the hpf-builder web page.
11. S. T.Hackstadt and A. D.Malony. Distributed array query and visualization for high performance fortran. In *Proc of Euro-Par'96. Lyon. France. August 1996*, Aout 1996.
12. Thinking Machines Corporation. *Prism User's Guide for the CM-5*, Dec. 1991.

Parallel and Distributed Computing in Education

Peter H. Welch

Computing Laboratory, University of Kent at Canterbury, CT2 7NF.
P.H.Welch@ukc.ac.uk

Abstract. The natural world is certainly not organised through a central thread of control. Things happen as the result of the actions and interactions of unimaginably large numbers of independent agents, operating at all levels of scale from nuclear to astronomic. Computer systems aiming to be of real use in this real world need to model, at the appropriate level of abstraction, that part of it for which it is to be of service. If that modelling can reflect the natural concurrency in the system, it ought to be much simpler

Yet, traditionally, concurrent programming is considered to be an advanced and difficult topic – certainly much harder than serial computing which, therefore, needs to be mastered first. But this tradition is wrong.

This talk presents an intuitive, sound and practical model of parallel computing that can be mastered by undergraduate students in the first year of a computing (major) degree. It is based upon Hoare's mathematical theory of *Communicating Sequential Processes* (CSP), but does not require mathematical maturity from the students – that maturity is pre-engineered in the model. Fluency can be quickly developed in both message-passing and shared-memory concurrency, whilst learning to cope with key issues such as race hazards, deadlock, livelock, process starvation and the efficient use of resources. Practical work can be hosted on commodity PCs or UNIX workstations using either Java or the occam multiprocessing language. Armed with this maturity, students are well-prepared for coping with real problems on real parallel architectures that have, possibly, less robust mathematical foundations.

1 Introduction

At Kent, we have been teaching parallel computing at the undergraduate level for the past ten years. Originally, this was presented to first-year students *before* they became too set in the ways of serial logic. When this course was expanded into a full unit (about 30 hours of teaching), timetable pressure moved it into the second year. Either way, the material is easy to absorb and, after only a few (around 5) hours of teaching, students have no difficulty in grappling with the interactions of 25 (say) threads of control, appreciating and eliminating race hazards and deadlock

Parallel computing is still an immature discipline with many conflicting cultures. Our approach to educating people into successful exploitation of parallel mechanisms is based upon focusing on parallelism as a powerful tool for *simplifying* the description of systems, rather than simply as a means for improving their performance. We never start with an existing serial algorithm and say: 'OK, let's parallelise that!'. And we work solely with a model of concurrency that has a semantics that is *compositional* – a fancy word for WYSIWYG – since, without that property, combinatorial explosions of complexity always get us as soon as we step away from simple examples. In our view, this rules out low-level concurrency mechanisms, such as spin-locks, mutexes and semaphores, as well as some of the higher-level ones (like monitors).

Communicating Sequential Processes (CSP)[1–3] is a mathematical theory for specifying and verifying complex patterns of behaviour arising from interactions between concurrent objects. Developed by Tony Hoare in the light of earlier work on monitors, CSP has a compositional semantics that greatly simplifies the design and engineering of such systems – so much so, that parallel design often becomes easier to manage than its serial counterpart. CSP primitives have also proven to be extremely lightweight, with overheads in the order of a few hundred nanoseconds for channel synchronisation (including context-switch) on current microprocessors [4, 5].

Recently, the CSP model has been introduced into the Java programming language [6–10]. Implemented as a library of packages [11, 12], JavaPP[10] enables multithreaded systems to be designed, implemented and reasoned about entirely in terms of CSP synchronisation primitives (channels, events, etc.) and constructors (parallel, choice, etc.). This allows 20 years of theory, design patterns (with formally proven good properties – such as the absence of race hazards, deadlock, livelock and thread starvation), tools supporting those design patterns, education and experience to be deployed in support of Java-based multithreaded applications.

2 Processes, Channels and Message Passing

This section describes a simple and structured multiprocessing model derived from CSP. It is easy to teach and can describe arbitrarily complex systems. No formal mathematics need be presented – we rely on an intuitive understanding of how the world works.

2.1 Processes

A *process* is a component that encapsulates some data structures and algorithms for manipulating that data. Both its data and algorithms are private. The outside world can neither see that data nor execute those algorithms. Each process is alive, executing its own algorithms on its own data. Because those algorithms are executed by the component in its own thread (or threads) of control, they express

the behaviour of the component from its own point of view[1]. This considerably simplifies that expression.

A *sequential process* is simply a process whose algorithms execute in a single thread of control. A *network* is a collection of processes (and is, itself, a process). Note that recursive hierarchies of structure are part of this model: a network is a collection of processes, each of which may be a sub-network or a sequential process.

But how do the processes within a network interact to achieve the behaviour required from the network? They can't see each other's data nor execute each other's algorithms – at least, not if they abide by the rules.

2.2 Synchronising Channels

The simplest form of interaction is synchronised message-passing along channels. The simplest form of channel is zero-buffered and point-to-point. Such channels correspond very closely to our intuitive understanding of a wire connecting two (hardware) components.

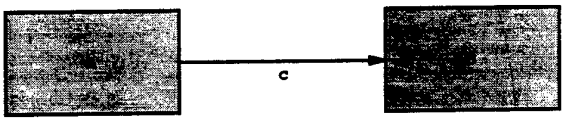


Fig. 1. A simple network

In Figure 1, `A` and `B` are processes and `c` is a channel connecting them. A wire has no capacity to hold data and is only a medium for transmission. To avoid undetected loss of data, channel communication is synchronised. This means that if `A` transmits before `B` is ready to receive, then `A` will block. Similarly, if `B` tries to receive before `A` transmits, `B` will block. When both are ready, a data packet is transferred – directly from the state space of `A` into the state space of `B`. We have a synchronised distributed assignment.

2.3 Legoland

Much can be done just with this simple model – from the design of self-timed digital logic (no global clock) through to the wide range of industrial multiprocessor embedded control for which occam[13–16] was orignally designed.

Here are some simple examples to build up fluency. First we introduce some elementary components from our 'teaching' catalogue – see Figure 2. All processes are cyclic and all transmit and receive just numbers. The `Id` process cycles

[1] This is in contrast with simple 'objects' and their 'methods'. A method body normally executes in the thread of control of the invoking object. Consequently, object behaviour is expressed from the point of view of its environment rather than the object itself. This is a slightly confusing property of traditional 'object-oriented' programming.

through waiting for a number to arrive and, then, sending it on. Although inserting an `Id` process in a wire will clearly not affect the data flowing through it, it *does* make a difference. A bare wire has no buffering capacity. A wire containing an `Id` process gives us a one-place FIFO. Connect 20 in series and we get a 20-place FIFO – sophisticated function from a trivial design.

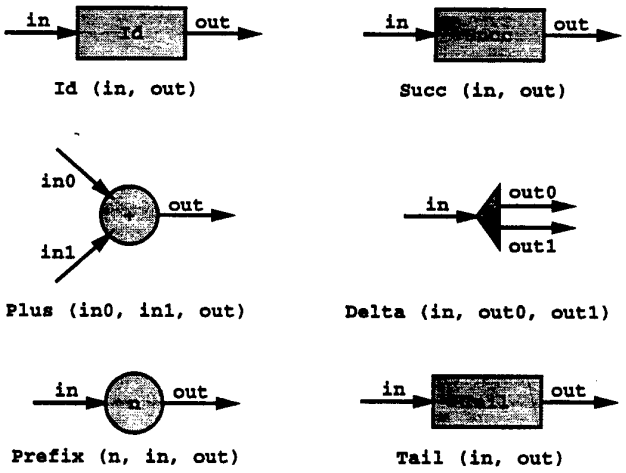


Fig. 2. Extract from a component catalogue

`Succ` is like `Id`, but increments each number as it flows through. The `Plus` component waits until a number arrives on each input line (accepting their arrival in either order) and outputs their sum. `Delta` waits for a number to arrive and, then, broadcasts it in parallel on its two output lines – both those outputs must complete (in either order) before it cycles round to accept further input. `Prefix` first outputs the number stamped on it and then behaves like `Id`. `Tail` swallows its first input without passing it on and then, also, behaves like `Id`. `Prefix` and `Tail` are so named because they perform, respectively, prefixing and tail operations on the streams of data flowing through them.

It's essential to provide a practical environment in which students can develop executable versions of these components and play with them (by plugging them together and seeing what happens). This is easy to do in occam and now, with the JCSP library[11], in Java. Appendices A and B give some of the details. Here we only give some CSP pseudo-code for our catalogue (because that's shorter than the real code):

```
Id (in, out) = in ? x --> out ! x --> Id (in, out)

Succ (in, out) = in ? x --> out ! (x+1) --> Succ (in, out)
```

```
Plus (in0, in1, out)
  = ((in0 ? x0 --> SKIP) || (in1 ? x1 --> SKIP));
    out ! (x0 + x1) --> Plus (in0, in1, out)

Delta (in, out0, out1)
  = in ? x --> ((out0 ! x --> SKIP) || (out1 ! x --> SKIP));
    Delta (in, out0, out1)

Prefix (n, in, out) = out ! n --> Id (in, out)

Tail (in, out) = in ? x --> Id (in, out)
```

[Notes: 'free' variables used in these pseudo-codes are assumed to be locally declared and hidden from outside view. All these components are *sequential* processes. The process (`in ? x --> P (...)`) means: "wait until you can engage in the input event (`in ? x`) and, then, become the process `P (...)`". The input operator (`?`) and output operator (`!`) bind more tightly than the `-->`.]

2.4 Plug and Play

Plugging these components together and reasoning about the resulting behaviour is easy. Thanks to the rules on process privacy[2], race hazards leading to unpredictable internal state do not arise. Thanks to the rules on channel synchronisation, data loss or corruption during communication cannot occur[3]. What makes the reasoning simple is that the parallel constructor and channel primitives are deterministic. Non-determinism has to be explicitly designed into a process and coded – it can't sneak in by accident!

Figure 3 shows a simple example of reasoning about network composition. Connect a `Prefix` and a `Tail` and we get two `Id`s:

```
(Prefix (in, c) || Tail (c, out))  =  (Id (in, c) || Id (c, out))
```

Equivalence means that no environment (i.e. external network in which they are placed) can tell them apart. In this case, both circuit fragments implement a 2-place FIFO. The only place where anything different happens is on the internal wire and that's undetectable from outside. The formal proof is a one-liner from the definition of the parallel (`||`), communications (`!`, `?`) and *and-then-becomes* (`-->`) operators in CSP. But the good thing about CSP is that the mathematics engineered into its design and semantics cleanly reflects an intuitive human feel for the model. We can see the equivalence at a glance and this quickly builds confidence both for us and our students.

[2] No external access to internal data. No external execution of internal algorithms (methods).

[3] Unreliable communications over a distributed network can be accommodated in this model – the unreliable network being another active process (or set of processes) that happens not to guarantee to pass things through correctly.

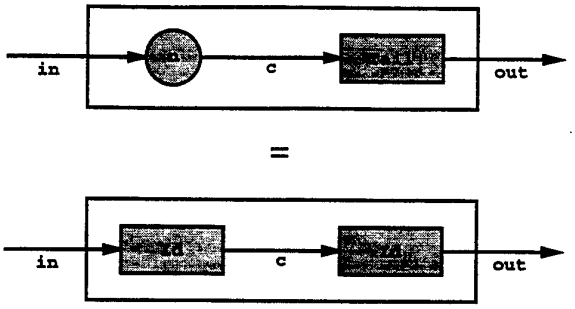
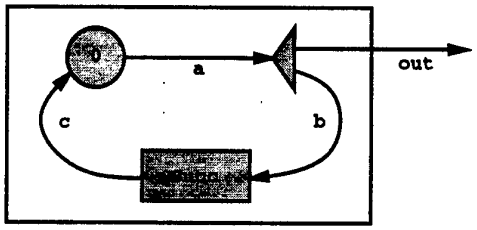
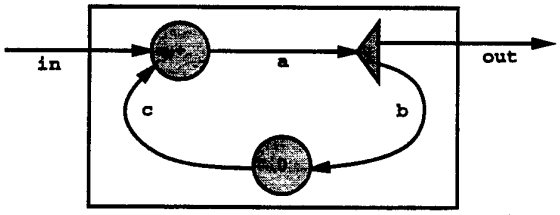


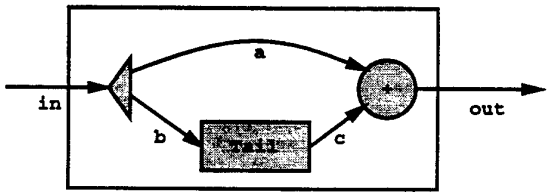
Fig. 3. A simple equivalence



Numbers (out)



Integrate (in, out)



Pairs (in, out)

Fig. 4. Some more interesting circuits

Figure 4 shows some more interesting circuits with the first two incorporating feedback. What do they do? Ask the students! Here are some CSP pseudo-codes for these circuits:

```
Numbers (out)
  = Prefix (0, c, a) || Delta (a, out, b) || Succ (b, c)

Integrate (in, out)
  = Plus (in, c, a) || Delta (a, out, b) || Prefix (0, b, c)

Pairs (in, out)
  = Delta (in, a, b) || Tail (b, c) || Plus (a, c, out)
```

Again, our rule for these pseudo-codes means that `a`, `b` and `c` are locally declared channels (hidden, in the CSP sense, from the outside world). Appendices A and B list occam and Java executables – notice how closely they reflect the CSP.

Back to what these circuits do: `Numbers` generates the sequence of natural numbers, `Integrate` computes running sums of its inputs and `Pairs` outputs the sum of its last two inputs. If we wish to be more formal, let `c<i>` represent the `i`'th element that passes through channel `c` – i.e. the first element through is `c<1>`. Then, for any `i >= 1`:

```
numbers:    out<i> = i - 1
integrate:  out<i> = Sum {in<j> | j = 1..i}
pairs:      out<i> = in<i> + in<i + 1>
```

Be careful that the above only details *part* of the specification of these circuits: how the values in their output stream(s) relate to the values in their input stream(s). We also have to be aware of how flexible they are in synchronising with their environments, as they generate and consume those streams. The base level components `Id`, `Succ`, `Plus` and `Delta` each demand one input (or pair of inputs) before generating one output (or pair of outputs). `Tail` demands two inputs before its first output, but thereafter gives one output for each input. This effect carries over into `Pairs`. `Integrate` adds 2-place buffering between its input and output channels (ignoring the transformation in the actual values passed). `Numbers` will always deliver to anything trying to take input from it.

If necessary, we can make these synchronisation properties mathematically precise. That is, after all, one of the reasons for which CSP was designed.

2.5 Deadlock – First Contact

Consider the circuit in Figure 5. A simple stream analysis would indicate that:

```
Pairs2:    a<i> = in<i>
Pairs2:    b<i> = in<i>
Pairs2:    c<i> = b<i + 1> = in<i + 1>
Pairs2:    d<i> = c<i + 1> = in<i + 2>
Pairs2:  out<i> = a<i> + d<i> = in<i> + in<i + 2>
```

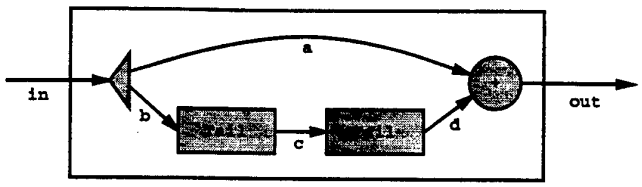


Fig. 5. A dangerous circuit

But this analysis only shows what would be generated *if* anything were generated. In this case, nothing is generated since the system deadlocks. The two `Tail` processes demand three items from `Delta` before delivering anything to `Plus`. But `Delta` can't deliver a third item to the `Tails` until it's got rid of its second item to `Plus`. But `Plus` won't accept a second item from `Delta` until it's had its first item from the `Tails`. Deadlock!

In this case, deadlock can be designed out by inserting an `Id` process on the upper (a) channel. `Id` processes (and FIFOs in general) have no impact on stream contents analysis but, by allowing a more decoupled synchronisation, *can* impact on whether streams actually flow. Beware, though, that adding buffering to channels is not a general cure for deadlock.

So, there are always two questions to answer: what data flows through the channels, assuming data does flow, and are the circuits deadlock-free? Deadlock is a monster that must – and can – be vanquished. In CSP, deadlock only occurs from a cycle of committed attempts to communicate (input or output): each process in the cycle refusing its predecessor's call as it tries to contact its successor. Deadlock potential is very visible – we even have a deadlock primitive (`STOP`) to represent it, on the grounds that it is a good idea to know your enemy!

In practice, there now exist a wealth of design rules that provide formally proven guarantees of deadlock freedom[17–22]. Design tools supporting these rules – both constructive and analytical – have been researched[23, 24]. Deadlock, together with related problems such as livelock and starvation, need threaten us no longer – even in the most complex of parallel system.

2.6 Structured Plug and Play

Consider the circuits of Figure 6. They are similar to the previous circuits, but contain components other than those from our base catalogue – they use components we have just constructed. Here is the CSP:

```
Fibonacci (out)
  = prefix (0, d, a) || prefix (1, a, b) ||
    delta (b, out, c) || pairs (c, d)

Squares (out)
  = Numbers (a) || Integrate (a, b) || Pairs (b, out)
```

```
Demo (out)
  = Numbers (a) || Fibonacci (b) || Squares (c) ||
    Tabulate3 (a, b, c, out)
```

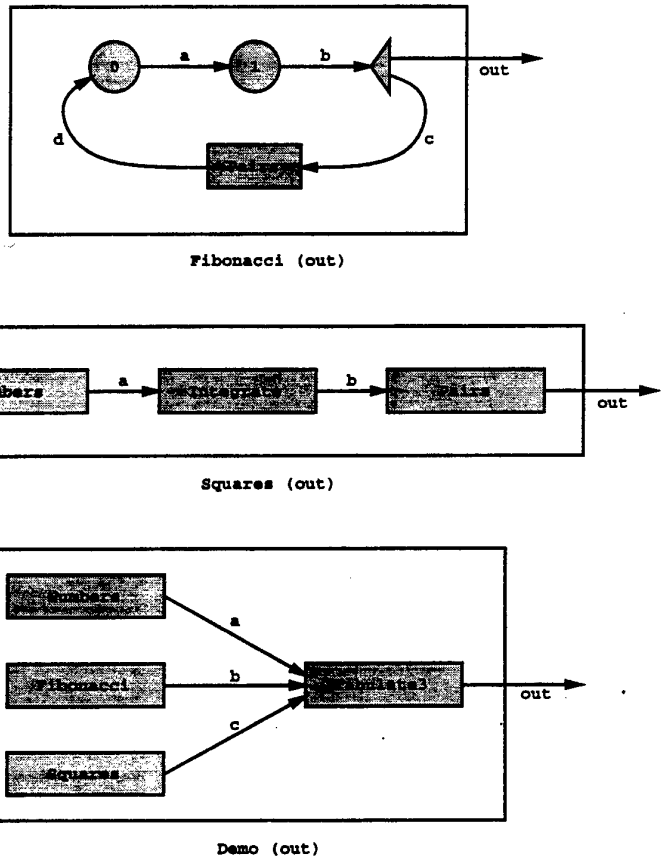


Fig. 6. Circuits of circuits

One of the powers of CSP is that its semantics obey simple composition rules. To understand the behaviour implemented by a network, we only need to know the behaviour of its nodes – not their implementations.

For example, `Fibonacci` is a feedback loop of four components. At this level, we can remain happily ignorant of the fact that its `Pairs` node contains another three. We only need to know that it requires two numbers before it outputs anything and that, thereafter, it outputs once for every input. The two `Prefixes` initially inject two numbers (`0` and `1`) into the circuit. Both go into `Pairs`,

but only one (their sum) emerges. After this, the feedback loop just contains a single circulating packet of information (successive elements of the `Fibonacci` sequence). The `Delta` process taps this circuit to provide external output.

`Squares` is a simple pipeline of three components. It's best not to think of the nine processes actually involved. Clearly, for `i >= 1`:

```
Squares:   a<i> = i - 1
Squares:   b<i> = Sum {j - 1 | j = 1..i} = Sum {j | j = 0..(i - 1)}
Squares: out<i> = Sum {j | j = 0..(i - 1)} + Sum {j | j = 0..i} = i * i
```

So, `Squares` outputs the increasing sequence of squared natural numbers. It doesn't deadlock because `Integrate` and `Pairs` only add buffering properties and it's safe to connect buffers in series.

`Tabulate3` is from our base catalogue. Like the others, it is cyclic. In each cycle, it inputs in parallel one number from each of its three input channels and, then, generates a line of text on its output channel consisting of a tabulated (15-wide, in this example) decimal representation of those numbers.

```
Tabulate3 (in0, in1, in2, out)
  = ((in0 ? x0 - SKIP) || (in1 ? x1 - SKIP) || (in2 ? x2 - SKIP));
    print (x0, 15, out); print (x1, 15, out); println (x2, 15, out);
    Tabulate3 (in0, in1, in2, out)
```

Connecting the output channel from `Demo` to a text window displays three columns of numbers: the natural numbers, the Fibonacci sequence and perfect squares.

It's easy to understand all this – thanks to the structuring. In fact, Demo consists of 27 threads of control, 19 of them permanent with the other 8 being repeatedly created and destroyed by the low-level parallel inputs and outputs in the `Delta`, `Plus` and `Tabulate3` components. If we tried to understand it on those terms, however, we would get nowhere.

Please note that we are not advocating designing at such a fine level of granularity as normal practice! These are only exercises and demonstrations to build up fluency and confidence in concurrent logic. Having said that, the process management overheads for the occam `Demo` executables are only around 30 microseconds per output line of text (i.e. too low to see) and three milliseconds for the Java (still too low to see). And, of course, if we are using these techniques for designing real hardware[25], we will be working at much finer levels of granularity than this.

2.7 Coping with the Real World – Making Choices

The model we have considered so far – parallel processes communicating through dedicated (point-to-point) channels – is *deterministic*. If we input the same data in repeated runs, we will always receive the same results. This is true regardless of how the processes are scheduled or distributed. This provides a very stable base from which to explore the real world, which doesn't always behave like this.

Any machine with externally operatable controls that influence its internal operation, but whose internal operations will continue to run in the absence of that external control, is not deterministic in the above sense. The scheduling of that external control will make a difference. Consider a car and its driver heading for a brick wall. Depending on when the driver applies the brakes, they will end up in very different states!

CSP provides operators for internal and external choice. An external choice is when a process waits for its environment to engage in one of several events – what happens next is something the environment can determine (e.g. a driver can press the accelerator or brake pedal to make the car go faster or slower). An internal choice is when a process changes state for reasons its environment cannot determine (e.g. a self-clocked timeout or the car runs out of petrol). Note that for the combined (parallel) system of car-and-driver, the accelerating and braking become internal choices so far as the rest of the world is concerned.

occam provides a constructor (`ALT`) that lets a process wait for one of many events. These events are restricted to channel input, timeouts and `SKIP` (a *null* event that has always happened). We can also set pre-conditions – run-time tests on internal state – that mask whether a listed event should be included in any particular execution of the `ALT`. This allows very flexible internal choice within a component as to whether it is prepared to accept an external communication[4]. The JavaPP libraries provide an exact analogue (`Alternative.select`) for these choice mechanisms.

If several events are pending at an `ALT`, an internal choice is normally made between them. However, occam allows a `PRI ALT` which resolves the choice between pending events in order of their listing. This returns control of the operation to the environment, since the reaction of the `PRI ALT`ing process to multiple communications is now predictable. This control is crucial for the provision of real-time guarantees in multi-process systems and for the design of hardware. Recently, extensions to CSP to provide a formal treatment of these mechanisms have been made[26, 27].

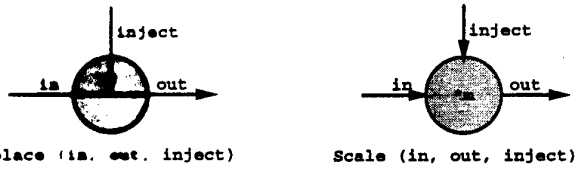


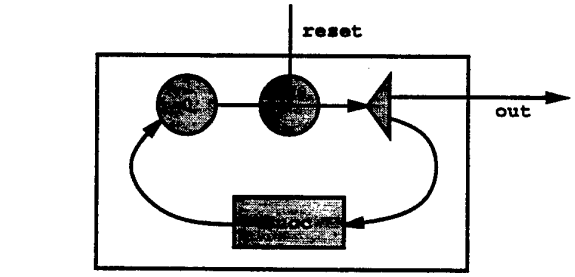
Fig. 7. Two control processes

[4] This is in contrast to *monitors*, whose methods cannot refuse an external call when they are unlocked and have to *wait* on condition variables should their state prevent them from servicing the call. The close coupling necessary between sibling monitor methods to undo the resulting mess is not WYSIWYG[9].

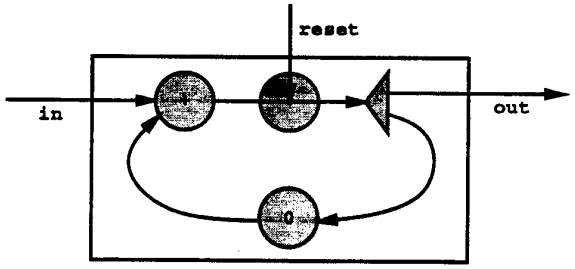
Figure 7 shows two simple components with this kind of control. `Replace` listens for incoming data on its `in` and `inject` lines. Most of the time, data arrives from `in` and is immediately copied to its `out` line. Occasionally, a signal from the `inject` line occurs. When this happens, the signal is copied `out` but, *at the same time*, the next input from `in` is waited for and discarded. In case both `inject` and `in` communications are on offer, priority is given to the (less frequently occurring) `inject`:

```
Replace (in, inject, out)
  = (inject ? signal --> ((in ? x --> SKIP) || (out ! signal --> SKIP))
     [PRI]
     in ? x --> out ! x --> SKIP
    );
    Replace (in, inject, out)
```

`Replace` is something that can be spliced into any channel. If we don't use the `inject` line, all it does is add a one-place buffer to the circuit. If we send something down the `inject` line, it gets injected into the circuit – replacing the next piece of data that would have travelled through that channel.



RNumbers (out, reset)



RIntegrate (in, out, reset)

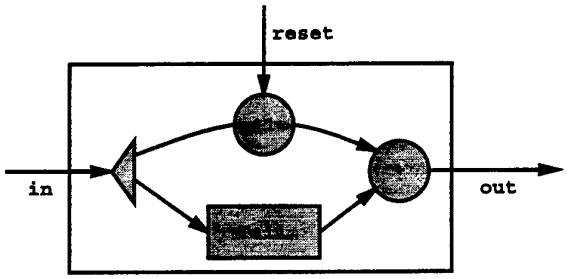
Fig. 8. Two controllable processes

Figure 8 shows RNumbers and RIntegrate, which are just Numbers and Integrate with an added Replace component. We now have components that are resettable by their environments. RNumbers can be reset at any time to continue its output sequence from any chosen value. RIntegrate can have its internal running sum redefined.

Like Replace, Scale (figure 7) normally copies numbers straight through, but scales them by its factor m. An inject signal resets the scale factor:

```
Scale (m, in, inject, out)
  = (inject ? m --> SKIP
     [PRI]
     in ? x --> out ! m*x --> SKIP
    );
    Scale (m, in, inject, out)
```

Figure 9 shows RPairs, which is Pairs with the Scale control component added. If we send just +1 or -1 down the reset line of RPairs, we control whether it's adding or subtracting successive pairs of inputs. When it's subtracting, its behaviour changes to that of a *differentiator* – in the sense that it undoes the effect of Integrate.



RPairs (in, out, reset)

Fig. 9. Sometimes Pairs, sometimes Differentiate

This allows a nice control demonstration. Figure 10 shows a circuit whose core is a resettable version of the Squares pipeline. The Monitor process reacts to characters from the keyboard channel. Depending on its value, it outputs an appropriate signal down an appropriate reset channel:

```
Monitor (keyboard, resetN, resetI, resetP)
  = (keyboard ? ch -->
       CASE ch
         'N': resetN !  0 --> SKIP
         'I': resetI !  0 --> SKIP
         '+': resetP ! +1 --> SKIP
         '-': resetP ! -1 --> SKIP
    );
    Monitor (keyboard, resetN, resetI, resetP)
```

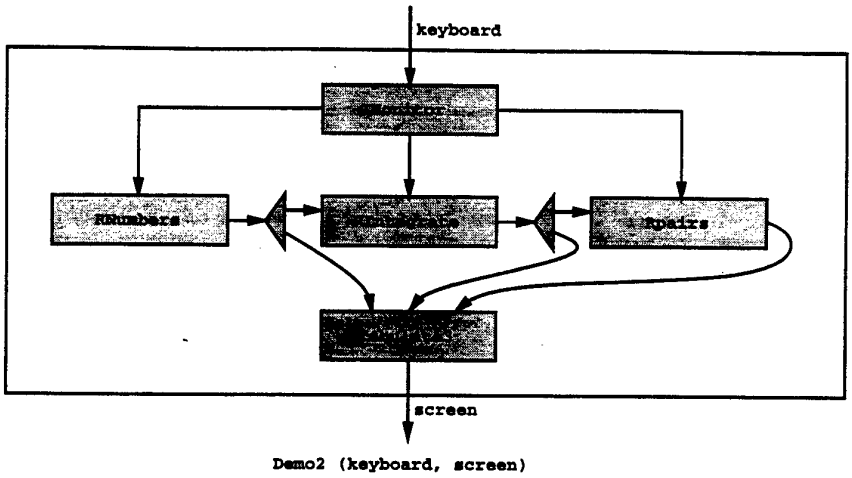


Fig. 10. A user controllable machine

When Demo2 runs and we don't type anything, we see the inner workings of the Squares pipeline tabulated in three columns of output. Keying in an 'N', 'I', '+' or '-' character allows the user some control over those workings[5]. Note that after a '-', the output from RPairs should be the same as that taken from RNumbers.

2.8 A Nastier Deadlock

One last exercise should be done. Modify the system so that output freezes if an 'F' is typed and unfreezes following the next character.

Two 'solutions' offer themselves and Figure 11 shows the *wrong* one (Demo3). This feeds the output from Tabulate3 back to a modified Monitor2 and then on to the screen. The Monitor2 process PRI ALTs between the keyboard channel and this feedback:

```
Monitor2 (keyboard, feedback, resetN, resetI, resetP)
  = (keyboard ? ch -->
       CASE ch
         ...  deal with 'N', 'I', '+', '-' as before
         'F': keyboard ? ch --> SKIP
     [PRI]
     feedback ? x --> screen ! x  --> SKIP
    );
    Monitor2 (keyboard, feedback, resetN, resetI, resetP)
```

[5] In practice, we need to add another process after Tabulate3 to slow down the rate of output to around 10 lines per second. Otherwise, the user cannot properly appreciate the immediacy of control that has been obtained.

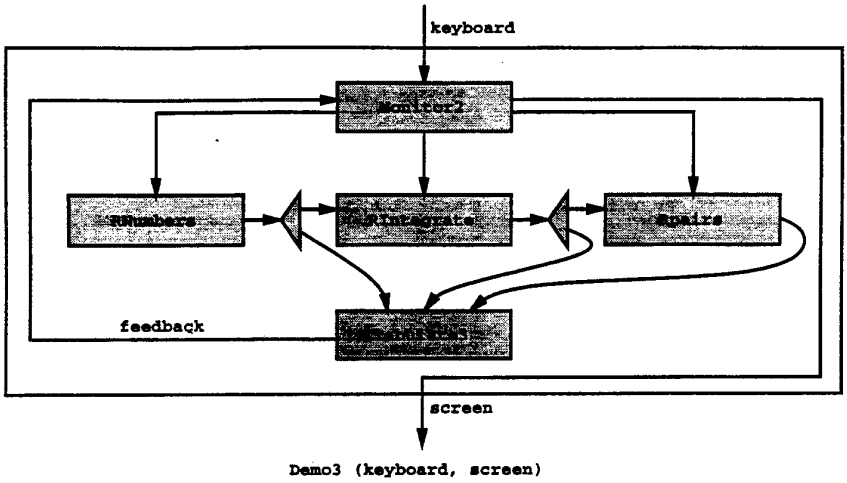


Fig. 11. A machine over which we may lose control

Traffic will normally be flowing along the `feedback-screen` route, interrupted only when `Monitor2` services the `keyboard`. The attraction is that if an 'F' arrives, `Monitor2` simply waits for the next character (and discards it). As a side-effect of this waiting, the `screen` traffic is frozen.

But if we implement this, we get some worrying behaviour. The freeze operation works fine and so, *probably*, do the 'N' and 'I' resets. *Sometimes*, however, a '+' or '-' reset deadlocks the whole system – the `screen` freezes and all further `keyboard` events are refused!

The problem is that one of the rules for deadlock-free design has been broken: *any data-flow circuit must control the number of packets circulating!* If this number rises to the number of sequential (i.e. lowest level) processes in the circuit, deadlock always results. Each node will be trying to output to its successor and refusing input from its predecessor.

The `Numbers`, `RNumbers`, `Integrate`, `RIntegrate` and `Fibonacci` networks all contain data-flow loops, but the number of packets concurrently in flight is kept at one[6].

In `Demo3` however, packets are continually being generated within `RNumbers`, flowing through several paths to `Monitor2` and, then, to the `screen`. Whenever `Monitor2` feeds a reset back into the circuit, deadlock is possible – although not certain. It depends on the scheduling. `RNumbers` is always pressing new packets into the system, so the circuits are likely to be fairly full. If `Monitor2` generates a reset when they are full, the system deadlocks. The shortest feedback loop is from `Monitor2`, `RPairs`, `Tabulate3` and back to `Monitor2` – hence, it is the '+' and '-' inputs from `keyboard` that are most likely to trigger the deadlock.

[6] Initially, `Fibonacci` has two packets, but they combine into one before the end of their first circuit.

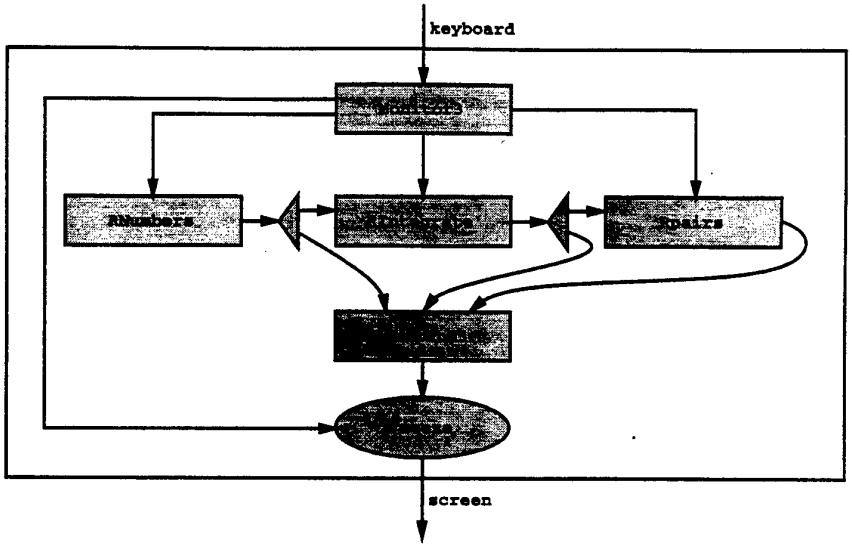


Fig. 12. A machine over which we will not lose control

The design is simply fixed by removing that feedback at this level – see Demo4 in Figure 12. We have abstracted the freezing operation into its own component (and catalogued it). It's never a good idea to try and do too many functions in one sequential process. That needlessly constrains the synchronisation freedom of the network and heightens the risk of deadlock. Note that the idea being pushed here is that, unless there are special circumstances, *parallel design is safer and simpler than its serial counterpart!*

Demo4 obeys another golden rule: *every device should be driven from its own separate process.* The **keyboard** and **screen** channels interface to separate devices and should be operated concurrently (in Demo3, both were driven from one sequential process – Monitor2). Here are the driver processes from Demo4:

```
Freeze (in, freeze, out)
  = (freeze ? x --> freeze ? x --> SKIP
     [PRI]
     (in ? x --> out ! x --> SKIP
    );
    Freeze (in, freeze, out)

Monitor3 (keyboard, resetN, resetI, resetP, freeze)
  = (keyboard ? ch -->
       CASE ch
         ...  deal with 'N', 'I', '+', '-' as before
         'F': freeze ! ch --> keyboard ? ch --> freeze ! ch --> SKIP
    );
    Monitor3 (keyboard, resetN, resetI, resetP, freeze)
```

A channel structure is just a record (or object) holding two or more CSP channels. Usually, there would be just two channels – one for each direction of communication. The channel structure is used to conduct a two-way conversation between two processes. To avoid deadlock, of course, they will have to understand protocols for using the channel structure – such as who speaks first and when the conversation finishes. We call the process that opens the conversation a *client* and the process that listens for that call a *server*[8].

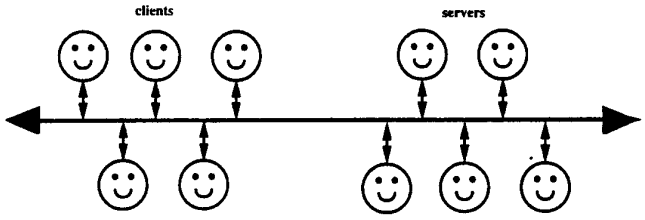


Fig. 13. A many-many shared channel

The CSP model is extended by allowing multiple clients and servers to share the same channel (or channel structure) – see Figure 13. Sanity is preserved by ensuring that only one client and one server use the shared object at any one time. Clients wishing to use the channel queue up first on a client-queue (associated with the shared channel) – servers on a server-queue (also associated with the shared channel). A client only completes its actions on the shared channel when it gets to the front of its queue, finds a server (for which it may have to wait if business is good) and completes its transaction. A server only completes when it reaches the front of its queue, finds a client (for which it may have to wait in times of recession) and completes its transaction.

Note that shared channels – like the choice operator between multiple events – introduce scheduling dependent non-determinism. The order in which processes are granted access to the shared channel depends on the order in which they join the queues.

Shared channels provide a very efficient mechanism for a common form of choice. Any server that offers a non-discriminatory service[9] to multiple clients should use a shared channel, rather than `ALT`ing between individual channels from those clients. The shared channel has a constant time overhead – `ALT`ing is linear on the number of clients. However, if the server needs to discriminate between its clients (e.g. to refuse service to some, depending upon its internal state), `ALT`ing gives us that flexibility. The mechanisms can be efficiently combined. Clients can be grouped into equal-treatment partitions, with each group clustered on its own shared channel and the server `ALT`ing between them.

[8] In fact, the client/server relationship is with respect to the channel structure. A process may be both a server on one interface and a client on another.

[9] Examples for such servers include window managers for multiple animation processes, data loggers for recording traces from multiple components from some machine, etc.

2.9 Buffered and Asynchronous Communications

We have seen how fixed capacity FIFO buffers can be added as active processes to CSP channels. For the occam binding, the overheads for such extra processes are negligible.

With the JavaPP libraries, the same technique *may* be used, but the channel objects can be directly configured to support buffered communications – which saves a couple of context switches. The user may supply objects supporting *any* buffering strategy for channel configuration, including normal blocking buffers, overwrite-when-full buffers, infinite buffers and black-hole buffers (channels that can be written to but not read from – useful for masking off unwanted outputs from components that, otherwise, we wish to reuse intact). However, the user had better stay aware of the semantics of the channels thus created!

Asynchronous communication is commonly found in libraries supporting inter-processor message-passing (such as PVM and MPI). However, the concurrency model usually supported is one for which there is only *one* thread of control on each processor. Asynchronous communication lets that thread of control launch an external communication and continue with its computation. At some point, that computation may need to block until that communication has completed.

These mechanisms are easy to obtain from the concurrency model we are teaching (and which we claim to be general). We don't need anything new. Asynchronous sends are what happen when we output to a buffer (or buffered channel). If we are worried about being blocked when the buffer is full or if we need to block at some later point (should the communication still be unfinished), we can simply spawn off another process[7] to do the send:

```
(out ! packet --> SKIP |PRI| SomeMoreComputation (...));
Continue (...)
```

The `Continue` process only starts when both the `packet` has been sent and `SomeMoreComputation` has finished. `SomeMoreComputation` and sending the `packet` proceed concurrently. We have used the priority version of the parallel operator (`|PRI|`, which gives priority to its left operand), to ensure that the sending process initiates the transfer before the `SomeMoreComputation` is scheduled. Asynchronous receives are implemented in the same way:

```
(in ? packet --> SKIP |PRI| SomeMoreComputation (...));
Continue (...)
```

2.10 Shared Channels

CSP channels are strictly point-to-point. occam3[28] introduced the notion of (securely) *shared* channels and channel structures. These are further extended in the KRoC occam[29] and JavaPP libraries and are included in the teaching model.

[7] The occam overheads for doing this are less than half a microsecond.

For deadlock freedom, each server must guarantee to respond to a client call within some bounded time. During its transaction with the client, it must follow the protocols for communication defined for the channel structure *and* it may engage in separate client transactions with other servers. A client may open a transaction at any time but may not interleave its communications with the server with any other synchronisation (e.g. with another server). These rules have been formalised as CSP specifications[21]. Client-server networks may have plenty of data-flow feedback but, so long as no cycle of client-server relations exist, [21] gives formal proof that the system is deadlock, livelock and starvation free.

Shared channel structures may be stretched across distributed memory (e.g. networked) multiprocessors[15]. Channels may carry all kinds of object – including channels and processes themselves. A shared channel is an excellent means for a client and server to find each other, pass over a private channel and communicate independently of the shared one. Processes will drag pre-attached channels with them as they are moved and can have local channels dynamically (and temporarily) attached when they arrive. See David May's work on *Icarus*[30, 31] for a consistent, simple and practical realisation of this model for distributed and mobile computing.

3 Events and Shared Memory

Shared memory concurrency is often described as being 'easier' than message passing. But great care must be taken to synchronise concurrent access to shared data, else we will be plagued with race hazards and our systems will be useless. CSP primitives provide a sharp set of tools for exercising this control.

3.1 Symmetric Multi-Processing (SMP)

The private memory/algorithm principles of the underlying model – and the security guarantees that go with them – are a powerful way of programming shared memory multiprocessors. Processes can be automatically and dynamically scheduled between available processors (*one object code fits all*). So long as there is an excess of (runnable) processes over processors and the scheduling overheads are sufficiently low, high multiprocessor efficiency can be achieved – with guaranteed no race hazards. With the design methods we have been describing, it's very easy to generate *lots* of processes with most of them runnable most of the time.

3.2 Token Passing and Dynamic CREW

Taking advantage of shared memory to communicate between processes is an extension to this model and must be synchronised. The shared data does not belong to any of the sharing processes, but must be globally visible to them – either on the stack (for occam) or heap (for Java).

The JavaPP channels in previous examples were only used to send data *values* between processes – but they can also be used to send objects. This steps outside the automatic guarantees against race hazard since, unconstrained, it allows parallel access to the same data. One common and useful constraint is only to send *immutable* objects. Another design pattern treats the sent object as a *token* conferring permission to use it – the sending process losing the token as a side-effect of the communication. The trick is to ensure that only one copy of the token ever exists for each sharable object.

Dynamic CREW (Concurrent Read Exclusive Write) operations are also possible with shared memory. Shared channels give us an efficient, elegant and easily provable way to construct an active *guardian* process with which application processes synchronise to effect CREW access to the shared data. Guarantees against starvation of writers by readers – and vice-versa – are made. Details will appear in a later report (available from [32]).

3.3 Structured Barrier Synchronisation and SPMD

Point-to-point channels are just a specialised form of the general CSP multi-process synchronising *event*. The CSP parallel operator binds processes together with events. When *one* process synchronises on an event, *all* processes registered for that event must synchronise on it before that first process may continue. Events give us structured multiway barrier synchronisation[29].

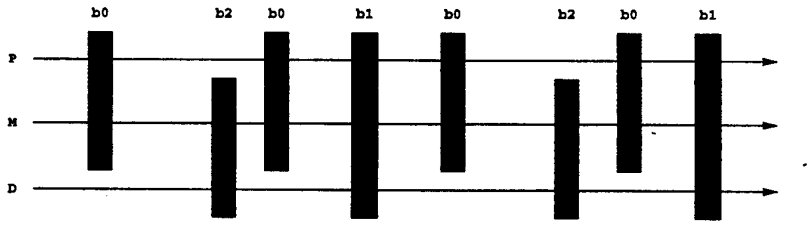


Fig. 14. Multiple barriers to three processes

We can have many event barriers in a system, with different (and not necessarily disjoint) subsets of processes registered for each barrier. Figure 14 shows the execution traces for three processes (P, M and D) with time flowing horizontally. They do not all progress at the same – or even constant – speed. From time to time, tha faster ones will have to wait for their slower partners to reach an agreed barrier before all of them can proceed. We can wrap up the system in typical SPMD form as

```
|| <i = 0 FOR 3>
   S (i, ..., b0, b1, b2)
```

where b0, b1 and b2 are events. The replicated parallel operator runs 3 instances of S in parallel (with i taking the values 0, 1 and 2 respectively in the different instances). The S process simply switches into the required form:

```
S (i, ..., b0, b1, b2)
  = CASE i
      0 : P (..., b0, b1)
      1 : M (..., b0, b1, b2)
      2 : D (..., b1, b2)
```

and where P, M and D are registered only for the events in their parameters. The code for P has the form:

```
P (..., b0, b1)
   = someWork (...); b0 --> SKIP;
     moreWork (...); b0 --> SKIP;
     lastBitOfWork (...); b1 --> SKIP;
     P (..., b0, b1)
```

3.4 Non-Blocking Barrier Synchronisation

In the same way that asynchronous communications can be expressed (section 2.9), we can also achieve the somewhat contradictory sounding, but potentially useful, *non-blocking* barrier synchronisation.

In terms of serial programming, this is a two-phase commitment to the barrier. The first phase declares that we have done everything we need to do this side of the barrier, but does not block us. We can then continue for a while, doing things that do not disturb what we have set up for our partners in the barrier and do not need whatever it is that they have to set. When we need their work, we enter the second phase of our synchronisation on the barrier. This blocks us only if there is one, or more, of our partners who has not reached the first phase of their synchronisation. With luck, this window on the barrier will enable most processes most of the time to pass through without blocking:

```
doOurWorkNeededByOthers (...);
barrier.firstPhase ();
privateWork (...);
barrier.secondPhase ();
useSharedResourcesProtectedByTheBarrier (...);
```

With our lightweight CSP processes, we do not need these special phases to get the same effect:

```
doOurWorkNeededByOthers (...);
(barrier --> SKIP [PRI] privateWork (...));
useSharedResourcesProtectedByTheBarrier (...);
```

The explanation as to why this works is just the same as for the asynchronous sends and receives.

3.5 Bucket Synchronisation

Although CSP allows choice over general events, the occam and Java bindings do not. The reasons are practical – a concern for run-time overheads[10]. So, synchronising on an event commits a process to wait until everyone registered for the event has synchronised. These multi-way events, therefore, do not introduce non-determinism into a system and provide a stable platform for much scientific and engineering modelling.

Buckets[15] provide a non-deterministic version of events that are useful for when the system being modelled is irregular and dynamic (e.g. motor vehicle traffic[33]). Buckets have just two operations: `jump` and `kick`. There is no limit to the number of processes that can jump into a bucket – where they all block. Usually, there will only be one process with responsibility for kicking over the bucket. This can be done at any time of its own (internal) choosing – hence the non-determinism. The result of kicking over a bucket is the unblocking of all the processes that had jumped into it[11].

4 Conclusions

A simple model for parallel computing has been presented that is easy to learn, teach and use. Based upon the mathematically sound framework of Hoare's CSP, it has a compositional semantics that corresponds well with out intuition about how the world is constructed. The basic model encompasses object-oriented design with active processes (i.e. objects whose methods are exclusively under their own thread of control) communicating via passive, but synchronising, wires. Systems can be composed through natural layers of communicating components so that an understanding of each layer does not depend on an understanding of the inner ones. In this way, systems with arbitrarily complex behaviour can be safely constructed – free from race hazard, deadlock, livelock and process starvation.

A small extension to the model addresses fundamental issues and paradigms for shared memory concurrency (such as token passing, CREW dynamics and bulk synchronisation). We can explore with equal fluency serial, message-passing and shared-memory logic and strike whatever balance between them is appropriate for the problem under study. Applications include hardware design (e.g. FFGAs and ASICs), real-time control systems, animation, GUIs, regular and irregular modelling, distributed and mobile computing.

occam and Java bindings for the model are available to support practical work on commodity PCs and workstations. Currently, the occam bindings are

[10] Synchronising on an event in occam has a unit time overhead, regardless of the number of processes registered. This includes being the last process to synchronise, when all blocked processes are released. These overheads are well below a microsecond for modern microprocessors.

[11] As for events, the `jump` and `kick` operations have constant time overhead, regardless of the number of processes involved. The bucket overheads are slightly lower than those for events.

the fastest (context-switch times under 300 nano-seconds), lightest (in terms of memory demands), most secure (in terms of guaranteed thread safety) and quickest to learn. But Java has the libraries (e.g. for GUIs and graphics) and will get faster. Java thread safety depends on following the CSP design patterns, but these are easy to acquire[12].

The JavaPP JCSP library[11] also includes an extension to the Java AWT package that drops channel interfaces on all GUI components[13]. Each item (e.g. a `Button`) is a process with a `configure` and `action` channel interface. These are connected to separate internal handler processes. To change the text or colour of a `Button`, an application process outputs to its `configure` channel. If someone presses the `Button`, it outputs down its `action` channel to an application process (which can accept or refuse the communication as it chooses). Example demonstrations of the use of this package may be found at [11]. Whether GUI programming through the process-channel design pattern is simpler than the listener-callback pattern offered by the underlying AWT, we leave for the interested reader to experiment and decide.

All the primitives described in this paper are available for KRoC occam and Java. Multiprocessor versions of the KRoC kernel targeting NOWs and SMPs will be available later this year. SMP versions of the JCSP[11] and CJT[12] libraries are automatic if your JVM supports SMP threads. Hooks are provided in the channel libraries to allow user-defined network drivers to be installed. Research is continuing on portable/faster kernels and language/tool design for *enforcing* higher level aspects of CSP design patterns (e.g. for shared memory safety and deadlock freedom) that currently rely on self-discipline.

Finally, we stress that this is *undergraduate* material. The concepts are mature and fundamental – *not* advanced – and the earlier they are introduced the better. For developing fluency in concurrent design and implementation, no special hardware is needed. Students can graduate to real parallel systems once they have mastered this fluency. The CSP model is neutral with respect to parallel architecture so that coping with a change in language or paradigm is straightforward. However, even for uni-processor applications, the ability to do safe and lightweight multithreading is becoming crucial *both* to improve response times *and* simplify their design.

The experience at Kent is that students absorb these ideas very quickly and become very creative[14]. Now that they can apply them in the context of Java, they are smiling indeed.

[12] Java active object (i.e. processes) do not invoke each other's methods and communicate only through shared passive objects with carefully designed synchronisation properties (e.g. channels and events). Shared use of user-defined passive objects will be automatically thread-safe so long as the shared memory usage patterns are kept. We do not need to get involved with the monitor model within Java.

[13] We believe that the new Swing GUI libraries from Sun (that will replace the AWT) can also be extended through a channel interface for secure use in parallel designs – despite the warnings concerning the use of Swing and multithreading[34].

[14] The JCSP libraries used in Appendix B were produced by Paul Austin, an undergraduate student at Kent.

References

1. C.A. Hoare. Communication Sequential Processes. *CACM*, 21(8):666–677, August 1978.
2. C.A. Hoare. *Communication Sequential Processes.* Prentice Hall, 1985.
3. Oxford University Computer Laboratory. *The CSP Archive.* <URL: http:// www.comlab.ox.ac.uk/ archive/ csp.html>, 1997.
4. P.H. Welch and D.C. Wood. KRoC - the Kent Retargetable occam Compiler. In B. O'Neill, editor, *Proceedings of WoTUG 19*, Amsterdam, March 1996. WoTUG, IOS Press. <URL:http:// www.hensa.ac.uk/ parallel/ occam/ projects/ occam-for-all/ kroc/>.
5. Peter H. Welch and Michael D. Poole. occam for Multi-Processor DEC Alphas. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 189–198, Amsterdam, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press.
6. Peter Welch et al. *Java Threads Workshop - Post Workshop Discussion.* <URL:http://www.hensa.ac.uk/parallel/groups/wotug/java/discussion/>, February 1997.
7. Gerald Hilderink, Jan Broenink, Wiek Vervoort, and Andre Bakkers. Communicating Java Threads. In *Parallel Programming and Java, Proceedings of WoTUG 20*, pages 48–76, 1997. (See reference [5]).
8. G.H. Hilderink. Communicating Java Threads Reference Manual. In *Parallel Programming and Java, Proceedings of WoTUG 20*, pages 283–325, 1997. (See reference [5]).
9. Peter Welch. Java Threads in the Light of occam/CSP. In P.H.Welch and A. Bakkers, editors, *Architectures, Languages and Patterns, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 259–284, Amsterdam, Netherlands, April 1998. World occam and Transputer User Group (WoTUG), IOS Press. ISBN 90-5199-391-9.
10. Alan Chalmers. *JavaPP Page - Bristol.* <URL:http://www.cs.bris.ac.uk/ ~alan/ javapp.html/>, May 1998.
11. P.D. Austin. *JCSP Home Page.* <URL:http://www.hensa.ac.uk/parallel/languages/ java/jcsp/>, May 1998.
12. Gerald Hilderink. *JavaPP Page - Twente.* <URL:http://www.rt.el.utwente.nl/ javapp/>, May 1998.
13. Ian East. *Parallel Processing with Communication Process Architecture.* UCL press, 1995. ISBN 1-85728-239-6.
14. John Galletly. *occam 2 - including occam 2.1.* UCL Press, 1996. ISBN 1-85728-362-7.
15. occam-for-all Team. *occam-for-all Home Page.* <URL:http://www.hensa.ac.uk/ parallel/occam/occam-for-all/>, February 1997.
16. Mark Debbage, Mark Hill, Sean Wykes, and Denis Nicole. Southampton's Portable occam Compiler (SPoC). In R. Miles and A. Chalmers, editors, *Progress in Transputer and occam Research, Proceedings of WoTUG 17*, Concurrent Systems Engineering, pages 40–55. Amsterdam, Netherlands, April 1994. World occam and Transputer User Group (WoTUG), IOS Press. <URL:http://www.hensa.ac.uk/ parallel/ occam/ compilers/ spoc/>.
17. J.M.R. Martin and S.A. Jassim. How to Design Deadlock-Free Networks Using CSP and Verification Tools - a Tutorial Introduction. In *Parallel Programming and Java, Proceedings of WoTUG 20*, pages 326–338, 1997. (See reference [5]).

18. A.W. Roscoe and N. Dathi. The Pursuit of Deadlock Freedom. Technical Report *Technical Monograph PRG-57*, Oxford University Computing Laboratory, 1986.
19. J. Martin, I. East, and S. Jassim. Design Rules for Deadlock Freedom. *Transputer Communications*, 2(3):121–133, September 1994. ISSN 1070-454X.
20. P.H. Welch, G.R.R. Justo, and C. Willcock. High-Level Paradigms for Deadlock-Free High-Performance Systems. In Grebe et al., editors, *Transputer Applications and Systems '93*, pages 981–1004, Amsterdam, 1993. IOS Press. ISBN 90-5199-140-1.
21. J.M.R. Martin and P.H. Welch. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4):215–232, October 1996. ISSN 1070-454X.
22. A.W. Roscoe. *Model Checking* CSP, *A Classical Mind.* Prentice Hall, 1994.
23. J.M.R. Martin and S.A. Jassim. A Tool for Proving Deadlock Freedom. In *Parallel Programming and Java, Proceedings of WoTUG 20*, pages 1–16, 1997. (See reference [5]).
24. D.J. Beckett and P.H. Welch. A Strict occam Design Tool. In *Proceedings of UK Parallel '96*, pages 53–69, London, July 1996. BCS PPSIG, Springer-Verlag. ISBN 3-540-76068-7.
25. M. Aubury, I. Page, D. Plunkett, M. Sauer, and J. Saul. Advanced Silicon Prototyping in a Reconfigurable Environment. In *Architectures, Languages and Patterns, Proceedings of WoTUG 21*, pages 81–92, 1998. (See reference [9]).
26. A.E. Lawrence. Extending CSP. In *Architectures, Languages and Patterns, Proceedings of WoTUG 21*, pages 111–132, 1998. (See reference [9]).
27. A.E. Lawrence. HCSP: Extending CSP for Co-design and Shared Memory. In *Architectures, Languages and Patterns, Proceedings of WoTUG 21*, pages 133–156, 1998. (See reference [9]).
28. Geoff Barrett. occam3 reference manual (draft). <URL:http:// www.hensa.ac.uk/ parallel/ occam/ documents/>, March 1992. (unpublished in paper).
29. Peter H. Welch and David C. Wood. Higher Levels of Process Synchronisation. In *Parallel Programming and Java, Proceedings of WoTUG 20*, pages 104–129, 1997. (See reference [5]).
30. David May and Henk L Muller. Icarus language definition. Technical Report CSTR-97-007, Department of Computer Science, University of Bristol, January 1997.
31. Henk L. Muller and David May. A simple protocol to communicate channels over channels. Technical Report CSTR-98-001, Department of Computer Science, University of Bristol, January 1998.
32. D.J. Beckett. *Java Resources Page.* <URL:http://www.hensa.ac.uk/parallel/ languages/java/>, May 1998.
33. Kang Hsin Lu, Jeff Jones, and Jon Kerridge. Modelling Congested Road Traffic Networks Using a Highly Parallel System. In A. DeGloria, M.R. Jane, and D. Marini, editors, *Transputer Applications and Systems '94*, volume 42 of *Concurrent Systems Engineering*, pages 634–647, Amsterdam, Netherlands, September 1994. The Transputer Consortium, IOS Press. ISBN 90-5199-177-0.
34. Hans Muller and Kathy Walrath. Threads and swing. <URL:http://java.sun.com/ products/jfc/swingdoc-archive/threads.html>, April 1998.

Appendix A: occam Executables

Space only permits a sample of the examples to be shown here. This first group are from the 'Legoland' catalogue (Section 2.3):

```
PROC Id (CHAN OF INT in, out)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out ! x
:
```

```
PROC Succ (CHAN OF INT in, out)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out ! x PLUS 1
:
```

```
PROC Plus (CHAN OF INT in0, in1, out)
  WHILE TRUE
    INT x0, x1:
    SEQ
      PAR
        in0 ? x0
        in1 ? x1
      out ! x0 PLUS x1
:
```

```
PROC Prefix (VAL INT n, CHAN OF INT in, out)
  SEQ
    out ! n
    Id (in, out)
:
```

Next come four two of the 'Plug and Play' examples from Sections 2.4 and 2.6:

```
PROC Numbers (CHAN OF INT out)
  CHAN OF INT a, b, c:
  PAR
    Prefix (0, c, a)
    Delta (a, out, b)
    Succ (b, c)
:
```

```
PROC Integrate (CHAN OF INT in, out)
  CHAN OF INT a, b, c:
  PAR
    Plus (in, c, a)
    Delta (a, out, b)
    Prefix (0, b, c)
:
```

```
PROC Pairs (CHAN OF INT in, out)
  CHAN OF INT a, b, c:
  PAR
    Delta (in, a, b)
    Tail (b, c)
    Plus (a, c, out)
:
```

```
PROC Squares (CHAN OF INT out)
  CHAN OF INT a, b:
  PAR
    Numbers (a)
    Integrate (a, b)
    Pairs (b, out)
:
```

Here is one of the controllers from Section 2.7:

```
PROC Replace (CHAN OF INT in, inject, out)
  WHILE TRUE
    PRI ALT
      INT x:
      inject ? x
        PAR
          INT discard:
          in ? discard
          out ! x
      INT x:
      in ? x
        out ! x
:
```

Asynchronous receive from Section 2.9:

```
SEQ
  PRI PAR
    in ? packet
    SomeMoreComputation (...)
  Continue (...)
```

Barrier synchronisation from Section 3.3:

```
PROC P (..., EVENT b0, b2)
  ...  local state declarations
  SEQ
    ...  initialise local state
    WHILE TRUE
      SEQ
        someWork (...)
        synchronise.event (b0)
        moreWork (...)
        synchronise.event (b0)
        lastBitOfWork (...)
        synchronise.event (b1)
:
```

Finally, non-blocking barrier synchronisation from Section 3.4:

```
SEQ
  doOurWorkNeededByOthers (...)
  PRI PAR
    synchronise.event (barrier)
    privateWork (...)
  useSharedResourcesProtectedByTheBarrier (...)
```

Appendix B: Java Executables

These examples use the JCSP library for processes and channels[11]. A process is an instance of a class that implements the `CSProcess` interface. This is similar to, but different from, the standard `Runable` interface:

```
package jcsp.lang;

public interface CSProcess {
  public abstract void run ();
}
```

For example, from the 'Legoland' catalogue (Section 2.3):

```
import jcsp.lang.*;

class Succ implements CSProcess {

  private ChannelInputInt in;
  private ChannelOutputInt out;

  public Succ(ChannelInputInt in, ChannelOutputInt out) {
    this.in = in;
    this.out = out;
  }

  public void run() {
    while (true) {
      int x = in.read ();
      out.write (x + 1);
    }
  }
}

class Prefix implements CSProcess {

  private int n;
  private ChannelInputInt in;
  private ChannelOutputInt out;

  public Prefix(int n, ChannelInputInt in, ChannelOutputInt out) {
    this.n = n;
    this.in = in;
    this.out = out;
  }

  public void run() {
    out.write (n);
    new Id (in, out).run ();
  }
}
```

JCSP provides a `Parallel` class that combines an array of `CSProcesses` into a `CSProcess`. It's execution is the parallel composition of that array. For example, here are two of the 'Plug and Play' examples from Sections 2.4 and 2.6:

```
class Numbers implements CSProcess {

  private ChannelOutputInt out;

  public Numbers (ChannelOutputInt out) {
    this.out = out;
  }

  public void run() {
    One2OneChannelInt a = new One2OneChannelInt ();
    One2OneChannelInt b = new One2OneChannelInt ();
    One2OneChannelInt c = new One2OneChannelInt ();
    new Parallel (
      new CSProcess[] {
        new Delta (a, out, b),
        new Succ (b, c),
        new Prefix (0, c, a),
      }
    ).run();
  }
}

class Squares implements CSProcess {

  private ChannelOutputInt out;

  public Squares (ChannelOutputInt out) {
    this.out = out;
  }

  public void run() {
    One2OneChannelInt a = new One2OneChannelInt ();
    One2OneChannelInt b = new One2OneChannelInt ();
    new Parallel (
      new CSProcess[] {
        new Numbers (a),
        new Integrate (a, b),
        new Pairs (b, out),
      }
    ).run();
  }
}
```

Here is one of the controllers from Section 2.7. The processes `ReadInt` and `WriteInt` just read and write a single integer (from and to a public `value` field):

```
class Replace implements CSProcess {

  private AltingChannelInputInt in;
  private AltingChannelInputInt inject;
  private ChannelOutputInt out;

  public Replace (AltingChannelInputInt in,
                  AltingChannelInputInt inject,
                  ChannelOutputInt out) {
    this.in = in;
    this.inject = inject;
    this.out = out;
  }

  public void run() {

    Alternative alt = new Alternative();
    AltingChannelInputInt[] altChans = {inject, in};

    CSProcess writeInt = new WriteInt (out);
    CSProcess readInt = new ReadInt (in);
    CSProcess parIO  = new Parallel (new CSProcess[] {readInt, writeInt});

    while (true) {
      switch (alt.select (altChans)) {
        case 0:
          writeInt.value = inject.read ();
          parIO.run ();
        break;
        case 1:
          out.write (in.read ());
        break;
      }
    }
  }
}
```

JCSP also has channels for sending and receiving arbitrary `Objects`. Here is an asynchronous receive (from Section 2.9) of an expected `Packet`:

```
// set up processes once (before we start looping ...)

CSProcess readObj = new ReadObj (in);
CSProcess someMore = new SomeMoreComputation (...);
CSProcess async = new PriParallel (new CSProcess[] {readObj, someMore});

while (looping) {
  async.run ();
  Packet packet =  Packet) readObj.object
  Continue (...);
}
```

An ISA comparison between Superscalar and Vector Processors

Francisca Quintana[1], Roger Espasa[2], and Mateo Valero[2]

[1]University of Las Palmas de Gran Canaria, Edificio de Informática y Matemáticas, Campus de Tafira, 35017 Las Palmas de Gran Canaria, Canary Islands, Spain
fquintan@dis.ulpgc.es
[2]U. Politècnica Catalunya--Barcelona, Computer Architecture Department, Campus Nord
{roger,mateo}@ac.upc.es

Abstract. This paper presents a comparison between superscalar and vector processors. First, we start with a detailed ISA analysis of the vector machine, including data related to masked execution, vector length and vector first facilities. Then we present a comparison of the two models at the instruction set architecture (ISA) level that shows that the vector model has several advantages: executes fewer instructions, fewer overall operations, and generally executes fewer memory accesses. We then analyse both models in terms of speculative execution, each one in its context. Results show that superscalar processors make an extensive use of speculation and that there is a large ammount of misspeculated instructions. In the vector model, speculation is achieved using vector masks and, in general, fewer operations are misspeculated.

1 Introduction

Traditionally, there have been different approaches aimed at improving microprocessor performance. One of them has been the exploitation of data level parallelism (DLP). The DLP paradigm uses vectorization techniques to discover data level parallelism in a sequentially specified program and expresses this parallelism using vector instructions[1][2][3] A single vector instruction specifies a series of operations to be performed on a stream of data Each operation performed on each individual element is independent of all others and, therefore, a vector instruction is easily pipelineable and highly parallel[4][5][6] Another approach aimed at reaching high performance in a program's execution is the exploitation of instruction level parallelism (ILP). Current state-of-the-art microprocessors all include 4-wide fetch engines coupled with sophisticated branch predictors, large reorder buffers to dynamically schedule instructions and non-blocking caches to allow multiple outstanding misses. All these techniques focus on a single goal: executing several instructions that are known to be independent, in parallel[7] The larger the number of instructions that can be launched on each cycle, the better the performance achieved.

There are two very important advantages in using vector instructions to express data-level parallelism. First, the total number of instructions that have to be executed to complete a program is reduced because each vector instruction has more semantic content that the corresponding scalar instructions. Second, the fact that the individual operations in a single vector instruction are independent allows a more efficient execution: once a vector instruction is issued to a functional unit, it will use it with useful work for many cycles. During those cycles, the processor can look for other vector instructions to be launched to the same or other functional units. It is very likely that, by the time a vector instruction completes all its work, there is already another vector instruction ready to occupy the functional unit. Meanwhile, in a scalar processor, when an instruction is launched to a functional unit, another instruction is required at the very next cycle to keep the functional unit busy. Unfortunately, many hazards can get in the way of this requirement: true data dependencies, cache misses, branch misspeculation, etc.

The combination of these two effects has many related advantages. First, the pressure on the fetch unit is greatly reduced. By specifying many operations with a single instruction, the total number of different instructions that have to be fetched is reduced. Many branches disappear embedded in the semantics of vector instructions. A second advantage is the simplicity of the control unit. With relatively few control effort, a vector architecture can control the execution of many different functional units, since most of them work in parallel in a fully synchronous way. A third advantage is related to the way the memory system is accessed: a single vector instruction can exactly specify a long sequence of memory addresses. Consequently, the hardware has considerable advance knowledge regarding memory references, can schedule these accesses in an efficient way[8], and needs to access no more data than is actually needed. In addition, a vector memory operation is able to amortize start-up latencies over a potentially long stream of vector elements.

In this paper we make a comparison between vector and superscalar processors by analysing the behaviour of a Mips R10000[9] superscalar processor and a Convex C4[10] vector processor. This study is carried out from different points of view. First of all we introduce an initial analysis of the Convex C4 vector processor. This includes an overview of several intrinsic characteristics of vector processing: we will analyze the effect of execution under mask and execution using the vector first facility. Then we will compare the superscalar and vector approaches from the ISA point of view. We will present data about the number of instructions and operations executed in both processors. Finally, we will present a comparison about speculative execution in the two approaches.

2 Convex C4 Analysis

We will start by analyzing the vector length and vector mask facilities of vector processors. We will also present the vector first facility which is specific of the Convex C4 machine. Then we will compare the number of instructions, operations and memory traffic of vector processors and superscalars.

This study will be carried out using the six more vectorizable programs from Specfp92. We have measured the vectorization percentage using the Dixie tool[11]. We have generated the execution traces of the Specfp92 programs when running on a Convex C4 machine, and then we have used the Jinks simulator to measure the amount of vector and scalar operations carried out by the programs. The vectorization percentage has been calculated as the ratio between vector operations and the addition of vector and scalar operations.

2.1 Operation Distribution

Table 1 presents the basic operation distribution for the five more vectorizable programs of the Specfp92. First column shows the total number of basic blocks (in millions) executed for each program. Next two columns present the total number of instructions broken down into scalar and vector instructions. We will distinguish between instructions and operations. A scalar instruction performs only one operation, while a vector instruction performs several operations, depending on the value of the vector length (VL) register. Fifth column is the percentage of vectorization for each program, defined as the ratio between the number of vector operations and the total number of operations performed. Finally column sixth presents the average vector length used in vector instructions. An interesting point from this table is the average vector length observed in the programs, which is not heavily related to the percentage of vectorization.

Table 1. Operation distribution

Program	# basic blocks	# instructions		# vector operations	% Vect	Avg. VL
		Scalar	Vector			
Swm256	2.57	27.46	74.82	8127.98	99.7	93
Hydro2d	4.74	38.85	35.43	3684.89	99.0	101
Nasa7	16.79	139.80	55.98	3885.02	96.5	62
Su2cor	22.53	143.95	24.08	3066.07	95.5	125
Tomcatv	19.95	126.66	6.37	644.41	83.6	99
Wave5	48.99	579.77	35.88	1615.04	73.6	43

2.2 Vector Length Distributions

Vector execution is based on executing a certain operation specified in one instruction over a large amount of independent data. The amount of data specified in each instruction is dinamically specified with the value of the Vector Length register (VL).The latency of the operation being carried out is then amortized across all VL elements. Therefore, the larger the VL, the better the performance. Fig. 1 presents the

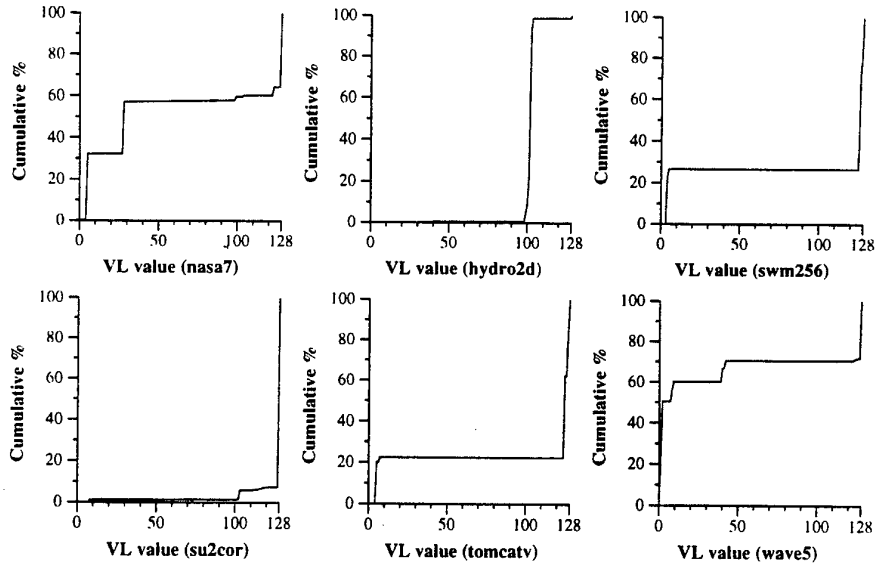


Fig. 1. VL Distribution for Specfp92 programs

VL distribution for the six more vectorizable Specfp92 benchmarks. As we can see, the vector length distributions follow several patterns. Swim256, *Tomcatv* and *Su2cor* have the majority of their vector lengths clustered around 128. *Hydro2d* has a single dominant vector length which is the number of grid points used in the z-direction of the problem. *Nasa7* and *Wave5* have a distribution that follows a staircase, having several dominant vector lengths. All this data suggest that even among vectorizable programs the utilisation of the vector registers varies a lot.

2.3 Vector First Capability

A new capability in the Convex C4 processor is the Vector First facility which allows specifying the first element in the vector register on which the instruction will be executed. That is, an instruction executes VL operations starting at element VF. This facility avoids having to reload data in the cases of recurrences as those presented in Fig. 2(a). In these cases, instead of executing two load instructions for matrix B (for position I and I+1, as presented in Fig. 2(b)), only one load instruction is executed. Fig. 2(b) shows the assembly code without vector first. Every add instruction involves two vector load instructions, which is redundant. In Fig. 2(c), using vector first, the same data can be reused in the loop body just using the appropriate vector first value, so just one vector load is needed for each add instruction. [Note that the notation '^v0' means execution under vector first].

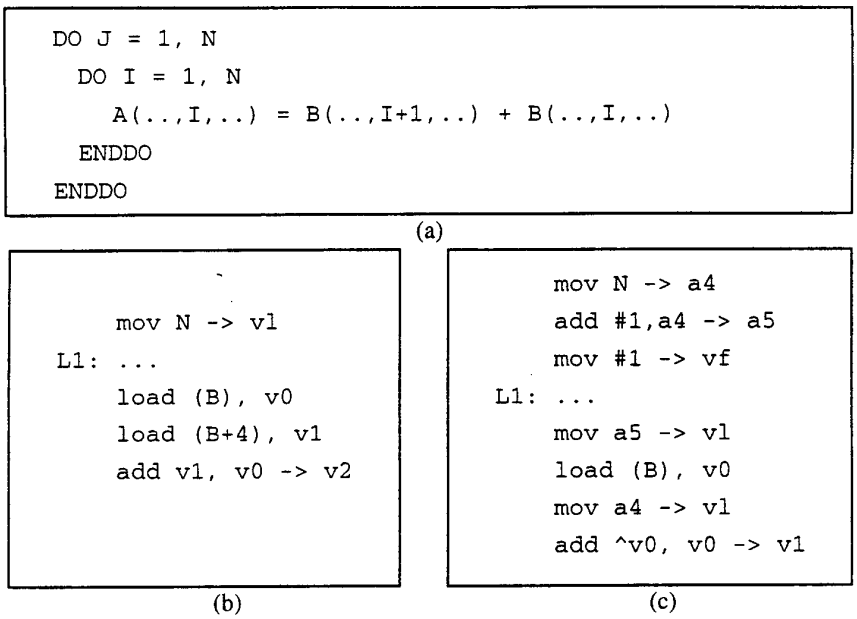


Fig. 2. Typical vector loop at *Hydro2d* benchmark. (a) Source code for a vector loop with a recurrence of distance 1. (b) Assembly code without using vector first facility, with add involving two load instructions. (c) Assembly code using vector first so that every data must be loaded just once

Table 2 presents the distribution of the vector first values for the same Specfp92 benchmarks as Fig. 1. This table shows the total number of operations carried out under vector first and the respective percentages of operations that have been executed with vector first equal to 1, 2 or other values. The compiler is not able to use the vector first neither in benchmark *Nasa7* nor in *Su2cor*. Moreover, these programs only present low order recurrences (with distance 1or 2).

Table 2. Vector First distribution for Specfp92 programs

Program	# Ops under VF (x 10^6)	VF Value (in percentages)		
		1	2	Other
Swm256	2.841	76	24	0
Hydro2d	11.060	100	0	0
Nasa7	--	--	--	--
Su2cor	--	--	--	--
Tomcatv	1.124	50	50	0
Wave5	1.449	97	3	0

2.4 Vector Mask Execution

The Convex C4 vector processor allows the execution of instructions under a calculated mask stored in the Vector Mask (VM) register. The VL operations will be carried out, but only those that have the correct value stored in the i^{th} position of the mask will be finally stored in the destination register of the instruction. We have made an analysis of the masks used during the execution of the benchmarks so to test the effectiveness of masked execution. Table 3 shows the total ammount of instructions executed under mask and the percentage of instructions with respect to the total ammount of instructions. This data shows a relatively small use of the execution under mask in the C4 vector processor. However, taking into account that each vector instruction implies the execution of VL operations, table 3 also shows the total ammount of operations executed under mask and the percentage of operations referred to the total ammount of operations. From this table we can see that the most intensive use of the masked execution is made by the *Hydro2d* benchmark with more than 15% of their operations executed under mask. Programs *Su2cor* and *Wave5* execute 3.95% and 3.64% of their operations under mask, respectively. The remaining programs execute either very few operations under mask (*Swm256* and *Nasa7*) or none at all (*Tomcatv*).

The execution of operations under mask can be considered as speculative execution, as all VL operations are carried out but only those that correspond to the right value in the mask are used. We can think of the extra operations as misspeculative execution. The analysis of the masks, as we will show, has allowed us to measure the amount of speculative work carried out by the vector processor.

Table 3. Instructions and operations executed under vector mask

Program	Instructions executed under vector mask		Operations executed under vector mask	
	Total Number (x 10^6)	% over total instructions	Total Number (x 10^6)	% over total operations
Swm256	0.01	0.015	0.13	0.016
Hydro2d	5.75	7.75	582.91	15.65
Nasa7	0.07	0.036	8.02	0.20
Su2cor	1.06	0.63	130.75	3.95
Tomcatv	0.00	0.00	0.00	0.00
Wave5	5.17	0.84	80.00	3.64

3 Scalar and Vector ISA's Comparison

In this section we present a comparison between superscalar and vector processors at the instruction set architecture level. We will look at three different issues that are

determined by the instruction set being used and by the compiler: number of instructions executed, number of operations executed and memory traffic generated. the distinction between instructions and operations is necessary because in the vector architecture, a vector *instruction* executes several *operations* (between 1 and 128 in our case).

3.1 Instructions Executed

As already mentioned, vector instructions contain a high semantic content in terms of operations specified. The result is that, to perform a given task, a vector program executes many fewer instructions than a scalar program, since the scalar program has to specify more address calculations, loop counter increments and branch computations that are typically implicit in vector instructions. The net effect of vector instructions is that, in order to specify all the computations required for a certain program, much less instructions are needed. Fig. 3(a) presents the total number of instructions executed in the Mips R10000 (using Mips IV Instruction Set [12]) and the Convex C4 machines for the six benchmark programs. In the Mips R10000 case, we use the values of graduated instructions gathered using the hardware performance counters. In the Convex C4 case we use the traces provided by Dixie[12]. As it can be seen, the differences are huge. Obviously, as vectorization degree decreases, this gap is diminished. Although several compiler optimizations (loop unrolling, for example) can be used to lower the overhead of typical loop control instructions in superscalar code, vector instructions are inherently more expressive. Having vector instructions allows a loop to do a task in fewer iterations. This implies fewer computations for address calculations and loop control, as well as less instructions dispatched to execute the loop body itself. As a direct consequence of executing less instructions, the instruction fetch bandwidth required, the pressure on the fetch engine and the negative impact of branches are all three reduced in comparison to a superscalar processor. Also, relatively simple control unit is enough to dispatch a large number of operations in a single go, whereas the superscalar processor devotes an always increasing part of its area to manage out-of-order execution and multiple issue. This simple control, in turn, can potentially yield a faster clocking of the whole datapath. It is interesting to note that the ratio of number of instructions can be larger than 128. Consider, for example, *Swm256*. In vector mode, it requires 102.28 million instructions while in superscalar mode requires 11466 million instructions. If, on average, each vector instruction performs 93 iterations then all these vector instructions would be roughly equivalent to 102.28*93 = 9512 million superscalar instructions. The difference between 9512 and 11466 is the extra overhead that the supercalar machine has to pay due to the larger number of loop iterations it performs.

3.2 Operations Executed

Although the comparison in terms of instructions is important from the point of view of the pressure on the fetch engine, a more accurate comparison between the super-

scalar and vector model comes from looking at the total number of operations performed. As already mentioned in the previous section, the reduction of overhead due to the semantic content of vector instructions should translate into an smaller number of operations executed in the vector model. Fig. 3(b) plots the total number of operations executed on each platform for each program. These data has been gathered from the internal performance counters of the Mips R1000 processor, and from the traces obtained with Dixie. As expected, the total number of operations in the superscalar platform is greater than in the vector machine, for all programs. The ratio of superscalar operations to vector operations can be favourable to the vector model by factors that go from 1.24 up to 1.88.

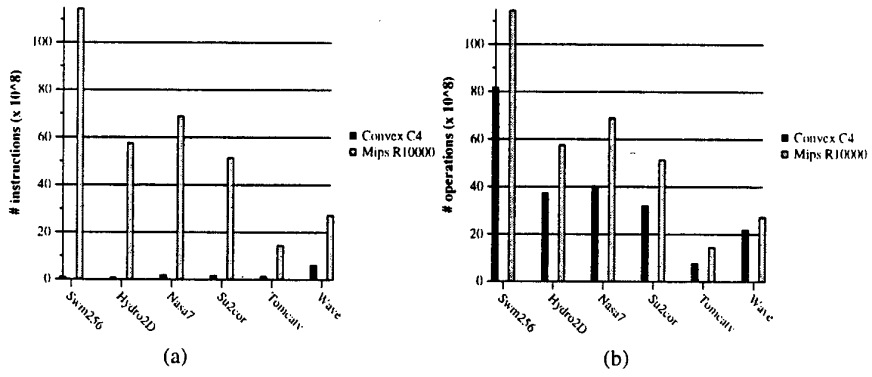


Fig. 3. Vector – Superscalar ISA comparison. (a) Instructions executed. (b) Operations executed

3.3 Memory Traffic

Another analysis that we have carried out is the study of memory traffic both in vector and superscalar processors. Superescalar processors have a memory hierarchy in which data is moved up and down in terms of cache lines. Some of this data is thrown away from the cache before it is used so there is an amount of traffic that is not strictly useful. In vector processors, every data item that is brought from main memory is used, so there is no useless traffic in vector processors. Moreover, depending on the data size of the program there will be different behaviours in superscalar processors. If data fits in L1, there will be almost no traffic between the L1 and the L2 caches. However, if data doesn't fit in L1 but fits in L2, there will be a lot of traffic between the L1 and L2 caches because of conflicts. If data doesn't fit in the L2 cache, traffic will increase a lot between the two memory hierarchy levels. These behaviours can be seen if Fig. 4.

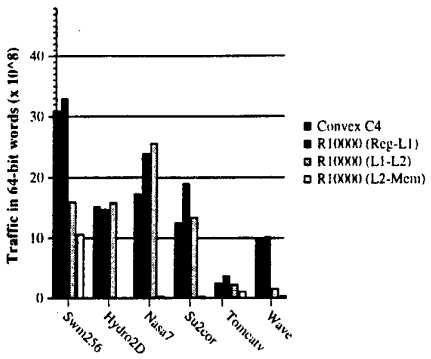


Fig. 4. Vector – Superscalar Memory traffic comparison

4 Speculative Execution in Superscalar and Vector Processors

In this section we will make a study about speculative execution in superscalar and vector processors. Each architecture is able to speculatively execute instructions, although each one in its particular way. Superscalar processors execute speculatively instructions based upon predictions of conditional branches. Vector processors execute instructions under vector masks and only those that have the correct value in the mask are definitely stored. This section is intended to study the effectiveness of the speculative execution in both architectures.

4.1 Speculation in Superscalar Processors

The increase in SS processors aggressiveness regarding issue width and out of order execution has made branch prediction and speculative execution essential techniques in taking advantage of processor capabilities. When a branch is reached, and the result of the condition evaluation is not known, a speculation of the final result of the branch is made, so that the execution continues along the speculated direction. When the actual result of the branch condition is obtained, the executed instructions are validated if the prediction was correct, and rejected if not.

The amount of misspeculative instructions in the SS processor is presented in Fig. 5. This data has been gathered using the Mips R10000 performance internal counters. This speculative work includes all types of instructions. As we can see in Fig.5(a) the misspeculated execution of instructions (referred to the total number of issued instructions) for the six programs goes from 14% to 25%.

Among the misspeculative work, the load/store misspeculation is specially important because it wastes non-blocking cache resources, bandwidth, and can pollute the cache (and memory hierarchy in general) by making data movements between differ-

ent levels that won't be used in the future. Fig. 5(b) shows the load/store misspeculation degree for the benchmarks with respect to the total number of load/store instructions. In some of them, the misspeculation percentage is as large as 40%, although the mean value is about 15%.

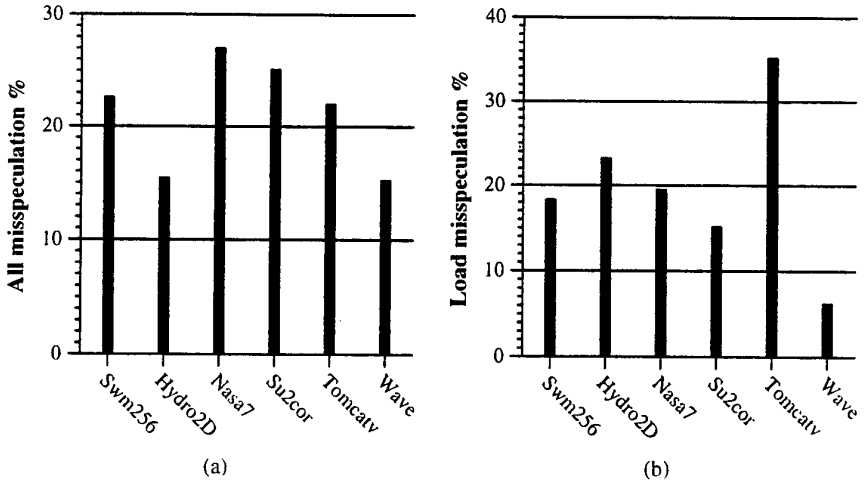


Fig. 5. (a) Misspeculative execution in superscalar processors. (b) Load misspeculation in superscalar processors

4.2 Speculation in Vector Processors

Vector processors are also able to speculatively execute instructions, but in a different way than superscalar processors. It is based on the execution under vector mask. When an instruction is executed under vector mask, all the operations are carried out, but only those having the correct value in the i^{th} position of the vector mask is definitely stored in the destination register. We have previously presented the values of masked executions referred to the total number of instructions and operations carried out by the programs. However, as masked execution is only carried out in vector mode, a more precise measure about the use of masked execution is presented in table 4. Measures in table 4 show that the behaviour differs from one program to another. Program *Hydro2d* executes a considerable ammount of operations under mask (16%). *Swm256* and *Nasa7* make almost no use of the execution under mask and finally, *Su2cor* and *Wave5* execute 4.23% and 4.95% of their operations under mask.

An interesting analysis independent from the use of masked execution, is the effectiveness of masked execution. All these instructions executed under mask, are properly speculated or not. An operation is speculated "right" if after the operation has been carried out the result is effectively stored in its destination. All those operation that were carried out but not stored are misspeculated work. Fig. 6(a) shows the

distribution of right and wrong speculated operations in the five programs (recall that program *Tomcatv* does not execute instructions speculatively). Three of the programs (*Nasa7*, *Su2cor* and *Wave5*) have good values of right prediction: *Nasa7* and *Wave5* are above 63% of right speculation and *Su2cor* is more than 56%. The other two programs (*Swm256* and *Hydro2d*) have low values of right speculation, with Swim256 being the program with the worst behaviour (only 2.58% of right speculation).

Table 4. Instructions and operations executed under vector mask

Program	Instructions executed under vector mask		Operations executed under vector mask	
	Total Number (x 10^6)	% over total vector instructions	Total Number (x 10^6)	% over total vector operations
Swm256	0.01	0.002	0.13	0.016
Hydro2d	5.75	16.25	582.91	16.00
Nasa7	0.07	0.12	8.02	0.20
Su2cor	1.06	4.41	130.75	4.23
Wave5	5.17	14.40	80.00	4.95

Another interesting consideration that we have studied regards the distribution of operations executed under mask among the different instruction types. This study has allowed us to establish the ammount of instructions executed under mask for each type of instructions. We have considered six types of instructions: add-like, mul-like, div, diadic, load and store.

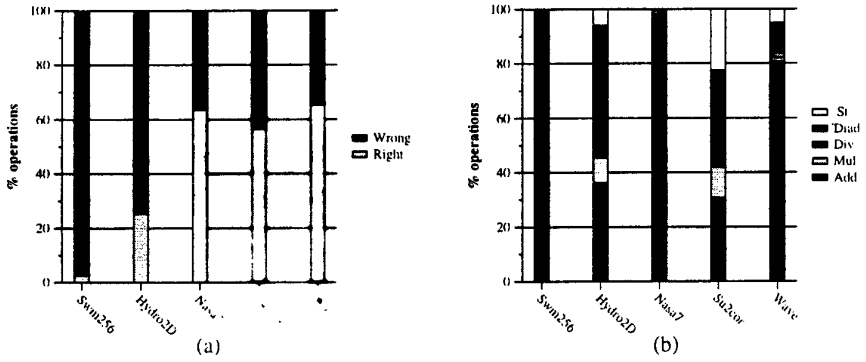


Fig. 6. (a) Distribution of Right Wrong speculation operation. (b) Distribution of instruction executed under vector mask among the different instruction types

The first consideration comes from the fact that none of the programs execute load instructions under mask, which may be explained because of the possibility of gather instructions. Fig 6(b) shows the breakdown of instructions executed under mask among the different instruction types. Division and add-like instructions are the most used instructions for execution under mask.

Finally, we have also studied the effectiveness of execution under mask among the different types of instructions. Results in Fig. 7 show that, in general, there is not a clear correlation between the instruction type and the misspeculation rate. Division instructions are an exception. For divisions the misspeculation rates are higher than for the rest of instruction in all cases. This result is not unreasonable since division instructions are typically executed in statements such as the following,

```
if A(i)<> 0 then B(i)=B(i)/A(i)
```

In such a case, misspeculation is determined by the value stored in A(i). In our programs, the A(i) vector is sparsely populated and causes large numbers of misspeculations.

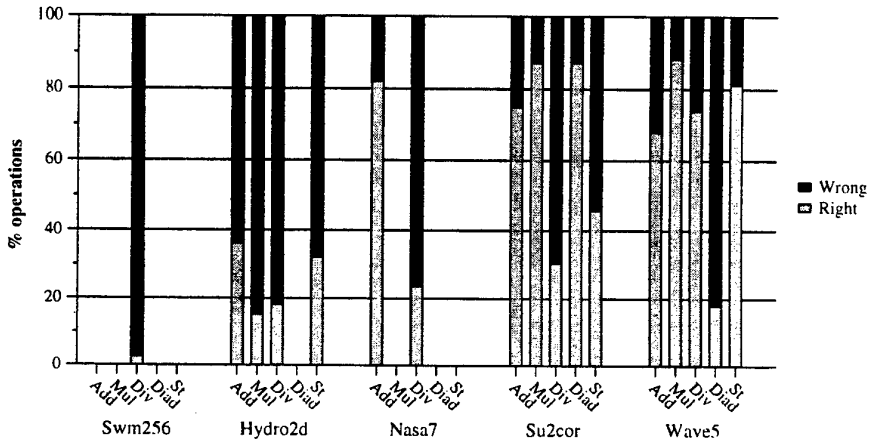


Fig. 7. Break-down of Right – Wrong speculated operations

5 Conclusions

We have outlined a comparison between superscalar and vector processors from several points of view. Vector processors have different possibilities that allow them to decrease the memory traffic and branch impact in a program's execution. Their SIMD model is especially interesting because the initial latency of the operations is amortized across the VL operations that each instruction executes.

We have studied the behaviour at the ISA level of the superscalar and vector processors. We looked at total number of instructions executed, number of operations

executed and memory traffic. The vector processor executes much less instructions than the superscalar machine due to the higher semantic content of its instructions. This translates into a lower pressure on the fetch engine and the branch unit. Moreover, the vector model executes less operations than the superscalar machine. The analysis of memory traffic reveals that, in general, and ignoring spill code effects, the vector machine performs less data movements than the superscalar machine.

We have also studied the speculative execution behaviour of superscalar and vector processors. Superscalar processors make an extensive use of speculative execution and the misspeculation rates are important. On the other hand, vector processors execute speculatively by using the vector mask. Vector processors make a lower use of execution under mask and the misspeculation rates are also important, although many of them are produced because of prediction in div instructions.

References

1. Roger Espasa and Mateo Valero. "Multithreaded Vector Architectures". Third Internationl Symposium on High-Performance Computer Architecture, IEEE Computer Society Press, pag. 237-249, San Antonio, TX, February 1997.
2. Roger Espasa and Mateo Valero. "Decoupled Vector Architectures". Second Internationl Symposium on High-Performance Computer Architecture, IEEE Computer Society Press, pag. 281-290, San Jose, CA, February 1996.
3. Luis Villa, Roger Espasa and Mateo Valero. "Effective Usage of Vector Registers in Advanced Vector Architectures". Parallel Architectures and Compilation Techniques (PACT'97), IEEE Computer Society Press, San Francisco, CA, USA, Nov. 1997
4. Roger Espasa, Mateo Valero. "Exploiting Instruction- and Data-Level Parallelism" IEEE Micro, Sept/Oct. 1997 (In conjunction with the IEEE Computer Special Issue on Computing with a Billion Transistors)
5. Roger Espasa and Mateo Valero. "A Victim Cache for Vector Registers". International Conference on Supercomputing, ACM Computer Society Press, Vienna, Austria, July 1997.
6. Roger Espasa, Mateo Valero and James E. Smith. "Out-of-order Vector Architectures". 30th International Symposium on Microarchitecture (MICRO-30) North Carolina, December 1-3, 1997.
7. Norman P. Jouppi and David W. Wall. "Available instruction level parallelism for superscalar and superpipelined machines", ASPLOS, pages 272-282, 1989.
8. M. Peiron, M. Valero, E. Ayguade and T. Lang. "Vector Multiprocessors with Arbitrated Memory Access". In 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligura, Italy, June 22-24, 1995, pp. 243-252.
9. K. Yeager et al. "The MIPS R10000 Superscalar Microprocessor". IEEE Micro, vol 16, No 2, April 1996, pp 28-40.
10. Convex Assembly Language Reference Manuel (C Series). Convex Press, 1991.
11. Charles Price. MIPS IV Instruction Set, revision 3.1. MIPS Technologies, Inc., Mountain View, California, January, 1995.
12. Roger Espasa and Xavier Martorell. "Dixie: a trace generation system for the C3480". Technical Report CEPBA-RR-94-08, Universitat Politecnica de Catalunya, 1994.

13.Doug Burger, Todd M. Austin and Steve Bennett. "Evaluating Future Microprocessors: The SimpleScalar ToolSet". University of Wisconsin-Madison. Computer Sciencies Department. Technical Report CS-TR-1308, July 1996.
14.F. Quintana, R. Espasa and M. Valero. "A Case for Merging the ILP and DLP Paradigms". In 6th Euromicro Workshop on Parallel and Distributed Processing, Madrid, Spain, January 21-23, 1998, pp. 217-224.

Implementing the Time-Warp Simulation Model in Java

Pedro Bizarro, Luís M. Silva, João Gabriel Silva

Departamento de Engenharia Informática
Universidade de Coimbra
POLO II, Vila Franca
3030 Coimbra
PORTUGAL
bizarro@dei.uc.pt, luis@dei.uc.pt, jgabriel@dei.uc.pt

Abstract. This paper presents JWarp, a Java library that implements an optimistic model of discrete-event parallel simulation: the Time-Warp model. Java fits well in the field of simulation and offers some important advantages over other languages: modularity, flexibility, robustness, support for multithreading and exception handling. The paper presents the main features of the library, the programming interface and some of its implementation details. JWarp is one of the first libraries to implement Time-Warp in Java.

1. Introduction

There are several areas like engineering, computer science, economics and military that are particularly interested in using simulation to study the behaviour of complex models. The execution of some of those simulation models can be a very time consuming task. For statistical reasons it might be necessary to simulate a model for quite a long time, or to perform the same simulation several times with different parameter values.

A possible solution to reduce the execution times of long-running simulations is by using multiple processors operating in parallel [Fujimoto90]. A typical simulation model involves several components or entities. By exploiting this inherent model of parallelism it would be possible to speed up the performance of the simulations by decomposing these components through several processors.

Every simulation model is a specification of the corresponding physical model and is composed by a set of states and events. In a discrete event simulation the state of the system only changes at discrete points in simulated time.

A natural decomposition strategy can result in an object-oriented system design, where an object corresponds to some component of the real system and is represented by a computational task that is assigned to a processor for execution. In this way, a logical process (LP) simulates every component of the model. A discrete-event

simulation requires the existence of multiple LP entities, a time-ordered event list holding timestamped events to be processed in the future, a global discrete clock that indicates the current simulation time and a set of state variables that define the state of the simulation.

The most simple way for managing the event-list would be based on a centralized strategy: the list of events is managed by a single process (master), and there would be a pool of slave processes running on the parallel system that would execute those events in a concurrent way. However, the existence of a centralized queue of events would represent a bottleneck to the simulation thereby clearly reducing the potential for parallelism.

The most permissive way of conducting parallel simulations is to eliminate the globally shared-event list and use a completely distributed list of events. Each LP will be assigned to a processor that maintains its own local simulation clock, a local event list and a set of state variables. Events are modeled as timestamped messages, which are exchanged between the physical objects of the application (LP).

However, the schemes that follow a distributed strategy would require some synchronization protocols to make sure the events are processed in a consistent order by all the LP entities. These synchronization protocols may increase the costs of communication between processors. Nevertheless, they have been deserved a considerable attention by the parallel simulation research community [Lin95].

In order to understand the main issue behind the use of distributed event-lists lets take a look at Figure 1. It represents the temporal execution of two logical processes (LP1 and LP2). The LP1 entity is processing event *alpha* while LP2 is processing event *beta*. The execution of event *alpha* generates a new event (*Gama*) that is sent to LP2. This *Gama* event has a lower timestamp than event *beta*, and thus should have been consumed before that one. Due to the asynchrony of the LP entities it was not possible to assure a consistent order in the processing of events, thereby resulting in a *causality error* [Fujimoto90].

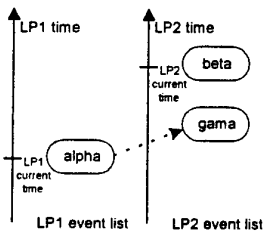


Fig. 1. The causality problem

The synchronization protocols have been broadly classified as *Conservative* or *Optimistic* [Reynolds88]. Both schemes are based on the sending of messages carrying some causality information.

The Conservative approach [Chandy79] strictly avoids the possibility of any causality error ever occurring. This is achieved by stopping each process until the system is sure that no other event will be scheduled by any other LP with a timestamp smaller than the one in the top of the local list of events. This method introduces some blocking on the execution of processes and restricts the potential for parallelism. Besides it is prone to the occurrence of deadlock and thus requires a deadlock detection and recovery scheme.

The Optimistic approach tries to exploit all the potential parallelism available in the simulations. The Time Warp mechanism is a well known optimistic approach

based upon the Virtual Time paradigm [Jefferson85]. It relies upon a scheme for causality error detection and a recovery scheme based on a rollback technique. An optimistic LP progresses simulation and advances its local virtual time as far as simulation is possible without occurring any causality error.

If an event is scheduled in some LP with a timestamp in the local past relative to the local virtual clock, i.e. out of chronological order (*straggler* message), then the LP entity is forced to roll back to the most recently saved state in the simulation history consistent with the arrival of that event message and restarts the simulation at that point thereby correcting the causality error.

In order to allow this rollback operation every LP entity is forced to save its simulation state from time to time. All the messages that were sent previously after that instant of time should be undone. This is achieved by sending some sort of anti-messages to annihilate the original messages. If these ones were already consumed by the destination processes they will be force to roll back as well to a previous saved state. It was proved by [Jefferson85] that the protocol will not roll back until the beginning of the simulation and always assures some forward progress for the computation.

Anti-messages (also called negative messages) are exact copies of normal (positive) messages with a single difference: they have the `sign` field with a different value. When a process sends an anti-message it passes part of its responsibility of rollback to the other process. The other may or may not rollback depending of its internal state: if the message corresponding to the anti-message was already consumed then it must rollback.

The major drawback of the Time Warp approach is the need to save each process state periodically [Jefferson87]. To free up some of the used memory the simulation system calculates a time limit, called Global Virtual Time (GVT) [Belenot90] beyond which no process is required to roll back and thereby the system can perform some garbage collection scheme. Alternative solutions are also required to optimize the rollback operation [Gafni88] and to achieve load-balanced simulations [Das94].

Time Warp is a relatively complex simulation protocol but it has been proved to be a very effective technique for running complex asynchronous simulations [Wieland89][Presley89]. We foresee that with an implementation in Java the use of Time Warp could become more widespread for use by the research community as well as for educational purposes.

2. The Importance of Java

In the past few years, Java as received a great deal of attention from several fields of computing including network and distributed programming.

A comprehensive list of computing platforms has been enhanced with the support of Java Virtual Machine (JVM)[Oasis]. Since Java programs are entirely portable across the systems that have a JVM we will be able to execute parallel simulations in heterogeneous systems, comprising networks of personal computers running a Microsoft Windows operating system or clusters of workstation machines running

some flavor of Unix. All this will be possible with a simulation tool like JWarp. Programmers are not required to change any line of code of their simulations since Java provides the necessary support to deal with the heterogeneity.

The main handicap of Java is still its poor performance. However, recent studies, have proved that the use of JIT, Java as enhanced its performance to the C++ level [Mangione98]. With the foreseed improvement in the JVMs available, Java will close the performance gap even more in the near future.

3. JWarp Internal Architecture

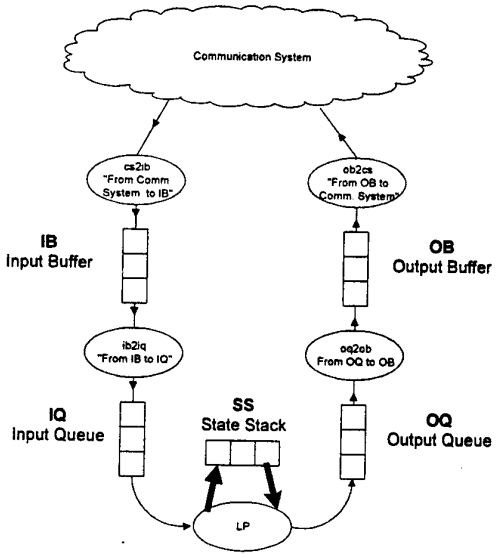


Figure 2 represents the JWarp internal architecture. In this Figure, the ovals represent threads, the rectangles represent data buffers and full lines represent data transfers. In this first approach, only positive messages (thin lines) and state saving and restoring (thick lines) are represented. It will be shown latter all the other kinds of message flows.

Events arrive to every `LP` by being first received in `cs2ib`, placed in `IB`, received in `ib2iq` and placed in `IQ`. Outgoing events are placed in `OQ` by `LP`, received by `oq2ob`, placed in `OB`, received by `ob2cs` and sent into the network.

Fig. 2. JWarp architecture

LP state variables (defined by the programmer) are saved from time to time in the State Stack (`SS`).

The threads `cs2ib`, `oq2ob` and `ob2cs` are just running an infinite cycle fetching data from one side and placing it in the other. Thread `ob2cs` analyses one field of the messages to know where to send them over the network.

Thread `ib2iq` detects the messages out-of-order and causality errors. It will command the state restoring, anti-message sending, and will process GVT calculation requests.

If there were no straggler messages the JWarp internal behaviour would be the following:

1. The message arrives at `cs2ib` from the network through TCP/IP.

2. The message is placed in IB **in arriving order**.
3. It is fetched by ib2iq.
4. A corresponding acknowledge message is put in OB by ib2iq.
5. The acknowledge message is sent by ob2cs.
6. ib2iq puts the received message in IQ **ordered by the simulation time**.
7. Depending on the checkpoint frequency, the LP's state is saved in SS.
8. Just after the state saving the message finally arrives to LP. LVT is updated to a new value that corresponds to the incoming message processing time.
9. LP consumes the message and responds by sending none, one or more messages, to one or more recipients, that are placed in the Output Queue **in arriving order**.
10. The messages are then fetched from OQ and placed in OB by oq2ob.
11. They are finally sent over the network if they are remote events or placed in IQ if they are local events.

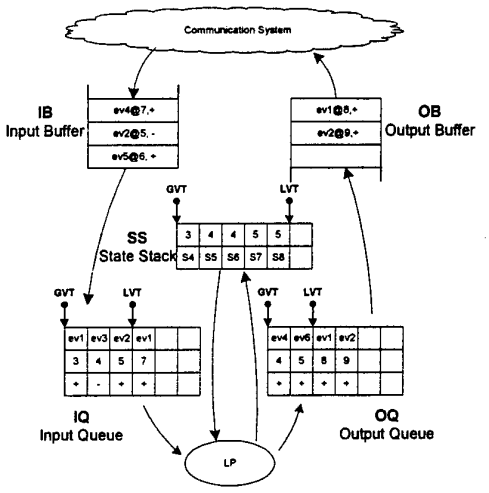


Fig. 3. Buffer behaviour

3.1. Buffers

In JWarp, when a buffer is asked to retrieve the next event it can do one of two things: i) retrieve, return and delete the message or ii) just retrieve and return. Buffers IB and OB delete retrieved messages while IQ and OQ do not. Events are kept and not deleted in IQ and OQ because when there is a rollback operation those events must be consumed again. Likewise, the events that were sent must be maintained because there could be a potential need to send anti-messages. Fetching an event in IQ or OQ only means to retrieve a copy of it and move LVT pointer forward.

Although the pointers are called LVT and GVT they do not store LVT and GVT time values. They are just a reference in the array buffers. Buffers IB and OB do not need to keep any of its messages. All the information needed for a rollback is stored in IQ, OQ and SS between the GVT and LVT pointers.

IQ - Input Queue

In IQ, the events after GVT pointer are the ones that have simulation time bigger than GVT time. Events after LVT pointer are the ones that have simulation time bigger

than LVT time. Thus events after `LVT` pointer have not been processed yet and the ones between `LVT` and `GVT` pointers have been processed but can not be discarded because they might be needed in a rollback situation.

Events in `IQ` are placed in increasing simulation time order. The fetched event is always the one with `(LVT pointer)+1`.

OQ - Output Queue

In `OQ`, the events after `LVT` pointer have not been sent already, and the ones between `GVT` and `LVT` pointers have been sent but can not be deleted because they might be needed for anti-messaging. Note that `IQ`'s `LVT` pointer is directly related with `LVT` value: it defines a frontier splitting events with simulation time smaller than `LVT` from those with simulation time bigger than `LVT`. However, in `OQ`, there is no such relationship. `LVT` pointer is just a frontier splitting sent and unsent events. This means that events in `OQ` are sent as soon as possible even if they have simulation times much bigger than LVT. Events in `OQ` are placed and retrieved in FIFO order.

SS - State Stack

States are saved from time to time and placed in `SS` in FIFO order. There are no state records above `LVT` pointer or below `GVT` pointer as it can be seen in Figure 3.

IB - Input Buffer & OB - Output Buffer

Events are put and get in FIFO order. When an event is get from `IB or OB` it is removed from there.

3.2. Threads

After the initial synchronization phase there will be the following threads: `cs2ib`, `ib2iq`, `LP`, `oq2ob`, `ob2cs` and `GVTmaster`.

cs2ib - From Communication System to Input Buffer

It is only listening for incoming messages. It will receive every kind of message (normal events, acknowledge messages, `GVT start` request and `GVT broadcasts`) and will treat them all with the same procedure: place incoming message in `IB`.

oq2ob - From Output Queue to Output Buffer

Runs an infinite loop fetching from `OQ` and putting in `OB`. This operation, updates automatically `OQ`'s `LVT` pointer

ob2cs - From Output Buffer to Communication System

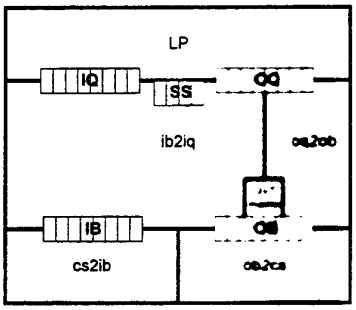
This thread fetches messages from `OB` and if the message is normal message or it is an anti-message, an acknowledge message, or a `GVT report` message, it will peek

into its receiver `ID` field and send it there. If the message is a `GVT start` or `GVT broadcast` message it will send the message to every possible LP over the network.

ib2iq - From Input Buffer to Input Queue
Upon receiving a message it acts as follows:

A. If it receives a normal message the message is placed in `IB` in simulation order, ready for being processed by LP. A corresponding acknowledge message is immediately sent to `OB`.
   A.1. If when trying to place the message it realizes there was a causality error, it initiates the rollback operation. The message is still placed in `IB` in simulation order regardless of the rollback.
   A.2. If there was a negative counterpart in queue then both messages are annihilated and no rollback will happen even if the negative message had past simulation time.
B. If is an anti-message:
   B.1. With a corresponding positive (normal) message in `IB` it annihilates both.
      B.1.1. If the positive message was already consumed then it starts the rollback operation.
   B.2. Without a corresponding positive counterpart then it is just placed in `IQ`.
C. If it is an acknowledge message it will search for the corresponding unacknowledged message in `OQ` and will set its status to acknowledged.
D. If it is a `GVT start` message it will start GVT calculation algorithm which finishes by sending a `GVT report` message to `OQ` and from there to the initiation GVT calculation process.
E. If it is a `GVT broadcast` message, the new GVT will be updated accordingly and the garbage collection will take place.

LP - Logical Process



The programmer's thread is completely unaware of negative and positive message differences, `GVT start`, `GVT report`, `GVT broadcast` and acknowledge messages. All messages received by LP are positive and therefore are treated in the same way. They are fetched from `IQ` and the rest is up to the programmer. When processing the event, none, one or more events may be produced and then placed in `OQ`.

Fig. 4. Thread layers

GVT Master
This thread only exists in one process. From time to time it wakes up and initiates the GVT calculation mechanism by sending a `GVT start` message into buffer `OB`. On

the other side of the buffer, thread `ob2cs` will fetch the message; it will see that it is a message to broadcast and does so.

Layer Relations Between Threads

All JWarp threads and its communication channels (buffers) are represented in Figure 4.

In the Time Warp model every message has at least four fields: `Sender ID`, `Receiver ID`, `Sender LVT`, `Receiver LVT`. `Receiver LVT` is also called simulation time, since the message will be simulated at that particular time.

3.3. Types of Messages

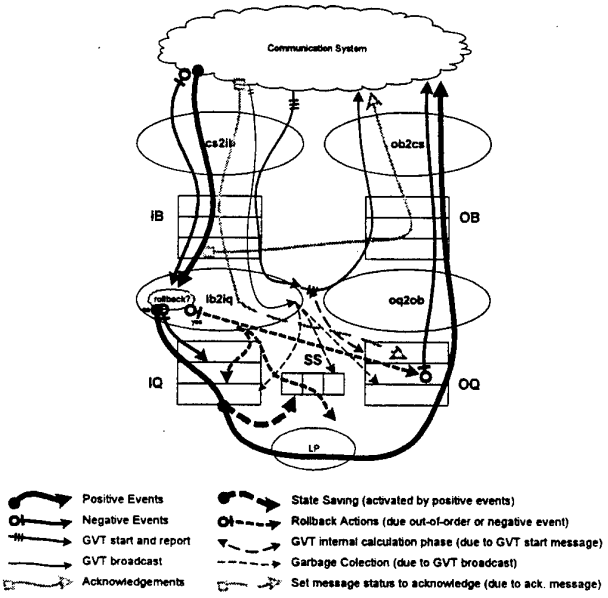


Fig. 5. Message flows - represents every kind of possible message and its consequences. Messages are represented with full lines and actions with dotted lines. Message and action arrows of the same style are cause and consequence.

Positive Messages

As it can be seen, only positive messages arrive to the LP. Just before arriving, the LP's state is saved in `SS`, as it can be seen in Figure 5 by the dotted State Saving line. After arriving to the LP, this message is processed and eventually some more messages are produced and sent to the network. However, if a positive message is

timestamped in the past, a rollback will happen. In a rollback operation, the state variables are restored from SS, the LVT pointer in IQ is adjusted to this state and the LVT pointer in OQ is also adjusted. In OQ the messages that where to the left of LVT pointer and are now to the right must be unsent. For every message in these conditions, a correspondent anti-message is created and is sent while the original to unsent message is deleted. All messages to the right of LVT pointer before adjustment are just deleted.

Negative Messages
If the incoming message is negative it will never get to the LP. Two things may happen: it creates a rollback or it does not produce a rollback. Other Time Warp models allow for a negative message to arrive before its positive part if the underlying communication system allows for out-of-order messages. JWarp uses TCP sockets thus this is guaranteed never to happen. However, if it is possible for a negative message to arrive before the positive, then what is needed to do is simply to place it in IQ and do not allow LP to fetch it. Whenever the positive message arrived both would be annihilated in the buffers.

Acknowledge Messages
When a positive message arrives to `ib2iq` an acknowledge message is produced and placed in OB. When an acknowledge message arrives, `ib2iq` will look for its corresponding message in OQ and change its status to "acknowledged".

GVT Start and GVT Report Messages
When a `GVT start` message arrives, OQ is consulted (GVT Internal Calculation Phase line) in order to obtain the proper values to respond with a `GVT report` message. That message is then sent to back to the master.

GVT Broadcast
Finally, when a `GVT broadcast` message arrives with a new GVT an operation of garbage collection is started which involves removing some data from IQ, OQ and SS.

4. JWarp Interface

Like many simulation languages and environments, the JWarp library offers a event list and functions to fetch and schedule events. Applications built with JWarp should typically run in a cycle fetching one event at a time from the event list and processing that event. The event processing operations may produce zero, one or more events either to be handled by the local processor or by a remote one.

To allow rollback operations, the state variables need to be saved periodically. JWarp offers special classes where the programmer is allowed to define which variables (or objects) make part of the application state and, therefore, which variables have their values restored after a rollback.

At the programming stage, the developer is asked to define the event types that is, the messages to be exchanged between processors at run-time. The programmer must also define which machines and ports will be used in the distributed simulation. The pool of processors is therefore static; removing, adding or changing any of these entries implies a new compilation of the package.

The interface used by the application consists in only a few functions to retrieve events and to schedule events. Network communications, location of other process, the operations of rollback, state saving and state restoring are completely invisible to the application.

4.1. JWarp Programming Example

Let us see through a small example of a Ping-Pong application how to program a JWarp simulation. This example is quite simple: one process sends an event message and the other replies with another event. Figure 6 presents the main Java file (`PingPong.java`) that specifies the LP entities and indicates the mapping of events to the corresponding LPs.

```
 1: package pingpong;
 2: import  jwarp.*;
 3:
 4: class PingPong{
 5:     static JwarpManager sim = new JwarpManager();
 6:
 7:     public static void main (String args[]){
 8:
 9:         Ping  pPing = new Ping("I process ping events");
10:         Pong  pPong = new Pong("I process pong events");
11:
12:         sim.mapsEvent2LP("ping", pPing);
13:         sim.mapsEvent2LP("pong", pPong);
14:
15:         sim.JWInit(args);
16:     }
17: }
```

Fig. 6. The main class that starts the whole simulation (PingPong.java)

The things that are required to do are:

1. First, create a class with a `public static main method`. In this example, this file is `PingPong.java`.
2. Define an `JwarpManager` object that will be responsible by the control of the simulation (line 5).
3. Declare our LP entities: Ping and Pong (lines 9 and 10).

4. Declare which events are handled by the Logical Processes by using the method `mapsEvent2LP` (lines 12 and 13).
5. Start everything with `JWinit(args)` (line 15).

The `main()` class and the help of a configuration class are used by the JWarp package to know which processes should run on which processors, which LPs should be executed and what is the mapping of events to LP entities.

Figure 7 shows the code of one the LP entities (Ping). The other one (Pong) is not shown since it is quite similar. Mainly there are two things that are required for a programmer to do:

1. Define the LP entities which will make part of the simulation. In this particular case they are defined in `Ping.java` and `Pong.java`. These classes are extensions to a JWarp abstract class: `Jwarp_LP`. Since this class `implements Runnable` the programmer must define its code inside the `run` method. These classes are the ones which define the model to be simulated;
2. Receiving and sending messages is accomplished with the `getEvent` and `putEvent` methods.

```
package pingpong;
import  jwarp.*;

public class Ping extends Jwarp_LP{
  public void run(){
    ppMessage pingOut;
    pingOut = new ppMessage( 2, 5, "ping", "pong", "Hi from
Ping!");
      putEvent(pingOut);
      System.out.println("Ping sent message: " + pingOut);
      ppMessage pongIn = (ppMessage) getEvent();
      System.out.println("Pong received message: " + pongIn);
   }
   public Ping(String name) { super(name);}
}
```

Fig. 7. The code of one LP entity that sends a ping event and receives a pong (`Pong.java`)

Finally Figure 8 presents the definition of an event message. The programmer is basically required to:

1. Define the message types necessary to the simulation. In this case we define a single one that must extend the class `Message`, a class belonging to the JWarp package;
2. To print the message contents the programmer may define the `toString` method.

```
package pingpong;
import  jwarp.*;

public class ppMessage extends Message {
   String sentence;

   public ppMessage( long sendingTime, long receivingTime,
                     String sender   , String receiver,
                     String sentence){
      super(sendingTime, receivingTime, sender, receiver);
      this.sentence = sentence;
   }

   public  String  toString(){
      return ("ppMessage " + this.getSendingTime() +
              "-->"        + this.getReceivingTime() +
              ". From: "   + this.getSender() +
              " To: "      + this.getReceiver());
   }
}
```

Fig. 8. Definition of an event message (`ppMessage.java`)

5. Related Work

Several work about the Time Warp model has been presented in the literature [Jefferson85][Fujimoto90][Lin95][Ferscha95]. It was firstly implemented as an operating system - TWOS - in the Jet Propulsion Laboratory [Jefferson87]. Later on, it was ported to several other systems [Fujimoto89][Turner94][Belenot92].

Several parallel simulation languages have also appeared in the last decade: OLPS [Abrams88], Maisie [Bagrodia90], ModSim [West88], SCE [Gill89], Sim++ [Baezer94] and YADDES [Preiss89].

Other approach has been followed by other researchers that chose to implement the parallel simulation system as a run-time library written in C++: examples include WARPED [Martin94], SPEEDES [Steinman91] and HASE++ [Howell97].

Until recently, there only two simulation libraries that were implemented in Java: SimJava [SimJava] and SimKit [SimKit]. However, these libraries only support sequential simulations. This year parallel discrete-event simulation Java libraries appeared: JTED [Cowie98] following the conservative approach and Formax [Halderen98] following a web-based optimistic approach.

6. Conclusions

This paper reports an implementation of the Time Warp model in Java. The library implements all the internal synchronization mechanisms included in that model and provides a very easy-to-use programming interface.

With JWarp it can be possible to execute parallel applications on clusters of workstations and personal computers that have the support of a Java Virtual Machine. Java assures the portability of the programs, solves the problems of heterogeneity and provides a quite flexible programming environment.

It can be used to execute long-running complex simulation models. With the appropriate visualization tools it can also be adopted in the class rooms for the teaching of parallel simulation techniques and concurrent programming.

7. References

[Abrams88] M.Abrams. *"The Object Library for Parallel Simulations (OLPS)"*, Proceedings Winter Simulation Conference, pp. 210-219, San Diego, California, December 1988

[Baezner94] D.Baezner, G.Lomow, B.Unger. *"A Parallel Simulation Environment Based on Time Warp"*, International Journal in Computer Simulation, Vol. 4 (2), pp. 183-207, 1994

[Bagrodia94] R.L. Bagrodia, V.Jha, J.Waldorf. *"The Maisie Environment for Parallel Simulation"*, Proceedings 27th Annual Simulation Symposium, California, pp. 4-12, 1994

[Belenot90] S.Bellenot *"Global Virtual Time Algorithms"*. Proceedings of SCS Multiconference on Distributed Simulation, Vol 22, No 2, pp. 122-127, January 1990.

[Belenot92] S.Belenot. *"A Network Version of the Time Warp Operating System"*, Proceedings of the Workshop on Cluster Computing, Florida, 1992, 1992

[Chandy79] K.M.Chandy, J.Misra. *"Distributed Simulation: A Case Study in Design and Verification of Distributed Programs"*, IEEE Transactions on Software Engineering, Vol. SE-5, No. 5, pp. 440-452, September 1979

[Cowie98] J Cowie. "JTED: Parallel Discrete-Event Simulation in Java", Proceedings of ACM 1998 Workshop on Java for High Performance Network Computing, pp 251-254, February 1998.

[Das94] S.R.Das, R.M.Fujimoto. *"An Adaptive Memory Management Protocol for Time Warp Parallel Simulation"*, Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Nashville, pp. 201-210, 1994

[Ferscha95] A.Ferscha, S.Tripathi. *"Parallel and Distributed Simulation of Discrete-Event Systems"*, Technical Report University of Vienna, 1995

[Fujimoto89] R.M. Fujimoto. *"Time-Warp on a Shared-Memory Multiprocessor"*, Proceedings 1989 International Conference on Parallel Processing, Vol. III, pp. 242-249, August 1989

[Fujimoto90] R.M. Fujimoto. *"Parallel Discrete Event Simulation"*, *Communications of the ACM*, Vol.33, No 10, pp. 30-53, October 1990.

[Gafni88] A.Gafni. *"Rollback Mechanisms for Optimistic Distributed Simulation Systems"*, Proceedings of the SCS Multiconference on Distributed Simulation 19, pp. 61-67, 1988

[Gill89] D.H.Gill, F.X.Maginnis. *"An Interface for Programming Parallel Simulations"*, Proceedings of the SCS Multiconference on Distributed Simulation, Vol. 21 (2), pp. 151-154, Tampa, Florida, March 1989

[Halderen98] B. Halderen, B. Overeinder, "Formax: Web-based Distributed Discrete Event Simulation in Java", Proceedings of ACM 1998 Workshop on Java for High Performance Network Computing, pp 113-122, February 1998.

[Howell97] F.Howell. "*HASE++: A Discrete-Event Simulation Library for C++*", Technical Report Department of Computer Science, University of Edinburgh, Available at: http://www.dcs.ed.ac.uk/home/hase/projects/hase++.html

[Oasis] The Java Oasis, "The Java Oasis: Java Developers Kit (JDK) - Index", http://www.oasis.leo.org/java/development/jdk/00-index.html

[Mangione98] C. Mangione, "Performance Tests Show Java as Fast as C++". Java World, http://www.javaworld.com, February 1998

[SimJava] SimJava, http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.1/

[SimKit] SimKit, http://www.cpsc.ucalgary.ca/~adi/simkit/workshop/index.htm

[Jefferson85] D.R.Jefferson. "*Virtual Time*". ACM Transactions on Programming Languages and Systems, Vol. 7, No 3, pp. 404-425, July 1985, pp. 404-425.

[Jefferson87] D.Jefferson, *et al.* "*Distributed Simulation and the Time Warp Operating System*". Proceedings of 11th ACM Symposium on Operating Systems Principles, Vol 21, No 5, pp 77-93, November 1987.

[Lin95] Y.B.Lin, P. Fishwick. "*Asynchronous Parallel Discrete Event Simulation*", IEEE Transactions on Systems, Man and Cybernetis, Vol. 26, No. 4, pp. 397-412, 1995

[Martin94] D.Martin, P:Wilsey, T.McBrayer. "*The WARPED Time-Warp Simulation Kernel*", Technical Report University of Cincinnati, USA, 1994

[Naughton96] P. Naughton. "*The Java Handbook*". Osborne McGraw-Hill, 1996.

[Preiss89] B.R.Preiss. "*The YADDES Distributed Discrete-Event Simulation Specification Language*", Proceedings of the SCS Multiconference on Distributed Simulation, Vol. 21 (2), pp. 139-144, Tampa Florida, March 1989

[Presley89] M.Presley, M.Ebling, F.Wieland, D.Jefferson. "*Benchmarking the Time Warp Opearing System with a Computer Network Simulation*", Proceedings SCS Distributed Simulation Conference, pp. 8-13, 1989

[Reynolds88] J.P.Reynolds. "*A Spectrum of Options for Parallel Simulation*", Proceedings Winter Simulation Conference, San Diego, California, pp. 325-332, December 1988

[Steinman91] J.S.Steinman. "*SPEEDES: Synchronous Parallel Environment for Emulation and Discrete-Event Simulation*", Proceedings of Parallel and Distributed Simulation Conference, pp. 95-103, 1991

[Turner94] S:Turner. "*Distributed Simulation with a Transputer Version of the Time Warp Operating System*", Proceedings Transputers'94, Besancon, France, pp. 39-54, September 1994

[West88] J.West, A.Mullarney. "*ModSim: A Language for Distributed Simulation*", Proceeedings of the 1988 SCS Multiconference on Distributed Simulation, Vol. 4(2), pp. 235-257, 1994

[Wieland89] F.Wieland, L. Hawley, A.Feinberg, M. DiLoreto, L.Blume, *et al.* "*The Performance of Distributed Combat Simulation with the Time Warp Operating System*". Concurrency: Pratice and Experience, Vol 1(1), pp. 35-40, September 1989.

Evaluation of High Performance Fortran for an Industrial Computational Fluid Dynamics Code

Thomas Brandes[1], Falk Zimmermann[1], Christian Borel[2], Marc Brédif[2]

[1] Institute for Algorithms and Scientific Computing (SCAI)
German National Research Center for Information Technology (GMD)
Schloß Birlinghoven, D-53754 St. Augustin, Germany
{Thomas.Brandes, Falk.Zimmermann}@gmd.de
[2] Service Aerodynamique Numerique PV21
MATRA BAe Dynamics France, CFD Group
37, avenue Lois Bréguet BP 1,
F-78146 Vélizy-Villacoublay Cedex, France
{cborel, mbredif}@matra-def.fr

Abstract. The PHAROS project, funded by the European Unions ESPRIT program for research and development in information technology, aimed to assess High Performance Fortran (HPF) as a paradigm for porting large FORTRAN 77 scientific applications to distributed memory architectures, in comparison to message-passing programming. The AEROLOG computational fluid dynamic software developed by MATRA was one of these applications that has been ported to HPF. It is devoted to the study of compressible fluid flows around complex geometries. This paper describes the port of the AEROLOG code to HPF based on the decomposition of subdomains. It outlines the parallelization strategy, the changes of the data structures and the tuning of the boundary conditions for the subdomains. Performance results for industrial test cases with different HPF compilers are given and compared with the results of the message-passing version.

1. Introduction

High Performance Fortran (HPF) [7,8] is a data parallel, high level programming language for parallel computing that is expected to be more convenient in terms of portability and maintainability than explicit message passing and to allow higher productivity in software development. But the porting of key commercial applications to HPF is still of critical importance for the continuing development and acceptance of HPF as a standard and for the improvement of HPF compilers.

The ESPRIT project "Open HPF Programming Environments" (PHAROS) aimed to assess HPF as a paradigm for porting large FORTRAN 77 scientific applications to shared and distributed memory architectures, in comparison to message-passing programming. The PHAROS project was funded by the European Union's ESPRIT pro-

gram for research and development in information technology. It was a two years project, running from January 1996 until December 1997.

To this end, four major commercial FORTRAN 77 application codes have been successfully ported to HPF (structural analysis, CFD and electromagnetism application codes). These codes already had message-passing parallel versions. The comparison of HPF to message-passing considered factors such as:

- the porting effort;
- the performance of the resulting code;
- the portability and maintainability of the resulting code.

One of the PHAROS applications was the AEROLOG computational fluid dynamics software of MATRA BAe Dynamics [1,5]. The AEROLOG code is a proprietary CFD software devoted to the study of compressible fluid flows around complex geometries. For many years, it has been systematically applied during the aeronautical development programs MISTRAL, MICA, and APACHE, reducing the experimental studies and consequently cutting down costs and delays.

Together with HPF experts and HPF tool providers, the version AEROLOG-v3.2e (e for Euler inviscid fluid model) has been ported to HPF. It is an industry relevant subset of the latest release AEROLOG-v3.2 which includes all the functionalities used in today's applications. This full release AEROLOG-v3.2 is composed of 102 subroutines with about 19000 lines of source code written in standard FORTRAN 77. The reduced code AEROLOG V3.2e is composed of 55 subroutines with about 11500 lines of source code. It is equivalent to the content of the message passing version of the AEROLOG code.

In accordance to the workplan of the PHAROS project, the rest of this paper is organized as follows. Section 2 describes the AEROLOG software and section 3 outlines the initial port to HPF. The code review in section 4 identified the problems of the initial version and resulted in the code tuning presented in section 5. We discuss our expectations for the next generation of HPF compilers in section 6 to overcome the still existing problems. Finally, we compare in section 7 the results of the HPF versions with the MPI version and conclude in section 8.

2. Description of the AEROLOG Software and the Test Cases

2.1 The AEROLOG CFD Code

The AEROLOG-v3.2 code allows the simulation of steady or unsteady inviscid and compressible fluid flows over three-dimensional geometries by solving the Euler system of partial differential equations. It utilizes an explicit time integration scheme of Lax-Wendroff type. It is second order accurate in time and allows steady flow simulation with the local time-stepping technique or unsteady simulation with a uniform time

step limited by the so-called CFD stability condition. Another functionality is the finite volume space integration scheme. It is a three-dimensional extension of the cell vertex Ni scheme [4] which is second order accurate in space on a Cartesian grid. The formulation is fully conservative so that shock and expansion waves are automatically captured.

The data layout is based on a multidomain meshing strategy. The global mesh is composed of an assembly of locally structured three-dimensional mesh blocks (I, J, K, families of mesh lines). All types of degenerations are allowed on the mesh boundaries, like mesh plans degenerating into a mesh line or a point. This is very useful for the meshing of complex geometries, but it requires the implementation of the convenient matching conditions.

The most time consuming part of the code is the subroutine that computes the time increments for the physical variables at each time step. It is composed of a succession of calls to subroutines that can be sorted into two groups:

- The "local" routines are called independently over subdomains. This group uses typically up to 90% of the total CPU time within the FORTRAN 77 code.
- The "boundary" routines perform the boundary conditions: flow conditions (inflow, outflow, walls, etc.) and numerical matching conditions (interfaces between subdomains). These routines make an intensive use of indirect addressing and involve dependencies between data belonging to different subdomains.

2.2 CFD Test Cases

Three meshes of increasing sizes (see Table 1) have been build around the same geometry of blunt body. The free flow conditions are: Mach number equal to 2.96 and angle of attack equal to 10o. With these conditions, the fluid flow over the blunt body shows strong shock and expansion waves: characteristic of MATRA industrial applications.

Test Case Name	Sizes and Number of Subdomains	Total Mesh Points	Processing Nodes
Small	[illegible] x 9) x 8	154440	1-8
Medium	[illegible] x 9) x 8	304200	2-16
Large	[illegible] x 9) x 8	603720	4-64

Table 1. Industrial Test Cases.

Due to the coarse grain parallelization strategy over subdomains (see next section), the number of HPF processes is limited by the number of subdomains. In order to run a number of HPF tasks higher than the initial number of subdomains, a pre-processor can be applied [6]. This pre-processor takes as input a mesh file with its associated topological description and generates automatically a new mesh data set with respect to the three following constraints :

- generate the given number of mesh blocks,
- optimize the load balancing (i.e. homogeneous mesh block sizes),
- minimize the size of blocks interfaces.

3. Initial Port to HPF

3.1 Porting to Fortran 90

Initially, the code was ported from FORTRAN 77 to Fortran 90. Apart from inserting F90 syntax, e.g. array syntax and interface blocks, we replaced the old one dimensional "work-array" with dynamic arrays. Some of the arrays became allocatable arrays, other ones, especially for local data, became automatic arrays. These changes made the code more flexible as the static size of the workspace is no longer given. But they were also absolutely necessary to allow the HPF distribution of the mesh data in a useful way.

The porting to Fortran 90 was supported by the Foresys (FORtran Engineering SYStem) tool from SIMULOG [11] that is a reverse-engineering, migration and development support system for Fortran. It was especially useful to generate interface blocks with intentions for the dummy arguments, and to take advantage of new syntax and new language features.

3.2 Coarse Grain Parallelization Strategy

For the HPF parallelization we have chosen the following strategy:

- The loops over the different subdomains calling the local routines provide *coarse grain* parallelism without communication. The HPF mapping directives have to guarantee that all data belonging to one subdomain is completely mapped to the same processor.
- The matching conditions of the interfaces of adjacent subdomains are difficult to handle. The initial strategy was the replication of the data and computations involved in the boundary conditions.

In the AEROLOG code, the data of the two- or three-dimensional subdomains is linearized and stored in a one-dimensional array. For using the coarse grain parallelism, it is essential that we can distribute the data in the program in such a way that one subdomain is completely owned by the processor that will work on this data. As the subgrids have different sizes, it would be necessary that HPF supports generalized block distributions where the user can pass to the compiler the corresponding block

sizes for each processor. Unfortunately, none of the commercial HPF compilers supported this feature already during the project time. For this reason, we had to reorganize the one-dimensional mesh data arrays to two-dimensional ones. The second dimension corresponds to the subdomains numbering and will be distributed by BLOCK (see **Fig. 1**). This imposed significant changes to the code, not only for the arrays containing mesh data, but also for all integer arrays used for indirect addressing of mesh data.

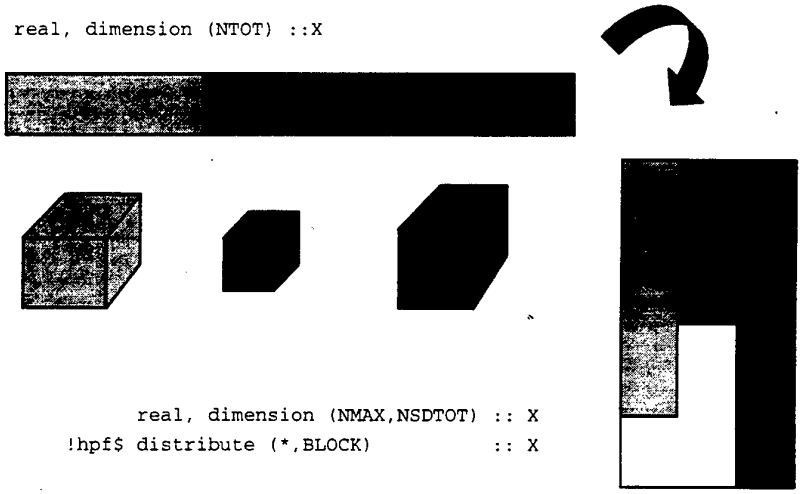


Fig. 1. New data structures for the mesh data and their distribution.

3.3 Coarse Grain HPF Implementation

With the help of the INDEPENDENT directive, we enabled the parallelization of the loops over the subdomains. The local routines are defined as PURE routines to allow their parallel execution for the different subdomains (see **Fig. 2**). Furthermore, the local routines have not to be parallelized at all and do not need any HPF directive.

The AEROLOG code takes advantage of sequence association. The subdomains are implicitly reshaped within the local subroutines. Within one subroutine, one subdomain is always considered as a three-dimensional rectangular grid. Though the HPF standard does not allow sequence association for mapped arguments, we could rely on it as long as it is only used for a single subdomain that is completely mapped to one processor.

For the boundary routines, the values of the boundary nodes of the different subdomains are gathered from the distributed mesh data. For the initial HPF port, the data and computations are replicated on all nodes and every processor updates the values of its boundary mesh nodes. The gathering of the distributed data is realized by replica-

tion of the whole mesh data. As the mesh data is not distributed within the boundary routine, implicit remapping at subroutine boundaries is utilized (see **Fig. 3**).

```
      integer  :: NSDTOT     ! total number of subdomains
      integer  :: NMAX       ! maximal size of one subdomain
      real, dimension (NMAX,NSDTOT) :: F !  mesh data, e.g. force
!hpf$ distribute F (*,block)              !  distribute the subdomains
      integer, dimension(NSDTOT) :: IM, JM, KM ! sizes
      ...
!hpf$ independent
      do NSD = 1, NSDTOT
        call LOCAL_ROUTINE (F(1,NSD),IM(NSD),JM(NSD),KM(NSD), ...)
      end do
      call BOUNDARY (F, NMAX, NSDTOT, ...)
      ....

      pure subroutine LOCAL_ROUTINE (F,IM,JM,KM,...)
      integer, intent(in) :: IM, JM, KM
      real, dimension(IM,JM,KM), intent(inout) :: F
      ...
      end subroutine LOCAL_ROUTINE
```

Fig. 2. Outline of the initial HPF AEROLOG Code.

```
      subroutine BOUNDARY (F, NMAX, NSDTOT, ISD_B, IJK_B, NB)
      integer, intent (in)     :: NMAX, NSDTOT, NB
      integer, dimension (2, NB), intent (in)  :: ISD_B, IJK_B
      real, dimension (NMAX, NSDTOT), intent (inout) :: F
!hpf$ distribute F(*,*)                  ! replicated mesh data
      real    :: X1, X2, X
      integer :: IB, IJK1, ISD1, IJK2, ISD2
      do IB=1,NB
        IJK2 = IJK_B(2,IB); ISD2 = ISD_B(2,IB)
        IJK1 = IJK_B(1,IB); ISD1 = ISD_B(1,IB)
        X = (F(IJK1,ISD1) + F(IJK2,ISD2)) * 0.5
        F (IJK1,ISD1) = X; F(IJK2,ISD2) = X
      end do
      end subroutine
```

Fig. 3. Computation of boundary conditions in the AEROLOG code.

4. Code Review

We tested the initial HPF port with the following compilers:

- NAS HPFPlus by NA Software Liverpool, Release 2.01, a commercial HPF compiler that was also the target compiler for all HPF codes in the PHAROS project;
- PG HPF by Portland Inc., Oregon [9], AIX Rel. 2.2-1, another commercial HPF compiler;
- ADAPTOR HPF compiler, version 5.1 (Oct. 1997) [3], developed at SCAI in GMD, a research compiler that is available as public domain.

All results have been measured on the IBM SP2 at the GMD. We give the execution times in seconds for 5 iterations on the 'small' test case that works on 8 subdomains, every subdomain contains *65 x 33 x 9* mesh points (see also Table 1).

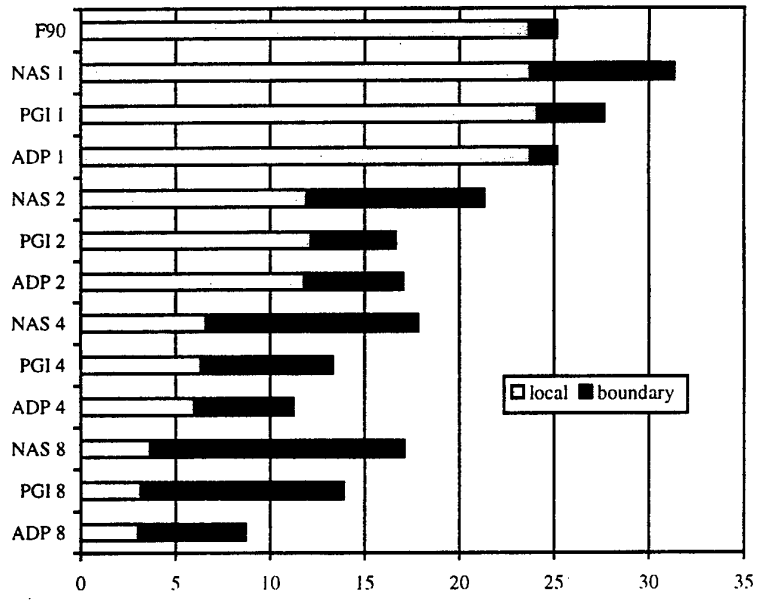


Fig. 4. Execution times (in seconds) of initial HPF version.

Fig. 4 shows the execution times of the initial HPF version, compiled by the native Fortran 90 compiler (xlf) and by the different HPF compilers running on 1, 2, 4, and 8 processors. The HPF version (considered as a Fortran 90 version without directives) has nearly the same performance as the original FORTRAN 77 version. The execution times, separated for the local and boundary routines, show that the local routines are parallelized perfectly. They scale well and the HPF parallelization causes no overhead.

But all boundary routines do not scale. In contrary, the execution time increases with the number of processors. This is due to the replication of distributed data that involves an all-to-all communication. Furthermore, it shows that the compilers have already different support for this kind of structured communication that follows a fixed communication pattern where every processor knows which data has to be sent and to be received.

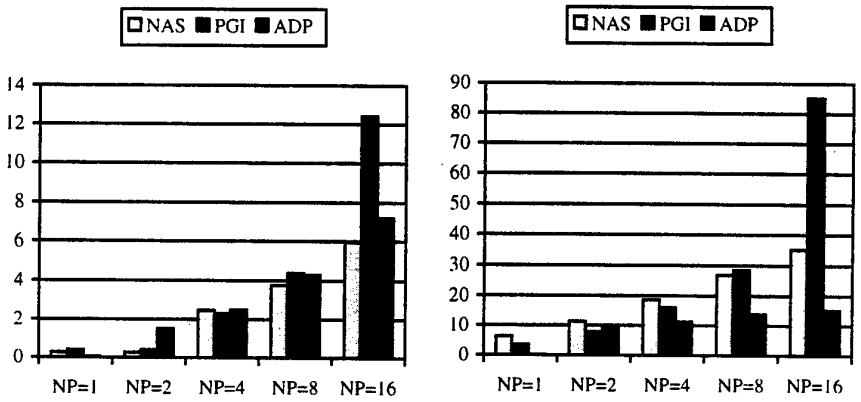


Fig. 5. Replication of distributed data.

In order to estimate the cost of replications, we benchmarked a sample code that performs only data replications of arrays with varying sizes. **Fig. 5** shows how much time (in milliseconds) the replication of distributed data needs for the different compilers and for the different number of processors. Array sizes of 16 Kbytes and 128 Kbytes are considered. The time for replicating distributed data increases with the size of the array and with the number of processors. The ADAPTOR runtime system is able to recognize at runtime that the replication of distributed data on a single processor does not require any copying at all.

5. Tuning of the HPF Code

As the results of the initial HPF port show, only the tuning of the boundary routines is necessary. We considered two strategies:

- We let the mesh arrays distributed for the boundary routines and relied on the capabilities of the HPF compiler to deal with unstructured communication. Unfortunately, all HPF compilers failed to generate more efficient code than for the initial

HPF version. Especially the NAS HPFPlus compiler provided absolutely no efficient support for indirect addressing.

- Instead of replicating the whole array containing the mesh data, we compressed the full mesh data to the boundary data before replicating it. This solution needed new data structures for packing and unpacking of boundary data (see **Fig. 6**). The packing of the data can be done independently for all subdomains. This approach required only HPF features that were supported by all HPF compilers.

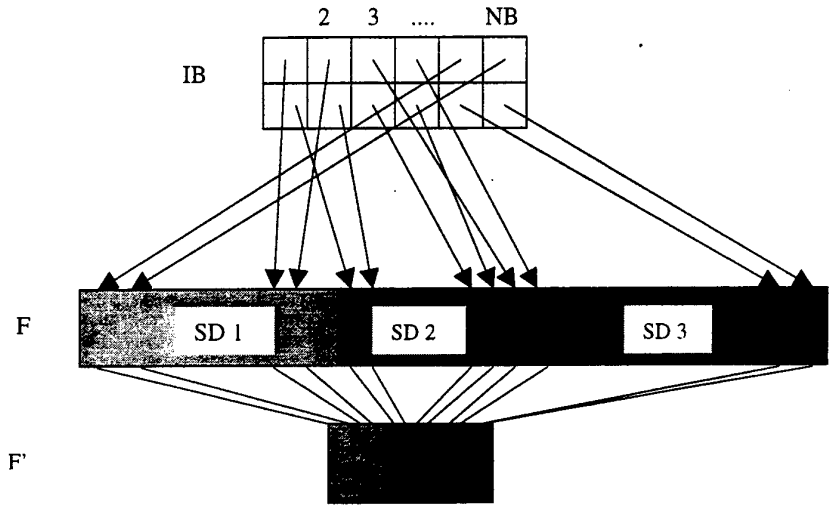


Fig. 6. Packing of boundary data.

By the packing of the boundary data, much less data has to be replicated between the different processors. Only the boundary data and not the whole mesh data is exchanged between the processors. The results shown in **Fig. 7** verify the effectiveness of the chosen approach. Compared to the sequential version, speedups from 4 to 5 on 8 processors are achieved.

6. Expectations for the Next Generation of HPF Compilers

The current tuned HPF version is still not fully portable between the different HPF compilers as the calling of local subroutines within an independent loop is supported differently. The NAS HPFPlus compiler did not act at all upon the `INDEPENDENT` directive, but scheduled the local computations, defined as `HPF_SERIAL` routines and not as `PURE` routines, on the processors owning the subdomain. The need for the

slightly different versions of the HPF code should become redundant with the next releases of the HPF compilers.

Considering HPF 2.0 [8], we expect support for general block distributions. This would avoid the additional dimension for the subdomains and there would be no more wasting of memory in case of different subdomain sizes (see also **Fig. 1**). This feature is absolutely necessary to combine the evolution of the serial and the HPF version of the AEROLOG code.

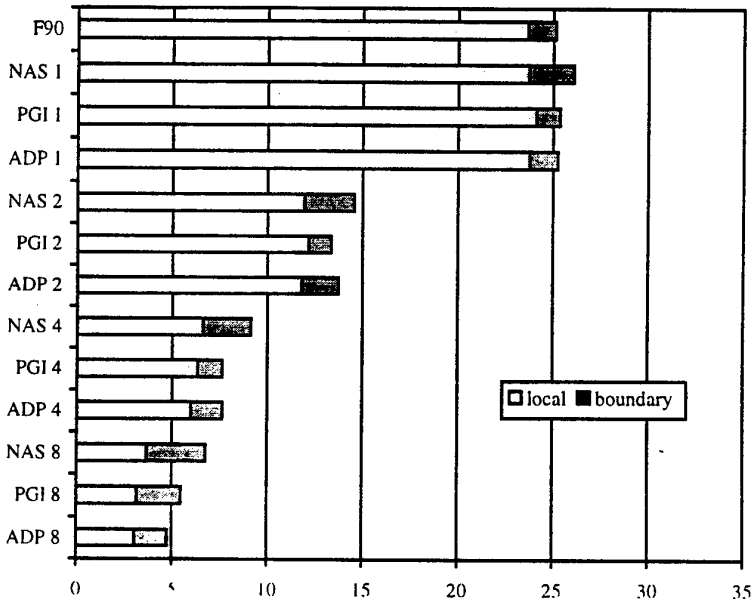


Fig. 7. Execution times (in seconds) of tuned HPF version.

The Amdahl limit restricts the maximum speed-up as long as the boundary computations are not parallelized. But then unstructured communication has to be supported. Therefore the compiler has to build a schedule required for accessing remote items of distributed arrays and for communication optimization. Unfortunately, schedules cannot be worked out at compile time, but only at run-time, when the values of the indirection arrays are known. The code design which first builds the schedule, then uses it to carry out the actual communication and computation, has been coined as the *inspector/executor* scheme [10]. The PGI and ADAPTOR compiler followed this design, but only the latest release of ADAPTOR tool provided sufficient support for reusing communication schedules by an additional directive [2].

Fig. 8 presents results for the 'medium' test case (100 iterations), for the local routines (local) and for different implementations of the boundary routines. The replication strategy of the initial and tuned HPF version does not scale. The unstructured communication (unstr.) for the parallelized boundary computations scales, but produces an unacceptable overhead due to the high costs for building the communication schedule. If the communication schedule can be reused, e.g. by tracing modifications of the indirection array as described in [2], the unstructured communication produces good results (traced).

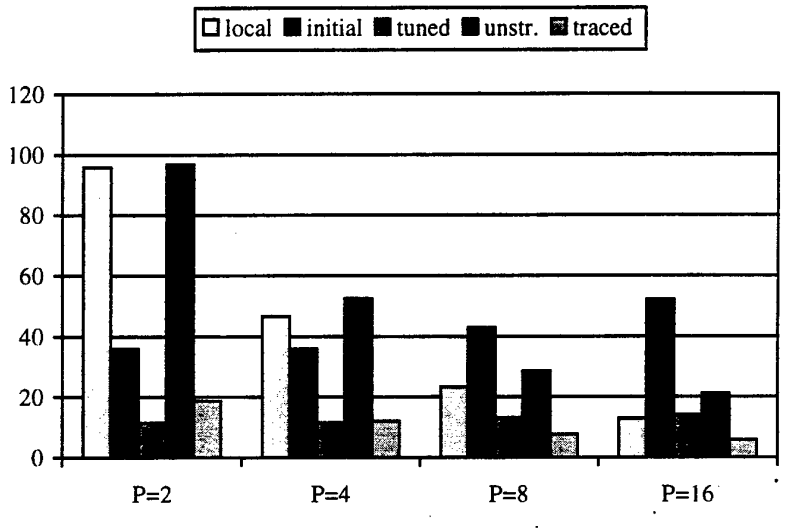


Fig. 8. Different tuning strategies of the boundary computations (ADAPTOR).

7. Benchmarking and Comparison with Message Passing

The porting of the AEROLOG code to HPF required important code changes as well as the message passing port, but it could be done step by step, always having a running version. This porting included useful code cleaning and modernized memory management. The replacement of the super-array technique in favor of Fortran 90 dynamic allocation of the local arrays brings simplification and flexibility to the code. But many code changes were only required due to the limited capabilities of the HPF compilers.

The HPF and the message passing version achieve nearly the same performance for smaller number of processors. But the message passing version of the AEROLOG code scales better (see **Fig. 9**). It parallelizes also the boundary routines and takes advantage of reusing explicitly communication schedules. But with an HPF compiler that supports unstructured communication and reuses schedules the scalability of the

MPI version can be nearly achieved as the results with the ADAPTOR compilation system verify.

From the code development and maintenance point of view, it is possible to replace smoothly the FORTRAN 77 reference code in favor of the Fortran 90/HPF code. The benefits of this migration will be the merging of the sequential/parallel shared/distributed memory versions of AEROLOG, which have reached very different levels of development at the moment. The migration of the complete AEROLOG code (implicit solver, Navier-Stokes solver, etc.) is eased by the choice of the coarse grain parallelization strategy based on the multidomain approach. In particular, it is not necessary to rewrite the local algorithms which can remain FORTRAN 77, saving a lot of porting efforts and bug risks. Experimental results with the ADAPTOR compiler have also shown that the HPF version is well suited for shared memory architectures by translating the HPF directives into parallelization directives for the native compiler. Due to the shared memory, runtime support for unstructured communication is not necessary.

In any case, we have seen from the PHAROS final benchmarks that it is not possible at the moment to get rid of the message-passing version of the code, which is the only one able to run efficiently enough on massively parallel computers. Enhancements of the HPF compiler technology are still required to a complete migration to HPF.

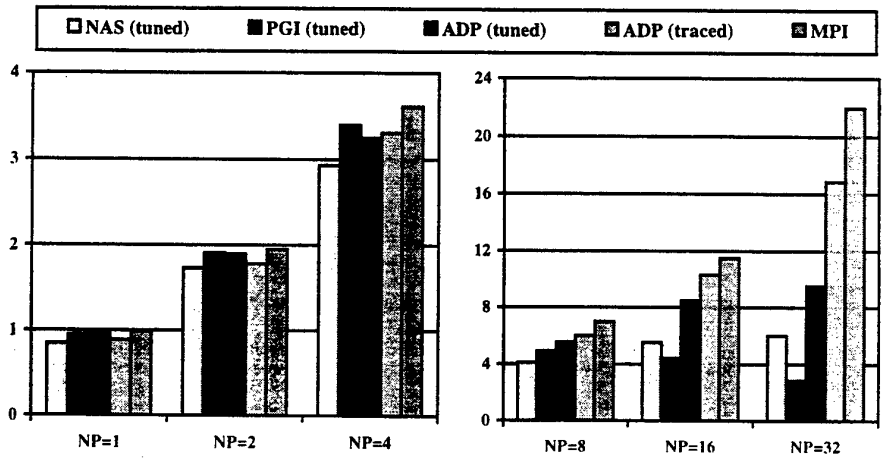


Fig. 9. Speedups on industrial test cases.

8. Conclusions

With the end of the PHAROS project, we have an HPF version of the AEROLOG code that runs with at least three HPF compilers and produces acceptable results for a limited number of processors. The porting effort was higher than expected because code restructuring was required in order to achieve the HPF implementation of the coarse grain parallel strategy. As HPF concepts are rather complicated for non specialists, the know-how transfer from tool providers and experts to the end-user was very important and might be considered as a major benefit of the PHAROS project.

At this time, the code is not fully portable as different language features are used for the two commercial HPF compilers. This is not only due to the missing support in the compilers, but also due to fact that the HPF standard was not not rigid enough, so that HPF directives led to various interpretations. With future releases of the HPF compilers, these problems will disappear. The tuning of the boundary conditions required a lot of effort. This effort might be less with advanced HPF compilers where unstructured communication is better supported.

The HPF version can directly be compiled for a serial machine achieving the same performance than the original code. While the independent computations over the subdomains scale well, the boundary conditions remain the critical part, even in the tuned version. Replication of mesh data is rather expensive, unstructured communication is not well supported. Due to the replication of the boundary computations, the scalability of this version is limited in any case.

Nevertheless, experimental results with the research compilation system ADAPTOR verify that a scalable and efficient HPF parallelization of the AERLOG software is possible if general block distributions and unstructured communication are sufficiently supported.

In conclusion of this project, we can state that HPF is a useful paradigm for porting large FORTRAN 77 applications to parallel architectures and in the long run the better alternative. But an efficient and portable parallelization and a higher productivity in software development can only be achieved if HPF compilers improve substantially.

Acknowledgements

The support of the PCF-CFORN group (SCAI, GMD) for access to the IBM SP2 at the GMD is appreciated

References

1. Borel, C., Brédif, M.: High Performance Parallelized Implicit Euler Solver for the Analysis of Unsteady Aerodynamic Flows. In Eccomas'92 : Conference Proceedings, pp. 1069-1076. Elsevier Science Publisher. 1992.
2. Brandes, Th., Germain, C.: A Tracing Protocol for Optimizing Data Parallel Irregular Computations. Accepted for publication at EuroPar'98, Southampton, Sep. 1998.
3. Brandes, Th., Höver-Klier, R.: ADAPTOR User's Guide (Version 5.1). Technical documentation, GMD, Oct. 1997. Available via anonymous ftp from ftp.gmd.de as gmd/adaptor/docs/uguide.ps.
4. Brédif, M., Chattol J., Koeck, P., Werlé, C: Simulation d'un système de Déviation de Jet à l'Aide des Équations d'Euler. *AGARD CP*, 412:13.1--13.11, 1986.
5. Brédif, M., Chapin, F., Borel, C., Simon, P.: Industrial Use of CFD for Missile Studies: New trends at MATRA BAe Dynamics France, in NATO-AVT panel on Missile Aerodynamics (11-15 May 98, Sorrento, Italy). Conference Proceedings, to be published.
6. Choukroun, F., Roux, F.X., Borel, C., Brédif, M.: Implementation of an Industrial CFD Code on a Massively Parallel Computer with Distributed Memory. In Parallel CFD'93 : New Algorithms and Applications, pp. 271-276, Elsevier Science, 1995.
7. High Performance Fortran Forum: High Performance Fortran Language Specification, Version 1.1, Department of Computer Science, Rice University, Nov. 1994.
8. High Performance Fortran Forum: High Performance Fortran Language Specification, Version 2.0, Department of Computer Science, Rice University, Jan. 1997.
9. PGHPF: Reference Manual, User's Guide, Technical Report, The Portland Group, Inc., Oregon, Nov. 1994.
10. Mirchandaney, R. et al: Principles of run-time support for parallel processing. In ACM Int. Conf. on Supercomputing, pages 140--152, 1988.
11. Foresys: SIMULOG. Foresys 2.0 User's Guide. Technical report, SIMULOG, 1998. http://www.simulog.fr/foresys.

Automatic Detection of Parallel Program Performance Problems[1]

A. Espinosa, T. Margalef, E. Luque.

Computer Science Department
Universitat Autònoma de Barcelona.
08193 Bellaterra, Barcelona, SPAIN.
e-mail: iinfd@cc.uab.es

Abstract. *Actual behaviour of parallel programs is of capital importance for the development of an application. Programs will be considered matured applications when their performance is under acceptable limits. Traditional parallel programming forces the programmer to understand the enormous amount of performance information obtained from the execution of a program. In this paper, we propose an automatic analysis tool that lets the programmers of applications avoid this difficult task. This automatic performance analysis tool main objective is to find poor designed structures in the application. It considers the trace file obtained from the execution of the application in order to locate the most important behaviour problems of the application. Then, the tool relates them with the corresponding application code and scans the code looking for any design decision which could be changed to improve the behaviour*

1. Introduction:

The performance of a parallel program is one of the main reasons for designing and building a parallel program [1]. When facing the problem of analysing the performance of a parallel program, programmers, designers or occasional parallel systems users must acquire the necessary knowledge to become performance analysis experts.

Traditional parallel program performance analysis has been based on the visualization of several execution graphical views [2, 3, 4, 5]. These high level graphical views represent an abstract description of the execution data obtained from many possible sources and even different executions of the same program [6].

---

[1] This work has been supported by the CICYT under contract TIC 95-0868

The amount of data to be visualized and analyzed, together with the huge number of sources of information (parallel processors and interconnecting network states, messages between processes, etc.) make this task of becoming a performance expert difficult. Programmers need a high level of experience to be able to derive any conclusions about the program behaviour using these visualisation tools. Moreover, they also need to have a deep knowledge of the parallel system because the analysis of many performance features must consider architectural aspects like the topology of the system and the interconnection network.

In this paper we describe a Knowledge-based Automatic Parallel Program Analyser for Performance Improvement (KAPPA-PI tool) that eases the performance analysis of a parallel program. Analysis experts look for special configurations of the graphical representations of the execution which refer to problems at the execution of the application. Our purpose is to substitute the expert with an automatic analysis tool which, based on a certain knowledge of what the most important performance problems of the parallel applications are, detects the critical execution problems of the application and shows them to the application programmer, together with source code references of the problem found, and indications on how to overcome the problem.

We can find other automatic performance analysis tools:

-Paradyn [7] focuses on minimising the monitoring overhead. The Paradyn tool performs the analysis "on the fly", not having to generate a trace file to analyse the behaviour of the application. It also has a list of hypotheses of execution problems that drive the dynamic monitoring.

- AIMS tool [8], is a similar approach to the problem of performance analysis. The tool builds a hierarchical account of program execution time spent on different operations, analyzing in detail the communications performed between the processes.

-Another approach to addressing the problem of analysing parallel program performance is carried out by [9] and [10]. The solution proposed is to build an abstract representation of the program with the help of an assumed programming model of the parallel system. This abstract representation of the program is analysed to predict some future aspects of the program behaviour. The main problem of this approach is that, if the program is modelled from a high level view, some important aspects of its performance may not be considered, as they will be hidden under the abstract representation.

- Performance of a program can also be measured by a pre-compiler, like Fortran approaches (P3T [11], this approach is not applicable to all parallel programs, especially those where the programmer expresses dynamic unstructured behaviour.

Our KAPPA-PI tool is currently implemented (in Perl language [12]) to analyse applications programmed under the PVM [13] programming model. The KAPPA-PI tool bases the search for performance problems on its knowledge of their causes. The analysis tool makes a "pattern matching" between those execution intervals which degrade performance and the "knowledge base" of causes of the problems. This is a process of identification of problems and creation of recommendations for their solution. This working model allows the "performance problem data base" to adapt to new possibilities of analysis with the incorporation of new problems (new knowledge data) derived from the experimentation with programs and new types of programming models.

In section 2, we describe the analysis methodology briefly, explaining the basis of its operations and the processing steps to detect a performance problem. Section 3 presents the actual analysis of a performance problem detected in an example application. Finally, section 4 exposes the conclusions and future work on the tool development.

2.- Automatic analysis overview.

The objective of the automatic performance analysis of parallel programs is to provide information regarding the behaviour of the user's application code.

This information may be obtained analysing statically the code of the parallel program. However, due to the dynamic behaviour of the processes that form the program and the parallel system features, this static analysis may not be sufficient.

Then, execution information is needed to effectively draw any conclusion about the behaviour of the program. This execution information can be collected in a trace file that includes all the events related to the execution of the parallel program. However, the information included in the trace file is not significant to the user who is only concerned with the code of the application.

The automatic performance analysis tool concentrates on analysing the behaviour of the parallel application expressed in the trace file in order to detect the most important performance problems. Nonetheless, the analysis process can not stop there and must relate the problems found with the actual code of the application. In this way, user receives meaningful information about the application behaviour.

In figure 1, we represent the basic analysis cycle followed by the tool to analyse the behaviour of a parallel application.

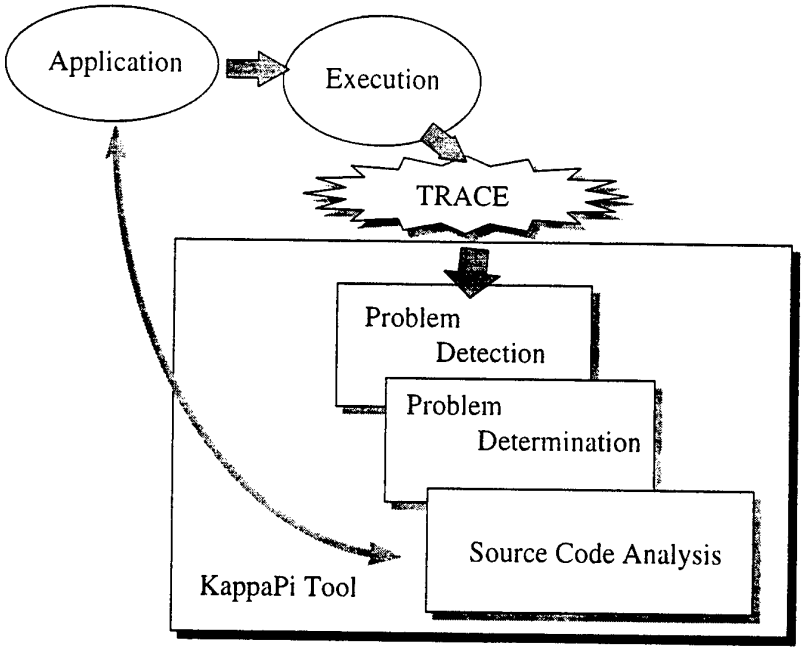


Fig. 1. Schema of the analysis of a parallel application

The analysis first considers the study of the trace file in order to locate the most important performance problems occurring at the execution. Once those problematic execution intervals have been found, they are studied individually to determinate the type of performance problem for each execution interval.

When the problem is classified under a specific category, the analysis tool scans the segment of application source code related to the execution data previously studied. This analysis of the code brings out any design problem that may have produced the performance problem. Finally, the analysis tool produces an explanation of the problems found at this application design level and recommends what should be changed in the application code to improve its execution behaviour.

In the following points, the operations performed by the analysis tool are explained in detail.

2.1. Problem Detection

The first part of the analysis is the study of the trace file obtained from the execution of the application. In this phase, the analysis tool scans the trace file, obtained with the use of TapePVM [14], with the purpose of following the evolution of the efficiency of the application. The application efficiency is basically found by measuring the number of processors that are executing the application during a certain time.

The analysis tool collects those execution time intervals when the efficiency is minimum. These intervals represent those situations where the application is not using all the capabilities of the parallel machine. They could be evidence of an application design fault. In order to analyse these intervals further, the analysis tool selects the most important inefficiencies found at the trace file. More importance is given to those inefficiency intervals that affect the most number of processors for the longest time.

2.2. Problem Determination

Once the most important inefficiencies are found, the analysis tool proceeds to classify the performance with the help of a "knowledge base" of performance problems. This classification is implemented in the form of a problem tree, as seen in figure 2.

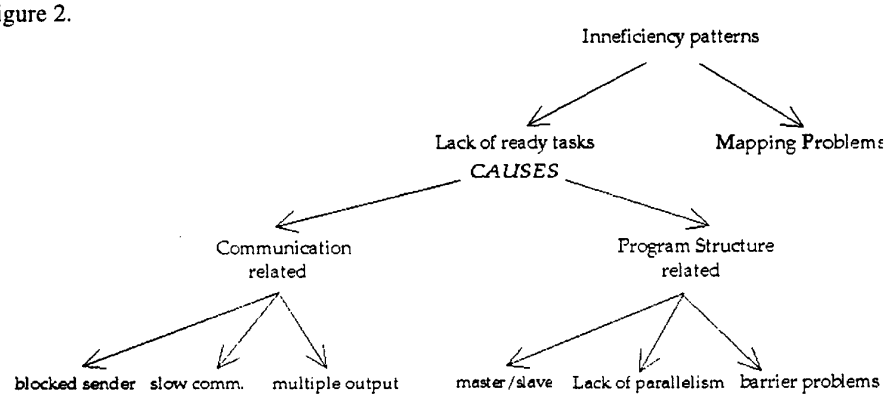


Fig. 2. *Classification of the performance problems of an application*

Each inefficiency interval at the trace is exhaustively studied in order to find which branches in the tree describe the problem in a more accurate way. When the classification of the problem arrives at the lowest level of the tree, the tool can proceed to the next stage, the source code analysis

2.3. Application of the source code analysis.

At this stage of the program evaluation, the analysis tool has found a performance problem in the execution trace file and has classified it under one category.

The aim of the analysis tool at this point is to point out any relationship between the application structure and the performance problem found. This detailed analysis differ from one performance problem to another, but basically consists of the application of several techniques of pattern recognition to the code of the application.

First of all, the analysis tool must select those portions of source code of the application that generated the performance problem when executed. In order to establish a relationship between the executed processes and the program code, the analysis tool builds up a table of process identificators and their corresponding code modules names.

With the help of the trace file, the tool is able to relate the execution events of certain operations, like sending or receiving a message, to a certain line number in the program code. Therefore, the analysis tool is able to find which instructions in the source code generated a certain behaviour at execution time. Each pattern-matching technique tries to test a certain condition of the source code related to the problem found. For each of the matches obtained in this phase, the analysis tool will generate some explanations of the problem found, the bounds of the problem and what possible alternatives there are to alleviate the problem.

The list of performance problems, as well as their implications of the source code of the application is shown at table 1. A more exhaustive description of the classification can be found at [15].

NAME	DESCRIPTION	TRACE INFORMATION	SOURCE CODE IMPLICATIONS
Mapping Problems			
Mapping problem	There are idle processors and ready-to-execute processes in busy processors	Processes assignments to busy processors, number of ready processors	Solutions affect the process-processor mapping
Lack of Ready Tasks Problems			
Communication Related			
Blocked Sender	A blocked process is waiting for a message from another process that is already blocked for reception.	Waiting receive times of the blocked processes. Process identifiers of the sender partner of each receive.	Study of the dependencies between the processes to eliminate waiting.
Multiple Output	Serialization of the output messages of a process.	Identification of the sender process and the messages sent by this process.	Study of the dependencies between the messages sent to all receiving processes.
Long Communication	Long communications block the execution of parts of the program.	Time spent waiting. Operations performed by the sender at that time.	Study of the size of data transmitted and delays of the interconnection network.
Program Structure Related			
Master/ Slave problems	The number of masters and collaborating slaves is not optimum.	Synchronization times of the slaves and master processes.	Modications of the number of slaves/masters.
Barrier problems	Barrier primitive blocks the execution for too much time.	Identification of barrier processes and time spent waiting for barrier end.	Study of the latest processes to arrive at the barrier.
Lack of parallelism	Application design does not produce enough processes to fill all processors	Analysis of the dependences of the next processes to execute.	Possibilities of increasing parallelism by dividing processes

Table 1. *Performance problems detected by the analysis tool.*

In the next section, we illustrate the process of analysing a parallel application with the use of an example.

3. Example: analysis of an application.

In this example we analyse a tree-like application with important amount of communications between processes. The application is executed mapping each process to a different processor. From the execution of the application we obtain a trace file, which is shown as a time-space diagram, together with the application structure, in figure 3.

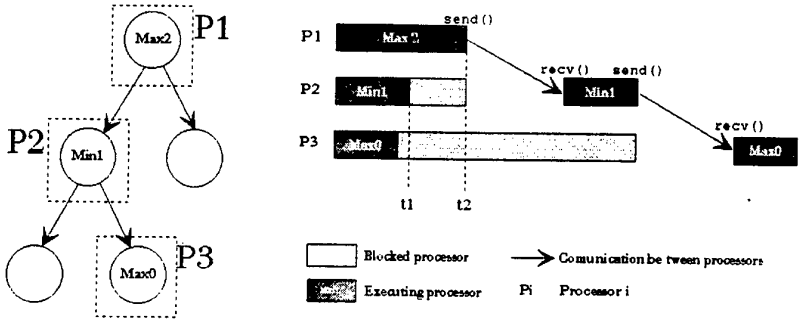


Fig. 3. *Application trace file space-time diagram*

In the next points we follow the operations carried out by the tool when analysing the behaviour of the parallel application.

3.1. Problem Detection

First of all, the trace is scanned to look for low efficiency intervals. The analysis tool finds an interval of low efficiency when processors P2 and P3 are idle due to the blocking of the processes "Min1" and "Max0". Then, the execution interval (t1,t2) is considered for further study.

3.2. Problem Determination

The analysis tool tries to classify this problem found under one of the categories. To do so, it studies the number of ready-to-execute processes in the interval. As there are no such kind of processes, it classifies the problem as "lack of ready processes". The analysis tool also finds that the processors are not just idle, but waiting for a message to arrive, so the problem is classified as a communication related.

Then, the analysis tool must find out what the appropriate communication problem is. It starts analyzing the last process (Max0) which is waiting for a message from Min1 process. When the tool tries to study what the Min1 process was doing at that

time, it finds that Min1 was already waiting for a message from Max2, so the analysis tool classifies this problem as a blocked sender problem, sorting the process sequence: Max2 sends a message to Min1 and Min1 sends a message to Max0.

3.3. Analysis of the source code.

In this phase of the analysis, the analysis tool wants to analyse the data dependencies between the messages sent by processes Max2, Min1 and Max0 (see figure 3).

First of all, the analysis tool builds up a table of the process identifiers and each source C program name of the processes.

When the program names are known, the analysis tool opens the source code file of process Min1 and scans it looking for the send and the receive operations performed. From there, it collects the name of the variables which are actually used to send and receive the messages. This part of the code is expressed on figure 4.

```
 1  pvm_recv(-1,-1);
 2
 3  pvm_upkfl(&calc,1,1);
 4
 5  calc1 = min(calc,1);
 6
 7  for(i=0;i<sons;i++)
 8  {
 9     pvm_initsend(PvmDataDefault);
10
11     pvm_pkfl(&calc1,1,1);
12
13     pvm_send(tid_son[i],1);
14  }
```

Fig. 4. *"Min1.c" relevant portion of source code*

When the variables are found ("calc" and "calc1" at the example) , the analysis tool starts searching the source code of process "Min1" to find all possible relationships between both variables. As these variables define the communication dependence of the processes the results of these tests will describe the designed relationship between the processes.

In this example, the dependency test is found true due to the instruction found at line 5, which relates "calc1" with the value of "calc". This dependency means that the message sent to process "Max0" depends on the message received from process "Max2".

The recommendation produced to the user explains this situation of dependency found. The analysis tool suggests the modification of the design of the parallel application in order to distribute part of the code of process "Min1" (the instructions that modify the variable to send) to process "Max0", and then send the same message to "Min1" and to "Max0". This message shown to the user is expressed in figure 5.

```
Analysing MaxMin....

A Blocked Sender situation has been found in the
execution.

Processes involved are:
Max0, Min1, Max2
Recommendation: A dependency between Max2 and Max0 has
been found.
The design of the application should be revised.
Line 25 of Min1 process should be distributed to Max0.
```

Fig. 5. *Output of the analysis tool*

The line referred in the recommendations of the tool (Line 5 of Min1 Process) should be executed in the process Max0, so variable "calc" must be sent to Max0 to solve the expression. Then, the codes of the processes may be changed as follows in figure 6.

```
...
pvm_recv(-1,-1);
pvm_upkfl(&calc,1,1);
calc1 = min(calc,1);.
...
```

Process Max0

```
...
pvm_recv(-1,-1);
pvm_upkfl(&calc,1,1);
calc1 = min(calc,1);
...
```

Process Min1

```
...
calc = min(old,myvalue);
pvm_initsend(PvmDataDefault);
pvm_pkfl(&calc1,1,1);
pvm_send(tid_Min1,1);
pvm_send(tid_Max2,1);
...
```

Process Max2

Fig. 6. *New process code*

In the new processes code, the dependencies between Min1 and Max2 processes have been eliminated. From the execution of these processes we obtain a new trace file, shown in figure 7. In the figure, the process Max0 does not have to wait so long

until the message arrives. As a consequence, the execution time of this part of the application has been reduced.

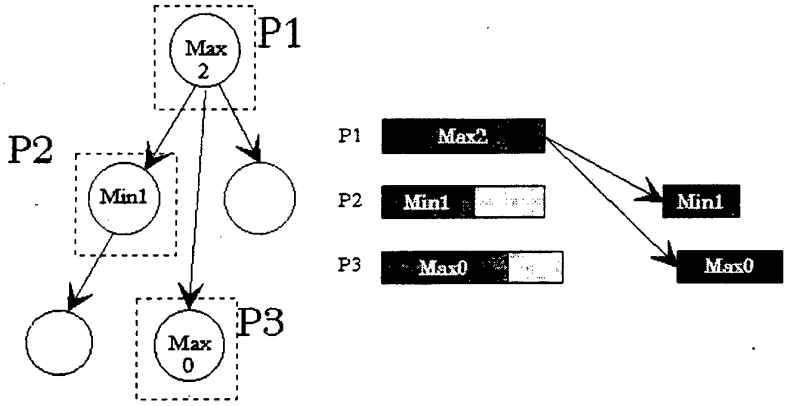


Fig. 7. *Space-state diagram of the new execution of the application*

4. Conclusions

This automatic analysis tool is designed for programmers of parallel applications that want to improve the behaviour of their applications. The application programmers' view of the tool is quite simple: the application is brought to the analysis tool as input and, after the analysis, the programmer receives a list of suggestions to improve the performance of the program. Those suggestions explain, at programmer level, which problems have been found in the execution of the application and how to solve them changing the program code.

Nonetheless, when applying the suggested changes to the application code, other new performance problems could appear. Programmers must be aware of the behaviour side-effects of introducing changes in the applications. Hence, once the application code is rebuilt, new analysis should be considered. This new analysis must be tested to find a set of representative input data in order to analyse the execution of the application comprehensively with a trace file.

Moreover, some problems may be produced by more than one cause. Sometimes it is difficult to separate the different causes of the problems and propose the most adequate solution. This process of progressive analysis of problems with multiple causes is one of the future fields of tool development.

Future work on the tool will consider the increment and refinement of the causes of performance problems, the "knowledge base". The programming model of the analysed applications must also be extended from the currently used (PVM) to other parallel programming paradigms.

Due to the general use of a few parallel execution trace formats [16, 4] and programming libraries, it is possible to have similar kind of performance data of many different applications running on different parallel systems. Although we have found that additional trace information (which is not easily obtained) can alleviate the analysis task to a high degree.

But far greater efforts must be focused on the optimisation of the search phases of the program. The search for problems in the trace file and the analysis of causes for a certain problem must be optimised to operate on very large trace files. The computational cost of analysing the trace file to derive these results is not irrelevant, although the tool is built not to generate much more overhead than the visual processing of a trace file.

The tree-structure of the problems helps to eliminate the testing of some hypotheses, but may complicate the analysis when considering problems with multiple causes (at different levels of the tree).

References:

[1] Pancake, C. M., Simmons, M. L., Yan J. C.: Performance Evaluation Tools for Parallel and Distributed Systems. IEEE Computer, November 1995, vol. 28, p. 16-19.

[2] Heath, M. T., Etheridge, J. A.: Visualizing the performance of parallel programs. IEEE Computer, November 1995, vol. 28, p. 21-28 .

[3] Kohl, J.A. and Geist, G.A.: "*XPVM Users Guide*". Tech. Report. Oak Ridge National Laboratory, 1995.

[4] Reed, D. A., Aydt , R. A., Noe , R. J., Roth, P. C., Shields, K. A., Schwartz , B. W. and Tavera, L. F .: Scalable Performance Analysis: The Pablo Performance Analysis Environment. Proceedings of Scalable Parallel Libraries Conference. IEEE Computer Society, 1993.

[5] Reed, D. A., Giles, R. C., Catlett, C. E.. Distributed Data and Immersive Collaboration. Communications of the ACM. November 1997. Vol. 40, No 11. p. 39-48.

[6] Karavanic, K. L., Miller, B. P.. Experiment Management Support for Performance Tuning. In Proceedings of SC'97 (San Jose, CA, USA, November 1997).

[7] Hollingsworth, J. K., Miller, B, P. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. International Conference on Supercomputing (Tokyo, July 19-23, 1993).

[8] Yan, Y. C., Sarukhai, S. R.: Analyzing parallel program performance using normalized performance indices and trace transformation techniques. Parallel Computing 22 (1996) 1215-1237.

[9] Crovella, M.E. and LeBlanc, T. J. . The search for Lost Cycles: A New approach to parallel performance evaluation. TR479. The University of Rochester, Computer Science Department, Rochester, New York, December 1994.

[10] Meira W. Jr. Modelling performance of parallel programs. TR859. Computer Science Department, University of Rochester, June 1995.

[11] Fahringer T. , Automatic Performance Prediction of Parallel Programs. Kluwer Academic Publishers. 1996.

[12] Wall, L. , Christiansen, T. ., Schwartz, R. L. , : Programming Perl. O'Reilly and Associates, 2nd Edition, Nov 96.

[13] Geist, A. , Beguelin, A. , Dongarra, J. , Jiang, W. , Manchek, R. and Sunderam, V. , PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Network Parallel Computing. MIT Press, Cambridge, MA, 1994.

[14] Maillet, E. : *TAPE/PVM* an efficient performance monitor for PVM applications-user guide, LMC-IMAG Grenoble, France. June 1995.

[15] Espinosa, A. , Margalef, T. and Luque, E. . Automatic Performance Evaluation of Parallel Programs. Proc. of the 6th EUROMICRO Workshop on Parallel and Distributed Processing, pp. 43-49. IEEE CS. 1998.

[16] Geist, G. A., Heath, M.T. , Peyton, B. W. and Worley, P. H. . PICL. A portable instrumented communication library. Tech. Report ORNL/TM-11130, Oak Ridge National Laboratory, July 1990.

Registers Size Influence on Vector Architectures

Luis Villa * Roger Espasa** Mateo Valero

Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya–Barcelona, Spain
e-mail: {luisv,roger,mateo}@ac.upc.es
http://www.ac.upc.es/hpc

Abstract. *In this work we have studied the influence of the vector register size over two different concepts of vector architectures. We have observed that, long vector registers play an important role in a conventional vector architecture. However, we observed that even using highly vectorizable codes, only a small fraction of that large vector registers is used. Nevertheless, we have observed that, reducing vector register size on a conventional vector architecture, result in a severe performance degradation, providing slowdowns in the range of 1.8 to 3.8. When we including an out-of-order execution on a vector architecture, the necessity of long vector registers, is reduced. We have used a trace driven approach to simulate a selection of the Perfect Club and Specfp92 programs. The results of the simulations show that, the register size reduction on an out-of-order vector architecture is less negative than in a conventional vector machine, providing slowdowns in the range of 1.04 to 1.9. Even when reducing the registers size to 1/4 the original size on an out-of-order machine, the slowdown provided is in the range of 1.04 to 1.5, but it still is better than a conventional vector machine. Finally, when comparing both architectures, using the same register file size, (8kb), we can see that the performance gained by using out-of-order execution is in the range of 1.13 to 1.40.*

1 Introduction

Numerical applications have been the area where vector architectures have proved their efficiency. This vector architectures have used in-order execution, limited form of ILP techniques and large latencies memory systems. In order to achieve good performance and to be able to tolerate the large latencies, this kind of processors have exploited the data level parallelism embedded in each vector instruction and have allowed the overlapping of vector and scalar instructions

* On leave from the Centro de Investigación en Cómputo, Instituto Politécnico Nacional – México D.F. This work was supported by the Instituto de Cooperación Iberoamericana (ICI), Consejo Nacional de Ciencia y Tecnologia (CONACYT).

** This work was supported by the Ministry of Education of Spain under contract 0429/95, and by the CEPBA.

when possible. Conventional vector architectures have used large vector registers as one of the principals resources to hide latency. When a vector instruction is started, it pays for some initial (potentially long) latency, but then it works on a long stream of elements end effectively amortizes this latency across all elements.

Taking into account this point of view, we can understand why that vector machines have been designed with vector registers as large as possible. Unfortunately large registers have several disadvantages :

- When the application can not make full use of the vector register size, a precious hardware resource is being wasted [1, 2].
- Large registers means, big number of transistors and expensive cost; this implies that only a few of them can be implemented on the design.
- If the number of registers that the compiler sees is small, then the amount of spill code introduced to support all live variables is considerably [5].

Reducing the vector registers length is certainly a solution to the problems just outlined. If most applications can not fully use all elements present in each vector register then, reducing the vector register length will reduce cost and increase the fraction of usage of registers. The drawback of register length reduction is the associated performance penalty. Each time a vector instruction is executed, its associated latencies are amortized over a smaller number of elements. This can have a significant negative impact on performance, especially for memory accesses. Moreover, more instructions have to be executed each with a shorter effective length, and, therefore, the number of times that latencies must be payed is larger.

Unless some extra latency tolerance mechanism is introduced in a vector architecture, vector length can not be reduced without a severe performance penalty. While many techniques have been developed to tolerate memory latency in superscalar processors, only a few studies have considered the same problem in the context of vector architectures [3, 4, 5].

In this paper will study the influence of the vector register size over two different concepts of vector architectures, on a conventional vector architecture and on an out-of-order vector machine. We will present data, confirming that we can not reduce the vector register size on a conventional vector architecture without suffering a severe performance penalty. We will show that combining an out-of-order execution and short registers, the performance degradation is quite small than the observed on a conventional vector machine. We have observed that this combination allows not only the vector register reduction with a good performance but also when comparing the performance between both architectures the performance of the new out-of-order vector machine is much better.

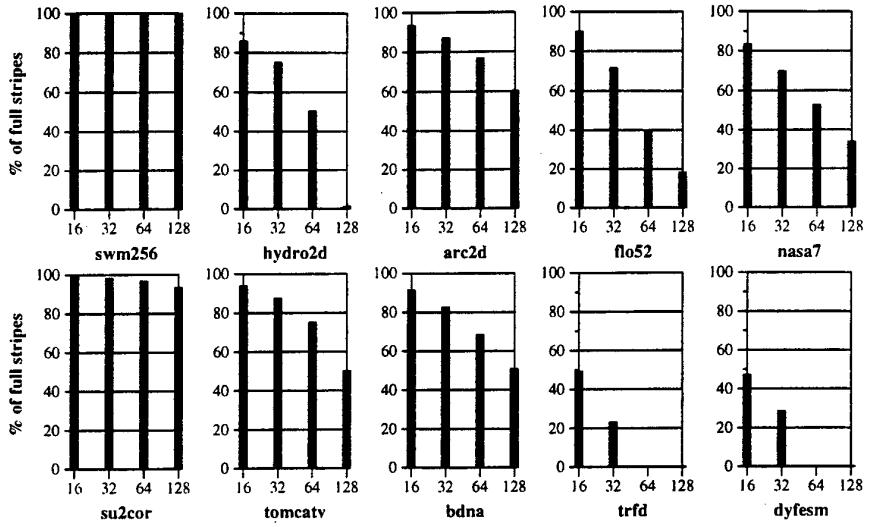


Fig. 1. Percentage of full stripes for different vector register sizes

2 Vector Registers Usage

In this section we will investigate the relationship between the next two parameters :

- Vector Register Size (VRZ).
- Benchmark Programs.

High memory latencies are common in vector architectures. In order to hide that latency, large vector registers have been a norm in the design of this kind of architectures. This point of view is correct, but unfortunately, with large vector registers not everything is positive :

- Large registers mean large hardware space and more cost. The Designer, normally, includes just few of them (eg. 8 or 16 with 128 element each).
- Having few registers, it is a drawback for the compiler because the quality of the code that it can generate is quite poor.

We have seen [1] that, when a machine has large registers, programs do not make use of their hardware. Many people are researching over new algorithms in order to execute their calculus as fast as possible, physics, chemistry, mathematics, and so on, field where this kind of architectures still excel. The algorithms characteristic are quite varied and the different architectures are trying to apply all their capacity, but some times the data structures from the applications are like a barrier.

In order to know how a set of applications make use of the register file on a vector architecture, we have done the following. Having a set of registers, where each register has as a maximum VL elements, we have executed our set of programs, using four possible values for the vector register size: 16, 32, 64 and 128 elements.

Figure 1 presents the percentage of full stripes of a program set. If we have an architecture where the VL maxim could be 128, and the structure of the programs permit the entire use of this available hardware, we will say that in this case we have a full stripe.

Now, if we consider a maximal vector register size of 64 elements and the program allows the use of bigger registers, then instructions would "translate" into two instructions that could operate on 64 elements each one. For example, the figure 1 shows how in most cases less than 50% of all executed vector instructions, used a vector register of size 128. When the vector register size was 16 elements, almost 85% of all executed vector instructions used full stripes except the program *dyfesm*.

As we have expected, there is a strong dependence between the whole performance and the program executed to get it. We have observed that, if an architecture have a long register, it does not mean that the applications will make total use of its resource. In most cases (for our applications) we will have better register usage when the vector registers are smaller.

We know, from [6], that a reduction of the vector registers on a conventional vector architecture must be enclosed by a technique which could hide that reduction, in order to keep or in the best of the cases improve, the performance.

3 Reducing Vector Registers Length

The architecture and compiler are reflected in the characteristics of the code that these could generate from an application. If these are an intelligent pair, it could be easy to obtain programs which use different vector register sizes; sections of a register, where each section could be considerate a independent register. The Fujitsu VPP500 [7] is an example of that kind of architectures. The VPP500 has a vector register file organized as 256 registers and each register has 64 elements (8 bytes each). Different register file configurations can be possible, from 256 registers of 64 elements each until 8 registers of 2048 elements each. For our purposes, this lower limit size (64 elements) is not enough, because we want to study shorter vector register, in order to have better register usage (see section 2).

Unfortunately, most of vector architectures does not have the VPP500 vector register reorganization. Our reference architecture falls into this category.

The procedure that we have followed, in order to obtain a set of binaries (from benchmarks) assuming different vector register lengths, is the following:

- For each program, we searched all the highly vectorized loops, with the help of the compiler information.

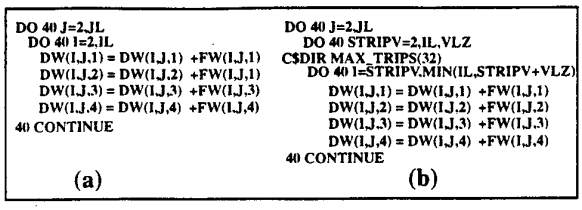


Fig. 2. (a) Flo52 loop without Strip-Mining, (b) Adding Strip-mining.

- First, we manually modified the benchmark sources and then, we manually added strip-mined loop (see figure 2) performing steps of desired length VLZ (vector length size).
- In this way, we constructed four different configurations for each source program using VLS=16, 32, 64 and 128 elements by register.

After applying this technique, we can notice that, the architecture sees more scalar and vector instructions. The vectorizable loop, will need more iterations to complete the same number of vector operations and due to the scalar operations are inside the loop, these are executed more times.

In the next section, we will describe the vector architectures examined in this study and then; we will show the performance reached by each one.

4 Vector Architectures and Simulations Tools

In this section, we describe the main characteristics of the architectures evaluated in this work. First, we will show the reference vector architecture used as a baseline. Second, we will introduce the *out-of-order* vector architecture used. Finally, we will describe the tools used to generate traces and for simulating each architecture.

4.1 The Baseline Architecture

We have used a machine loosely based on a Convex C3400 [8], as a baseline vector architecture. Even though this machine is a multiprocessor architecture, our work assumes a uniprocessor vector machine.

Figure 3 show a basic description of a C3400.

- Scalar Unit
  - The scalar unit executes all instructions that involve scalar registers (A and S registers) adds subtracts compares, shifts, logical operations and integer converts. And can issues a maximum of one instruction per cycle.
  - The scalar unit has eight 32 bits address registers and eight 64 bit scalar registers.
  - This unit has a 16 KB data cache, with 32 bytes line size.

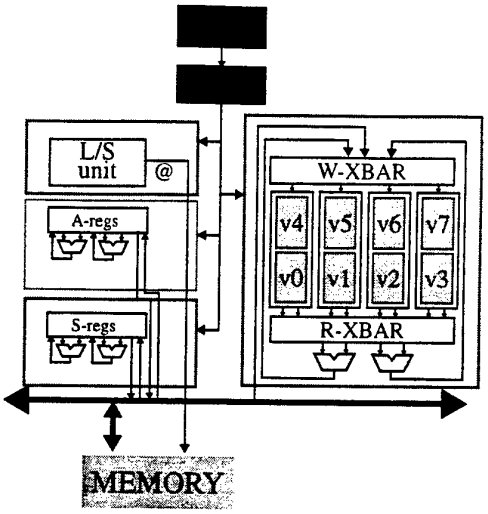


Fig. 3. The reference vector architecture.

- Vector Unit
  - The vector unit consists of two computation units (`FU1` and `FU2`) and one memory accessing unit. The `FU2` unit, is a general purpose arithmetic unit capable of executing all vector instructions. The `FU1` unit, is a restricted functional unit that executes all vector instructions *except* multiplication, division and square root.
  - The vector unit has 8 vector registers, grouped in pairs. Each register holds up to 128 elements of 64 bits each. Each group share two read ports and a write port, that link them to the functional units.
- Requesting memory is done through only one data bus (Loads and Stores).
- The reference machine implements vector chaining, from functional units to other functional units and to store unit. Memory load does not chain with any functional unit.

4.2 The Out-of-order Vector Architecture

For our simulations we used the out-of-order vector architecture introduced in [5]. The out-of-order and renaming version of the reference architecture is shown in figure 4. It has the same computing capacity as the reference machine but it is extended to use a renaming technique very similar to that found in the R10000 [9]. We will refer to this architecture as 'OOO'. Instructions flow in-order through the Fetch and Decode/Rename stages and then go to one of the four queues present in the architecture based on instruction type. At the rename stage, a mapping table translates each virtual register into a physical

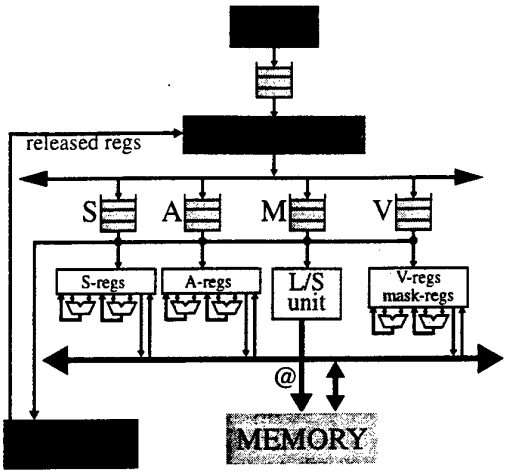


Fig. 4. The Out-of-Order vector architecture studied in this paper.

register. There are 4 independent mapping tables, one for each type of register: A, S, V and mask registers. Each mapping table has its own associated list of free registers. When instructions are accepted into the decode stage, a slot in the reorder buffer is also allocated. Instructions enter and exit the reorder buffer in strict program order. When an instruction defines a new logical register, a physical register is taken from the free list, the mapping table entry is updated with the new physical register number and the old mapping is stored in the reorder buffer slot allocated to the instruction. When the instruction commits the old physical register is returned to the free list.

The A, S and V queues monitor the ready status of all instructions held in the queue and as soon as one instruction is ready, it is sent to the appropriate functional unit for execution. All instruction queues can hold up to 16 instructions. The machine has a 64 entry BTB, where each entry has a 2-bit saturating counter for predicting the outcome of branches. Both scalar register files (A and S) have 64 physicals registers each. The mask register file has 8 physical registers. The fetch stage, the decode stage and all four queues only process a maximum of 1 instruction per cycle. Committing instructions proceeds at a faster rate, and up to 4 instructions may commit per cycle. The functional unit latencies of the architecture are very similar to the R10000 ones. See [5] for further details of the architecture.

The most important aspect of the architecture when considering final performance is the number of physical vector registers available for renaming vector instructions. In [5] it is shown that 16 physical vector registers is the optimum point that maximizes performance at a reasonable cost. Unless otherwise stated, we will use 16 physical vector registers for our simulations. In section 5, we will vary the number of physical vector registers from 16 to 32 and to 64 to study

how the number of physical registers interacts with the length of each register.

As we did for the traditional machine, we define four different versions of the OOO architecture, each having a different vector register length. The four versions will be referred to as the *OOO128, OOO64, OOO32 and OOO16* architectures and will have a vector length of *128, 64, 32 and 16* elements respectively.

4.3 Simulations Tools

For our simulations, we have used a trace-driven simulations to generate all the data, that we will show.

We have used a pixie-like tool called Dixie [10] that is able to produce a trace of basic blocks executed as well as a trace of the values contained in the *vector length* (`vl`) register and Jinks [11] a parameterizable simulator that implements the reference architecture model before described. The ability to trace the value of the *vector length* register is critical to have a detailed simulation of the program execution.

5 Performance

Using the binaries gathered (see section 3), we will study different variations of our vector architectures. For each binary (program), we have eight different configurations. The difference among each program is the maximal vector register size allow to use. The eight models under study, will be referred to as the *REF128, REF64, REF32, REF16, OOO128, OOO64, OOO32 and OOO16*, where *128, 64, 32 and 16*, are the vector register size used by each model.

Both architectures have the same number of logical registers, that means that the same code was introduced in both architectures. But, because the *o-o-o* architecture implements renaming, it uses a total of 16 physical registers, which are invisible for the compiler and for the user.

We will cover two points in this section. For three different latencies of 1, 50 and 100 cycles, we will show:

- How each architecture tolerates the vector register reduction plus memory latencies effect.
- The performance of each architecture, using different vector register sizes (Speed-Up).

5.1 Reference Architecture

In Figure 5, we can see the effect of reducing vector register sizes on the reference vector architecture.

In this figure, we have selected the *REF128* as a baseline in order to study the register reduction effect. Using one cycle latency and register sizes of *128 and 64*, the behavior seems to be constant, an ideal vector architecture behavior. When we reduce the register size and we use a more real memory latency, of 50

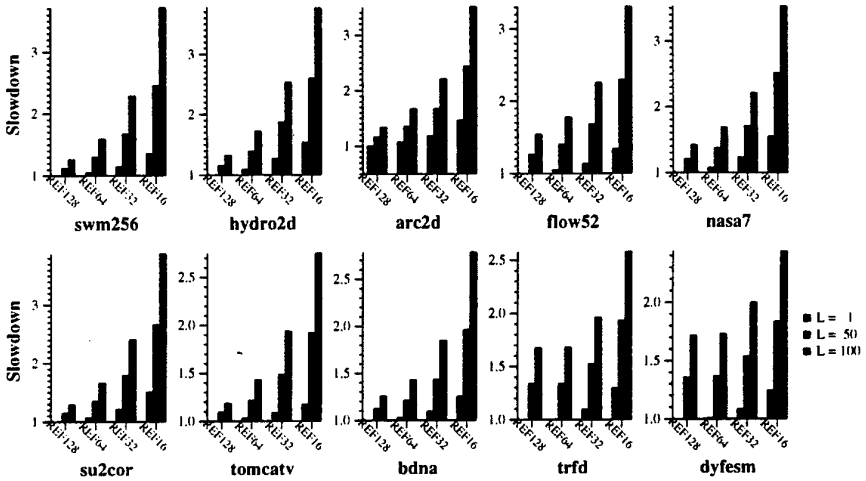


Fig. 5. Effects of memory latency and vector register size on a Conventional Vector Architecture. X-axis is memory latency

and 100 cycles, the effect is clearly negative. It is most remarkable when the memory latency is bigger and the vector registers are shorter.

Even though, the architecture uses large registers (*REF128*), the performance degradation is quite important. The slowdown degradation can take values from *1.1-1.7*. This is an important point to emphasize because large registers on vector architectures have been once of the best tools used to attack memory latency, but we can see that it is not sufficient.

If we compare the *REF128*, with the other configurations, *REF64*, *REF32* and *REF16* the slowdown can reach up to *3.5*.

This behavior is not a surprise, and as we expected, reducing the vector register size on a conventional vector architecture can be a quite negative factor.

5.2 OOO Vector Architecture

Figure 6, shows the vector register reduction effect but now on the *out-of-order* vector architecture. Again, the baseline is the best configuration, in this case is *OOO128*. Clearly we can observe that, this architecture has better vector reduction tolerance. Reducing the vector register size up to 1/4 (from 128 to 32), *line OOO32*, the execution time is degraded by an factor of *1.0-1.5*.

When we evaluated the memory latency effect, we saw that, the *OOO128, OOO64 and OOO32*, in most cases (programs *swm256, hydro2d, arc2d, nasa7, tomcatv, bdna*) have a very good memory latency tolerance, with slowdown in the range *1.0-1.3*. Other programs, such as *flow52, trfd, dyfesm and su2cor*, do not have good behavior using short registers, but it is still better than the tolerance showed by the reference architecture, with slowdowns in the range *1.22-1.98*.

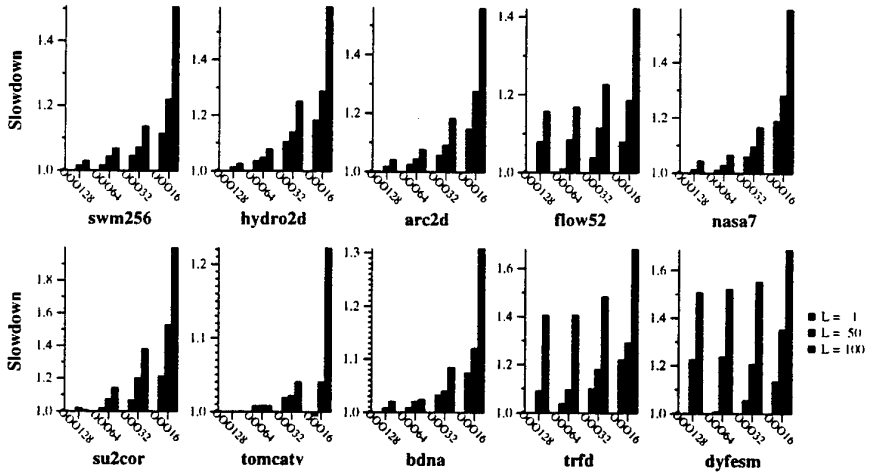


Fig. 6. Effects of memory latency and vector register length on a *Out-of-order* Vector Architecture. X-axis is memory latency.

Until this point we can conclude that if an architecture uses advanced ILP techniques like an *out-of-order*, it will be able to tolerate the vector register reduction better, even across large latency range.

5.3 Performance Comparison

In this section we will present a comparison performance between both architectures. We will make this comparison using the same or less, register file size. That is REF128 versus OOO16, OOO32 and OOO64.

Figure 7, plots the simulated performance using three different memory latencies. For each program, each configuration and each value of memory latency, we compute the speedup relative to the performance of the REF128 configuration at latency 1.

We can observe in Figure 7 that, using the same register file size, *8Kb, REF128 and OOO64* lines, the performance over the *REF128* is much better for all the programs and all the memory latencies, with speedups in the range of *1.09-1.4*.

Even reducing the register file size (on *OOO64*) up to 1/2, line *OOO32*, it is still better than the reference machine with large registers, for all programs and all memory latencies, with speedups in the range of *1.04-1.34*.

Nevertheless, when reducing the size up to 1/4 on *OOO64*, (*OOO16 line*), the performance of the *o-o-o machine* is not always better than the *REF128*. The programs *hydro2d, flo52, tomcatv, bdna* and *trfd*, show better performance than the reference machine with speedups in the range of *1.03-1.1*. Four programs, namely *swm256. arc2d and nasa7*, have performance that is slightly better or

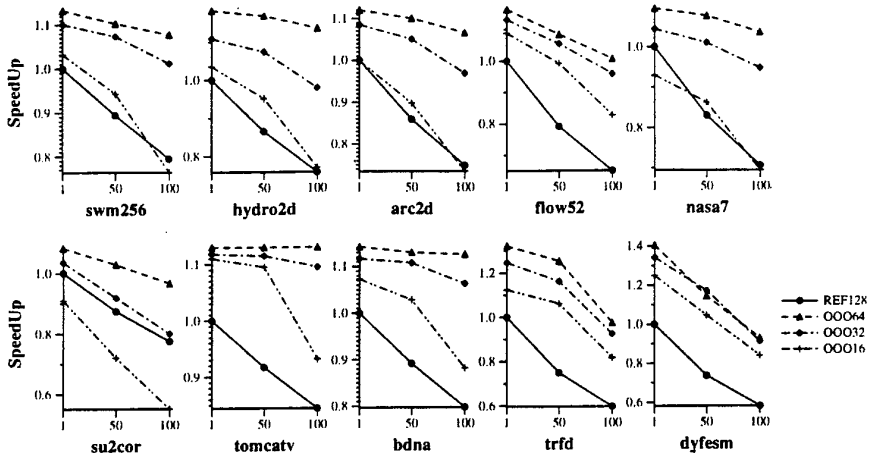


Fig. 7. Performance comparison of the OOO architecture and the Reference Architecture using the same or less, register file size. X-axis is memory latency in cycles and Y-axis represent SpeedUp.

slightly worse than the *REF128*, but the difference is typically around the *8%*. And finally, the worse case was the performance of the program *su2cor*, with a slowdown around *40%*.

6 Summary

In this paper we have studied, the influence of reducing the vector register size, over two different concepts of vector architectures.

The in order execution, traditionally used on vector architectures, and the long latencies payed on a memory request, have been always used with the use of long vector registers in order to hide and amortize, this latency and this strict program order. Nevertheless, we have showed that long registers were rarely fully used for a set of highly vectorizable programs. Less than *40%* of all the registers being used are completely filled with 128 elements of data.

As expected, reducing the vector register length on a traditional vector machine results in a remarkable loss of performance. The cost savings is clearly out-weighted by the execution time degradation. Halving the vector length yields slowdowns in the range of *1.1-3.5*. Unless some latency tolerance technique is added to a traditional vector machine, vector register length should be kept as long as possible.

We have used an ILP technique, out-of-order execution, in order to reduce the need for very large vector registers without a remarkable lost on performance. Simulations show that when the out-of-order execution is exploited, is possible

reduce the vector register size up to 1/4, without a considerable degradation in performance (slowdowns of *1.0-1.5*).

Finally we have compared the performance between architectures, where the *out-of-order* vector architecture used the same or less, register file size than the baseline architecture. Simulations showed that, using an *out-of-order* it is possible to reduce the size of each vector register up to 4Kb (REF128/4) with a better performance (speedups of *1.04-1.34*) than the conventional architecture and up to 2Kb (REF128/8), with speedup in the range *0.9-1.3*.

With this work we showed that, when ILP is exploited using *out-of-order* architecture, the need for very large vector registers, as we noted in our previous studies, it is substantially reduced. The vector register reduction can be used in several different ways: either to decrease processor cost by reducing the total amount of storage devoted to register values or to improve performance by more effectively using the available storage. Using *out-of-order* execution and short register, the vector architecture concept like a big and expensive supercomputers could change, because designers could use the actual technology and ideas (caches, memory systems, no blocking loads, Clustering, etc.) in order to improve the performance.

References

1. Luis Villa, Roger Espasa, and Mateo Valero. Effective usage of vector registers in advanced vector architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT97)*, San Francisco Cal., 1997.
2. R. Espasa, M. Valero, D. Padua, M. Jiménez, and E. Ayguadé. Quantitative analysis of vector code. In *Euromicro Workshop on Parallel and Distributed Processing*. IEEE Computer Society Press, January 1995.
3. Roger Espasa and Mateo Valero. Decoupled vector architectures. In *HPCA-2*, pages 281–290. IEEE Computer Society Press, Feb 1996.
4. Roger Espasa and Mateo Valero. Multithreaded vector architectures. In *HPCA-3*, pages 237–249. IEEE Computer Society Press, Feb 1997.
5. Roger Espasa, Mateo Valero, and James E. Smith. Out-of-order Vector Architectures. In *MICRO-30*, pages 160–170. IEEE Press, December 1997.
6. Luis Villa, Roger Espasa, and Mateo Valero. Effective usage of vector registers in decoupled vector architectures. In *Parallel and Distributed Processing (PDP98)*, Madrid, Spain 1997.
7. T. Utsumi, M. Ikeda, and M. Takamura. Architecture of the VPP500 Parallel Supercomputer. In *Proceedings of Supercomputing'94*, pages 478–487, Washington D.C., November 1994. IEEE Computer Society Press.
8. Convex Press, Richardson, Texas, U.S.A. *CONVEX Architecture Reference Manual (C Series)*, sixth edition, April 1992.
9. Keneth C. Yeager. The Mips R10000 Superscalar Microprocessor. *IEEE Micro*, pages 28-40, April 1996.
10. Roger Espasa and Xavier Martorell. Dixie: a trace generation system for the C3480. Technical Report UPBA-RR-94-08, Universitat Politècnica de Catalunya, 1994.
11. Roger Espasa. DIXIE: A parametrizable simulator for vector architectures. Technical Report UPC-DEPBA-1995-31, Universitat Politècnica de Catalunya, 1995.

The Adaptive Restarted Procedure for ORTHOMIN(k) Algorithm

Takashi NODERA and Naoto TSUNO

Department of Mathematics
Keio University
3-14-1 Hiyoshi Kohoku Yokohama 223 Japan
email: {nodera, tsuno}@math.keio.ac.jp

Abstract. An ORTHOMIN(k) algorithm, a truncated version of GCR (generalized conjugate residual) algorithm proposed by Eisenstat *et al.* [4], has been widely used for solving large and sparse nonsymmetric linear systems of equations $Ax = b$. In order to accelerate the convergence of the ORTHOMIN(k) method, we generally use a restart technique. But, it is not so easy to find out the restarting timing of its algorithm. In this paper, we will propose new adaptive restarted procedure which will find the restart timing of the ORTHOMIN(k) automatically. At last, numerical experiments are reported that demonstrate the efficacy of the adaptive restated procedure combined with the ORTHOMIN(k) algorithm on a distributed memory parallel machine AP1000.

1 Introduction

In this paper, we consider the iterative solution of large and sparse linear systems of equations

$$Ax = b \tag{1}$$

in which the coefficient A is a non-singular $n \times n$ matrix and b is a given n-vector. To simplify in this paper, we will presume A and b to be a real and large nonsymmetric matrix. The class of non-stationary iterative methods is characterized by the fact that update for the residual vector is computed separately from the current approximation to the solution. A major class of these methods is Krylov subspace or conjugate gradient type algorithms, like GCR (generalized conjugate residual) [4], GMRES (generalized minimal residual) [5], BiCG (bi-conjugate gradient) [2], and BiCGStab (bi-conjugate gradient stabilized) [8, 11, 17, 18].

The ORTHOMIN(k) algorithm [1] is the important variant of GCR algorithm [4]. This algorithm converges very quickly under certain condition among the GCR algorithm's family. However, in some case, the residual of the ORTHOMIN(k) algorithm may not have a faster convergence. So we present an adaptive restarted procedure on the ORTHOMIN(k) algorithm, principally the combined algorithm can be better deal with a faster convergence. The adaptive restarted procedure with the PRES (pseudo residual) [9] algorithm was primarily proposed by Inadu and Nodera [13, 16]. In this paper, the adaptive restarted procedure

for the ORTHOMIN(k) algorithm will be proposed, and it will be recognized to decide the restart timing, automatically.

This paper is organized as follows. In section 2, we briefly review the ORTHOMIN(k) algorithm and its associated properties. In section 3, we show the main idea on which the adaptive restarted procedure for the ORTHOMIN(k) algorithm. In section 4, we report the numerical experiments to show the convergence behavior of the adaptive restarted ORTHOMIN(k) algorithm on the MIMD parallel machine AP1000, followed by some concluding remarks in section 5.

2 Review of ORTHOMIN(k)

One kind of the most successful scheme is based on the orthogonal projection, typified by GCR [4](generalized conjugate residual) or ORTHOMIN [1, 4] and ORTHODIR [3, 11] or ORTHORES [9, 12] and GMRES [5] algorithm. The GCR algorithm is mathematically equivalent to GMRES algorithm. The GCR algorithm begins with the initial approximate solution x_0 and initial residual $r_0 = b - Ax_0$ and characterizes kth approximate solution as $x_k = x_0 + z_k$, where z_k solves

$$\min_{z \in \mathcal{K}_k} \|b - A(x_0 + z)\|_2 = \min_{z \in \mathcal{K}_k} \|r_0 - Az\|_2.$$

Here, $\mathcal{K}_k$ is the kth Krylov subspace determined by the coefficient matrix A and r_0, which defined

$$\mathcal{K}_k \equiv \text{span}\{r_0, Ar_0, A^2r_0, \ldots, A^{k-1}r_0\}.$$

In some sense, GCR algorithm finds the best approximate solution in the Krylov subspace. In contrast to the BiCG like algorithms [11, 8, 17] based on the Lanczos process, GCR algorithm uses long recurrences. This work and storage per step grows drastically as the number of steps increase and the algorithm becomes impractical for lots of iterations. As a consequence, we must restart this algorithm in practice, which may results in very slow convergence. In order to overcome this advantage of the long recurrences, a popular technique is to resort to truncated strategies. It uses only a few, say k, rather than all the vectors generated previously in recurrences to get the next vectors and can be significantly less expensive at each restart.

The ORTHOMIN(k) algorithm, primarily proposed by Vinsome [1] as a truncated version of the GCR algorithm. Figure 1 displaies the standard ORTHOMIN(k) algorithm without correction. In this algorithm, the direction vector update can be truncated so that at most $k \ll n$ previous direction vectors are used after iteration k.

$$p_i = r_i + \sum_{j=i-k}^{i-1} \beta_i^{(j)} p_j \tag{2}$$

In this case, the x_{i+1} is local minimum, the point in

$$x_{i-k} + \text{span}\{p_{i-k}, \ldots, p_i\}$$

1: Choose x_0.
2: $r_0 = b - Ax_0$
3: **for** $i = 0, 1, 2, \ldots$
  3.1: **if** $i = 0$ **then**
    3.1.1: $p_0 = r_0$
    else
    3.1.2: **for** $j = \sigma, \sigma + 1, \ldots, i - 1$
      3.1.2.1: $\beta_i^{(j)} = -(Ar_i, Ap_j)/(Ap_j, Ap_j)$
      endfor
    3.1.3: $p_i = r_i + \sum_{j=\sigma}^{i-1} \beta_i^{(j)} p_j$
    endif
  3.2: $\alpha_i = (r_i, Ap_i)/(Ap_i, Ap_i)$
  3.3: $x_{i+1} = x_i + \alpha_i p_i$
  3.4: $r_{i+1} = r_i - \alpha_i Ap_i$
  3.5: If converge, escape the loop.
  endfor

Where, $\sigma = \max\{0, i - k\}$

Fig. 1. The ORTHOMIN(k) algorithm

whose residual norm $\|r_{i+1}\|_2$ is minimized.

The following theorem was proposed by Eisenstat *et al.* [4].

[Theorem 2.1] *Let $M = (A + A^T)/2$ denote the symmetric part of A, and $R = (A - A^T)/2$ denote the skew-symmetric part of A. When M is positive definite, residuals generated by the ORTHOMIN(k) method fulfill the following relation.*

$$\|r_i\|_2 \leq \left[1 - \frac{\lambda_{\min}(M)^2}{\lambda_{\min}(M)\lambda_{\max}(M) + \rho(R)^2}\right]^{i/2} \|r_0\|_2.$$

where $\lambda_{\min}(M)$ and $\lambda_{\max}(M)$ imply the smallest and largest eigenvalues of M, respectively. Also, $\rho(R)$ denotes the spectral radius of R.

This theorem states that the residual norm of the ORTHOMIN(k) algorithm is decreased in every iteration steps. Namely, we will get the approximate solution by using this algorithm. In practice, we have found that even if this bound to be pessimistic, this algorithm is an effective solution technique for large and sparse nonsymmetric matrix problems. This algorithm is very easy to implement, but in some case the ORTHOMIN(k) algorithm slows down the convergence of residual norm. In this case, we make the choice of new starting vector and then restarts the algorithm once again. So an suitable restarting is usually necessary for this algorithm to make the acceleration of convergence of residual. In the next section, we devote to the study of automatic restart of the ORTHOMIN(k) algorithm in adaption.

3 The adaptive restarted procedure

The restarts of ORTHOMIN(k) algorithm are ordinarily needed to reduce for the round off errors and the amount of the necessary computational time to satisfy the convergence criterion. However, so many restarts slow down the convergence of the ORTHOMIN(k) algorithm. So the suitable restart of this algorithm can be accelerated the convergence of the residuals. We have designed an adaptive procedure with the automatic restart for the ORTHOMIN(k) algorithm.

The adaptive restarted procedure, which was proposed by Inadu and Nodera [13, 16], is the technique which is introduced to the ORTHORES(k) algorithm for solving the large sparse sets of nonsymmetric linear systems of equations. The ORTHORES(k) algorithm belongs to the class of the pseudo residual algorithm [9, 10]. This technique improves the convergence of ORTHORES(k) method by using the restart of its algorithm, appropriately. In order to work this approach effectively, we need to find out the timing of performing the restart. For the pseudo residual algorithm, we decided the timing of the restart from the two points of view: one is the observation of oscillating residual norm, and the another is the observation of the scalar coefficients of the ORTHORES(k) algorithm. On the other hand, the ORTHOMIN(k) algorithm has a good property which minimizes the residual norm. Therefore, we consider to use a different strategy that does find out about the timing of restart for the ORTHOMIN(k) algorithm, adaptively.

The ORTHOMIN(k) algorithm has very slow convergence behavior, when the scalar $|\alpha_i|$ is the smallest enough. One of the reasons that the degree of the direction polynomials does not come up higher order. So in order to improve the convergence of its residual, we consider the timing of the restart of ORTHOMIN(k) algorithm, which is based on the scalar $|\alpha_i|$. Also, the scalar $|\alpha_i|$ has a meaning called the distance that proceeds along a direction vector. In fact, while the norm of residual decreasing sharply, we have a property that the scalar $|\alpha_i|$ stalls at the small value. Let us consider about the execution of the adaptive restart with the following rule of the determination of the timing.

(1) **Rule of deciding the timing of restart**

When the scalar $||\alpha_i Ap_i||/||r_i||$ is even small more than the parameter given ε in advance, we are not able to expect a faster convergence of the ORTHOMIN(k) method in the continuous iteration of k steps, and then we consider to do the restart. In fact, while the restart is difficult to be executed for smaller value of parameter ε, the restart is easy to be executed for the larger value of the parameter ε. We have shown that the adaptive restarted procedure stabilized to the many problems around the parameter $\varepsilon = 1.0$, as the results of numerous experiments coming from the discretization of the boundary value problem of partial differential equation, etc.

For the next iteration steps of the ORTHOMIN(k) method after performed the restart, we expect that the scalar $||\alpha_i Ap_i||/||r_i||$ becomes the larger value. While for smaller value of $||\alpha_i Ap_i||/||r_i||$ we performed the restart tentatively,

```
1:  Choose x_0 and ε.
2:  α'_max = 0, adapt_restart := on ..... (2-a)
3:  r_0 = b − Ax_0, k_count = 0
4:  for i = 0, 1, 2, ...
    4.1:  Calculate α_i and p_i, using ORTHOMIN(k) method.
    4.2:  x_{i+1} = x_i + α_i p_i
    4.3:  r_{i+1} = r_i − α_i A p_i
    4.4:  If converge, escape the loop.
    4.5:  α'_i = ||α_i A p_i|| / ||r_i||
    4.6:  If α'_i > α'_max, then adapt_restart := on. ..... (2-b)
    4.7:  if α'_i < ε then
          4.7.1:  k_count = k_count + 1
          else
          4.7.2:  k_count = 0, adapt_restart := on ..... (2-d)
          endif
    4.8:  if k_count = k then ..... (1)
          4.8.1:  if adapt_restart = on then ..... (2-c)
                  4.8.1.1:  α'_max = max_{i−k+1≤j≤i} α'_j
                  4.8.1.2:  adapt_restart := off
                  4.8.1.3:  x_0 = x_{i+1} and restart (goto step 3).
                  endif
          endif
    endfor
```

Fig. 2. The algorithm of adaptive restarted procedure for the ORTHOMIN(k) method, (AR-ORTHOMIN(k))

even if the convergence of the ORTHOMIN(k) method is still slow, the situation becomes more worse from which the residual polynomial has remained in this time. In this case, we better do not have to perform the restart. However, after we restarted the algorithm, in order to know the scalar $\|\alpha_i A p_i\|/\|r_i\|$ in advance, the additional computational cost, which is equal to the iteration steps, is needed. Consequently, one might not expect with efficient. Therefore, we perform the restart in an unconditional judgment of the first restart, and then we shall decide whether we do perform or do not perform the restart in according to the circumstances of the former update restart after the second restart.

(2) **Rule of the execution of restart**

(a) The restart is done in an unconditional judgment of the 1st restart.

(b) When we performed the restart, comparing the maximum value of distance that proceeds in k iteration steps before the restart and after the restart, we examine the efficiency of the restart. If one of the maximum value of distance proceeding after the restart is larger, we consider that the restart is working effectively.

Table 1. AP1000 specification

Architecture	Distributed Memory, MIMD
Number of processors	64
Inter processor networks	Broadcast network(50MB/s) Two-dimensional torus network (25MB/s/port) Synchronization network

4 Numerical experiments

We now give some numerical results to demonstrate the behavior of convergence associated with the AR-ORTHOMIN(k) algorithm. We use the test problems coming from the boundary value problems of partial differential equation in the scientific and industrial applications. We shall show the efficiency of the adaptive restarted procedure. All the computations were done in double precision (64 bits) on the MIMD parallel machine Fujitsu AP1000 with 64 processors. The Specification of AP1000 is given in Table 1. Each cell of AP1000 employs RISC-type SPARC or SuperSPARC processor chip. For simplicity we did not use any preconditioner in numerical experiments.

[Example 1] Firstly, we consider a finite difference problem, namely, central finite differencing applied to the following Dirichlet problem:

$$\begin{aligned} -u_{xx} - u_{yy} + \sigma u_x(x,y) + \tau u_y(x,y) \\ = f(x,y) \text{ on } \Omega = [0,1]^2, \\ u(x,y) \mid_{\partial\Omega} = 1 + xy. \end{aligned}$$

with $f(x,y)$ is chosen so that the true solution $u(x,y) = 1 + xy$ on Ω. Let h represent the mesh size in each direction. This yields a matrix of size $n = 66536$ (where, $h = 1/257$), after boundary points have been eliminated. In our numerical computations, the initial guess is chosen as $x_0 = 0$, and an approximate solution x_k is considered to have converged if the residual satisfies $\|r_k\|_2/\|r_0\| \leq 10^{-12}$. Also, the iteration was stopped, when the number of iteration exceeded $6654(\approx 0.1 \times n)$. By varying the constant σ and τ, the amount of nonsymmetricity of the coefficient matrix A may be varied.

In Table 2, we are displayed the numerical results obtained by the standard ORTHOMIN(k) and AR-ORTHOMIN(k) method. For this problem, AR-ORTHOMIN(5) and AR-ORTHOMIN(10) method applied to this problem worked quite well. On the other hand, the standard ORTHOMIN(k), $k = 5$, or 10, method gave an excessive computational times and the number of iterations. Figure 4 gives representative plots of the convergence behavior of ORTHOMIN(5), ORTHOMIN(10), AR-ORTHOMIN(5), and AR-ORTHOMIN(10) method for the case of $h = 1/257$, and $(\sigma+\tau)h/4 = 5.0$. As you can seen clearly, only the AR-ORTHOMIN(k) method is successful in this example. The ORTHOMIN(k) without the adaptive restarted procedure has some trouble from the beginning, which

Table 2. The numerical results for example 1, $((\sigma + \tau)h/4 = 0.5)$

$(\sigma : \tau)$	8 : 0	7 : 1	6 : 2	5 : 3	4 : 4
⟨⟨execution time (Sec) ⟩⟩					
ORTHOMIN(5)	64.01	63.98	55.20	47.76	45.37
AR-ORTHOMIN(5)	53.23	54.75	51.53	46.51	47.46
ORTHOMIN(10)	114.06	103.27	94.08	82.08	83.55
AR-ORTHOMIN(10)	86.00	82.00	84.60	77.61	77.16
⟨⟨number of iteration ⟩⟩					
ORTHOMIN(5)	1031	1030	891	772	732
AR-ORTHOMIN(5)	838	852	798	737	745
ORTHOMIN(10)	1258	1139	1039	906	920
AR-ORTHOMIN(10)	964	915	943	865	858
⟨⟨number of restart ⟩⟩					
AR-ORTHOMIN(5)	5	4	3	3	2
AR-ORTHOMIN(10)	5	3	3	3	2

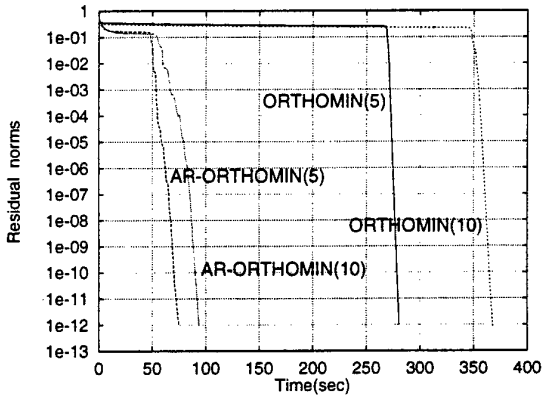


Fig. 4. The convergence behavior of residual norms vs. computational time for example 2 $((\sigma + \tau)h/4 = 5.0, \sigma : \tau = 8 : 0)$

causes the stagnation. Note that in this case the AR-ORTHOMIN(5) method is preferable, because it is more efficient: the working cost of AR-ORTHOMIN(5) method less than AR-ORTHOMIN(10) method. This result shows that the AR-ORTHOMIN(k) method keeps the residual size better behaved than the standard ORTHOMIN(k) method, which without the adaptive restarted procedure, over the course of run. We found that in most cases the AR-ORTHOMIN(k) method was more efficient than the standard ORTHOMIN(k) method in CPU times.

[Example 2] We now consider a little bit difficult class of finite difference dis-

Table 3. The numerical results for example 2

σh	2^{-2}	2^{-1}	2^{0}	2^{1}	2^{2}
⟨⟨execution time (Sec) ⟩⟩					
ORTHOMIN(5)	213.55	290.96	344.28	—	—
AR-ORTHOMIN(5)	191.54	217.57	325.40	—	—
ORTHOMIN(10)	227.48	309.03	431.88	—	—
AR-ORTHOMIN(10)	243.68	276.74	482.00	—	—
⟨⟨number of iteration ⟩⟩					
ORTHOMIN(5)	3212	4369	5185	(4e-12)*	(1e-8)*
AR-ORTHOMIN(5)	2896	3338	4945	(2e-10)*	(2e-8)*
ORTHOMIN(10)	2499	3400	4751	(3e-12)*	(2e-8)*
AR-ORTHOMIN(10)	2723	3111	5447	(3e-11)*	(2e-8)*
⟨⟨number of restart ⟩⟩					
AR-ORTHOMIN(5)	18	63	58	43	35
AR-ORTHOMIN(10)	15	24	53	52	28

*The relative residual norm after the maximum iterations

cretization of the Dirichlet boundary value problem as follows:

$$-u_{xx} - u_{yy} + \sigma \left\{ \left(y - \frac{1}{2} \right) u_x + \left(x - \frac{1}{3} \right) \left(x - \frac{2}{3} \right) u_y \right\} = f(x, y) \text{ on } \Omega = [0, 1]^2$$

$$u(x, y)|_{\partial\Omega} = 1 + xy.$$

Central differencing, with uniform mesh spacing h in each direction, yields a $n \times n$ sparse coefficient matrix. The right hand side of the above equation is taken such that the true solution is $u(x, y) = 1 + xy$. Problems of this type arise frequently in many scientific problem and are significant practical importance. The initial approximation vector is $x_0 = 0$ and no preconditioning is used for these numerical experiments.

For the test problem we let $h = 1/257$ and use several value of σ. We give comparative results in Table 3 with $\sigma h = 2^{-2}, 2^{-1}, 2^{0}, 2^{1}, 2^{2}$, respectively. In the item of execution time in this table, runs for which convergence is not possible maximum iterations are labeled by (—).

In the Table 3, in most cases AR-ORTHOMIN(5) method worked quite well. For the case of $\sigma h = 2^{-2}$, and 2^{-1}, the AR-ORTHOMIN(10) method gave an excessive number of iterations and the computational times.

Figure 5 gives representative plots of the convergence behavior of the above mentioned methods with no preconditioning for the case $\sigma h = 2^{0}$.

The following observations on this problem can be made. The AR-ORTHOMIN(5) method worked well in most cases, particularly in $\sigma h = 2^{-1}$. As you can see that, for large k such as the AR-ORTHOMIN(10) method, the improvement of the computational cost is not impressive, but the residual norms of the AR-ORTHOMIN(10) method stay well below those of the standard ORTHOMIN(10)

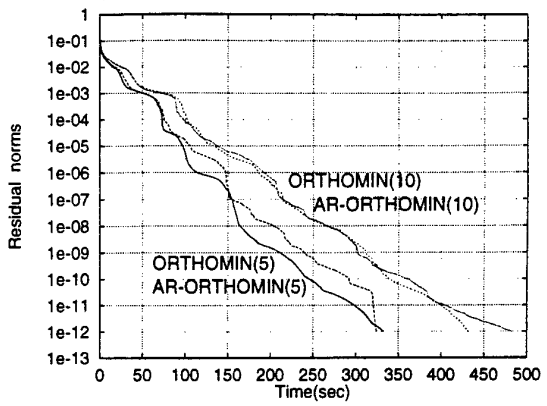


Fig. 5. The convergence behavior of residual norms vs. computational time for example 2 ($\sigma h = 2^0$)

method. We note that, as expected from these numerical experiments, the AR-ORTHOMIN(5) method is slightly more efficient than the AR-ORTHOMIN(10) method.

[Example 3] Our last example is taken from the example of Reichel *et al.* [6] and Gutknecht [7].

$$A := \begin{bmatrix} 1 & 0.5 & & & 0 \\ 0 & 1 & 0.5 & & \\ \sigma & 0 & 1 & 0.5 & \\ & \sigma & 0 & \cdots & \\ & & \cdots & \cdots & 0.5 \\ 0 & & & \sigma & 0 \quad 1 \end{bmatrix} \in \boldsymbol{R}^{4096\times 4096}, \quad (\sigma > 0)$$

Since all the eigenvalues of $M = (A + A^T)/2$ are distributed in the interval $[-2\sigma, 2 + 2\sigma]$, the condition number of M becomes large so that the element σ is large. Also, the property of positive definite of M is not guaranteed. On the other hand, the spectral radius of $R = (A - A^T)/2$ is satisfied the following inequality $\rho(R) \leq 1 + 2\sigma$.

Table 4 shows the numerical results for several σ. In this example, since the behavior of residuals of standard ORTHOMIN(k) method showed linear convergence by all cases, there is no restat performed by the AR-ORTHOMIN(k) method.

5 Conclusion

Our study involved a new approach to the adaptive restarted procedure for the ORTHOMIN(k) algorithm. One interesting feature of this technique is the

Table 4. The numerical results for example 3

σ	0.1	0.3	0.5	0.7	0.9
⟨⟨execution time (sec) ⟩⟩					
ORTHOMIN(5)	0.45	0.45	0.83	1.69	—
AR-ORTHOMIN(5)	0.46	0.46	0.84	1.72	—
ORTHOMIN(10)	0.68	0.68	1.17	2.29	6.85
AR-ORTHOMIN(10)	0.68	0.68	1.18	2.31	6.90
⟨⟨number of iteration ⟩⟩					
ORTHOMIN(5)	32	32	57	115	(4e-10)*
AR-ORTHOMIN(5)	32	32	57	115	(4e-10)*
ORTHOMIN(10)	32	32	52	98	285
AR-ORTHOMIN(10)	32	32	52	98	285
⟨⟨number of restart ⟩⟩					
AR-ORTHOMIN(5)	0	0	0	0	0
AR-ORTHOMIN(10)	0	0	0	0	0

*The relative residual norm after the maximum iterations

fact that extra calculation is not explicitly needed, which may be used only implicitly given as calculations of the standard ORTHOMIN(k) algorithm. The results presented in this paper suggest that the adaptive restarted procedure with ORTHOMIN(k) algorithm, which we called the AR-ORTHOMIN(k), can be one of the useful tools for computing the approximate solution of large and sparse nonsymmetric linear systems of equations on parallel machines with modern high performance architectures. The details of the parallel implementation of this strategy and the further numerical experiments are given in Tsuno and Nodera [15].

References

1. P. Vinsome, "ORTHOMIN, an iterative method for solving sparse sets of simultaneous linear equations," *In Proceedings of the Fourth Symposium on Reservoir Simulation*, Society of Petroleum Engineering of AIME, pp. 149-159 (1976).
2. R. Fletcher, "Conjugate gradient methods for indefinite systems, *Lecture Notes in Math.*, Vol. 506, pp. 73-89 (1976).
3. D. M. Young and K. C. Jea, "Generalized conjugate gradient acceleration of nonsymmetrizable iterative methods," Linear Algebra and its Applications, Vol. 34, pp. 159-194 (1980).
4. S. Eisenstat, H. Elman and M. Schultz, "Variational iterative methods for nonsymmetric systems of linear equations," *SIAM J. Numer. Anal.*, Vol. 20, No. 2, pp. 345-357 (1983).
5. Y. Saad and M. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, Vol. 7, No. 3, pp. 856-869 (1986).

6. L. Reichel and L. N. Trefethen, "Eigenvalues and pseudo-eigenvalues of Toeplitz matrices," Lin. Alg. Appl., Vol. 162, pp. 153-185 (1992).
7. M. H. Gutknecht, "Variants of BiCGSTAB for matrices with complex spectrum," SIAM J. Sci. Comput., Vol. 14, pp.1020-1033 (1993).
8. H. A. van der Vorst, "BiCGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems," *SIAM J. Sci. Sat. Comput.*, Vol. 13, pp. 631-644 (1992).
9. R. Weiss, "Properties of generalized conjugate gradient methods," *Numer. Lin. Alg. Appl.*, Vol. 1, No. 1, pp. 45-63 (1994).
10. R. Weiss, "A theoretical overview of Krylov subspace methods," *Appl. Numer. Math.*, Vol. 19, No. 3, pp. 207-234 (1995).
11. A. M. Bruaset, "A survey of preconditioned iterative methods," *Pitman Research Notes in Math* (1995).
12. T. Nodera and T. Inadu, "The convergence acceleration of pseudo residual method using a restarted procedure," *Transaction on Information Processing Society of Japan*, Vol. 37, No. 6 (in Japanese), pp. 1237-1240 (1996).
13. T. Inadu and T. Nodera, "An adaptive restarting procedure for pseudo-residual methods," *Transaction on Information Processing Society of Japan*, Vol. 37, No. 9 (in Japanese), pp. 1637-1645 (1996).
14. N. Tsuno and T. Nodera, "Convergence properties for an adaptive restarted procedure of non-stationary iterative methods (Part I, and Part II)," *SIG HPC, Information Processing Society of Japan*, Vol. 96, No. 81 (in Japanese), pp. 105-109 (1996) and Vol. 96, No. 81 (in Japanese), pp. 7-12 (1996).
15. N. Tsuno and T. Nodera, "The adaptive restarted procedure for ORTHOMIN(k) method," *SIG HPC, Information Processing Society of Japan*, Vol. 97, No. 75 (in Japanese), pp. 7-12 (1997).
16. T. Nodera and T. Inadu, "A note on adaptive restarting procedure for pseudo residual algorithms," *Scientific Computing* (eds. G. H. Golub, *et al.*), Springer-Verlag, pp. 265-272 (1997).
17. T. Nodera and Y. Noguchi, "Effectiveness of BiCGStab(ℓ) method on AP1000," *Transaction on Information Processing Society of Japan*, Vol. 38, No. 11 (in Japanese), pp. 2089-2101 (1997).
18. T. Nodera and Y. Noguchi, "A note on BiCGStab(ℓ) method on AP1000," *IMACS Lecture Note on Computer Science* (1998), to appear.

This article was processed using the LaTeX macro package with LLNCS style

Reconfigurable Systems Past and Next 10 Years

Jean Vuillemin[1]

Ecole Normale Supérieure, 45 rue d'Ulm, 75230 Paris cedex 05, France.
This research was partly done at *Hewlett Packard Laboratories*, Bristol U.K.

Abstract. A driving factor in *Digital System DS* architecture is the *feature size* of the silicon implementation process. We present Moore's laws and focus on the shrink laws, which relate chip performance to feature size. The theory is backed with experimental measures from [14], relating performance to feature size, for various memory, processor and FPGA chips from the past decade. Conceptually shrinking back existing chips to a common feature size leads to common architectural measures, which we call *normalized*: area, clock frequency, memory and operations per cycle. We measure and compare the normalized *compute density* of various chips, architectures and silicon technologies.
A *Reconfigurable System RS* is a standard processor tightly coupled to a *Programmable Active Memory PAM*, through a high bandwidth digital link. The PAM is a FPGA and SRAM based coprocessor. Through software configuration, it may emulate any specific custom hardware, within size and speed limits. RS combine the flexibility of software programming to the performance level of application specific integrated circuits ASIC.
We analyze the performance achieved by P1, a first generation RS [13]. It still holds some significant absolute speed records: RSA cryptography, applications from high-energy physics, and solving the Heat Equation. We observe how the software versions for these applications have gained performance, through better microprocessors. We compare with the performance gain which can be achieved, through implementation in P2, a second-generation RS [16].
Recent experimental systems, such as the *Dynamically Programmable Arithmetic Array* in [19] and others in [14], present advantages over current FPGA, both in storage and compute density. RS based on such chips are tailored for *video processing*, and similar compute, memory and IO bandwidth intensive. We characterize some of the architectural features that a RS must posses in order to be *fit to shrink*: automatically enjoy the optimal gain in performance through future shrinks. The key to scale, for any general purpose system, is to embed memory, computation and communication at a much deaper level than presently done.

1 Moore's Laws

Our modern world relies on an ever increasing number of Digital Systems DS: from home to office, through car, boat, plane and elsewhere. As a point in case,

the shear economic magnitude of the Millenium Bug [21], shows how futile it would be to try and list *all* the functions which DS serve in our brave new digital world.

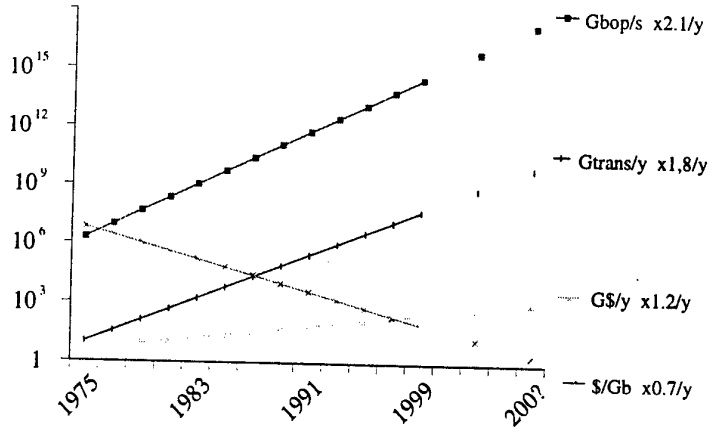


Fig. 1. Estimated number and world wide growth rate: $G = 10^9$ transistors fabricated per year; G bit operations computed each second; Billion $ revenues from silicon sold world wide; $ cost per $G = 2^{30}$ bits of storage.

Through recent decades, earth's combined raw compute power has more than doubled each year. Somehow, the market remains elastic enough to find applications, and people to pay, for having twice as many bits automatically switch state than twelve months ago. At least, many people did so, each year, for over thirty years - fig. 1.

An ever improving silicon manufacturing technology meets this ever increasing demand for computations: more transistors per unit area, bigger and faster chips. On the average over 30 years, the cost per bit stored in memory goes down by 30% each year. Despite this drop in price, selling 80% more transistors each year increases revenue for the semi-conductor industry by 20% - fig. 1.

The number of transistors per mm^2 grows about 40% each year, and chip size increases by 15%, so:

The number of transistors per chip doubles in about 18 months.

That is how *G. Moore*, one of the founders of *Intel*, famously stated the laws embodied in fig. 1. That was in the late sixties, known since as *Moore's Laws*.

More recently, *G. Moore* [18] points out that we will soon fabricate more transistors per year than there are living ants on earth: an estimated 10^{17}.

Yet, people buy computations, not transistors. How much computation do they buy? Operating all of this year's transistors at 60 MHz amounts to an

aggregate *compute power* worth 10^{24} *bop/s - bit operation per second.* That would be on the order of 10 million *bop/s* per ant!

This estimate of the world's compute power could well be off by some order of magnitude. What matters is that *computing power at large has more than doubled each year* for three decades, and it should do so for some years to come.

1.1 Shrink Laws

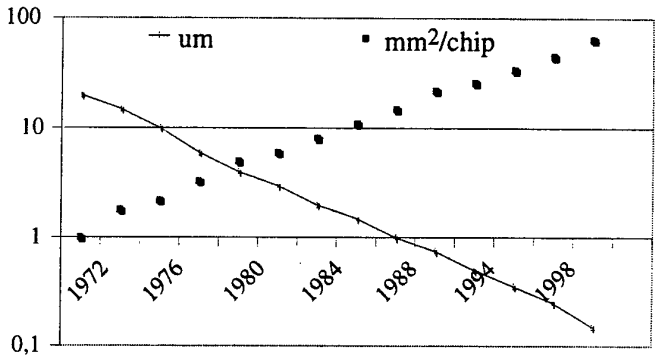


Fig. 2. Shrink of the feature size with time: minimum transistor width, in $\mu m = 10^{-6} m$. Growth of chip area - in mm^2.

The economic factors at work in fig. 1 are separated from their technological consequences in fig 2. The feature size of silicon chips *shrinks*: over the past two decades, the average shrink rate was near 85% per year. During the same time, chip size has increased: at a yearly rate near 10% for DRAM, and 20% for processors.

The effect on performance of scaling down all dimensions and the voltage of a silicon structure by 1/2: the area reduces by 1/4, the clock delay reduces to 1/2 and the power dissipated per operation by 1/8.

Equivalently, the clock frequency doubles, the transistor density per unit area quadruples, and the number of operations per unit energy is multiplied by 8, see fig. 2. This shrink model was presented by [2] in 1980, and intended to cover feature sizes down to 0.3 μm - see fig. 3.

Fig. 4 compares the shrink model from fig. 3 with experimental data gathered in [14], for various DRAM chips, published between in the last decade. The last entry - from [15] - accounts for synchronous SDRAM, where access latency is traded for throughput. Overall, we find a rather nice fit to the model. In fig. 7, we also find agreement between the theoretical fig. 3 and experimental data for microprocessors and FPGA, although some architectural trends appear.

A recent update of the shrink model by Mead [9] covers features down to 0.03 μm. The optimists conclusion, from [9]:

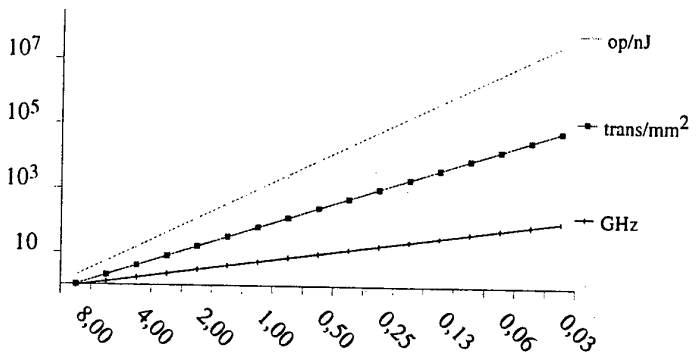


Fig. 3. Theoretical chip performance, as the minimum transistor width (feature size) shrinks from 8 to 0.03 micron μm: transistors per square millimeter; fastest possible chip wide synchronous clock frequency, in giga hertz; number of operations computed, per nano joule.

We can safely count on at least one more order of magnitude of scaling.

The pessimist will observe that it takes 2 pages in [2] to state and justify the *linear* shrink rules; it takes 15 pages in [9], and the rules are *no longer linear*. Indeed, thin oxide is already nearly 20 atoms thick, at current feature size 0.2 μm. A linear shrink would have it be less than one atom thick, around 0.01 μm. Other fundamental limits (*quantum mechanical effects, thermal noise, light's wavelength,* ...) become dominant as well, near the same limit. Although *C. Mead* [9] does not *explicitly* cover finer sizes, the *implicit* conclusion is:

We cannot count on two more orders of magnitude of scaling.

Moore's law will thus eventually either run out of fuel - demands for *bop/s* will some year be under twice that of the previous - or it will be out of an engine - *shrink laws* no longer apply below 0.01 μm. One likely possibility is some combination of both: feature size will shrink ever more slowly, from some future time on.

On the other hand, there is no fundamental reason why the size of chips cannot keep on increasing, even if the shrink stops. Likewise, we can expect new architecture to improve the currently understood technology path. No matter what happens, how to *best use the available silicon* will long remain an important question. Another good bet: the amount of storage, computation and communication, available in each system will grow, ever larger.

2 Performance Measures for Digital Systems

Communication, processing and storage are the three building blocks of DS. They are intimately combined at all levels. At micron scale, wires, transistors

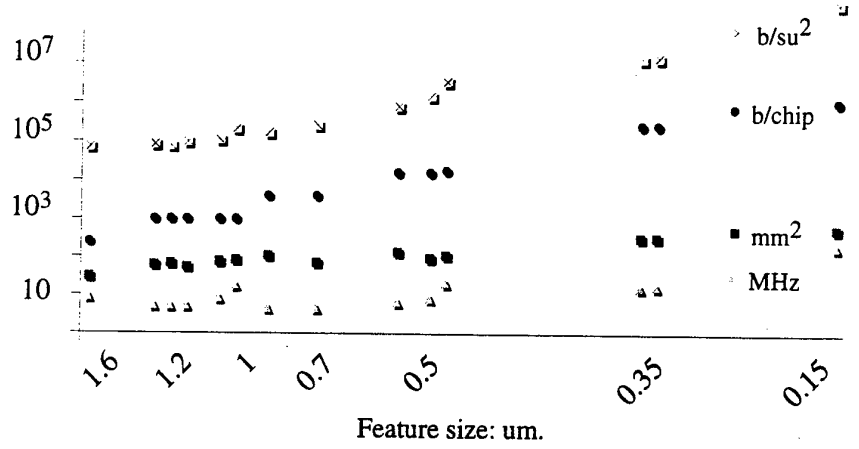


Fig. 4. : Actual DRAM performance as feature size shrinks from 0.8 to 0.075 μm: clock frequency in Mega hertz; square millimeters per chip; bits per chip; power is expressed in bit per second per square micron.

and capacitors implement the required functions. At human scale, the combination of a modem, microprocessor and memory in a PC box does the trick. At planet scale, communication happens through more exotic media - waves in the electromagnetic ether, or optic fiber - at either end of which one finds more memory, and more processing units.

2.1 Theoretical performance measures

Shannon's Mathematical Theory of Communication [1] shows that physical measures of information (bits b) and communication (bits per second b/s) are related to the abstract mathematical measure ot statistical entropy H, a positive real number $H > 0$. Shannon's theory does not account for the cost of any computation. Indeed, the global function of a communication or storage device is the identity $X = Y$.

On the other hand, source coding for MPEG video is among the most demanding computational tasks. Similarly, *random* channel coding (and decoding), which gets near the optimal for the communication purposes of Shannon as coding blocks become bigger, has a computational complexity which increases exponentially with block size.

The basic question in Complexity Theory is to determine how many operations $C(f)$, are necessary and sufficient for computing a digital function f. All operations in the computation of f are accounted for, down to the bit level, regardless of when, where, or how the operation is performed. The unit of measure for $C(f)$ is one *Boolean operation bop*. It is applicable to all forms of compu-

tations - sequential, parallel, general and special purpose. Some relevant results (see [5] for proofs):

1. The complexity of n bit binary addition is $5n - 3$ bop. The complexity of computing one bit of sum is 1 *add* = 5 *bop* (full adder: 3 in, 2 out).
2. The complexity of n bit binary multiplication can be reduced, from $6n^2$ *bop* for the naive method (and $4n^2$ *bop* through *Booth Encoding*), down to $c(\epsilon)n^{1+\epsilon}$, for any real number $\epsilon > 0$. As $c(\epsilon) \mapsto \infty$ when $\epsilon \mapsto 0$, the practical complexity of binary multiplication is only improved for n large.
3. Most Boolean functions f, with n bits of input and one output, have a *bop* complexity $C(f)$ such that $2n/n < C(f) < 2n/n(2+\epsilon)$, for all $\epsilon > 0$ and n large enough. To build one, just choose at random! No explicitly described Boolean function has yet been proved to posses more than linear complexity (including multiplication). An efficient way to compute a *random* Boolean function is through a *Lookup Table LUT*, implemented with a RAM or a ROM.

Computation is free in Shannon's model, while communication and memory are free within Complexity Theory. The *Theory of VLSI Complexity* aims at measuring, for all physical realizations of digital function f, the combined complexity of *communication, memory*, and *computation*. The *VLSI complexity* of function f is defined with respect to all possible chips for computing f. Implementations are all within the same silicon process, defined by some feature size, speed and design rules. Each design computes f within some area A, clock frequency F and T clock periods per IO sample. The silicon area A is used for storage, communication and computation, through transistors and wires. Optimal designs are selected, based on some performance measure. For our purposes: minimize the area A for computing function f, subject to the real time requirement $F/T < F_{io}$. In theory, one has to optimize among all designs for computing f. In practice, the search is reduced to structural decompositions into well known *standard components*: adders, multipliers, shifters, memories, ...

2.2 Trading size for speed

VLSI design allows trading area for speed. Consider, for example, the family of adders: their function is to repeatedly compute the binary sum $S = A+B$ of two n bits numbers A, B. Fig. 5 shows four adders, each with a different structure, performance, and mapping of the operands through time and IO ports. Let us analyze the VLSI performance of these adders, under simplifying assumptions: $a_{fa} = 2a_r$ for the area (based on transistor counts), and $d_{fa} = d_r$ for the combinatorial delays of $fadd$ and reg (setup and hold delay).

1. Bit serial (base 2) adder $sA2$. The bits of the binary sum appear through the unique output port as a time sequence $s_0, s_1, ..., s_n, ...$ one bit per clock cycle, from least to most significant. It takes $T = n + 1$ cycles per sum S. The area is $A = 3a_r$: it is the *smallest of all* adders. The chip operates at clock frequencies up to $F = 1/2d_r$: the *highest possible.*

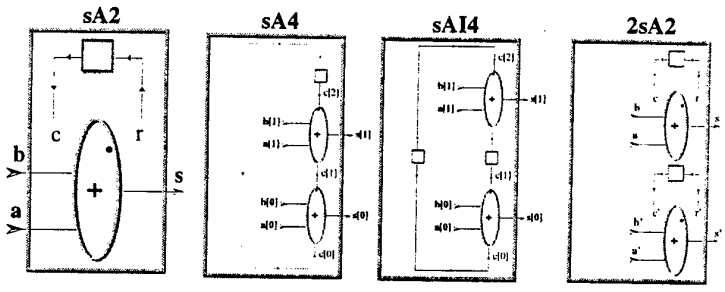


Fig. 5. Four serial adders: $sA2$ - base 2, $sA4$ - base 4, $sAI4$ - base 4 interleaved, and $2sA2$ - two independent $sA2$. An oval represents the full adder *fadd*; a square denotes the register *reg* (one bit synchronous *flip-flop*; the clock is implicit in the schematics).

2. Serial two bits wide (base 4) adder $sA4$. The bits of the binary sum appear as two time sequences s_0, s_2, ..., s_{2n}, ... and s_1, s_3, ... two bits per cycle, through two output ports. Assuming n to be odd, we have $T = (n+1)/2$ cycles per sum. The area is $A = 5a_r$ and the operating frequency $F = 1/3d_r$.
3. Serial interleaved base 4 adder $sAI4$. The bits of the binary sum S appear as two time sequences s_0, *, s_2, *, ..., s_{2n}, *, ... and *, s_1, *, s_3, ... one bit per clock cycle, even cycles through one output port, odd through the other. The alternate cycles (the *) are used to compute an independent sum S', whose IO bits (and carries) are interleaved with those for sum S. Although it still takes $n+1$ cycles in order to compute each sum S and S', we get *both* sums in so many cycles, at the rate of $T = (n+1)/2$ cycles per sum. The area is $A = 6a_r$ and the maximum operating frequency $F = 1/2d_r$.
4. Two independent bit serial adders $2sA2$. This circuit achieves the same performance as the previous: $T = (n+1)/2$ cycles per sum, area $A = 6a_r$ and frequency $F = 1/2d_r$.

The transformation that unfolds the base 2 adder $sA2$ into the base 4 adder $sA4$ is a special instance of a general procedure. Consider a circuit C which computes some function f in T cycles, within gate complexity G *bop* and memory M bits. The procedure from [11] unfolds C into a circuit C' for computing f: it trades cycles $T' = T/2$ for gates $G' = 2G$, at constant storage $M' = M$.

In the case of serial adders, the area relation is $A' = 5A/3 < 2A$, so that $A'T' < AT$. On the other hand, since $F' = 1/3d$ and $F = 1/2d$, we find that $A'T'/F' > AT/F$. An equivalent way to measure this, is to consider the density of full adders fadd per unit area $a_{fa} = 2a_r$, for both designs C and C': as $2/A = 0.66 < 4/A' = 0.8$, the unfolded design has a better *fadd* density than the original. Yet, since $F' = 1.5F$, the *compute density* - in *fadd* per unit area and time $d_{fa} = d_r$ - is lower for circuit C': $F/A = 0.16 > 2/A'F' = 0.13$. When we unfold from base 2 all the way to base $2n$, the carry register may be simplified away: it is always 0. The fadd densities of this n-bit wide carry propagate adder

is 1 per unit area, which is optimal; yet, as clock frequency is $F = 1/n$, the compute density is low: $1/n$.

Circuits $sAI4$ and $2sA2$ present two ways of *optimally* trading time for area, at *constant operator and compute density*. Both are instances of general methods, applicable to any function f, besides binary addition. From any circuit C for computing f within area A, time T and frequency F, we can derive circuits C' which optimally trades area $A' = 2A$ for time $T' = T/2$, at constant clock frequency $F' = F$. The trivial unfolding constructs $C' = 2C$ from two independent copies of C, which operate on separate IO. So does the interleaved adder $sAI4$, in a different manner. Generalizing the interleaved unfolding to arbitrary functions does not always lead to an optimal circuit: the extra wiring required may force the area to be more than $A' > 2A$. Also note that while these optimal unfolding double the throughput ($T = n/2$ cycles per add), the *latency* for each individual addition is not reduced from the original one ($T = n$ cycles per addition). We may constrain the unfolded circuit to produce the IO samples in the standard order, by adding reformatting circuitry on each side of the IO: a buffer of size n-bit, and a few gates for each input and output suffice. As we account for the extra area (for *corner turning*), we see that the unfolded circuit is no longer optimal: $A' > 2A$. For a complex function where a large area is required, the loss in *corner turning* area can be marginal. For simpler functions, it is not.

In the case of addition, area may be optimally traded for time, for all integer data bit width $D = n/T$, as long as $D < \sqrt{n}$. Fast wide $D = n$ parallel adders have area $A = nlog(n)$, and are structured as binary trees. The area is dominated by the wires connecting the tree nodes, their drivers (the longer the wire, the bigger the driver), and by pipelining registers, whose function is to reduce all combinatorial delays in the circuit below the clock period $1/F$ of the system.

Transitive functions permute their inputs in a rich manner (see [4]): any input bit may be mapped - through an appropriate choice of the external controls - into any output bit position, among N possible per IO sample. It is shown in [4] that computing a transitive function at IO rate $D = NF/T$, requires an area A such that:

$$A > a_m N + a_{io} D + a_w D^2, \tag{1}$$

where a_m, a_{io} and a_w are proportional to the area per bit respectively required for memory, IO and communication wires. Note that the gate complexity of a transitive function is zero: input bit values are simply permuted on the output. The above bound merely accounts for the area - IO ports, wires and registers - which is required to acquire, transport and buffer the data at the required rate. Bound (1) applies to shifters, and thus also to multipliers. Consider a multiplier that computes $2n$-bit products on each cycle, at frequency F. The wire area of any such multiplier is proportional to n^2, as $T = 1$ in (1). For high bandwidth multipliers, the area required for wires and pipelining registers is bigger than that for arithmetic operations.

The bit serial multiplier (see [11]) has a minimal area $A = n$, high operating frequency F, and it requires $T = 2n$ cycles per product. A parallel nave multiplier has area $A' = n^2$ and $T' = 1$ cycle per product. In order to maintain high

frequency $F' = F$, one has to introduce on the order of n^2 *pipelining registers*, so (perhaps) $A' = 2n^2$ for the *fully pipelined* multiplier. These are two extreme points in a range of optimal multipliers: according to bound (1), and within a constant factor. Both are based on nave multiplication, and compute n^2 mul per product. High frequency is achieved through deep pipelining, and the latency per multiplication remains proportional to n. In theory, latency can be reduced to T, by using reduced complexity $n^{1+\epsilon}$ shallow multipliers (see [3]); yet, shallow multipliers have so far proved bigger than nave ones, for practical values such as $n < 256$.

2.3 Experimental performance measures

Consider a VLSI design with area A and clock frequency F, which computes function f in T cycles per N-bit sample. In theory, there is another design for f which optimally trades area $A' = 2A$ for cycles $T' = T/2$, at constant frequency $F' = F$. The frequency F and the AT product remain invariant in such an optimal tradeoff. Also invariant:

- The gate density (in bop/mm^2), given by $D_{op} = c(f)/A = C(f)/AT$. Here $c(f)$ is the *bop* complexity of f per cycle, while $C(f)$ is the bop complexity per sample.
- The compute density (in bop/smm^2) is $c(f)F/A = FD_{op}$.

Note that trading area for time at constant gate and compute density is equivalent to keeping F and AT invariant.

Let us examine how various architectures trade size for performance, in practice. The data from [14] tabulates the area, frequency, and feature size, for a representative collection of chips from the previous decade: sRAM, DRAM, mPROC, FPGA, MUL.

The normalized area A/λ^2 provides a performance measure that is independent of the specific feature size λ. It leads [14] to a quantitative assessment of the gate density for the various chips, fig. 6 and 7.

Unlike [14], we also normalize clock frequency: the product by the operation density is the normalized compute power. To define the normalized the system clock frequency ϕ, we follow [9] and use $\phi = 1/100\tau(\lambda)$, where $\tau(\lambda)$ is the minimal inverter delay corresponding to feature size λ.

- The non linear formula used for $\tau((l) = cl^e$ is taken from [9]: the exponent $e = 1 - \epsilon(l)$ decreases from 1 to 0.9 as l shrinks from 0.3 to 0.03 μm. The non linear effect is not yet apparent in the reported data. It will become more significant with finer feature sizes, and clock frequency will cease to increase some time before the shrink itself stops.
- The factor 100 leads to normalized clock frequencies whose average value is 0.2 for DRAM, 0.9 for SRAM, 2 for processors and 2 for FPGA.

In the absence of architectural improvement, the normalized gate and compute density of the same function on two different feature size silicon implementations should be the same, and this indicates an optimal shrink.

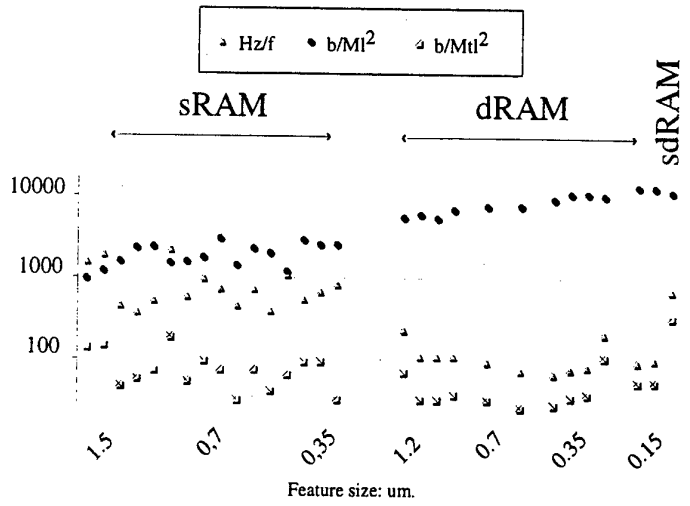


Fig. 6. Performance of various SRAM and DRAM chips, within to a common feature size technology: normalized clock frequency Hz/ϕ; bit density per normalized area $10^6\lambda^2$; binary gate operations per normalized area per normalized clock period $1/\phi$.

- The normalized performance figures for SRAM chips in fig. 6 are all within range: from one half to twice the average value.
- The normalized bit density for DRAM chips in the data set is 4.5 times that of SRAM. Observe in fig. 6 that it has increased over the past decade, as the result of improvements in the architecture of the memory cell (*trench* capacitors). The average normalized speed of DRAM is 4.5 times slower than SRAM. As a consequence the average normalized compute density of SRAM equals that of DRAM. The situation is different with SDRAM (last entry in fig. 6): with the storage density of DRAM and nearly the speed of SRAM, the normalized compute density of SDRAM is 4 times that of either: a genuine improvement in memory architecture.

A *Field Programmable Gate Array FPGA* is a mesh made of programmable gates and interconnect [17]. The specific function - Boolean or register - of each gate in the mesh, and the interconnection between the gates, is coded in some binary bitstream, specific to function f, which must first be downloaded into the *configuration memory* of the device. At the end of configuration, the FPGA switches to user mode: it then computes function f, by operating just as any regular ASIC would.

The comparative normalized performance figures for various recent microprocessors and FPGA is found in fig. 7.

- Microprocessors in the survey appear to have maintained their normalized compute density, by trading lower normalized operation density, for a higher

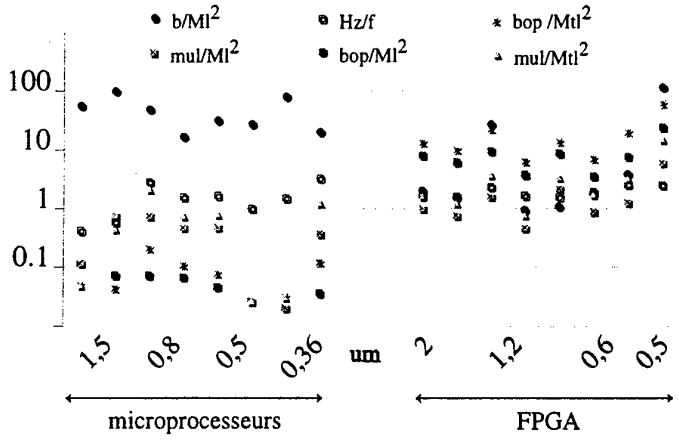


Fig. 7. Performance of various microprocessor and FPGA chips from [14], within a common feature size technology: normalized clock frequency Hz/ϕ; normalized bit density; normalized gate and compute density: for Boolean operations, additions and multiplication's.

normalized clock frequency, as feature size has shrunk. Only the microprocessors with a built-in multiplier have kept the normalized compute density constant. If we exclude multipliers, the normalized compute density of microprocessors has actually decreased through the sample data.

- FPGA have stayed much closer to the model, and normalized performances do not appear to have changed significantly over the survey (rightmost entry excluded).

3 Reconfigurable Systems

A *Reconfigurable System RS* is a standard sequential processor (the host) tightly coupled to a *Programmable Active Memory PAM*, through a high bandwidth link. The PAM is a reconfigurable processor, based on FPGA and SRAM. Through software configuration, the PAM emulate any specific custom hardware, within size and speed limits. The host can write into, and read data from the PAM, as with any memory. Unlike conventional RAM, the PAM processes data between write and read cycles: it an active memory. The specific processing is determined by the contents of its configuration memory. The content of configuration memory can be updated by the host, in a matter of milliseconds: it is programmable.

RS combine the flexibility of software programming to the performance level of application specific integrated circuits ASIC. As a point in case, consider the system P1 described in [13]. From the abstract of that paper:

> We exhibit a dozen applications where PAM technology proves superior, both in performance and cost, to every other existing technology, including supercomputers, massively parallel machines, and conventional custom hardware.
>
> The fields covered include computer arithmetics, cryptography, error correction, image analysis, stereo vision, video compression, sound synthesis, neural networks, high-energy physics, thermodynamics, biology and astronomy.
>
> At comparable cost, the computing power virtually available in a PAM exceeds that of conventional processors by a factor 10 to 1000, depending on the specific application, in 1992.

RS P1 is built from chips available in 92 - SRAM, FPGA and processor. Six long technology years later, it still holds at least 4 significant absolute speed records. In theory, it is a straightforward matter to port these applications on a state of the art RS, and enjoy the performance gain from the shrink. In the practical state of our CAD tools, porting the highly optimized P1 designs on oher systems would require time and skills. On the other hand, it is straightforward to estimate the performance without doing the actual hardware implementation. We use the Reconfigurable System P2 [16] - built in 97 - to conceptually implement the same applications as P1, and compare. The P2 system has 1/4 the physical size and chip count of P1. Both have roughly the same logical size (4k CLB), so the applications can be transferred without any redesign. The clock frequency is 66 MHz on P2, and 25MHz on P1 (and 33MHz for RSA). So, the applications will run at least twice faster on P2 than on P1. Of course, if we compare equal size and cost systems, we have to match P1 against 4P2, and the compute power has been multiplied by at least 8. This is expected by the theory, as the feature size of chips in P1 is twice that of chips in P2.

What has been done [20] is to port and run on recent fast processors, the software version for some of the original P1 applications. That provides us with a technology update on the respective compute power of RS and processors.

3.1 3D Heat Equation

The fastest reported software for to solving the Heat Equation on a supercomputer, is presented in [6]. It is based on the finite differences method. The Heat Equation can be solved more efficiently on specific hardware structures [7]:

- Start from an initial state - at time $t\Delta t$ - of the discrete temperatures in a discrete 3D domain, all stored in RAM.
- Move to the next state - at time $(t+1)\Delta t$ - by traversing the RAM three times, along the x, y and z axis.
- On each traversal, the data from the RAM feeds a pipeline of *averaging operators*, and the output of the pipeline is stored back in RAM.

Each *averaging operator* computes the average value $(a_t + a_{t+1})/2$ of two consecutive samples a_t and a_{t+1}. In order to be able to reduce the precision of internal

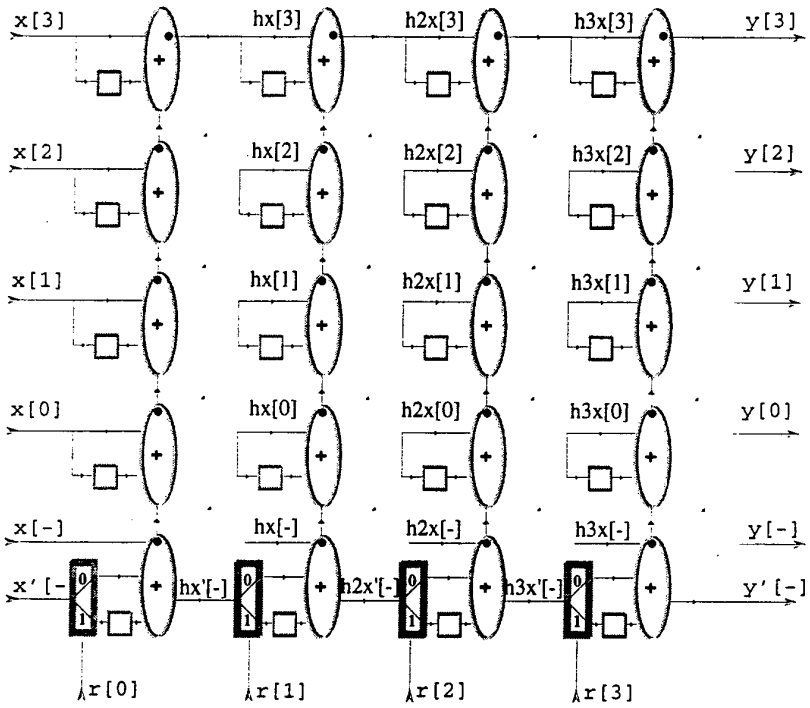


Fig. 8. Schematics of a hardware pipeline for solving the Heat equation. It is drawn with a pipeline depth of 4, and bit width of 4, plus 2 bits for randomized round off. The actual 1 pipeline is 256 deep, and 16+2 wide. Pipelining registers, which allow the network to operate at maximum clock frequency, are not indicated here. Neither is the random bit generator.

temperatures down to 16 bits, it is necessary, when division by two is odd, to distribute that low-order bit randomly between the sample and its neighbor. All deterministic round-off schemes lead to parasitic effects that can significantly perturb the result. The pseudo-randomness is generated by a 64-bit *linear feedback shift-register LFSR*. The resulting pipeline is shown in fig. 8. Instead of being shifted, the least significant sum bit is either delayed or not, based on a random choice in the LFSR.

P1 standing design can accurately simulate the evolution of temperature over time in a 3D volume, mapped on 512^3 discrete points, with arbitrary power source distributions on the boundaries. In order to reproduce that computation in real time, it takes a 40,000 MIPS equivalent processing power: 40 G instructions per second, on $32b$ data. This is out of the reach of microprocessors, at least until 2001.

3.2 High Energy Physics

The *Transition Radiation Tracker TRT* is part in a suite of benchmarks proposed by CERN [12]. The goal is to measure the performance of various computer architectures in order to build the electronics required for the Large Hadron Collider LHC, soon after the turn of the millennium. Both benchmarks are challenging, and well documented for a wide variety of processing technologies, including some of the fastest current computers, DSP-based multiprocessors, systolic arrays, massively parallel arrays, Reconfigurable Systems, and full custom ASIC based solutions.

The TRT problem is to find straight lines (particle trajectories) in a noisy digital black and white image. The rate of images is at 100 kHz; the implied IO rate close to 200 MB/s, and the low latency requirement (2 images) preclude any implementation solution other specialized hardware, as shown by [12].

The P1 implementation of the TRT is based on the *Fast Hough Transform* [10], an algorithm whose hardware implementation trades computation for wiring complexity. To reproduce the P1 performance reported in [12], a 64-bit sequential processor needs to run at over 1.2 GHz. That is about the amount of computation one gets, in 1998, with a dual processor, 64-bit machine, at 600 MHz. The required external bandwidth (up to 300 MB/s) is what still keeps such application out of current microprocessor reach.

3.3 RSA cryptography

The P1 design for RSA cryptography combines a number of algorithm techniques, presented in [8]. For 512-bit keys, it delivers a decryption rate in excess of 300 kb/s, although it uses only half the logical resources available in P1.

The implementation takes advantage of hardware reconfiguration in many ways: a rather different design is used for RSA encryption and decryption; a different hardware modular multiplier is generated for each different prime modulus: the coefficients of the binary representation of each modulus is hardwired

into the logical equations of the design. None of these techniques is readily applicable to ASIC implementations, where the same chip must do both encryption and decryption, for all keys.

As of printing time, this design still holds the acknowledged shortest time per block of RSA, all digital species included. It is surprising that it has held five years against other RSA hardware. According to [20], the record will go to a (soon to be announced) Alpha processor (one 64b multiply per cycle, at 750MHz) running (a modified version of) the original software version in [8]. We expect the record to be claimed back in the future by a P2 RSA design; yet, the speedup between P1 was 10x reported in 92, and we estimate that it should be only be 6x on 2P2, in 97. The reason: the fully pipelined multiplier, found in recent processors, is fully utilized by RSA software. A normalized measure of the impact of multiplier on theoretical performance can be observed in fig. 7.

For the Heat Equation, the actual performance ratio between P1 and the fastest processor (64b, 250MHz) was 100x in 92; with 4P2 against the 64b, 750MHz processor, the ratio should be over 200x in 98. Indeed, the computation in fig. 8 combines 16 b add and shift, with Boolean operations on three low order bits: software is not efficient, and the multiplier is not used.

4 What will Digital Systems shrink to?

Consider a DS whose function and real time frequency remain fixed, once and for all. Examples: digital watch, $56kb/s$ modem and GPS.

How does such DS shrink with feature size?

To answer, start from the first chip (feature size 1) which computes function f: area A, time T, and clock frequency F. Move in time, and shrink feature size to $1/2$. The design now has area $A' = A/4$, and the clock frequency doubles $F' = 2F$ ($F' = (2-\epsilon)F$ with non-linear shrink). The number of cycles per sample remains the same: $T' = T$. The new design has twice (or $2-\epsilon$) the required real time bandwidth: we can (in theory) further fold space in time: produce a design C'' for computing f within area $A'' = A'/2 = A/8$ and $T'' = 2T$ cycles, still at frequency $F'' = F' = 2F$. The size of any fixed real time DS shrinks very fast with technology, indeed. At the end of that road, after so many hardware shrinks, the DS gets implemented in software.

On the other hand, microprocessors, memories and FPGA actually grow in area, as feature size shrinks. So far, such commodity products have each aimed at delivering ever more compute power, on one single chip. Indeed, if you look inside some recent digital device, chances are that you will see mostly three types of chips: RAM, processor and FPGA. While a specific DS shrinks with feature size, a general purpose DS gains performance through the shrink, ideally at constant normalized density.

4.1 System on a chip

There are compelling reasons for wanting a Digital System to fit on a single chip. Cost per system is one. Performance is another:

- Off-chip communication is expensive, in area, latency and power. The bandwidth available across some on-chip boundary is orders of magnitude that across the corresponding off-chip boundary.
- If one quadruples the area of a square, the perimeter just doubles. As a consequence, when feature size shrinks by $1/x$, the internal communication bandwidth grows faster than the external IO bandwidth: $x^{3-\epsilon}$ against $x^{2-\epsilon}$. This is true as long as silicon technology remains planar: transistors within a chip, and chips within a printed circuit board, must all layed out side by side (not on top of each other).

4.2 Ready to Shrink Architecture

So far, normalized performance density has been maintained, through the successive generations of chip architecture.

Can this be sustained in future shrinks?

A dominant consideration is to keep up the system clock frequency F. The formula for the normalized clock frequency $1/\phi = 100\tau(\lambda)$ implies that each combinatorial sub-circuit within the chip must have delay less than 100x that of a minimal size inverter. The depth of combinatorial gates that may be traversed along any path between two registers is limited. The length of combinatorial paths is limited by wire delays. It follows that only finitely many combinatorial structures can operate at normalized clock frequency ϕ. There is a limit to the number N of IO bits to any combinatorial structure which can operate at such a high frequency. In particular, this applies to combinatorial adders (say $N < 256$), multipliers (say $N < 64$) and memories.

4.3 Reconfigurable Memory

The use of fast SRAM with small block size is common in microprocessors: for registers, data and instruction caches. Large and fast current memories are made of many small monolithic blocks. A recent SDRAM is described in [15]: $1Gb$ stored as 32 combinatorial blocks of $32Mb$ each. A 1.6 GB/s bandwidth is obtained: data is $64b$ wide at 200MHz.

By the argument from the preceding section, a large N bit memory must be broken into N/B combinatorial blocks of size B, in order to operate at normalized clock frequency $F = \phi$. A N bit memory with minimum latency may be constructed, through recursive decomposition into 4 quad memories, each of size $N/4$ - layed out within one quarter of the chip. The decomposition stops for $N = B$, when a block of combinatorial RAM is used. The access latency is proportional to the depth $log(N/B)$ of the hierarchical decomposition.

A *Reconfigurable Memory RM* is an array of high speed dense combinatorial memory blocks. The blocks are connected through a reconfigurable pipelined

wiring structure. As with FPGA, the RM has a configuration mode, during which the configuration part of the RM is loaded. In user mode, the RM is some group of memories, whose specific interconnect and block decomposition is coded by the configuration. One can trade data width for address depth, from $1 \times N$ to $N/B \times B$ in the extreme cases.

A natural way to design a RM is to imbed blocks of SRAM within a FPGA structure. In CHESS [19], the atomic SRAM block has size 8×256. The SRAM blocks form a regular pitch matrix within the logic, and it occupies about 30% of the area. As a consequence, the storage density of CHESS is over 1/3 that of a monolithic SRAM. This is comparable to the storage density of current microprocessors; it is much higher than the storage density of FPGA, which rely (so far) on off-chip memories.

After configuration, the FPGA is a large array of small SRAM: each is used as LUT - typically LUT4. Yet, most of the configuration memory itself is not accessible as a computational resource by the application. In most current FPGA, the process of downloading the configuration is serial, and it writes the entire configuration memory. In a 0.5x shrink, the download time doubles: 4x bits at (2-e)x the frequency. As a consequence, the download takes about 20 ms on P1, and 40 ms on P2.

A more efficient alternative is found in the X6k [17] and CHESS: in configuration mode, configuration memory is viewed as a single SRAM by the host system. This allows for faster complete download. An important feature is the ability to randomly access the elements of the configuration memory. For the RSA design, this allows for very fast partial reconfigurations: as we change the value of the $512b$ key which is hardwired into the logical equations, only few of the configuration bits have to updated. Configuration memory can also be used as a general-purpose communication channel between the host and the application.

4.4 Reconfigurable Arithmetic Array

The normalized gate density of current FPGA is over 10x that of processors, both for Boolean operations and additions - fig. 7. This is no longer true for the multiply density, where common FPGA barely meets the multiply density of processors which recently integrate one (or more) pipelined floating point multiplier.

The arithmetical density of RS can be raised: MATRIX [DeHon], which is an array of 8b ALU, with Reconfigurable Interconnect, does better than FPGA. CHESS is based on $4b$ ALU, which are packed as the *white* squares in a chessboard. It follows that CHESS has an arithmetic density which is near 1/3 that of custom multipliers. The synchronous registers in CHESS are $4b$ wide, and they are found both within ALU and routing network, to as to facilitate high speed systematic pipelining.

Another feature of CHESS [19], is that each *black* square in the chessboard may be used either as a switchbox, or as a memory, based on a local configuration bit. As a switchbox, it operates on $4b$ *nibbles*, which are all routed together. In

memory mode, it may implement various specialized memories, such as a depth 8 shift register, in place of eight 4b wide synchronous registers. In memory mode, it can also be used as a 4b in, 4b out $4LUT4$. This feature provides CHESS with a LUT4 density which is as high as for any FPGA.

4.5 Hardware or Software?

In order to implement digital function $Y = f(X)$, start from a specification by a program in some high level language. Some work is usually required to have the code match the digital specification, bit per bit - high level languages provide little support for funny bit formats and operations beneath the word size.

Once done, compile and unwind this code so as to obtain the run-code C_f. It is the sequence of machine instructions, which a sequential processor executes, in order to compute output sample Y_t from input sample X_t. This computation is to be repeated indefinitely, for consecutive samples: t=0, 1, For the sake of simplicity, assume the run-code to be straight-line: each instruction is executed once in sequence, regardless of individual data values; there is no conditional branch. In theory, the run-code should be one of minimal length, among all possible for function f, within some given instruction set. Operations are performed in sequence through the Arithmetic and Logic Unit ALU of the processor. Internal memory is used to feed the ALU, and provide (memory-mapped) external IO. For W the data width of the processor, the complexity of so computing f is $W|C_f|$ bop per sample. It is greater than the gate complexity $G(f)$. Equality $|C_f| = G(f)/W$ only happens in ideal cases. In practice, the ratio between the two can be kept close to one, at least for straight-line code.

The execution of run-code Cf on a processor chip at frequency F computes function f at the rate of F/C samples per second, with $C = |C_f|$. The feasibility of a software implementation of the DS on that processor depends on the real time requirement F_{io} - in samples per second.

1. If $F/C > F_{io}$, the DS can be implemented on the sequential processor at hand, through straightforward software.
2. If $F/C < F_{io}$, one needs a more parallel implementation of the digital system.

In case 1, the full computing power - WF in bop/s - of the processor is only used when $F/C = F_{io}$. When that is not the case, say $F/C > 2F_{io}$, one can attempt to trade time for area, by reducing the data width to $W/2$, while increasing the code length to $2C$: each operation on W bits is replaced by two operations on $W/2$ bits, performed in sequence. The invariant is the product CW, which gives the complexity of f in *bop* per sample. One can thus find the smallest processor on which some sequential code for f can be executed within the real time specification. The end of that road is reached for $W = 1$: a *single bit wide sequential processor*, whose run-code has length proportionnal to $G(f)$.

In case 2, and when one is not far away from meeting the real time requirement - say $F/C < 8F_{io}$ - it is advised to check if code C could be further reduced, or moved to a wider and faster processor (either existing or soon to come when

the feature size shrinks again). Failing that software solution, one has to find a hardware one. A common case mandating a hardware implementation, is when $F \approx F_{io}$: the real time external IO frequency F_{io} is near the internal clock frequency F of the chip.

4.6 Dynamic Reconfiguration

We have seen how to fold time in space: from a small design into a larger one, with more performance. The inverse operation, which folds space in time, is not always possible: how to fold any bit serial circuit (such as the adder from fig 5) into a half-size and half-rate structure is not obvious. Known solutions involve dynamic reconfiguration.

Suppose that function f may be computed on some RS of size $2A$, at twice the real-time frequency $F = 2F_{io}$. We need to compute f on a RS of size A at frequency F_{io} per sample. One technique, which is commonly used in [13], works when $Y = f(X) = g(h(X))$, and both g and h fit within size A.

1. Change the RS configuration to design h.
2. Process N input samples X; store each output sample $Z = h(X)$ in an external buffer.
3. Change the RS configuration to design g.
4. Process the N samples Z from the buffer, and produce the final output $Y = g(Z)$.
5. Go to 1, and process the next batch of N samples.

Reconfiguration takes time R/F, and the time to process N samples is $2(N + R)/F = (N + R)/F_{io}$. The frequency per sample $F_{io}/(1 + R/N)$ gets close to real-time F_{io}, as N gets large. Buffer size and latency are also proportional to N, and this form of dynamic reconfiguration may only happen at a low frequency.

The opposite situation is found in the ALU of a sequential processor: the operation may change on every cycle. The same holds in *dynamically programmable* systems, such as arrays of processors and DPGA [14]. With such a system, one can reduce by half the number of processors for computing f, by having each execute twice more code. Note that this is a more efficient way to fold space in time than previously: no external memory is required, and the latency is not significantly affected.

The ALU in CHESS is also dynamically programmable. Although no specialized memory is provided for storing instructions (unlike DPGA), it is possible to build specialized *dynamically programmed* sequential processors, within the otherwise *statically configured* CHESS array. Through this feature, one can modulate the amount of parallelism in the implementation of a function f, in the range between serial hardware and sequential software, which is not accessible without dynamic reconfiguration.

5 Conclusion

We expect it to be possible to build *Reconfigurable Systems* of arbitrary size, which are fit to shrink: they can exploit all the available silicon, with a high

normalized density for storage, arithmetic and Boolean operations, and operate at high normalized clock frequency.

For how long will the market demands for operations keep-up with the supply which such RS promise?

Can the productivity in mapping applications to massively parallel custom hardware be raised at the pace set by technology?

Let us take the conclusion from Carver Mead [9]:

> There is far more potential in a square centimeter of silicon than we have developed the paradigms to use.

References

1. C. E. Shannon, W. Weaver, *The Mathematical Theory of Communication*, University of Illinois Press, Urbana, 1949.
2. C. Mead, L. Conway *Introduction to VLSI systems*, Addison Wesley, 1980.
3. F.P. Preparata and J. Vuillemin. *Area-time optimal VLSI networks for computing integer multiplication and Discrete Fourier Transform*, Proceedings of I.C.A.L.P (Springer-Verlag), Haifa, Israel, Jul. 1981.
4. J. Vuillemin, *A combinatorial limit to the computing power of VLSI circuits*, IEEE Transactions on Computers, C-32:3:294-300, 1983.
5. I. Wegener *The Complexity of Boolean Functions*, John Wiley & sons, 1987.
6. O. A. McBryan, P. O. Frederickson, J. Linden, A. Schüller, K. Solchenbach, K. Stüben, C-A. Thole and U. Trottenberg, "Multigrid methods on parallel computers—a survey of recent developments", *Impact of Computing in Science and Engineering*, vol. 3(1), pp. 1–75, Academic Press, 1991.
7. J. Vuillemin. *Contribution à la résolution numérique des équations de Laplace et de la chaleur*, Mathematical Modelling and Numerical Analysis, AFCET, Gauthier-Villars, RAIRO, 27:5:591–611, 1993.
8. M. Shand and J. Vuillemin. *Fast implementation of RSA cryptography*, 11-th IEEE Symposium on Computer Arithmetic, Windsor, Ontario, Canada, 1993.
9. C. Mead, *Scaling of MOS Technology to Submicrometre Feature Sizes*, Journal of VLSI Signal Processing, V 8, N 1, pp. 9-26, 1994.
10. J.Vuillemin. *Fast linear Hough transform*, The International Conference on Application-Specific Array Processors, IEEE press, 1–9, 1994.
11. J. Vuillemin. *On circuits and numbers*, IEEE Trans. on Computers, 43:8:868–79, 1994.
12. L. Moll, J. Vuillemin, P. Boucard and L. Lundheim, *Real-time High-Energy Physics Applications on DECPeRLe-1 Programmable Active Memory*, Journal of VLSI Signal Processing, Vol 12, pp. 21-33, 1996.
13. J. Vuillemin, P. Bertin , D. Roncin, M. Shand, H. Touati, P. Boucard *Programmable Active Memories: the Coming of Age*, IEEE Trans. on VLSI, Vol. 4, NO. 1, pp. 56-69, 1996.
14. A. DeHon *Reconfigurable Architectures for General-Purpose Computing*, MIT, Artificial Intelligence Laboratory, AI Technical Report No. 1586, 1996.
15. N. Sakashita & al., *A 1.6-GB/s Data-Rate 1-Gb Synchronous DRAM with Hierarchical Square-Shaped Memory Block and Distributed Bank Architecture*, IEEE Journal of Solid-state Circuits, vol. 31, No 11, pp 1645-54, 1996.

16. M. Shand, *Pamette*, a Reconfigurable System for the PCI Bus, 1998.
http://www.research.digital.com/SRC/pamette/
17. Xilinx, Inc., *The Programmable Gate Array Data Book*, Xilinx, 2100 Logic Drive, San Jose, CA 95124 USA, 1998.
18. G. Moore. *An Update on Moore's Law*, 1998.
http://www.intel.com/pressroom/archive/speeches/gem93097.htm
19. Alan Marshall, Tony Stansfield, Jean Vuillemin *CHESS: a Dynamically Programmable Arithmetic Array for Multimedia Processing*, Hewlett Packard Laboratories, Bristol, 1998.
20. M. Shand. *An Update on RSA software performance*, private communication, 1998.
21. *The millenium bug: how much did it really cost?*, your newspaper, 2001.

A Method Based on Orthogonal Transformation for the Design of Optimal Feedforward Network Architecture

Bachiller P., Pérez R.M., Martínez P., Aguilar P.L., Calle J.E.

Department of Computer Sciences. University of Extremadura. Escuela Politécnica. 10071 Cáceres. Spain.
{Pilarb, Rosapere, Pablomar, Paguilar}@unex.es

Abstract. The problem of determining the optimum size of a feedforward neural network is recognized to be crucial for its practical implications in such important issues as learning and generalization. Several approaches for designing optimum size networks have been proposed in the literature, which consist of training a larger than necessary network and then removing the unnecessary links and nodes. In this kind of approaches, commonly known as pruning, before computing the optimum number of links and nodes it is necessary to train the network and, once they have been identified, the reduce-size network has to be retrained. In this paper, a direct method to obtain an optimum size network during its training process is presented. We use orthogonal transformations for computing the optimum number of nodes on each iteration of the training process. These transformations lead to a decorrelation of the information, which is the key of network size reduction.

1 Introduction

The back-propagation algorithm has emerged as one of the most popular for supervised training neural networks. This algorithm is extremely computation and storage demanding. An enormous amount of computation has to be spent on training the network and, in the retrieving phase, high throughputs are required for real-time processing which hinges on its massively parallel processing capability.

Multiprocessors, array of processors and massively parallel processors provide a natural solution to the BP algorithm, which can be expressed in basic matrix operations, such as inner-product, outer-product and matrix multiplications. For instance, this kind of operations can be mapped to basic processor arrays, systolic or wavefront arrays. They have the following key advantages:

- The exploitation of pipelining is very natural in regular and locally connected networks. They yield high throughput and simultaneously save the cost associated with communication.
- They provide a good balance between computation and communication, which is critical to the effectiveness of array computing.

An open question related to neural networks is how to determine the most appropriate network size for solving an specific task. To be representative, the network should have an optimum number of links and nodes. Moreover, from an implementation standpoint, small networks only require limited resources in any physical computational environment. The network will be overparametrized if the number of links is very high. In such cases, if the training set of data is not noise-free, the NN will try to learn the information along with the noise in the data, leading to poor validation results.

There are several approaches to solve the problem of determining the optimum size of a neural network. The first approach, called growing algorithm, adds gradually hidden units to an initial small network until it reaches the convergence [1]-[4]. The second one, known as pruning, consists of training a larger than necessary network, then remaining nodes are eliminated and finally the reduced-size network has to be retrained [5][6].

Pratim Kangilal and Narayan Banerjee [5] have proposed an approach for the optimization of the size of feedforward neural networks using orthogonal transformations. They used two orthogonal transformations, the singular value decomposition (SVD) [7] and the QR with column pivoting factorization (QRcp) [7]. Using SVD, the rank of a matrix can be computed and so the optimum number of parameters is determined. QRcp coupled with SVD is used for subset selection, which is the key of the design of optimal networks.

The use of the above orthogonal transformations for the NN size optimization depends on which nodes (input or hidden nodes) are going to be optimized:

1. Optimum number of input nodes: Let PxN matrix A comprises the input data sets, where P is the number of sets of data points (training patterns), and N is the number of inputs. The aim is to determine which of the N features are relatively redundant and, hence, can be eliminated. Performing SVD on A, the optimum number of input nodes of the neural network (say L) is determined for the input data sets. QRcp provides L of the N features, for the P sets of data points, which are enough for a correct training process.
2. Optimum number of hidden links and nodes: Consider a network, which has been trained with P input data sets. A PxM matrix B is formed with the M pseudo outputs of the concerned hidden layer for each of the P input data sets. SVD is performed on B for determining the enough number of hidden nodes for the given network. In case of a non-homogeneous network, i.e. when hidden nodes are fed with different sets of inputs, QRcp transformation is performed on B and the specific links between the hidden layers to be retained are identified. Once remaining nodes have been eliminated, the reduced-size network is retrained.

Castellano et al. [6] have developed a pruning algorithm based on the idea of iteratively removing hidden units of a large trained network and then adjusting the remaining weights in order to maintain the original input-output behavior.

In the above approaches, it is necessary to train the network before computing the optimum number of hidden nodes and, once they have been identified, remaining weights have to be adjusted. In this paper, we propose a method to obtain a network with optimum number of hidden nodes at the same time as it is trained. It is based on

computing the optimum number of hidden units on each iteration of the training process, and then updating only the weights connected to those hidden units.

2 Orthogonal Transformations

In order to compute the optimum size of a feedforward neural network, we apply orthogonal transformations. An important property of them is that the vector 2-norm, as well as the matrix 2-norm, and the Frobenius norm, are invariant under the application of this kind of transformations.

In particular, we use the properties of Householder reflections for computing the optimum number of hidden nodes on each iteration of the training process of the network. These transformations are described as follows [7]:

Let $v \in \Re^n$ be nonzero. An nxn matrix P of the form

$$P = I - 2vv^T/v^Tv \quad (1)$$

is called a Householder reflection. The vector v is called a Householder vector.

It can be shown easily that matrix P is symmetric and orthogonal:

- Symmetric:

$$P^T = I - 2(vv^T)^T/v^Tv = I - 2vv^T/v^Tv = P \quad (2)$$

- Orthogonal:

$$P^TP = I + 4vv^Tvv^T/v^Tvv^Tv - 4vv^T/v^Tv = I \quad (3)$$

Householder reflections can be used to zero selected components of a vector. Given a vector $0 \neq x \in \Re^n$, if we want Px to be multiple of e_1 (the first column of the nxn identity matrix), then, for any $x \in \Re^n$, v must be defined as follows:

$$Px = (I - 2vv^T/v^Tv)x = x-(2v^Tx/v^Tv)v \quad (4)$$

Setting $v=x+\alpha e_1$ gives

$$v^Tx = x^Tx + \alpha x_1 \quad (5)$$

$$v^Tv = x^Tx + 2\alpha x_1 + \alpha^2 \quad (6)$$

If we assume $\alpha = \pm \|x\|_2$ (2-norm of the vector x)

$$v = x \pm \|x\|_2 e_1 \Rightarrow Px = (I - 2vv^T/v^Tv)x = \pm \|x\|_2 e_1 \quad (7)$$

Given m vectors $\in \Re^n$ $[x_1,x_2,...,x_m]$, Householder reflections are used to determine which of them are linearly independent. Firstly, a Householder matrix (H_1) to zero the last $n-1$ components of x_1 is calculated. Next, the vector $y=H_1x_2$ is obtained. If $\|y'\|_2=\|x_2\|_2$, then x_2 is linearly dependent on x_1; otherwise, another Householder matrix (H_2) has to be computed to zero the last $n-2$ components of y, and matrix H_1 must be updated with the product H_2H_1. The vector y' is formed by the l first components of vector y, where l represents the number of linearly independent vectors obtained on each step. Now the vector y is obtained by the product of H_1 and x_3 and the equality

$\|y'\|_2=\|x_3\|_2$ is proved to determine the correlation degree among x_1, x_2 and x_3. Remaining vectors are used of the same way to prove the linear dependencies between all of them.

For instance, assume that we have three vectors $\in \Re^n$, x_1, x_2 and x_3, and that x_3 is a linear combination of x_1 and x_2. In such case, the linear dependency between those vectors can be observed applying Householder reflections. The first step is to compute the Householder matrix (H_1) that transforms x_1 into a multiple of e_1. Next, the vector $y=H_1x_2$ is computed. It can be observed that the equality $\|y'\|_2=\|x_2\|_2$, where y' is the first component of y, doesn't hold due to x_1 and x_2 are linearly independent. A new Householder reflection is computed in order to zero the last $n-2$ components of vector y. To prove the linear dependency of x_1, x_2 and x_3, a new vector y_2 has to be obtained by the product $H_2H_1x_3$. As x_3 is a linear combination of x_1 and x_2, it can be expressed as ax_1+bx_2 (with a and $b \in \Re$), so the product $H_2H_1x_3$ can be obtained as follows:

$$y_2 = H_2H_1x_3 = H_2H_1(ax_1 + bx_2) = aH_2H_1x_1 + bH_2y \tag{8}$$

$$y_2 = a[c_1\ 0\ \ldots\ 0]^T + b[d_1\ d_2\ 0\ \ldots\ 0]^T = [n_1\ n_2\ 0\ \ldots\ 0]^T \tag{9}$$

Equation (9) shows that all the components of y_2 are equal to zero, excepting the two first ones. Since Householder matrices are orthogonal, the equality $\|y_2\|_2=\|x_3\|_2$ holds. So, if the 2-norm of x_3 can be computed using only the two first components of y_2, the linear dependency among x_1, x_2 and x_3 is verified.

3 The Proposed Optimizing and Training Algorithm

The optimization of the size of a feedforward neural network is a very important issue of its design, since any network should have an optimum number of links and nodes to be representative. This aim can be achieved retaining only the most representative nodes and deleting all the others. The selection process hinges upon the linear dependency of the nodes. For instance, assume a feedforward neural network with three nodes on its hidden layer, where the output value for the third hidden node is linearly dependent on the output values of the rest of the hidden nodes for the set of training patterns. In such case, the third hidden node could be eliminated due to the net inputs of the subsequent layer can be obtained using only the first two hidden nodes.

The method we propose in this paper is based on the idea of determining, on each iteration of the training process, the number of linearly independent outputs of the hidden layer, say l, and then updating only the weights of the links connected with the first l hidden nodes.

In order to compute the optimum number of hidden nodes using Householder reflections, the N outputs of the concerned hidden layer have to be obtained for each pattern of the set of training patterns. Thus, P N-dimensional vectors are formed containing the outputs at the hidden layer for the input data set, where P represents the total number of training patterns. After presentation of the first training pattern,

the first vector x_1 is obtained and the Householder reflection H_1 to zero the last N-1 components of this vector is computed. Next, for each pattern i, a new vector x_i is composed and it is proved its linear dependency with regards to the i-1 vectors computed in previous steps.

Assume that L is the number of linearly independent vectors found at the i-th step. This means that the Householder matrix H computed in previous steps zeroes at least the last N-L components of each vector of the set $[x_1, ..., x_{i-1}]$. If the new vector x_i is linearly dependent on $[x_1, ..., x_{i-1}]$, then the product Hx_i must be a vector of the form:

$$Hx_i = [n_1 \ldots n_L\ 0 \ldots 0]^T \qquad (10)$$

However, if equation (10) does not hold, matrix H has to be updated using a new Householder reflection H_{L+1} to zero the last N-L-1 components of the vector obtained in (10). Thus, matrix H must be computed as the product $H_{L+1}H$.

3.1 Our Algorithm

Consider a feedforward neural network with N input nodes, M hidden nodes and O output nodes and P training patterns. Assume that L is the optimum number of hidden nodes computed at each iteration of the algorithm, being L equal to M at the beginning of the first iteration. The proposed optimizing and training algorithm is as follows:

1) Update the connection weights (w_{ji}) from the input layer to the hidden layer for each of the P training patterns, using the back-propagation algorithm:

$$w_{ji} = w_{ji} + \alpha \sum_{k=1}^{O} (\delta_{pk}\, w'_{kj}) f'(Net_j) x_{pi} \qquad 1 \le p \le P \qquad (11)$$

where f is the activation function of each neuron j, α is a constant which determines the learning rate, x_{pi} is the i-th input of the pattern p, δ_{pk} is the error of the k-th output node for the pattern p and w'_{kj} is the connection weight from the j-th hidden node to the k-th output node.

Since w'_{kj} is zero for j greater than L, only the weights connected to the first L hidden nodes will be updated.

2) Compute the number of non-redundant hidden nodes (L) and update the connection weights from such nodes to the subsequent layer. At the beginning of this step, L is equal to zero.

> **2.1)** Obtain a vector x_p formed by the hidden outputs for the concerned training pattern (p). Next, a new vector y is computed by the product Hx_p, being MxM matrix H the product of all the Householder reflections computed at the previous p-1 iterations of this step. At iteration 1, H is the MxM identity matrix.
>
> **2.2)** In order to prove if vector x_p is linearly dependent on the vectors (x_1, ..., x_{p-1}) formed by the hidden outputs for the previous p-1 training patterns, the following equation has to be verified:
>
> $$\|y'\|_2 = \|x_p\|_2 \qquad (12)$$

where y' is an L-dimensional vector composed by the first L components of vector y.

If equation (12) holds, then x_p is linearly dependent on $(x_1, ..., x_{p-1})$. Otherwise, the optimum number of hidden nodes is increased ($L=L+1$) and matrix H is updated with the product $H'H$, where H' is the Householder matrix that zeroes the last $M-L-1$ components of vector y.

2.3) Update the connection weights (w'_{jk}) from the first L hidden nodes to the subsequent layer:

$$w'_{jk} = w'_{jk} + \alpha(d_{pj}-y_{pj})f'(Net_j)y_{pk} \qquad (13)$$

where d_{pj} and y_{pj} are the desired and obtained outputs of the j-th output node for the pattern p, respectively, and y_{pk} is the output of the k-th hidden node for such pattern.

2.4) Go to step 2.1 until connection weights of the concerned hidden layer are updated for all the training patterns.

3) Go to step 1 until the network reaches the convergence.

Remarks of the algorithm:

1. Network outputs are computed using only the optimum hidden nodes of the previous algorithm iteration. So, at the beginning of the algorithm, network outputs are obtained considering M hidden nodes.
2. At step 2.2, it is not necessary to calculate explicitly matrix H' and then compute the product $H'H$, since the structure of a Householder reflection can be applied directly for updating a matrix.
3. The initial number of hidden units depends on the specific problem to solve. However it will be always less or equal than the number of training patterns.
4. Once the network is trained, the last M-L nodes of the hidden layer can be eliminated, since its weights to the subsequent layer are zero.
5. In case of a network with more than one hidden layer, once the weights of the first hidden layer have been updated, step 2 has to be applied again for the subsequent hidden layers.

3.2 Comparison between the original back-propagation method and our optimizing and training algorithm

In order to show the performance of our algorithm, we make a comparison in terms of computational cost between this approach and the original back-propagation algorithm.

The following table shows the differences on number of operations between both algorithms assuming an NxMxO neural network, P training patterns and L optimum hidden nodes.

Table 1. Number of operations at different steps of both, the original back-propagation algorithm and the optimizing and training algorithm.

Step	Back-propagation Algorithm	Optimizing and Training Algorithm
Compute Network Outputs	PxNxM + PxMxO	PxNxM + PxLxO
Update Input Layer Weights	PxNxM	PxNxL
Compute Optimum Number of Hidden Nodes	-	PxMxL
Compute Householder Reflections	-	LxMx(M+1)
Update Hidden Layer Weights	PxMxO	PxLxO

Network outputs are obtained applying the following equations:

$$y_j' = f(\sum_{i=1}^{N} w_{ji}\, x_i) \qquad 1 \le j \le M \tag{14}$$

$$y_k = f(\sum_{j=1}^{M} w'_{jk}\, y'_j) \qquad 1 \le k \le O \tag{15}$$

where y_j' are the outputs of the hidden layer, y_k are the outputs of the output layer, x_i are the network inputs and w_{ji} and w'_{jk} are the weights of input and hidden layers, respectively.

In the original Back-propagation algorithm, M hidden nodes are used to compute the network outputs for each training pattern, so PxNxM + PxMxO operations are required. In the optimizing and training algorithm L hidden nodes are only needed to compute the outputs of the last layer. However the M outputs of the hidden layer have to be obtained in order to prove equation (12), so this step entails PxNxM + PxLxO operations for the proposed algorithm.

To compute the optimum number of hidden nodes, it is necessary to verify equation (12) at each process iteration. Vector y' is obtained by the first L components of the product Hx, so MxL operations are needed for each iteration at this step. Since P is the number of iterations, the total number of operations is PxMxL.

If equation (12) does not hold, a new Householder reflection H' is computed and matrix H has to be updated with the product $H'H$. Instead of forming explicitly matrix H' and then computing $H'H$, which implies a matrix-matrix multiplication, the structure of H' can be applied directly using the equation:

$$H'H = (I - 2vv^T/v^Tv)H = H - v(2v^TH/v^Tv) \tag{16}$$

where v is the Householder vector for the matrix H'.

Thus, a Householder update of a matrix involves a matrix-vector multiplication followed by an outer product update, which entails Mx(M+1) operations.

Since L is the optimum number of hidden nodes computed by the algorithm, L Householder reflections are needed to prove equation (12). Hence, the total number of operations required on this step is LxMx(M+1).

It should be taken into account that the number of operations of table 1 for the optimizing and training algorithm is an upper limit of the actual number of operations. It is due to the optimum number of hidden nodes, at any step of the process, is always less or equal than L. Moreover, the number of hidden units used on each process iteration depends on the order of presentation of training patterns, so establishing a general quantitative comparison between both algorithms is a difficult task. This evaluation must be done for an specific network application

4 Simulation Results

To test the effectiveness of our algorithm, the chaotic time-series generated by the Mackey-Glass equation have been studied using three-layer feedforward networks.

A system is said to be chaotic if the evolutionary trajectory of the system is generated by a deterministic mechanism, but it is very sensitive to the system's initial condition [8]. Since under certain conditions a chaotic system behaves randomly, the identification of such system is difficult. Under those conditions, a model capable of identifying the underlying deterministic mechanism can greatly improve system performance, predictability and control.

The discrete time representation of the Mackey-Glass equation is given by

$$x(k+1) - x(k) = \alpha x(k-\tau)/(1+ x^{\gamma}(k-\tau)) - \beta x(k) \qquad (17)$$

Consider the series generated with α=0.2, β=0.1, γ=10 and τ=17. This combination generates a quasiperiodic time series, where a quasiperiodic process is a linear combination of several periodic processes.

The objective is to model the Mackey-Glass series to produce ahead predictions. The Mackey-Glass series $\{x(k)\}$ can be expressed as

$$x(k+p) = f(x(k), x(k-\tau), x(k-2\tau), ..., x(k-(N-1)\tau)) \qquad (18)$$

where p is the prediction time, which is chosen according to the need for long-term or short-term prediction, and N is generally between four and eight [8][9]. We have chosen N=6, so a six-input neural network is considered where $x(k)$, $x(k-\tau)$, ...,$x(k-5\tau)$ are used as the inputs and $x(k+p)$ is used as the output.

Simulation results have been obtained from several neural networks with different number of hidden units using 300 data sets for training. For each of those neural networks both, the back-propagation algorithm and the optimizing and training algorithm, have been applied. When the proposed method is applied, a reduced 6x3x1 network is obtained.

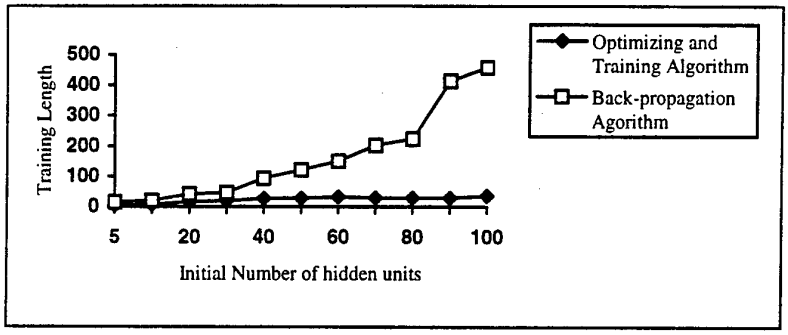


Fig. 1. Training length for several networks with different number of hidden nodes

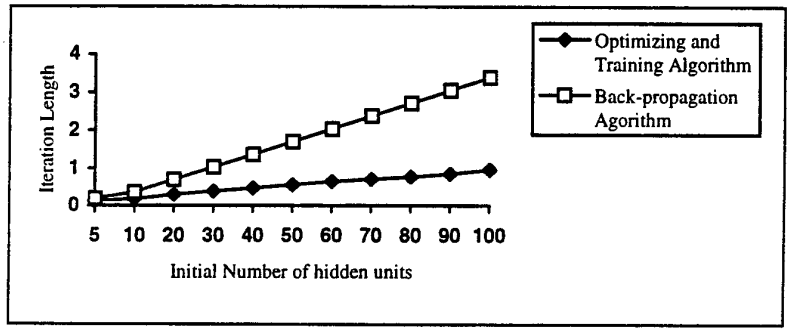


Fig. 2. Iteration length for MG series using networks with different number of hidden units.

Figure 1 and 2 show the training and iteration lengths using different number of hidden nodes in the proposed and the original back-propagation algorithms. As it can be seen, although training time increases for large networks in both algorithms, the optimizing and training method provides better results than the back-propagation algorithm, even when the optimum number of hidden nodes is near to the initial number of hidden nodes.

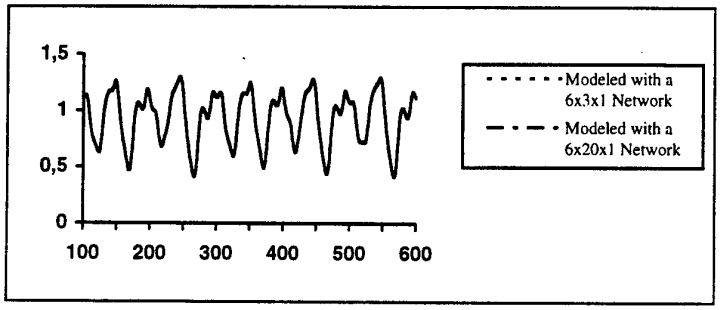


Fig. 3. Mackey-Glass series modeled using 6x20x1 and 6x3x1 networks.

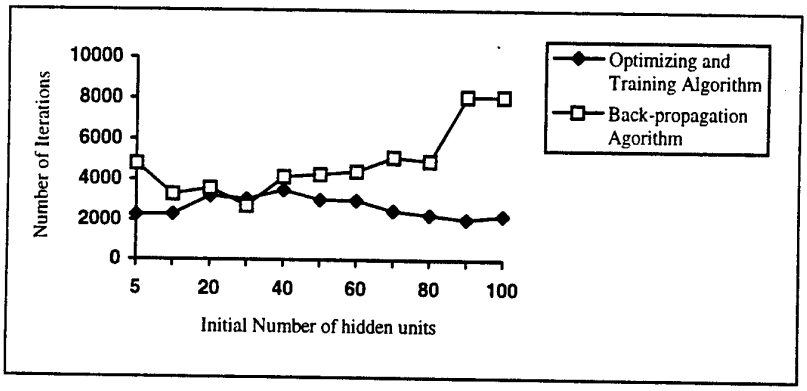


Fig. 4. Number of iterations required for the back-propagation algorithm and the proposed method.

The representation, in figure 3, of the Mackey-Glass series modeled using a 6x20x1 network, trained with the original back-propagation algorithm, and a 6x3x1 network, obtained by means of the proposed method, shows that the performance of both networks is equally good.

Figure 4 shows the number of iterations required to train the networks. From the results obtained we can observed that small networks need less number of iterations than large networks to reach a low mean-squared error (MSE). However the learning speed depends on many other factors such as weights initialization and learning rate parameter (α).

5 Conclusions

In this paper a method for training and reducing the size of feedforward neural networks has been presented. The key idea of this approach consists of iteratively computing the optimum hidden nodes and then updating only the weights connected to those nodes. Using this method the retraining process of the reduce-size network is avoided.

We apply Householder reflections to compute the optimum network size on each process iteration. These orthogonal transformations lead to a decorrelation of the network information using few operations, which accelerate the training process.

From experimental results, an improvement on the network training length can be observed with regards to the original back-propagation algorithm and hence, in relation to existing pruning approaches.

The proposed algorithm can be expressed in basic matrix operations and so its implementation can be easily achieved using processor arrays, systolic or wavefront arrays.

References

1. S. E. Fahlman and C.Lebiere: *The cascade-correlation learning architecture.* Advances in Neural Information Processing, Ed. San Mateo, 1990, pp. 524-432.
2. S. I. Gallant: *Optimal linear discriminants.* Proc. 8th Int. Conf. Pattern Recognition, Paris, France, 1986, pp. 849-852.
3. T. Ash: *Dynamic node creation in backpropagation networks.* Connection Sci., vol. 1, no. 4, 1989, pp. 365-375.
4. M. Mézard and J. P. Nadal: *Learning in feedforward layered networks: The Tiling algorithm.* J. Phys. A, vol. 22, 1989, pp. 2191-2204.
5. P. Kanjilal and N. Banerjee: *On the application of orthogonal transformation for the design and analysis of feedforward networks.* IEEE Transactions on Neural Networks, vol. 5, no. 5, 1995, pp. 1061-1070.
6. G. Castellano, A. M. Fanelli and M. Pelillo: *An iterative pruning algorithm for feedforward neural networks.* IEEE Transactions on Neural Networks, vol. 8, no. 3, 1997, pp. 519-531.
7. G. H. Golub and C. F. Van Loan: *Matrix computations.* Baltimore, MD: John Hopkins Univ. Press, 1989.
8. M. F. Tenorio: *Self-organizing network for optimum supervised learning.* IEEE Transactions on Neural Networks, vol. 1, no. 1, 1990, pp. 100-110
9. A. Lapedes and R. Farder: *Nonlinear signal processing using neural networks.* Los Alamos National Lab. Tech. Rep. LA-UR-2662, 1987.

Preprocessor Based Implementation of the Versatile Advection Code for Workstations, Vector and Parallel Computers

Gábor Tóth

Department of Atomic Physics, Eötvös University,
Puskin u. 5-7, Budapest 1088, Hungary,
gtoth@hercules.elte.hu, http://www.fys.ruu.nl/~toth

Abstract. The Versatile Advection Code is a single scientific software package designed and implemented to solve various hydrodynamic and magnetohydrodynamic problems typical of astrophysical research. It runs on workstations, and on vector and parallel supercomputers as well. The versatility for applications is ensured by the Loop Annotation Syntax preprocessor and the modular design of the software, while portability to different hardware platforms is achieved by the preprocessors that can translate the code from Fortran 90 both to High Performance Fortran and Fortran 77. Performance results are presented for several platforms.

1 Introduction

The Versatile Advection Code (VAC) [1, 2] has been developed since 1994 as a general purpose tool for hydrodynamic and magnetohydrodynamic astrophysical applications. VAC uses various shock capturing numerical methods [3], explicit, semi-implicit, or fully implicit time stepping [4, 5] on 1, 2, or 3 dimensional finite volume grids. The software package is complete with 120 pages of manual written in hypertext, a user interface based on web browsers, and visualization macros for the most popular visualization softwares. The ever growing number of users and applications proves that the concept of a single well designed general purpose scientific software package is a good alternative to the typical specialized scientific codes.

The most original software solution in VAC is the Loop Annotation Syntax (LASY) [6], which was developed to provide a compact notation for expressions occuring in a multidimensional hydrodynamic code independent of the number of represented spatial dimensions. The other important feature is the modular design, which allows VAC to solve different equations with different methods, and lets the user add extra terms in the equations, define special initial and boundary conditions, or specify non-default input/output data format by writing a few well specified subroutines.

VAC is designed from the beginning to run on workstations, where most scientists do their simulations, and on vector and parallel supercomputers, required for big 2D and 3D simulations, as well. The source code, after it is translated

from the LASY notation, uses Fortran 90 (F90) array syntax and High Performance Fortran (HPF) style FORALL statements for all the expressions that operate on the whole computational grid. Thus it is easy to add HPF compiler directives in an automated fashion and run the code on a parallel machine under HPF. It is also trivial to translate the FORALL statements back to ordinary DO loops for a Fortran 90 compiler on a non-parallel machine.

Although Fortran 90 is becoming available on most scientific computing facilities, it is still necessary to be able to translate the source code to Fortran 77 (F77). A simple translator program is implemented to carry out this task for the limited number of language constructs that are used from the rich Fortran 90 language. Not using all the features of F90 is a restriction for the developer, but it is beneficial for the users, who are more familiar with the simpler F77 language, and for the compilers, which usually do a better job on simpler program constructs.

2 Preprocessors

The use of the preprocessors can be best demonstrated on a small piece of code. The purpose of the `gradient` subroutine is simple: calculate the gradient `gradq` of the quantity `q` in direction `idir` within a rectangle, defined by `ix^L` indices. From the actual, more general, subroutine used in VAC, I extracted the part which is valid for Cartesian grids and uses central differences. The subroutine is shown in Figure 1.

```
subroutine gradient(q,ix^L,idir,gradq)

include 'vacdef.f90'
double precision:: q(ixG^T),gradq(ixG^T)
integer:: ix^L,idir,jx^L,hx^L

!SHIFT
jx^L=ix^L+kr(idir,^D);
!SHIFT MORE
hx^L=ix^L-kr(idir,^D);
!SHIFT BEGIN
gradq(ix^S)=0.5D0*(q(jx^S)-q(hx^S))/dx(ix^S,idir)
!SHIFT END

return
end
```

Fig. 1. Example source code with LASY.

The included `vacdef.f90` file declares the global parameters and variables. The array dimensions `ixG^T`, the grid spacing `dx(ixG^T,ndim)`, and the Kronecker delta array `kr(3,3)`, which is used to shift indices in a certain direction,

are all declared and initialized before this subroutine is called. The meaning of the LASY patterns starting with the special character ^ is briefly the following: ^D stands for dimensions, ^L for limits, ^S for array segments, and ^T for the total size of arrays. The VAC Preprocessor (VACPP) substitutes the patterns with substitute strings, whose number depends on the number of spatial dimensions, which is a parameter for VACPP. The preprocessor not only replaces the patterns with their substitute strings, but it also repeats the source code attached to the pattern, and the repetitions are separated appropriately. The detailed rules of LASY are described in [6], here I simply show the code translated to 2 dimensions in Figure 2.

```
subroutine gradient(q,ixmin1,ixmin2,ixmax1,ixmax2,idir,gradq)

include 'vacdef.f90'
integer:: ixmin1,ixmin2,ixmax1,ixmax2,idir,&
   jxmin1,jxmin2,jxmax1,jxmax2,hxmin1,hxmin2,hxmax1,hxmax2
double precision:: q(ixGlo1:ixGhi1,ixGlo2:ixGhi2),&
   gradq(ixGlo1:ixGhi1,ixGlo2:ixGhi2)

!SHIFT
jxmin1=ixmin1+kr(idir,1);jxmin2=ixmin2+kr(idir,2);
jxmax1=ixmax1+kr(idir,1);jxmax2=ixmax2+kr(idir,2);
!SHIFT MORE
hxmin1=ixmin1-kr(idir,1);hxmin2=ixmin2-kr(idir,2);
hxmax1=ixmax1-kr(idir,1);hxmax2=ixmax2-kr(idir,2);
!SHIFT BEGIN
gradq(ixmin1:ixmax1,ixmin2:ixmax2)=0.5D0*&
   (q(jxmin1:jxmax1,jxmin2:jxmax2)&
   -q(hxmin1:hxmax1,hxmin2:hxmax2))&
   /dx(ixmin1:ixmax1,ixmin2:ixmax2,idir)
!SHIFT END

return
end
```

Fig. 2. Source code translated to Fortran 90 for 2 spatial dimensions.

It is quite easy to imagine what the 1 or 3 dimensional versions would look like. Clearly, the LASY notation is not only more general, but also more compact than the translated F90 source code. The VACPP preprocessor is implemented as the *vacpp.pl* Perl script.

In case the user has no F90 compiler available, the Fortran 90 source is further translated to Fortran 77 by the *f90tof77* Perl script. The translation changes the free source format to fixed one, and replaces the array syntax by do loops. The *f90tof77* script can also deal with the differences between F90

```
      subroutine gradient(q,ixmin1,ixmin2,ixmax1,ixmax2,idir,gradq)

      include 'vacdef.f'
      integer ixmin1,ixmin2,ixmax1,ixmax2,idir,
     &   jxmin1,jxmin2,jxmax1,jxmax2,hxmin1,hxmin2,hxmax1,hxmax2
      double precision q(ixGlo1:ixGhi1,ixGlo2:ixGhi2),
     &   gradq(ixGlo1:ixGhi1,ixGlo2:ixGhi2)

*SHIFT
      jxmin1=ixmin1+kr(idir,1)
      jxmin2=ixmin2+kr(idir,2)
      jxmax1=ixmax1+kr(idir,1)
      jxmax2=ixmax2+kr(idir,2)
*SHIFT MORE
      hxmin1=ixmin1-kr(idir,1)
      hxmin2=ixmin2-kr(idir,2)
      hxmax1=ixmax1-kr(idir,1)
      hxmax2=ixmax2-kr(idir,2)
*SHIFT BEGIN
      do ix_2=ixmin2,ixmax2
      do ix_1=ixmin1,ixmax1
         gradq(ix_1,ix_2)=0.5D0*
     &      (q(ix_1+(jxmin1-ixmin1),ix_2+(jxmin2-ixmin2))
     &      -q(ix_1+(hxmin1-ixmin1),ix_2+(hxmin2-ixmin2)))
     &      /dx(ix_1,ix_2,idir)
      enddo
      enddo
*SHIFT END

      return
      end
```

Fig. 3. Source code further translated to Fortran 77.

and F77 regarding the variable declaration, and it can translate some functions like `sum, product, maxval, minval, any, all`, which operate on arrays and return scalars. The `where, forall, case` constructs can also be translated. Other features of Fortran 90, like dynamic allocation, modules, array valued functions, pointers, structures, etc. are not used in VAC, and cannot be translated by *f90tof77*, which is a short and simple program. The `gradient` subroutine in 2 dimensions and in F77 is shown in Figure 3. The loop variables `ix_1, ix_2` are declared in the included file.

```
subroutine gradient(q,ixmin1,ixmin2,ixmax1,ixmax2,idir,gradq)

include 'vacdef.hpf'
integer:: ixmin1,ixmin2,ixmax1,ixmax2,idir,&
   jxmin1,jxmin2,jxmax1,jxmax2,hxmin1,hxmin2,hxmax1,hxmax2
double precision:: q(ixGlo1:ixGhi1,ixGlo2:ixGhi2),&
   gradq(ixGlo1:ixGhi1,ixGlo2:ixGhi2)
!HPF$ DISTRIBUTE q(BLOCK,*) ONTO PP
!HPF$ DISTRIBUTE gradq(BLOCK,*) ONTO PP

!SHIFT
jxmin1=ixmin1+kr(idir,1);jxmin2=ixmin2+kr(idir,2);
jxmax1=ixmax1+kr(idir,1);jxmax2=ixmax2+kr(idir,2);
!SHIFT MORE
hxmin1=ixmin1-kr(idir,1);hxmin2=ixmin2-kr(idir,2);
hxmax1=ixmax1-kr(idir,1);hxmax2=ixmax2-kr(idir,2);
!SHIFT BEGIN
IF (hxmin1==ixmin1-1.and.hxmin2==ixmin2.and.&
    jxmin1==ixmin1+1.and.jxmin2==ixmin2) THEN
   gradq(ixmin1:ixmax1,ixmin2:ixmax2)=0.5D0*&
      (q(ixmin1+1:ixmax1+1,ixmin2:ixmax2)&
      -q(ixmin1-1:ixmax1-1,ixmin2:ixmax2))&
      /dx(ixmin1:ixmax1,ixmin2:ixmax2,idir)
ELSE IF(hxmin1==ixmin1.and.hxmin2==ixmin2-1.and.&
        jxmin1==ixmin1.and.jxmin2==ixmin2+1) THEN
   gradq(ixmin1:ixmax1,ixmin2:ixmax2)=0.5D0*&
      (q(ixmin1:ixmax1,ixmin2+1:ixmax2+1)&
      -q(ixmin1:ixmax1,ixmin2-1:ixmax2-1))&
      /dx(ixmin1:ixmax1,ixmin2:ixmax2,idir)
ELSE
   stop 'SHIFT did not optimize!'
ENDIF
!SHIFT END

return
end
```

Fig. 4. Source code with HPF directives and optimized index shifts.

The *f90tohpf* script inserts the HPF directives into the Fortran 90 source code automatically. All arrays defined on the full grid are declared with the `ixGlo1:ixGhi1,...` index limits, and they can be distributed among the processors according to the parameters given to *f90tohpf*. On different parallel architectures and/or for different problem sizes, different distributions may be optimal. The automatic insertion of the directives makes it extremely simple to, e.g., change a (BLOCK,BLOCK) distribution to (BLOCK,*) or (*,BLOCK).

Unfortunately, HPF compilers are not as mature as F77 or F90 compilers. Several HPF compiler bugs were found while VAC was tested on parallel computers. Due to the simplicity of the source code, there were relatively few problems, and they could be avoided relatively easily. Even if the code compiles and runs correctly, the performance can be very poor if the HPF compiler does not recognize the simple shift operations in the **gradient** subroutine and elsewhere in the source. The general global communication is much slower than the fast specialized shifts, which are supported by the hardware and the communication libraries of most parallel computers. To help the compiler, the VAC preprocessor can replace the general shift statement marked with the `!SHIFT` comments, with shifts in specific directions placed in the appropriate branches of an `if, else if` construct. The resulting code, shown in Figure 4, is longer and more difficult to read, but it usually compiles to a faster code under HPF. The physical layout of processors `PP` is defined in the include file. When only one spatial dimension is distributed, one can use the HPF directive `!HPF$ PROCESSORS PP(NUMBER_OF_PROCESSORS())`.

The code can also be translated to Connection Machine Fortran (CMFortran) with the `f90tocmf` script. Unfortunately the CM Fortran compiler recognizes index shifts for a very limited type of syntax, thus communication is not optimal without rewriting the critical shifts by hand. In principle, one could automate this optimization, but, since CM Fortran is disappearing from the scene, there is little motivation to write the necessary Perl script.

3 Results and Conclusions

VAC is being used by approximately 25 researchers, mostly astrophysicists. Most applications are hydrodynamic and magnetohydrodynamic simulations, but VAC is also used as a test suite for different numerical methods. Most users have access to powerful workstations, thus the code has been tested and used on DEC, SUN, IBM, SGI, HP workstations, and even on Pentium PC-s under LINUX.

Due to the simplicity of the loops, which is implied by the F90 array syntax, the code vectorizes very well. On a single node of the traditional vector supercomputer Cray C90, VAC runs about 23 times faster than on a DEC Alpha/400 workstation, while the ratio is 4.2 for the J90. These measurements were done for a specific problem [7], but the speed ratios are typical for all timings tried so far.

VAC has also been tested on the IBM SP, Cray T3E, Cray T3D, and Connection Machine 5 (CM5) parallel machines, and on a cluster of workstations,

under different HPF compilers [8,7]. The scaling is close to linear up to 8 processors on the IBM SP and on the Cray T3E for a rather moderate and *fixed* problem size, which proves that good scaling is possible under HPF even for a code as complex as VAC. The single node performance is a factor of 5.2 and 1.7 improvement relative to the DEC Alpha/400 workstation for the SP and the T3E machines, respectively. On a 16-node CM5, after optimizing the array shift operations by hand, the code runs about 15 times faster than on the DEC Alpha. VAC was tested on a cluster of workstations as well. The code compiled and ran successfully, but the multiuser environment did not allow for meaningful timing.

The Versatile Advection Code proves that it is possible to write one source code for several different applications and computer platforms with the aid of simple but powerful preprocessor and translator programs. All the preprocessor programs, `vacpp.pl`, `f90tof77`, `f90tohpf`, `f90tocmf`, `forall2do`, are implemented in Perl, which is a free software, and is installed on almost all scientific computers. Actually, the preprocessing step and the final compilation can be done on different computers if necessary.

Currently we are working on the HPF compatible implementation of the implicit time stepping module. As a first step the Poisson solver using Conjugate Gradient type iterative schemes (CG and BiCGSTAB), originally implemented in F77, has been rewritten to the LASY notation and now it runs successfully on parallel machines with HPF. The next step involves rewriting and testing the preconditioner [9] for the block penta- and heptadiagonal Jacobian matrices that arise in implicit time stepping schemes.

Acknowledgement. This work was performed as part of the project on 'Parallel Computational Magneto-Fluid Dynamics', funded by the Dutch Scientific Research Foundation (NWO) Priority Program on Massively Parallel Computing. It was sponsored by the Dutch National Computing Facilities Foundation (NCF) for the use of supercomputer facilities. The author receives a postdoctoral fellowship (D 25519) from the Hungarian Science Foundation (OTKA), and is supported by the OTKA grant F 017313. Collaboration on testing the code with R. Keppens, P. Meyer, and E. van der Zalm is gratefully acknowledged. The author also thanks the Astronomical Institute at Utrecht for its hospitality.

References

1. Tóth, G.: A general code for modeling MHD flows on parallel computers: Versatile Advection Code, Astrophys. Lett. & Comm. **34** (1996) 245–250
2. Tóth G.: Versatile Advection Code, in *Proceedings of High Performance Computing and Networking Europe 1997*, Lecture Notes in Computer Science, **1225**, edited by B. Hertzberger and P. Sloot (Springer-Verlag, 1997), p. 253–262
3. Tóth, G., Odstrčil, D.: Comparison of some Flux Corrected Transport and Total Variation Diminishing Numerical Schemes for Hydrodynamic and Magnetohydrodynamic Problems. J. Comput. Phys. **128** (1996) 82–100
4. Keppens, R., Tóth, G., Botchev, M. A., van der Ploeg, A.: Implicit and Semi-Implicit Schemes in the Versatile Advection Code: algorithms, submitted for publication to the Int. J. Num. Meth. in Fluids (1997)

5. Tóth, G., Keppens, R., Botchev, M. A.: Implicit and semi-implicit schemes in the Versatile Advection Code: numerical tests, Astron. & Astroph. **332** (1998) 1159-1170
6. Tóth, G.: The LASY Preprocessor and its Application to General Multi-Dimensional Codes, J. Comput. Phys. **138** (1997) 981-990
7. Tóth G., Keppens R.: Comparison of Different Computer Platforms for Running the Versatile Advection Code, in *Proceedings of High Performance Computing and Networking Europe 1998*, Lecture Notes in Computer Science, **1401**, edited by P. Sloot, M. Bubak, and B. Hertzberger (Springer-Verlag, 1998), p. 368-376
8. Keppens, R. and Tóth, G.: Simulating Magnetized Plasma with the Versatile Advection Code, in this volume of proceedings (1998)
9. van der Ploeg, A., Keppens, R., Tóth, G.: Block Incomplete LU-preconditioners for Implicit Solution of Advection Dominated Problems, in *Proceedings of High Performance Computing and Networking Europe 1997*, Lecture Notes in Computer Science, **1225**, edited by B. Hertzberger and P. Sloot (Springer-Verlag, 1997), p. 421–430

A Parallel N-Body Integrator Using MPI

Nuno Sidónio Andrade Pereira *

Politechnical Institute of Beja, School of Technology and Management
Largo de São João, nº 16, 1º Esq. C e D, 7800 Beja, Portugal

Abstract. The study of the astrophysical N-body problem requires the use of numerical integration to solve a system of 6N first-order differential equations. The particle-particle codes (PP) using direct summation methods are a good example of algorithms where parallelization can speed up the computation in an efficient way. For this purpose, a serial version of the PP code *NNEWTON* developed by the author was parallelized using the MPI library and tested on the CRAY-T3D at the EPCC. The results of the parallel code here presented show very good efficiency and scaling, up to 128 processors and for systems up to 16384 particles.

1 Introduction

We begin by an introduction to the Astrophysical N-body problem and the mathematical models used in our work. We also present an overview of particle simulation methods, and discuss the implementation of a direct summation method: the PP algorithm. A parallel version of this algorithm as well as the performance analysis are presented. Finally, the conclusions regarding the discussion of results are offered.

2 The Astrophysical N-Body Problem

The gravitational N-body problem refers to a system of interacting bodies through their mutual gravitational attraction, confined to a delimited region of space. In the universe we can select systems of bodies according to the observation scale. For instance, we can consider the Solar System with $N = 10$ (a restricted model: Sun + 9 planets). Increasing the observation scale, we have systems like open clusters (systems of young stars with typical ages of the order of 10^8 years, and $N \sim 10^2 - 10^3$), globular clusters (systems of old stars with ages of 12-15 billion years, extremely compact and spherically symmetric with $N \sim 10^4 - 10^6$), and galaxies ($N \sim 10^{10} - 10^{12}$). On the other extreme of our scale, on a cosmological scale, we have clusters of galaxies and superclusters. If we want to consider the whole universe, the total number of galaxies in the observable part is estimated to be of the order of 10^9 (see [2], [18], and [9]).

* This work was supported by EPCC/TRACS under Grant ERB-FMGE-CT95-0051 and partly supported by PRAXIS XXI under GRANT BM/594/94.

In our work we are interested in the dynamics of systems with N up to the order of 10^4 (open clusters and small globular clusters).

2.1 The Mathematical Model

In our mathematical model of the physical system each body is considered as a mass point (hereafter refered to as particle) characterized by a mass, a position, and a velocity. We also define an inertial cartesian coordinate system, suitably chosen in three-dimensional Euclidean space, and an independent variable t, the *absolute time* of Newtonian mechanics.

The state of the system is defined by the set $\mathcal{S}_N$ of $3N$ parameters: the masses, positions, and velocities of all particles. Hence:

$$\mathcal{S}_N = \{(m_i, \mathbf{r}_i, \dot{\mathbf{r}}_i), i = 1, \ldots, N\}, \tag{1}$$

where $\mathbf{r}_i$ and $\dot{\mathbf{r}}_i$ are the position and velocity vector of particle i, respectively.

Comments. The physical state of the system can be represented as a point in a $6N$-dimensional phase-space with coordinates $(\mathbf{r}_1, \ldots, \mathbf{r}_N, \dot{\mathbf{r}}_1, \ldots, \dot{\mathbf{r}}_N)$ (see [3]). However, we will use this representation of the system which is more suitable for the discussion of the parallelization of the N-body integrator, on Sect. 3.3.

The force exerted by particle j on particle i is given by Newton's Law of Gravity:

$$\mathbf{F}_{ij} = -Gm_i m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{\| \mathbf{r}_i - \mathbf{r}_j \|^3}, \tag{2}$$

and the total force acting on particle i is

$$\mathbf{F}_i = \sum_{j=1, j \neq i}^{N} \mathbf{F}_{ij}. \tag{3}$$

The right-hand side of equation (3) represents the contribution of the other $N-1$ particles to the total force.

We can now write the equations of motion of particle i:

$$\ddot{\mathbf{r}}_i = \frac{1}{m_i} \mathbf{F}_i. \tag{4}$$

Defining $\mathbf{v}_k = \dot{\mathbf{r}}_i$ we can write the system of $6N$ first-order differential equations:

$$\dot{\mathbf{r}}_i = \mathbf{v}_i, \dot{\mathbf{v}}_i = \frac{1}{m_i} \mathbf{F}_i \tag{5}$$

with $i = 1, \ldots, N$. The evolution of the N-body system is determined by the solution of this system of differential equations with initial conditions (1).

For systems with $N = 2$, the two-body problem known as the Kepler problem, (e.g. the Earth-Moon system) the equations of motion (5) can be solved analytically. However, for the general N(>2)-body problem that is not the case

(see [3]), and we must use numerical methods to solve the system of differential equations. In Sect. 3 we will discuss the problem of numerical integration of N-body systems.

In every mathematical model of a physical system there is always the problem of the validity of the model, that is, how suitable the model is to describe the physics of the system. In our case we are representing bodies with finite and, in general, different sizes by material points: bodies endowed with mass, but no extension. The physics of the interior of the bodies is not taken into account. However, for dynamical studies this model has proven to be suitable, and has been used to study the evolution of clusters of stars, galaxies, and the development of strutures in single galaxies (see [9]).

2.2 Exponential Instabilities in N-body Systems

The initial motivation of this work was the study of the exponential instability is self-gravitating N-body systems (see [16]). In this problem we are interested in the growth of a pertubation in one or more components of the system. For a given system of N particles we consider the set

$$\mathcal{S}_N^o = \{(m_i, \mathbf{r}_i^o, \dot{\mathbf{r}}_i^o), i = 1, \ldots, N\} \tag{6}$$

of initial conditions (at time $t = t_o$), and define the set of perturbed initial conditions:

$$\Delta\mathcal{S}_N^o = \{(m_i, \Delta\mathbf{r}_i^o, \Delta\dot{\mathbf{r}}_i^o), i = 1, \ldots, N\} \tag{7}$$

where $\Delta\mathbf{r}_i^o$ and $\Delta\dot{\mathbf{r}}_i^o$ are the position and the velocity perturbation vectors for the initial conditions. To evaluate the growth of the perturbations we must solve the system of $3N$ second-order differential equations (see [6] and nsap):

$$\Delta\ddot{\mathbf{r}}_i = -\sum_{j=1, j\neq i}^{N} \mathbf{f}(\Delta\mathbf{r}_i, \Delta\mathbf{r}_j, \mathbf{r}_i, \mathbf{r}_i)\frac{m_j}{\| \mathbf{r}_i - \mathbf{r}_j \|^3} \tag{8}$$

with $i = 1, \ldots, N$, and

$$\mathbf{f}(\Delta\mathbf{r}_i, \Delta\mathbf{r}_j, \mathbf{r}_i, \mathbf{r}_j) = \Delta\mathbf{r}_i - \Delta\mathbf{r}_j - 3(\Delta\mathbf{r}_i - \Delta\mathbf{r}_j).(\mathbf{r}_i - \mathbf{r}_j)\frac{\mathbf{r}_i - \mathbf{r}_j}{\| \mathbf{r}_i - \mathbf{r}_j \|^2}. \tag{9}$$

Defining $\Delta\mathbf{v}_i = \Delta\dot{\mathbf{r}}_i$ we can rewrite (8) in the form:

$$\Delta\dot{\mathbf{r}}_i = \Delta\mathbf{v}_i, \;\; \Delta\dot{\mathbf{v}}_i = -\sum_{j=1, j\neq i}^{N} \mathbf{f}(\Delta\mathbf{r}_i, \Delta\mathbf{r}_j, \mathbf{r}_i, \mathbf{r}_i)\frac{m_j}{\| \mathbf{r}_i - \mathbf{r}_j \|^3} \tag{10}$$

with $i = 1, \ldots, N$, $\Delta\mathbf{r}_i = (\Delta x_i, \Delta y_i, \Delta z_i)$, and $\Delta\mathbf{v}_i = (\Delta\dot{x}_i, \Delta\dot{y}_i, \Delta\dot{z}_i)$. This system of $6N$ first-order differential equations, the variational equations, must be solved together with equations (5).

We now define several metrics as functions of the components of the perturbation vectors (see [6] and [16]):

$$\Delta R = \max_{i=1,\ldots,N}(|\Delta x_i| + |\Delta y_i| + |\Delta z_i|) \tag{11}$$

$$< \Delta R >= \frac{1}{N} \sum_{i=1}^{N} (|\Delta x_i| + |\Delta y_i| + |\Delta z_i|) \tag{12}$$

for the pertubations in the position vectors, and

$$\Delta V = max_{i=1,\ldots,N}(|\Delta \dot{x}_i| + |\Delta \dot{y}_i| + |\Delta \dot{z}_i|) \tag{13}$$

$$< \Delta V >= \frac{1}{N} \sum_{i=1}^{N} (|\Delta \dot{x}_i| + |\Delta \dot{y}_i| + |\Delta \dot{z}_i|) \tag{14}$$

for the pertubations in the velocity vectors. Each metric is evaluated for each time step of the numerical integration of equations (5) and (10).

The analysis of the quantities given by equations (11), (12), (13), and (14) is very important to understand some aspects of the dynamical behavior of N-body systems (see [8], [10], [11], and [13]). In particular, we are interested in the relation between collisions and the growth of perturbations. The collisions between bodies are an important mechanism in the evolution of systems like open clusters and globular clusters (see [2] and [9]).

3 Numerical Simulation of N-Body Systems

In this section, we will briefly discuss the use of particle methods to solve the N-body problem with special attention to the direct summation method: the PP method (see [9], for an excellent and detailed presentation of these methods). We present a serial version of the PP method and discuss a parallel version of that method.

3.1 Overview of Particle Simulation Methods

Particle methods is the designation of a class of simulation methods in which the physical phenomena are represented by particles with certain attributes (such as mass, position, and velocity), interacting according to some physical law that determines the evolution of the system. In most cases we can establish a direct relation between the computational particles and the physical particles. In our work each computational particle is the numerical representation of one physical particle. However, in simulations of physical systems with large N, such as galaxies of 10^{11} to 10^{12} stars, each computational particle is a superparticle with the mass of approximately 10^6 stars.

We will now discuss the three principal types of particle simulation methods: a direct summation method, a particle-in-cell (PIC) method, and a hybrid method.

The Particle-Particle Method (PP). This is a direct summation method: the total force on the i^{th} particle is the sum of the interactions with each other particles of the system. To determined the evolution of a N-body system we consider the interaction of every pair of particles, that is, $N(N-1)$ pairs (i,j), with $i,j = 1,\ldots,N \wedge i \neq j$. The numerical effort (number of floating-point operations) is observed to be proportional to N^2.

The Particle-Mesh Method (PM). This is a particle-in-cell method: the physical space is discretized by a regular mesh where a density function is defined according to the attributes of the particles (e.g. mass density for a self-gravitating N-body system). Solving a Poisson equation on the mesh, the forces at particle positions are then determined by interpolation on the array of mesh-defined values. The numerical effort is observed to be proportional to N. The gain in speed is obtained at the cost of loss of spatial resolution. This is particularly important for the simulation of N-body systems if we are interested in exact orbits.

The Particle-Particle-Particle-Mesh Method (P^3M). This is a hybrid method: the interaction between one particle and the rest of the system is determined considering a short-range contribution (evaluated by the PP method) and a long-range contribution (evaluated by the PM method). The numerical effort is observed to be also proportional to N, as in the PM method. The advantage of this method over the PM method is that it can represent close encounters as accurately as the PP method. On the other hand the P^3M method calculates long-range forces as fast as the PM method.

Comments. We base the choice of method according to the physics of the system under investigation. For our work we use the PP method: we are interested in simulating clusters of stars where collisions are important and, therefore, spatial resolution is important. On the other hand, for the values of N used in some of our simulations ($N \sim 16 - 1024$) the use of a direct summation method has the advantage of providing forces that are as accurate as the arithmetic precision of the computer.

3.2 The PP Serial Algorithm

In our previous work (see [16]) we have implemented the PP method using FORTRAN 77. Several programs were writen (the *NNEWTON* codes) but only two versions are considered here: a PP integrator of the equations of motion, and a PP integrator of the equations of motion + variational equations. These two versions use a softened point-mass potential, that is, the force of interaction between two particles i and j is defined as (see [1], [2], and [9]):

$$\mathbf{F}_{ij} = -Gm_i m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{\| (\mathbf{r}_i - \mathbf{r}_j)^2 + \epsilon^2 \|^{3/2}}. \tag{15}$$

The parameter ϵ is often called the softening parameter and is introduced to avoid numerical problems during the integration of close encounters between particles: as the distance between particles becomes smaller the force changes as $1/\| \mathbf{r}_i - \mathbf{r}_j \|^2$ in equation (2) and extremely small time steps must be used in order to control the local error of truncation of the numerical integrator. The softening parameter will prevent the force to go to infinity for zero distance causing overflow errors.

3.3 The PP Parallel Algorithm (P-PP)

The PP method has been used to implement parallel versions of N-body integrators by several authors (see [14], for instance). Having this in mind, our first goal was to write a simple algorithm with good load-balance: each processor should perform the same amount of computations. On the other hand, the algorithm should be able to take advantage of an increased number of processors (scalability).

In our algorithm the global task is the integration of the system of equations (5), for N particles, and the sub-tasks are the integration of sub-sets $\mathcal{S}_{N_k}$ of N_k particles, with $k = 0, ..., p$, where $P = p + 1$ is the number of available processors. The parallel algorithm implements a *single program multiple data* (SPMD) programming model: each sub-task is executed by the same program operating on different data (the sub-sets $\mathcal{S}_{N_k}$ of particles).

The diagram in figure 1 shows the structure of the parallel algorithm and the main communication operations. The data are initially read from a file by one processor and a broadcast communication operation is performed to share the initial configuration of the system between every available processor. To each processor (k) is then assigned the integration of a sub-set $\mathcal{S}_{N_k}$ of particles. The global time step is also determined by a global communication operation, and at the end of each time iteration the new configuration of the particles (in each sub-set $\mathcal{S}_{N_k}$) is shared between all processors.

The load-balance problem is completely avoided in this algorithm since each processor is responsible for the same number of particles. The defined sub-sets of particles are such that

$$\sum_{k=0}^{p} \#(\mathcal{S}_{N_k}) = \sum_{k=0}^{p} N_k = N \tag{16}$$

and

$$N_i = N_j, \;\; i, j = 0, ..., p.$$

4 Implementation of the Parallel Algorithm

4.1 The Message Passing Model

The implementation of the P-PP algorithm was done in the framework of the message passing model (see [5] and [7]). In this model we consider a set of processes (each identified with a unique name) that have only local memory but are able to communicate with other processes by sending and receiving messages.

Most of the message passing systems implement a SPMD programming model: each process executes the same program but operates on different data. However, the message passing model does not preclude the dynamic creation of processes, the execution of multiple processes per processor, or the execution of different programs by different processes.

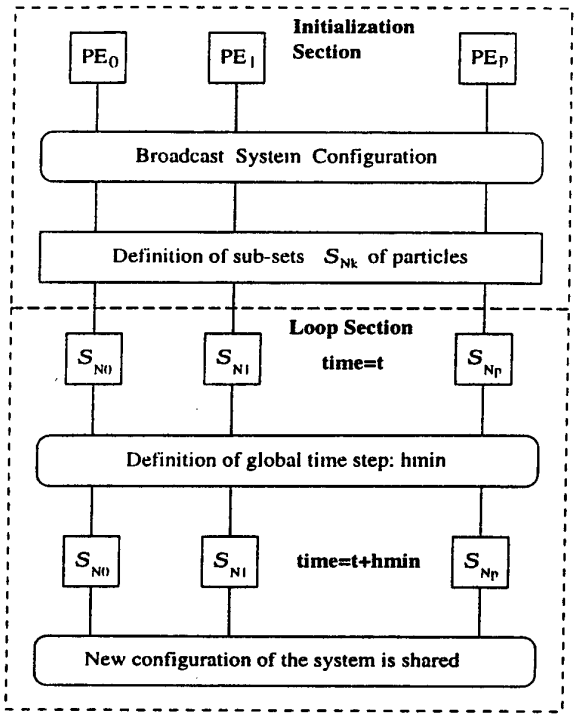


Fig. 1. The diagram shows the structure of the parallel algorithm and the main communication operations: broadcasting the initial configuration of the system to all processors, determination of the global time step and the global communication between processors to share the new configuration of the system after one time step. Each processor $PE_k, (k = 0, ...p)$ is responsible for the integration of its sub-set S_{N_k} of particles.

For our work, this model has one important advantage: it fits well on separate processors connected by a communication network, thus allowing the use of a supercomputer as well as a network of workstations.

4.2 The MPI Library

To implemented the parallel algorithm the Message Passing Interface (MPI) library (see [5]-[12]) was chosen for the following reasons:

- source-code portability and efficient implementations across a range of architectures are available,
- functionality and support for heterogeneous parallel architectures.

Using the MPI library was possible to develop a parallel code that runs on a parallel supercomputer like the Cray-T3D and on a cluster of workstations. On

the other hand, from the programming point of view is very simple to implement a message passing algorithm using the library functions.

4.3 Analysis of the MPI Implementation

The MPI implementation of the P-PP algorithm was possible with the use of a small number of library functions. Two versions of the codes writen in FORTRAN 77, the *NNEWTON* codes, (see [16]) were parallelized using the following functions (see [7]):

Initialization

1. `MPI_INIT`: Initializes the MPI execution environment.
2. `MPI_COM_SIZE`: Determines the number of processors.
3. `MPI_COMM_RANK`: Determines the identifier of a processor.

Data Structures: Special data structures were defined containing the system configuration.

4. `MPI_TYPE_EXTENT`: Returns the size of a datatype.
5. `MPI_TYPE_STRUCT`: Creates a structure datatype.
6. `MPI_TYPE_COMMIT`: Commits a new datatype to the system.
7. `MPI_TYPE_FREE`: Frees a no longer needed datatype.

Communication: One of the processors broadcasts the system configuration to all other processores.

8. `MPI_BCAST`: Broadcasts a message from processor with rank "root" to all other processors of the group.

Global Operations: Used to compute the global time step, and to share the system configuration between processors after one iteration.

9. `MPI_ALLREDUCE`: Combines values from all processors and distribute the result back to all processors.
10. `MPI_ALL_GATHERV`: Gathers data from all processors and deliver it to all.

Finalization

11. `MPI_FINALIZE`: Terminates MPI execution environment.

5 Performance Analysis

To analyse the performance of a parallel program several metrics can be considered depending on what characteristic we want to evaluate. In this work we are interested in studying the scalability of the P-PP algorithm, that is, how effectively it can use an increased number of processors. The metrics we used to evaluate the performance are functions of the program execution time (T), the problem size (N, number of particles), and processor count (P). In this section we will define the metrics (as in [5] and [14]) and discuss their application.

5.1 Metrics of Performance

We will consider three metrics for performance evaluation: execution time, relative efficiency, and relative speedup.

Definition 1. The *execution time* of a parallel program is the time that elapses from when the first processor starts executing on the program to when the last completes execution.

The execution time is actually the sum over the number of processors of three distinct times: computation time (during which the processor is performing calculations), communication time (time spent sending and receiving messages), and idle time (the processor is idle due to lack of computation or lack of data).

In this study the program is allowed to run for 10 iterations and the execution time is mesured by the time of one iteration ($T_{one} = T_{ten}/10$).

Definition 2. The *relative efficiency* (E_r) is the ratio between time T_1 of execution on one processor and time T_P of execution on P processors,

$$E_r = \frac{T_1}{PT_P}. \tag{17}$$

The relative efficiency represents the fraction of time that processors spend doing usefull work. The time each processor spends communicating with other processors or just waiting for data or tasks (idle time) will make efficiency always less than 100% (this may not be true is some cases where we have a superlinear regime due to cache effects but we will not discuss it in this work).

Definition 3. The *relative speedup* (S_r) is defined as the ratio between time T_1 of execution on one processor and time T_P of execution on P processors,

$$S_r = \frac{T_1}{T_P}. \tag{18}$$

The relative speedup is the factor by which execution time is reduced on P processors. Ideally, a parallel program running on P processors would be P times faster than on one processor and we would get $S_r = P$. However, communication

time and idle time on each processor will make S_r always smaler than P (except on the superlinear regime).

These quantities are very useful to analyse the scalability of a parallel program however, efficiency an speedup as defined above do not constitute an absolute figure of merit since the time of execution on a single processor is used as the baseline.

5.2 Performance Results of the PNNEWTON Code

For the performance analysis of the algorithm we mesured the time of one iteration for a range of values of two parameters: problem size, and number of processors. The relative efficiency and relative speedup were then evaluated using equations (17) and (18).

The objectives of this analysis are two-fold. First, we want to investigate how the metrics vary with increasing number of processors for a fixed problem size. Second, we want to investigate the behavior of the algorithm for different problem sizes within the range of interest for our N-body simulations. For that purpose the parallel code (*PNNEWTON*) was tested on the Cray-T3D system at the Edinburgh Parallel Computer Centre (EPCC). The system consists of 512 DEC Alpha 21064 processors arranged on a tridimensional torus and running at 150 MHz. The peak performance of the T3D array itself is 76.8 Gflop/s (see [4]).

The next figures show the results of the tests for systems with $N = 2^6, \ldots, 2^{14}$. The code was integrating equations (5). Similar tests were performed for another version of the *PNNEWTON* code which integrates equations (5) and (10), and identical results were obtained.

6 Conclusions

The purpose of this work was the development of a parallel code suitable to study N-body systems with $N \sim 10 - 10^4$. The required features of the program were portability, scalability and efficiency. The tests performed on both versions (*PNNEWTON* 1.0 and 2.0) showed an almost linear speedup and a relative efficiency between 60% and 98%. The worst cases ($E_r \approx 60\%$ and $E_r \approx 65\%$) correspond to a system with 64 particles running on 64 processors, and to a system with 128 particles running on 128 processors. With those configurations the communication costs are comparable to the computational costs and the efficiency drops.

Using a message passing model and the MPI library for the parallelization of the PP algorithm is possible to write a portable code with high efficiency and good scalability. Our parallel algorithm appears to be appropriate to develop parallel versions of the PP method.

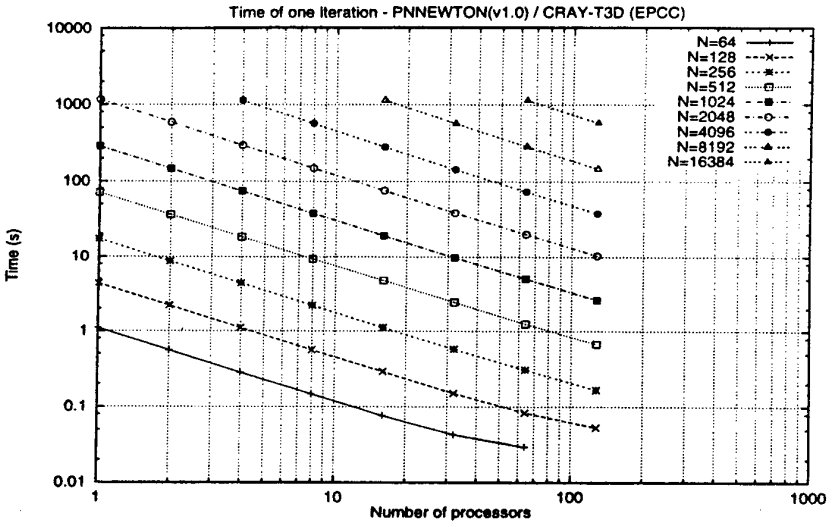


Fig. 2. For each value of N=2^k, ($k = 6, ..., 14$) the system is allowed to evolve during ten time steps. The computation was performed on a different number of processors. The variation of the time of one iteration with the number of processors for the tested systems shows a good scalling.

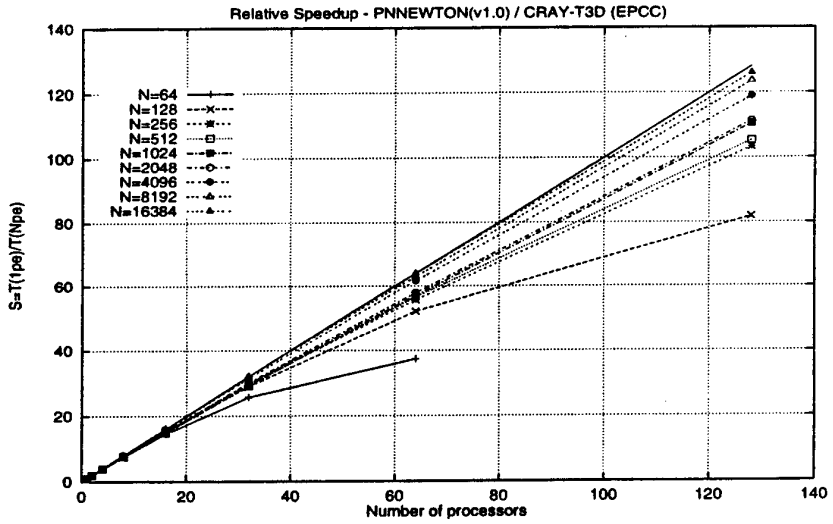


Fig. 3. The program is showing a good scalability for the tested configurations. The speed up is almost linear.

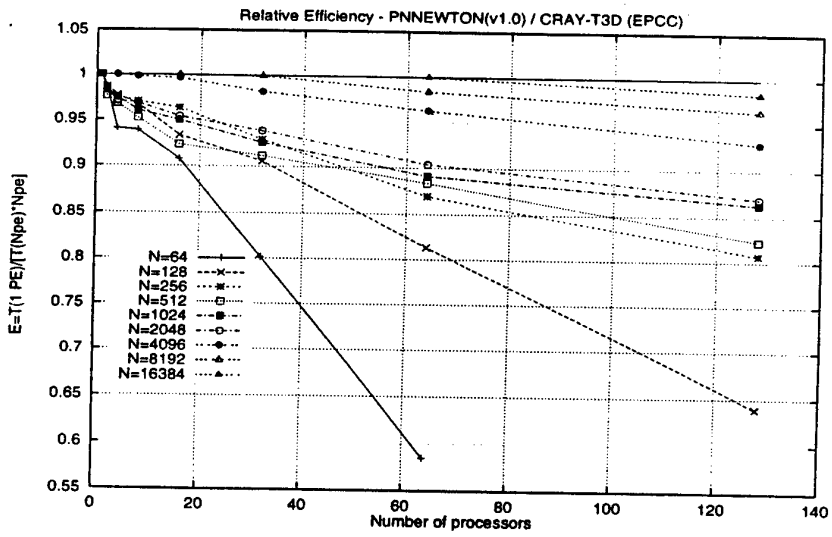


Fig. 4. The program shows high efficiency for most of the configurations tested. The lowest efficiencies correspond to cases where the cost of communications is relevant (the number of particles is the same as the number of processors).

References

1. Aarseth, S. J.: Galactic Dynamics and N-Body Simulations, Lecture Notes in Physics, Springer-Verlag, 1993.
2. Binney, J., Tremaine, S.: Galactic Dynamics, Princeton Series in Astrophysics, 1987.
3. Boccaletti, D., Pucacco, G.: Theory of Orbits, 1: Integrable Systems and Non-perturbative Methods. A&A Library, Springer-Verlag, 1996.
4. Booth, S., Fisher, J., MacDonald, N., Maccallum, P., Malard, J., Ewing, A., Minty, E., Simpson, A., Paton, S., Breuer, S.: Introduction to the Cray T3D, Edinburgh Parallel Computer Centre, The University of Edinburgh, 1997.
5. Foster, Ian.: Designing and Building Parallel Programs, Addison-Wesley, 1995.
6. Goodman, J., Heggie D. C., & Hut P.: The Astrophysical Journal, **515**:715-733, 1993.
7. Gropp, W., Lusk E., Skjellum, A.: USING MPI Portable Parallel Programming with the Message-Passing Interface, The MIT Press London, England, 1996.
8. Heggie, D. C.: Chaos in the N-Body Problem of Stellar Dynamics. Predictability, Stability, and Chaos in N-Body Dynamical Systems, Plenum Press, 1991.
9. Hockney, R. W., & Eastwood, J. W.:Computer Simulation Using Particles, Institute of Physics Publishing, Bristol and Philadelphia, 1992.
10. Kandrup, E. H., Smith, H. JR.: The Astrophysical Journal, **347**:255-265, 1991 June 10.
11. Kandrup, E. H., Smith, H. JR.: The Astrophysical Journal, **386**:635-645, 1992 February 20.

12. MacDonald, N., Minty, E., Malard, J., Harding, T., Brown, S., Antonioletti, M.: MPI Programming on the Cray T3D, Edinburgh Parallel Computer Centre, The University of Edinburgh, 1997.
13. Miller, R. H.: The Astrophysical Journal, 140,250, 1964.
14. Velde, Eric F. Van de.: Concurrent Scientific Computing, Springer-Verlag, 1994.
15. MPI: A Message-Passing Interface Standard, Message Passing Interface Forum, June 12, 1995.
16. Pereira, N. S. A.: Master Thesis, University of Lisbon, 1998.
17. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference, The MIT Press London, England, 1996.
18. Zeilik, M., Gregory, S. A., Smith, E. v. P.: Introductory Astronomy and Astrophysics, Saunders, 1992.

Efficient Molecular Dynamics on a Network of Personal Computers

G. Ciaccio[1], V. Di Martino[2]

[1] DISI, Università di Genova
via Dodecaneso 35, 16146 Genova, Italy
E-mail: ciaccio@disi.unige.it

[2] CASPUR, c/o Università di Roma "La Sapienza"
P.le A.Moro 5, 00185 Roma, Italy
E-mail: vincenzo@caspur.it

Abstract. The Genoa Active Message Machine (GAMMA) is a high-performance Active Messages-like communication layer implemented at kernel level as an extension of the Linux Operating System, and made available to user applications through a programming library. On low-cost clusters of Personal Computers (PCs) connected by Fast Ethernet, GAMMA achieves much better communication performance compared to public domain implementations of MPI and PVM.
We have considered an existing PVM Molecular Dynamics (MD) parallel application, designed to be portable across various MPP as well as NOW platforms. The goal of our work is to show how much migrating such a complex application from PVM to GAMMA is convenient in terms of absolute performance improvement as well as price/performance ratio in the perspective of running MD on a low-cost cluster of PCs. The "migration" approach is then compared to other two alternatives, namely: running the PVM version of MD "as is" on a cluster of PCs and trying tuning the PVM version of MD to match the underlying cluster architecture. It is shown that neither of such two alternatives lead to satisfactory performance.

Keywords: Fast Ethernet; Molecular Dynamics; Network of workstations; Parallel processing; Personal computers.

1 Introduction

Molecular Dynamics (MD) is one of the most frequent parallel applications in the scientific community. MD typically exhibits fairly good speed-up figures on a wide range of parallel computers with good intrinsic load balancing. This offers the opportunity to investigate the behaviour of large size samples of material by numerical simulation.

Network Of Workstations (NOWs) have emerged as the first cost-effective parallel architecture. Cluster of high-end Personal Computers (PCs) are emerging

as an even better solution, with unbeated price/performance ratio and potentially good absolute performance levels.

A serious obstacle to running MD on a cluster of PCs is the high communication latency exhibited by standard parallel programming environments like PVM [6] and MPI [7] running atop industry-standard communication protocols like TCP and UDP. Recently several teams have been engaged in producing efficient solutions using faster networks and optimized communication software to keep latency as low as possible. Many of such attempts gave rise to non-standard programming interfaces for high-performance communication. Porting a non-trivial parallel application on a non-standard communication layer may be an expensive task. However a better price/performance ratio and a satisfactory absolute performance level on a cluster of PCs may justify the porting effort.

In this paper we discuss three experiences of porting an existing MD parallel application on a low-cost cluster of PCs. The original MD code is a FORTRAN program with calls to PVM communication routines. The low-cost cluster is a pool of sixteen Pentium 133 MHz PCs, each equipped with 32 MByte of RAM and 256 KByte of second-level cache, networked by a shared 100base-TX Ethernet LAN. Each PC runs Linux, a POSIX-compliant Unix operating system.

The first experience [3] consists of migrating MD from PVM to the the Genoa Active Message MAchine (GAMMA) [1, 2], an efficient communication system based on Active Messages [8] and designed for best efficiency on 100base-T clusters of PCs. Porting MD to GAMMA required replacing PVM calls with calls to communication routines from the GAMMA library, as well as changing some communication patterns in order to achieve better exploitation of the capabilities of the underlying network hardware fully exposed by GAMMA. Therefore the corresponding porting effort was not negligible. The obtained MD application shall be called MD-GAMMA hereafter.

The second porting experience (also described in [3]) consists of running the original PVM version of MD "as is" on our cluster. This corresponds to a zero porting effort.

The third porting experience consists of trying tuning the communication patterns of the original PVM version of MD in order to increase the match with the network architecture of our cluster. This implies a very limited porting effort. The obtained application shall be called MD-TOKEN hereafter, as a circulating token has been added to reduce network contention.

2 The Genoa Active Message MAchine (GAMMA)

The Genoa Active Message MAchine (GAMMA) [1, 2] is an efficient messaging system based on Active Messages [8]. GAMMA is mainly implemented as a custom network device driver plus a small number of additional system calls extending the Linux kernel. Currently only the 3COM 3c595 and 3c905 Fast Ethernet adapters are supported. The GAMMA programming interface is a small yet complete set of communication functions supporting SPMD as well as MIMD

programming styles, and made available to user applications through a programming library.

The efficiency of GAMMA is mainly due to three features, namely:

- A "zero copy" communication protocol, that is no temporary buffers for messages along the whole communication path, thanks to the adoption of the Active Messages communication paradigm. This enables low-latency communication.
- A pipelined communication path, that is the various stages of the communication path work in parallel for best communication throughput. Every messaging system works in a pipelined way when delivering large messages fragmented into smaller units, but GAMMA allows a pipelined path yet with small, unfragmented messages. This allows best throughput for small as well as large messages.
- Broadcast primitives which directly expose the Ethernet hardware broadcast features to the applications. This allows efficient broadcast communication.

With GAMMA, any process of a given parallel application owns, and may activate and use thereof, 255 *communication ports* through which it can send and receive messages. Useful communication ports are numbered in the range from zero to 254. Port number 255 is currently reserved to the implementation of the barrier synchronization. Prior to using any of its own ports, the process may bind it to:

- A port of a destination process, for messages that will be sent throughout the port.
- A destination buffer in user space for storing incoming messages.
- A program-defined function acting as *receiver handler* for the port.
  A GAMMA receiver handler is an application-defined function which will be run at each message arrival. Such function will "consume" the message itself and possibly prepare a fresh final destination for the next incoming message. For instance, in order to avoid that a subsequent incoming message overlaps the previous one in the same user-space destination buffer, the receiver handler may re-bind the port to a fresh destination for the next incoming message.
- A program-defined function acting as *error handler* for the port. A GAMMA error handler is like a receiver handler, but it is issued in case of communication errors rather than upon successful message receptions. The purpose of error handlers is to help building application-level error recovery policies.

After a port is bound, its number fully defines the destination of messages sent through the port, as well as the user-space final destination of messages incoming through the port and the actions performed by the process in order to consume them.

With GAMMA the programmer is forced to bind a port for input before receiving messages from that port. This implies that the kernel is notified the address of the destination user-space buffer in advance w.r.t. the message arrival.

Therefore the activity of storing incoming messages into their final destinations can be performed directly by the GAMMA device driver rather than by the user-defined receiver handlers, and does not require any temporary kernel buffer.

2.1 Synchronous receive in GAMMA

With Active Messages there is a "send" but no "receive" operation. Instead the receiver handlers act as independent threads of the application triggered by message arrivals to perform the receive activities. Additional programming effort must be spent to ensure that receiver threads correctly cooperate with the main process thread. A very frequent problem is when the main thread needs to synchronize with a message arrival before continuing computation (e.g. when the process needs to receive data before processing them). A general solution is to use application-defined synchronization flags as follows:

1. A flag F of the application is initially reset.
2. In order to wait for one incoming message from a port P, the receiver process starts busy-waiting in a loop until F is set.
3. The receiver handler bound to port P sets F upon message arrival.

GAMMA offers a more flexible and reliable solution in the form of two semaphore-oriented library functions, namely `gamma_wait()` and `gamma_signal()`. Such functions give safe access to per-port semaphores embedded into the GAMMA library. The example above becomes as follows:

1. In order to wait for one incoming message from port P, the receiver process issues `gamma_wait(P,1)`
2. The receiver handler bound to port P issues `gamma_signal(P)` upon message arrival.

2.2 Communication performance

On our low-cost cluster of PCs, GAMMA achieves one-way "ping-pong" user-to-user message latency as low as 13 μs, with asymptotic bandwidth as high as 12.2 MByte/s (98% of the maximum 100base-T Ethernet throughput). Half the asymptotic bandwidth is achieved with messages as short as 200 byte. Such performance numbers are measured at application level, that is they represent the communication performance effectively delivered to user applications.

In terms of latency GAMMA rivals many much more expensive massively parallel platforms. Obviously GAMMA cannot compete with such platforms in terms of bandwidth as well as scalability. On the other hand no massively parallel computer can compete with GAMMA in terms of price/performance ratio.

3 The Molecular Dynamics application

Our MD application [4, 5] is a typical Molecular Dynamics code used for simulating the behaviour of polarizable fluids. The current release of MD is written in FORTRAN with calls to PVM routines, and is structured as a MIMD application.

The simulation of material samples with larger number of molecules turns the behaviour of MD from communication intensive to computation intensive. In our investigations the number of molecules has been kept as low as 4000 to stress the communication side.

MD performs a standard Lennard-Jones calculation plus the solution of the induced polarizability on each molecule taking in account first dipole momentum. Each step of MD consists of evaluating the induced dipoles $\bar{p}_i$ consistent with the values of $\bar{E}_i^{(q)}$ due to a given distributions of the point charges. This part of the calculation requires an iterative procedure with small computation time and many communications to exchange the values of the induced polarizability at each iteration among all processors. For a small number of molecules the cutoff radius is of the same size as the replicated box and the number of force vectors between molecule pairs grows almost quadratically with the total number of molecules. In such a situation any domain decomposition technique based on the spatial position of each molecule in the box is not feasible.

In the parallel implementation each processor maintains a copy of the position of each molecule. However each processor will compute force pairs only on a predefined subset of molecules which has been previously assigned to it. In this way the list of interacting particles, which is by far the larger data structure of MD, could be partitioned among the computation nodes and the total memory occupancy per processor is expected to decrease with increasing number of computation nodes.

When using high-latency communication systems like PVM, an important optimization is to keep the number of distincts messages as low as possible in order not to pay too much for the communication start-up costs. This is achieved by packing all the variables to be communicated (i.e. forces,virial,energy) in a single outgoing message whenever possible. Keeping the number of distinct messages as small as possible reduces the possibility of using multicast/broadcast communication primitives, since in PVM such collective communications are implemented as bare repetitions of point-to-point communications. Almost all communications were point-to-point ones, but a few of them, i.e. the exchange of the new coordinates of the molecules.

4 Migrating the application from PVM to GAMMA

In order to migrate MD from PVM to GAMMA to obtain the MD-GAMMA application, the GAMMA programming library has been extended with FORTRAN stubs to the original GAMMA communication C functions in a straighforward way.

Our PC cluster is equipped with low-cost shared 100base-T Ethernet hardware. This implies that the communication patterns of MD may cause lots of Ethernet collisions, with heavy communication delays. This could be partially avoided if the Fast Ethernet hub be replaced by a switch, but at a higher price. The alternative is to explicitly program a proper serialization of network accesses at the application level and to take best advantage of the Ethernet's hardware

broadcast facility that the GAMMA programming interface directly exposes. The serialization of communications during collective all-to-all data exchanges has been obtained in MD-GAMMA by considering all processes as circularly ordered by instance number and implicitly granting broadcast transmission right to a process after it has received broadcast messages from all its predecessors.

Another source of performance degradation with MD is the need of application-level temporary storage for incoming messages. Even with a "zero-copy" messaging system like GAMMA, MD-GAMMA must implement a temporary storage for received messages, because some broadcast messages carry information to be scattered among many processors and summed component-wise to existing local information arranged as arrays.

A potential problem with GAMMA is that the receiver is forced to accept messages in their final destination at any time the sender starts a communication. This may cause race conditions in the memory of the receiver process during the all-to-all exchange phase of MD. Such all-to-all exchange is a two-steps operation structured as two communication phases interleaved by one computation phase. In the computation phase the fresh data from the first communication phase are manipulated i.e. summed to previous data. If data from the second communication were delivered in the same data structure as data from the first communication, an inconsistency would arise if the second communication occurs before the intermediate computation step is complete. To avoid such race conditions in MD-GAMMA we had to implemented FIFO queues of application receive buffers for storing incoming GAMMA messages. Computations are carried out directly on the FIFOs' head arrays, whereas fresh incoming data are stored in the FIFOs' tail arrays. This way data from the second communication phase do not overwrite data from the first phase which have not yet been processed.

Migrating MD from PVM to GAMMA required one week of work from the first author of this paper to replace PVM calls with GAMMA calls, change some communication patterns and implement Active Messages-like receive policies, plus an additional week of work from the second author to debug and run the obtained MD-GAMMA application.

5 Tuning the existing PVM application

Another possibility for porting an existing PVM application on a given target platform is to retain the original message passing interface and to tune the communication patterns of the application in order to increase performance by matching the target architecture.

In the case of MD, an obvious drawback of the original version when running on a bus-interconnected pool of processing nodes like a PC cluster with shared Fast Ethernet is bus contention, which may cause unacceptably large communication delays due to collision storms. The easiest way to overcome such problem is to serialize processes when accessing the network by adding a circulating token, implemented by ordered exchanges of null PVM messages.

In our preliminary study we added a circulating token only in one subroutine of MD, which turns out to be heavily used in the program run. The obtained MD-TOKEN application required a very limited working effort. The token overhead is negligibe compared to the overall communication overhead as well as the MD computation time.

6 Performance results

Let us consider the speed-up curves depicted in Figures 1. The slow-down exhibited by MD as the number of processors increases beyond eight is clearly apparent. Given the low computational power of Pentium 133 MHz CPUs, such behaviour accounts for the poor efficiency of the PVM messaging systems involving many temporary copies of messages during the traversal of many layers of communication protocols, as well as the collision storms arising from processes simultaneously accessing the shared LAN during the exchange phases of the program execution.

However the excellent speed-up curve of MD-GAMMA up to 16 nodes, with the promise of a good scaling over even more processors, is mainly due to the following reasons:

- the relatively poor floating-point computational power of Pentium 133 MHz CPUs
- the high efficiency of GAMMA inter-process communications
- the fine tuning of the communication patterns in the GAMMA version of the application, based on the knowledge of features (broadcast) and limitations (shared LAN) of the underlying communication hardware.

In spite of its lower collision rate, MD-TOKEN shows a speed-up curve which is even worse than MD. The reason is that serializing network accesses by a circulating token implies serializing the software overhead of communications as well. When communication overhead is high, as with ordinary PVM, the potential advantage of eliminating collisions is by far recovered by the loss of parallelism in the execution of low-level communication software. Thus, coordinating processes at application level in the hope of making better use of the network may result into a counter effect with high-latency messaging systems. It is worth noting that the overhead of the circulating token itself is negligible (less than 5% with 16 nodes).

Figure 2 reports the average completion time per time-step for MD as well as MD-GAMMA and MD-TOKEN on our PC cluster. The curve of average completion time per time-step of MD on an eight-"thin-nodes" IBM SP2 is reported too. MD-GAMMA appears to outperform the IBM SP2 if more than twelve processors are engaged in the computation, besides performing better than the other two MD versions. When reading such curves it is important to pay attention to both the absolute performance and the cost of the hardware platform. It is worth pointing out that the current cost on the marketplace of a 16-nodes GAMMA leveraging shared 100base-T Ethernet and Pentium 133 MHz CPUs is comparable to the cost of one single high-end workstation.

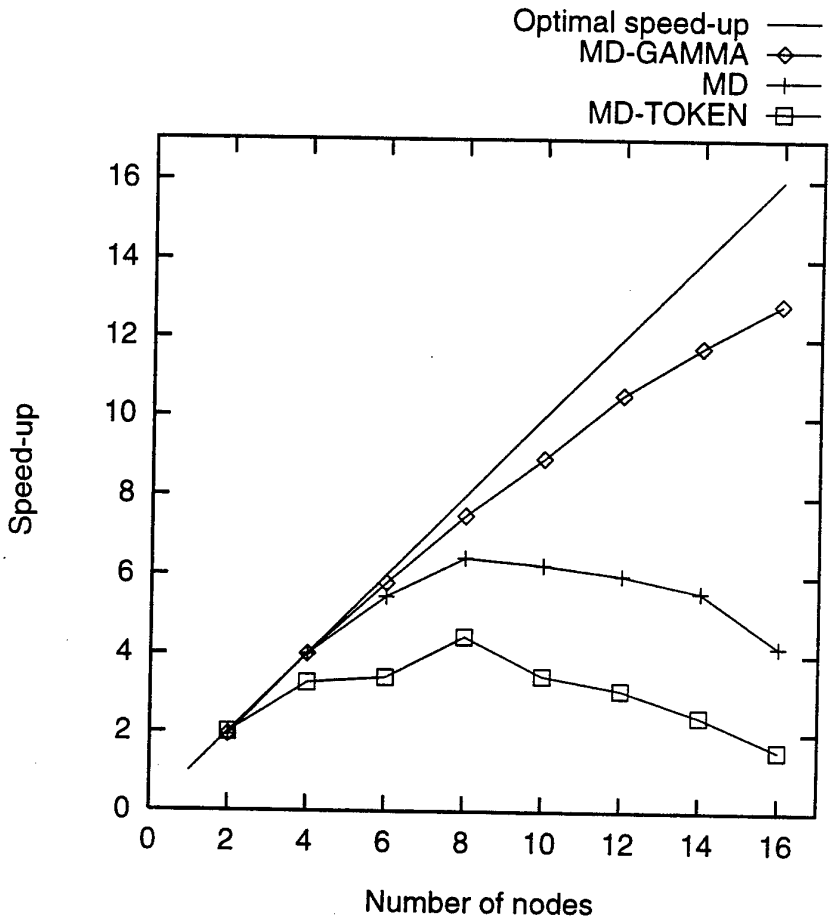


Fig. 1. Molecular Dynamics, GAMMA vs. PVM: speed-up comparison with same hardware platform (shared 100base-T Ethernet network of Pentium 133 PCs).

7 Conclusions

By using a low-latency messaging system like GAMMA, a significant number of networked PCs may be successfully exploited to run parallel code even with a low-cost interconnect like shared 100base-T Ethernet. Indeed low-latency as well as native broadcast communications offer more flexibility at the programming level to implement collision-free collective communication patterns. Similar collision-free patterns are not feasible with high-latency messaging systems like PVM providing a poor implementation of broadcast and too high a communication

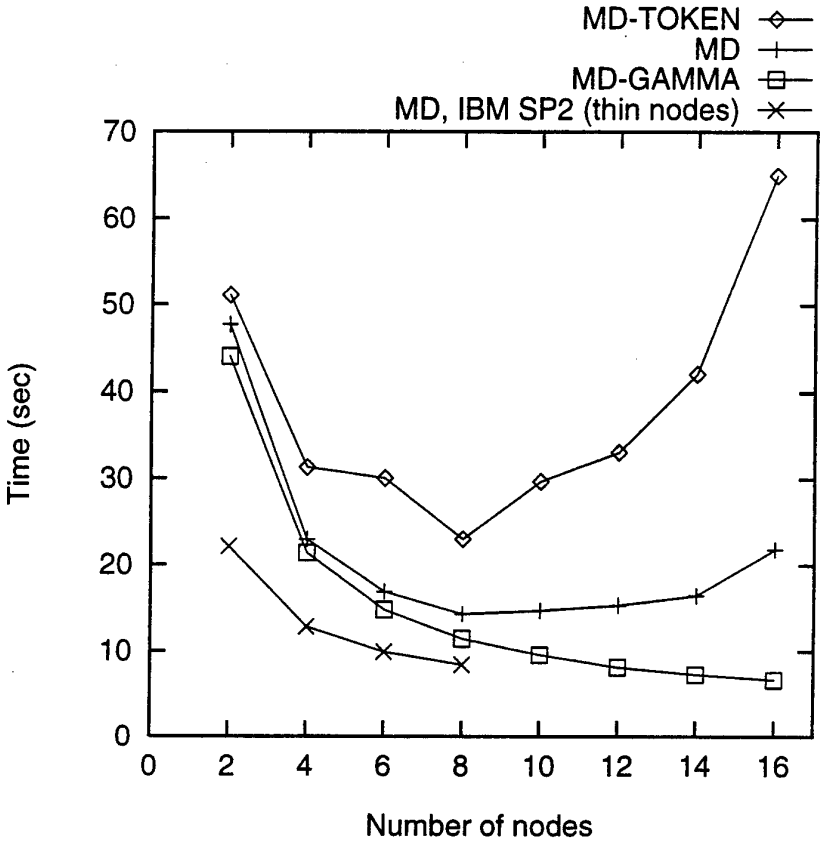


Fig. 2. Molecular Dynamics: average completion time per time-step on various parallel platforms including GAMMA.

overhead, which are not expected to decrease at the same rate at which the peak communication bandwidth offered by the Ethernet technology is increasing (not to mention the additional loss of efficiency when moving to SMP processing nodes).

In the case of MD it is apparent that exploiting a low-latency messaging system like GAMMA is the only way to turn a low-cost cluster of PCs into a cost-effective solution for parallel processing. The same holds for the large class of "non-embarassingly parallel" well-balanced parallel applications. The gain in price/performance as well as the good absolute performance level obtained on such kind of inexpensive platforms makes the porting effort worthwhile, at least in the case of well documented applications.

References

1. G. Chiola and G. Ciaccio. Implementing a Low Cost, Low Latency Parallel Platform. *Parallel Computing*, (22):1703–1717, 1997.
2. G. Ciaccio. Optimal Communication Performance on Fast Ethernet with GAMMA. In *Proc. Workshop PC-NOW, IPPS/SPDP'98*, pages 534–548, Orlando, Florida, April 1998. LNCS 1388, Springer-Verlag.
3. G. Ciaccio and V. Di Martino. Porting a Molecular Dynamics Application on a Low-cost Cluster of Personal Computers running GAMMA. In *Proc. Workshop PC-NOW, IPPS/SPDP'98*, pages 524–533, Orlando, Florida, April 1998. LNCS 1388, Springer-Verlag.
4. V. Di Martino. Computer Simulation of Polarizable Fluids. In *First European PVM Meeting*, Rome, October 1994.
5. V. Di Martino, G. Ruocco, and M. Sampoli. Molecular dynamics of polarizable fluids on parallel systems. In *HPC-ASIA '95*, Taipei, Taiwan, September 1995.
6. V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, pages 315–339, December 1990.
7. The Message Passing Interface Forum. MPI: A Message Passing Interface Standard. Technical report, University of Tennessee, Knoxville, Tennessee, 1995.
8. T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, Gold Coast, Australia, May 1992. ACM Press.

This article was processed using the LaTeX macro package with LLNCS style

Limits of Instruction Level Parallelism with Data Value Speculation

José González and Antonio González
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
{joseg,antonio}@ac.upc.es

Abstract. Increasing the instruction level parallelism (ILP) is one of the key issues to boost the performance of future generation processors. Current processor organizations include different mechanisms to overcome the limitations imposed by name and control dependences but no mechanisms targeting to data dependences. Thus, these dependences will become one of the main bottlenecks in the future. Data value speculation is gaining popularity as a mechanism to overcome the limitations imposed by data dependences by predicting the values that flow through them. In this work, we present a study of the potential of data value speculation to boost the limits of instruction level parallelism using both perfect and realistic predictors. Speedups obtained by data value speculation are very huge for an infinite window and still significant for a limited window. Different prediction schemes oriented to single thread and multiple threads (from a single program) architectures have been studied. The latter shows a significant improvement respect to the former for FP benchmarks although the difference is much smaller for integer programs.

1 Introduction

The performance of superscalar processors is limited by the necessity to obey the dependences existing among the program instructions. These dependences can be classified into three types[5]: name dependences, control dependences and data dependences.

Name dependences appear when the values generated by two instructions are to be written in the same storage location, either a register or memory. They can be eliminated by renaming the storage location that causes the dependence (i.e. changing the name of the locations where the values are to be written). Register renaming is a well known technique that deals with this kind of dependences. It is implemented dynamically by many current microprocessors such as DEC Alpha 21264 [4] or MIPS R10000 [23].

Control dependences are caused by branch instructions. They slow down the processor since it has to stall the fetch of instructions until the branch is solved, i.e. the destination address is computed and the condition is evaluated. Branch prediction is the mechanism that current microprocessors implement in order to overcome control dependences. It is based on the prediction of the outcome of branches which allows instructions that depend on a branch to be executed before the result of such branch is known.

Data dependences or true dependences appear when an instruction consumes the value produced by another previous instruction. These dependences are enforced in current microprocessors by executing the consumer after the producer. Thus, data dependences limit the amount of instruction level parallelism (ILP) by imposing a serialization on the execution of some instructions.

In the same way as control dependences are managed predicting the behavior of branches, it may be feasible to predict the result of some instructions in order to avoid the ordering imposed by data dependences, allowing the consumer instruction to be issued before the execution of the producer. The term *data value speculation* is used to refer to those mechanisms that predict the operands of an instruction, either source or destination, and execute speculatively the instructions dependent on it before the actual value is computed, allowing the processor to avoid the ordering imposed by data dependences.

In this work, we present a study of the ILP improvement that data value speculation techniques can provide. We present an evaluation of the limits of ILP that can be exploited by dynamically scheduled processors with infinite resources and data value speculation, and compare it with that of the same processor without data value speculation. We evaluate the benefits of predicting individual types of instructions (loads, stores, simple arithmetic, and multiplications) and the improvement achieved by predicting all of them. We consider both ideal prediction schemes and realistic ones. Finally, the impact of data value speculation for a limited instruction window is also evaluated. The results shows that data value speculation can significantly increase the ILP that dynamically scheduled processors can exploit, and therefore, it is a promising technique to be considered for future generation microprocessors.

The rest of this paper is organized as follows. Section 2 reviews the related work. The methodology to evaluate the ILP that can be exploited by an ideal processor, either with or without data value speculation, is described in section 3. The value predictors considered in this work are presented in section 4. The results of this study are detailed in section 5. Finally, section 6 summarizes the main conclusions of this work.

2 Related work

There have been a plethora of works dealing with the limits of the ILP [1][2][6][10][16][20][21]. Each work studies the ILP that could be exploited under some constraints such as fetch width, instruction window size, branch prediction, register renaming, memory aliasing, etc. A conclusion that can be extracted from all these works is that one of the main features that limit the parallelism are data dependences. For instance, in [5] it is shown that the maximum ILP that a processor could achieve with infinite resources and perfect branch prediction is not much higher than a few hundred instructions per cycle (IPC) and for some applications it is about a few tens of IPC.

Data value speculation has been the focus of several recent works. It is performed in [14] by predicting the address of load instructions whereas in [9] the address of stores is also predicted. In both cases the prediction is carried out using a history table of memory instructions and a stride based predictor. In [12], data value speculation is based on predicting the value that load instructions read from memory. The proposed mechanism exploits the feature that the authors call *value locality*, which refers to the fact that many

load instructions repeatedly bring the same value from memory. Value locality is extended for all type of instructions in [11]. In [8] data value speculation is performed by predicting the value read by load instructions. Unlike the mechanism proposed in [12], the load values are predicted by predicting their effective address and prefetching the data from memory into the history table. In [15] Sazeides and Smith show that the results that an instruction generates may follow a repetitive pattern that stride predictors cannot predict and propose a context-based predictor. In [22] Wang and Franklin present a hybrid predictor. The implementation of this predictor is similar to that of a 2-level branch predictor. In [7] the impact of different value predictors on the performance of a processor is studied using a limited instruction window.

The main contributions of this work are the following: This is the first work to our knowledge that evaluates the limits of ILP in an ideal dynamically scheduled superscalar processor that exploits data value speculation and compares it with that of the same processor without data value speculation. In [11], value prediction is evaluated for a perfect machine, as it is called by the authors. However, that machine is limited by a finite instruction window (4096 entries), branch prediction and fetch bandwidth. Besides, in this paper we study the benefits of predicting individual types of instructions for both ideal and realistic predictors.

3 Methodology

This section describes the methodology that we have used to obtain the ILP under different scenarios regarding prediction schemes and hardware resources.

3.1 Experimental framework

The evaluation methodology is trace-driven. The trace of each program has been generated using the ATOM tool [19]. For each instruction, the instrumentation routine obtains: its operation code, the source and target registers, the effective address (if the instruction is either a load or a store), and the value generated in the case of arithmetic and load instructions. These data are fed into the analysis program, which computes the performance achieved by the particular architectural model. Performance is reported as Instructions per Cycle (IPC).

The whole SPEC95 benchmark suite has been used for the different experiments. All the benchmarks have been compiled for a DEC AlphaStation600 5/266 with '-O4' optimization flag, and executed with their largest input set. Each program has been run for 5 billion of instructions, except gcc and ijpeg, which have been run until completion (1,569,885,184 and 684,497,921 instructions respectively). Figure 1 details the percentage of different types of instructions executed for the whole SPEC95 benchmark suite.

3.2 Architectural model

The first study of the limits of ILP is achieved assuming an ideal microprocessor with infinite resources, perfect branch prediction, infinite instruction fetch bandwidth, an infinite cache memory with infinite number of ports, perfect memory disambiguation, dynamic renaming with an infinite number of registers and memory renaming with infi-

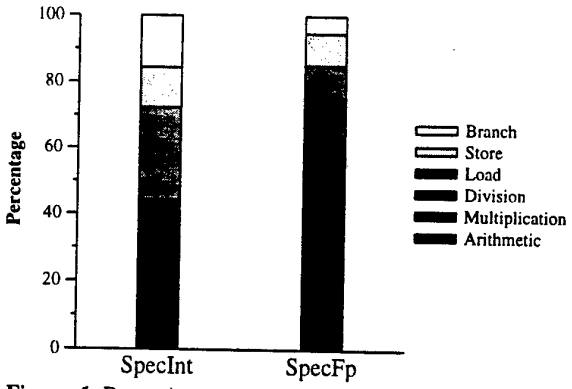


Figure 1. Dynamic percentage of each type of instructions

nite storage locations for renaming. Both an infinite and a limited instruction window are considered. In all the cases, precise exceptions [17] and an infinite retirement (commit) bandwidth are assumed.

3.3 IPC computation for an ideal architecture without data value speculation

The IPC of a given program for a particular architectural model is obtained by determining the time (measured in number of cycles) when the latest result of any instruction of the program is computed, and then, dividing the number of executed instructions by such number of cycles.

We will refer to the cycle when the result of an instruction i is available as the *completion time* of i, or CT_i for short. CT_i is computed as the maximum CT_j for any j such that j produces a result that is a source operand of i plus the latency of the operation i. This approach is similar to the one used in [1].

Each instruction of the trace produced by the execution of the instrumented program is analyzed in order to know the time when its operands are available. For each storage location the analysis program keeps track of the CT of the last instruction that wrote to it. This is implemented by means of two tables that are called the *register write table (RWT)* and the *memory write table (MWT)*. RWT_r stores the CT of the last instruction so far that its destination operand was the logical register r. MWT_a stores the CT of the last store that wrote into address a.

Therefore, when an arithmetic instruction is processed, the RWT is accessed in order to obtain the cycle that the source operands are available. Then, its CT is computed and the RWT entry associated to its destination register is updated with the new computed CT. That is:

$$RWT_{dest} = max\,(RWT_{src1}, RWT_{src2}) + Latency_{operation} \qquad (1)$$

In a similar way, when a load from address a is processed, the MWT is accessed to obtain the cycle that a previous store wrote into that memory position. Then, the RWT is updated as follows:

$$RWT_{dest} = max\,(RWT_{src1}, RWT_{src2}, MWT_a) + Latency_{load} \quad (2)$$

Finally, when a store to address a is processed, the *MWT* is updated to reflect the new write to this memory location:

$$MWT_a = max\,(RWT_{src1}, RWT_{src2}) + Latency_{store} \quad (3)$$

Notice that the new RWT_{dest} or MWT_a can be lower that the previous one because register and memory renaming is assumed. Dynamic register renaming is very common in current architectures. Memory renaming is much more complex and it is implemented to some extent by some mechanisms like the ARB of the Multiscalar [3]. In this paper, we assume unlimited renaming capabilities for both registers and memory.

When a new value for *RWT* or *MWT* is computed, the previous value is overwritten because any further instruction in the trace will always refer to the last value stored into a register or a memory location. However, in order to compute the IPC, we have to determine the maximum *CT* for any instruction of the program. To obtain such value, the analysis program keeps a variable that stores the maximum *CT* up to the current execution point (*Max_CT*).

3.4 IPC computation for a limited instruction window

A limited instruction window with W entries and in-order retirement implies that an instruction cannot start execution until the instruction W locations above in the trace and all previous instructions have completed and retired. Thus, the restriction of having a limited instruction window can be modeled by keeping track of the *CT* of the last W instructions. This is accomplished by means of a table, which is called *window retirement time* (*WRT*), that has W entries and stores the retirement time of the last W instructions processed so far.

Thus, when computing the *CT* of an instruction, in addition to consider the *CT* of its source operands, the *WRT* of the instruction W locations above has also to be considered. For instance, for each arithmetic instruction processed by the analysis program, the corresponding entry in the *RWT* is updated as follows:

$$RWT_{dest} = max\,(RWT_{src1}, RWT_{src2}, WRT_{n\_inst\%W}) + Latency_{operation} \quad (4)$$

where *n_inst* refers to the ordinal number of the current instruction in the trace. Expressions (2) and (3) are modified in a similar way to account for the effect of the limited instruction window.

For each new instruction, the *WRT* is updated to reflect the retirement (commit) time of the current instruction. This time is the maximum *CT* of any previous instruction, including the current one, and it is stored in the same entry of the *WRT* that was occupied by the instruction W locations above since it is not useful any more:

$$WRT_{n\_inst\%W} = Max\_CT \quad (5)$$

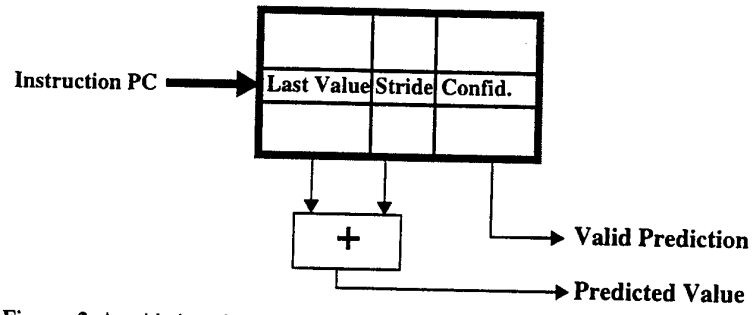


Figure 2. A stride-based predictor.

3.5 IPC computation for data value speculation

Data value speculation is based on predicting the source and/or the destination operands of some instructions. In this section, we present a methodology to compute the IPC when data value speculation is incorporated into a superscalar processor, independently of the particular predictor being used. In this way, we consider a predictor as a system that given an instruction (usually its program counter), provides its source and/or destination operands. In addition, each individual prediction is characterized by the time when the prediction is available (*PT*) and the correctness of the prediction.

In this paper, we consider data value speculation for the following type of instructions: Loads, Stores, Integer Arithmetic, Integer Multiplication, Float Arithmetic and Float Multiplication.

In all the cases, if a prediction is not correct, the *RWT* and *MWT* are updated as if prediction were not used. If the prediction is correct, the *RWT* and *MWT* are updated with the minimum between the completion time, given by expressions (1), (2) and (3), and the prediction time, which is a characteristic of the particular predictor being used. Section 4 discusses the predictors considered in this work and in particular, the time when predictions are available.

4 Data predictors

In this work we consider stride-based predictors, although the presented methodology could be applied for any other data predictor. A stride predictor has the structure shown in Figure 2. It is implemented by means of a table of 4096 entries that is direct-mapped, non-tagged and it is indexed with the least significant bits of the instruction address (PC) whose source or destination operands are to be predicted. Each entry stores the following information:

- Last value: This is the last value seen by that instruction. This value corresponds to the destination operand for all predictors except for the load and store address predictors. In these cases, it corresponds to the last effective address.
- Stride: This field contains the stride observed for the values of the corresponding instruction.

- Confidence: This field is used to assign confidence to the prediction. It is implemented by means of a 2-bit up/down saturated counter. A prediction is considered correct only if the most significant bit is set.

Predictor for arithmetic instructions stores the last result in the last value field. Load address predictors store the last effective address. Load value predictors store the last value read from memory. Finally, store predictors uses two tables: one for predicting the effective address and the other for predicting the value to be written.

When an instruction is to be predicted (either its result or its effective address, depending on the particular predictor), the prediction table is accessed and the predicted value is computed adding the stride to the previous last value. If the most significant bit of the confidence field is set (i.e., the prediction is considered to be correct) and the prediction is correct, the predicted value can be used instead of the actual value if the former is available earlier. The stride field is only updated if the confidence counter is below 10_2 after being updated.

In addition, we consider a perfect predictor that is assumed to produce always correct predictions. This is used to determined the upper bound of the performance that data value speculation can achieve.

4.1 Prediction time

An important feature of a predictor is the time when the predicted value is available. This time is used to update the *RWT* and *MWT* as explained in section 3.5.

Regarding the prediction time, two different types of predictors have been considered:

- Serialized: Every time the prediction table is accessed, only one prediction per static instruction can be performed at most. That is, an instruction is not predicted until the last execution of the same static instruction has been predicted.
- Non-serialized: Every time the prediction table is accessed, multiple predictions for each static instruction can be performed. In particular, all the subsequent executions of the same static instruction are predicted until the first one that is incorrect. That is, once the corresponding entry of the table has the correct stride, successive executions of the same static instructions can be predicted all at once.

The serialized predictors may be suitable for superscalar processors. In fact, most of the studies on value prediction assume this type of predictors [8][9][11][12][14]. A non-serialized predictor could be useful for architectures supporting multiple threads of control obtained from a single program, such as multiscalar processors [18] and the speculative multithreaded processors [13].

To determine the time when a prediction is available we consider a parameter that reflects the time required to perform a prediction operation (either of a single value for the serialized approach or multiple values for the non-serialized one). This parameter is called the *prediction latency (PL)*. This is the time required for a table look-up plus its update.

The prediction time of each instruction is determined by means of an additional field that is added to each entry of the prediction table for evaluation purposes. This field stores the cycle in which the entry has been used/updated for the last time. This field will be called *last update time* (*LUT*).

The prediction time for an instruction is just the sum of the last update time plus the prediction latency. That is:

$$PT = LUT + PL \tag{6}$$

The *LUT* is updated in a different way for serialized and non-serialized predictors. For the former, for each new instruction of the trace, the corresponding *LUT* is updated with the time when its operand is available (either computed or predicted, whichever occurs first):

$$LUT = RWT_{dest} \text{ for load and arithmetic instructions with destination register dest}$$

$$LUT = MWT_a \text{ for stores to address a} \tag{7}$$

For non-serialized predictors, the *LUT* field is updated in the same way as the serialized case but only for those instructions that are mispredicted or are considered not predictable as stated by the confidence field.

5 Results

The results of this section assume a one-cycle latency for all instructions and one-cycle prediction latency.

Table 1 shows the IPC achieved by the ideal processor described in section 3.2 with an infinite instruction window and without data value speculation

This results will be used as a baseline to compare the performance of data value speculation techniques. They represent the maximum parallelism that is possible to achieve in an ideal processor that is only constrained by data dependences whereas data value speculation removes this constraint. Notice that even for this ideal machine, the average IPC is only 37.39 for integer programs and 790.29 for floating point applications. When we add the constraint of a limited instruction window of 128 instructions, the IPC goes down to 9.64 and 17.51 respectively. This may suggest that relieving the restrictions imposed by data dependences through data value speculation can be and interesting mechanism to boost performance. In the following results, only the average result for integer and FP programs will be shown.

Figure 3 shows the speedup (in logarithmic scale) achieved by data value speculation with perfect prediction in relation to the infinite machine without data value speculation. In this figure and the following ones the speedup is computed as follows:

$$\text{Speedup} = \frac{\text{IPC with data value speculation}}{\text{IPC without data value speculation}}$$

In each bar, only a single type of instructions is predicted individually. With perfect prediction, when an instruction is predicted its result is considered to be available at cycle 0. Looking at the graphs, one can see that the potential performance of predicting

Table 1. IPC achieved with infinite resources and no data value speculation

SpecInt	IPC	SpecFP	IPC
go	89.45	tomcatv	397.79
m88ksim	17.14	swim	1403.82
gcc	47.02	su2cor	56.64
compress	35.71	hydro2d	181.09
li	27.62	applu	578.31
ijpeg	34.12	mgrid	4735.11
perl	18.72	turb3d	140.19
vortex	29.34	apsi	231.21
		fpppp	105.71
		wave5	73.02
Average	37.39	**Average**	790.29

memory instructions, both loads and stores, is less than the speedup achieved by predicting arithmetic instructions. This suggests that for the analyzed programs, there are much more arithmetic than memory instructions on critical paths. The speedup achieved by predicting multiplications is almost negligible. In addition to not being on critical paths, this may be due to the small percentage of multiplication operations, as shown in Figure 1.

Figure 4 shows the speedup obtained for a realistic prediction scheme based on a stride predictor, as it was described in previous sections. The instruction window is considered to be infinite and the prediction is non-serialized. The speedup achieved by predicting arithmetic instruction is very huge and it suggests that arithmetic prediction may be the most effective approach to remove the serialization imposed by data dependences. The IPC of data value speculation just for arithmetic instructions is 531 times higher than the IPC achieved without data value speculation, for an infinite machine and the FP benchmarks. When data value speculation is implemented for all the instructions, the speedup goes up to 2368. The speedup for integer programs is not so high (42 when predicting all the instructions). On the other hand, the speedup achieved by predicting memory instructions is much more limited (1.4 and 4.8 for integer and FP benchmarks respectively when predicting stores and load values). Predicting multiplications is not considered any more due to the poor results observed for the perfect predictor.

The speedup obtained with a serialized predictor is depicted in Figure 5. Notice that, as pointed out before, this scheme would correspond to the implementation of data value speculation on a superscalar processor since in such processors there is only one

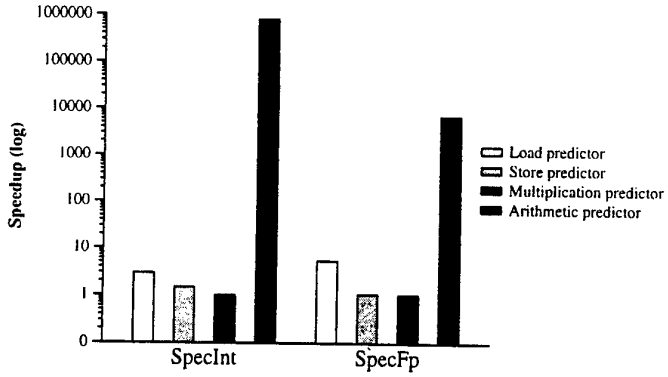


Figure 3. Speedup achieved by data speculation with perfect prediction, for different types of predictors.

flow of control and a given execution of a static instruction can be predicted only if its previous execution has updated the prediction table. On the other hand, a non-serialized predictor can be exploited by an architecture supporting multiple threads of control.

The speedup achieved by serialized prediction is still quite significant. The IPC achieved by these schemes is 30 and 35 times higher than the IPC achieved without data value speculation for integer and FP programs respectively. These results also show that the potential gain that load prediction may achieve is slightly higher for value prediction than for address prediction, but this gain is insignificant when compared to arithmetic prediction.

If we compare the speedup achieved by non-serialized prediction (Figure 4) against the speedup achieved by serialized prediction (Figure 5) we can observe that for integer benchmarks there is not much difference (e.g. it goes from 42 to 30 when predicting all the instructions) whereas for FP benchmarks the difference is huge (e.g. it goes from 2368 to 35 when predicting all the instructions). The main reason for this different behavior in the two types of benchmarks can be explained through the figures in Table. 2. This table shows the percentage of correctly predicted arithmetic instructions for which the completion time (*CT*) is lower than prediction time (*PT*). For these instructions, the prediction does not provide any improvement in spite of being correct. As expected, this percentage is greater when the predictions are serialized than when they are not since the prediction time of the serialized scheme is in general higher. Besides, the difference between serialized and non-serialized schemes for FP benchmarks is much higher than for integer benchmarks, which explains the higher impact of serialized prediction for FP benchmarks, as observed in Figure 4 and Figure 5.

The speedup achieved by predicting instructions relies on the amount of strided values existing among the applications. Figure 6 shows the percentage of strided values for the different instruction types for the whole Spec95 benchmark suite. It can be seen that load addresses have the greatest percentage of strided references and therefore one may expect a speedup for load address speculation higher than it actually is (see Figure 4 and Figure 5). However, even when the address of a load is predicted, it has to wait for previous stores to the same address to finish. On the other hand, predicting the value of a

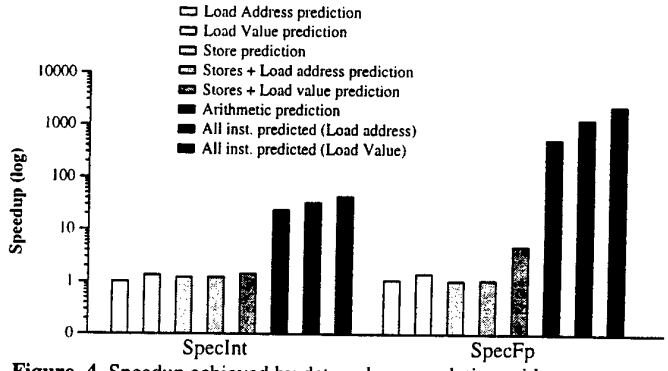


Figure 4. Speedup achieved by data value speculation with non-serialized prediction

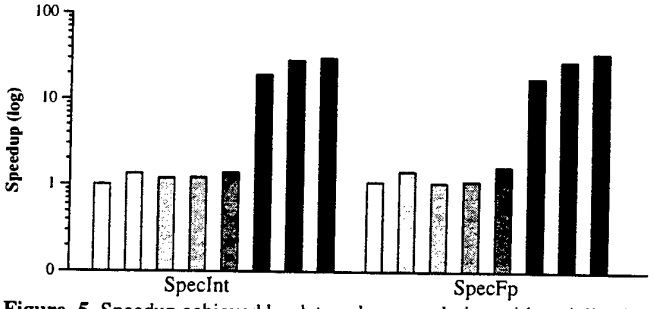


Figure 5. Speedup achieved by data value speculation with serialized predictions

Table 2. Percentage of correctly predicted instructions whose CT is lower than its PT.

	Non-serialized	**Serialized**
SpecInt	59.65	70.85
SpecFp	48.31	90.64

load or the result of any other instruction avoids completely the order imposed by data dependences. Simple arithmetic instructions (mainly integer arithmetic) has a high percentage of strided values. This fact, along with the significant weight of arithmetic instructions on the critical path (as confirmed in the evaluation of the prefect prediction scheme), makes arithmetic prediction to be the most effective type of speculation among the ones evaluated in this work.

Finally, we consider the impact of data value speculation with a limited instruction window. Figure 7 shows the speedup of data value speculation (IPC achieved by data value speculation divided by IPC achieved without data value speculation) when all types of instructions are predicted using separate history tables for each class, and predicting the value of loads. A non-serialized predictor is considered since it outperforms a serialized predictor for an infinite window (notice that the speedup is not depicted in logarithmic scale but in linear scale). It can be seen in this figure that the impact of the size of the instruction window its very significant since, for instance, the speedup is decreased from 2368 to only 1.75 for a window of 512 instructions in the SpecFp pro-

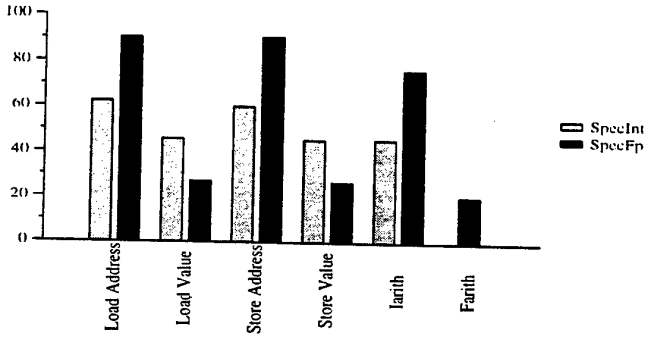


Figure 6. Percentage of strided values for each type of instruction

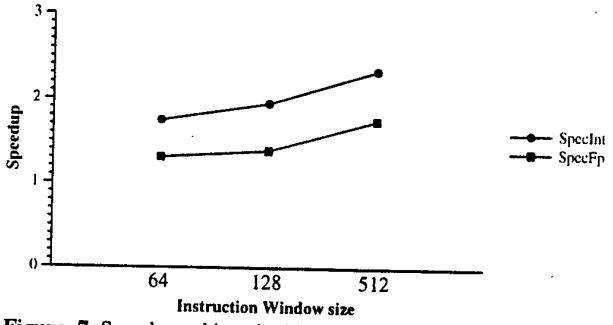


Figure 7. Speedup achieved with a finite instruction

grams. Furthermore, the gain due to data value speculation for the SpecInt outperforms the gain for SpecFp, which is the opposite to what happened with an infinite instruction window.

A main conclusion of the study of the effect of data value speculation on a limited instruction window is that it is an effective technique that could be considered for future generation microprocessors. A speedup around 2 can be achieved with simple stride-based predictors. However, the potential benefits of data value speculation are much higher for very large instructions windows. In this scenario, conventional superscalar microprocessors have been shown to be rather limited in the amount ILP that they can exploit due mainly to data dependences. This limitation can be significantly relieved by data value speculation techniques. Thus, novel organizations to support large instructions windows, like the multiscalar architecture [18] and speculative multithreaded processor[13] can be benefitted from data value speculation to a larger extent than superscalar processors.

6 Conclusions

In this work we have presented a study of the limits of instruction level parallelism (ILP) that can be exploited by a machine with infinite resources, infinite instruction window, perfect branch prediction and ideal memory. We have shown that avoiding the

ordering imposed by data dependences is a promising approach to improve the performance of superscalar processors for future generations. This can be accomplished by data value speculation techniques. These techniques are based on predicting the source or destination operands of instructions and execute speculatively the instructions dependent on them.

Data value speculation has been approached by means of both perfect and stride-based predictors. Two different types of prediction schemes have been studied: serialized and non-serialized. The former is oriented to superscalar processors whereas the latter is more suitable for multithreaded architectures (i.e., machines that support multiple threads of control from a single program). We have measured the benefits of data value speculation techniques by comparing the limits of ILP that can be exploited with such technique with that of a superscalar processor with the same features but without data value speculation. Results show an important speedup for arithmetic instructions both for serialized and non-serialized prediction schemes. We have also observed that the difference between these two schemes is very high for FP programs (non-serialized outperforms always serialized schemes) but it is relatively low for integer programs.

Finally, we have evaluated the impact of data value speculation with a limited instruction window. We have observed that the speedup suffers an important reduction but it is still significant. However, the benefits of data value speculation increases with the instruction size. We believe that data value speculation may play an important role when it is combined with mechanisms to support large instruction windows.

7 Acknowledgements

This work has been supported by the Spanish Ministry of Education under grant CYCIT TIC 429/95, the ESPRIT project MHAOTEU (24942) and the Direcció General de Recerca of the Generalitat de Catalunya under grant 1996FI-03039-APDT.

The research described in this paper has been developed using the resources of the Center of Parallelism of Barcelona (CEPBA).

References

1. T.M. Austin and G. S. Sohi. "Dynamic Dependency Analysis of Ordinary Programs". In *Proc. of Int. Symp. on Computer Architecture,* pp 342-351, 1992.
2. M. Butle T.Y. Yeh, Y. Patt, M. Alsup, H. Scales and M. Shebanowr "Single Instruction Stream Parallelism is Greater than Two". In *Proc. of Int. Symp. on Computer Architecture*, pp. 276-286, 1991.
3. M. Franklin and G. S. Sohi. "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References". *IEEE Transactions on Computer,*45(6), pp. 552-571, May 1996.
4. L. Gweunnap. "Digital 21264 Sets New Standard". *Microprocessor Report,* 9(3), March 1995.
5. J.L Hennessy and D.A. Patterson. *Computer Architecture. A Quantitative Approach.* Second Edition. Morgan Kaufmann Publishers, San Francisco 1996.
6. N.P. Jouppi and D.W. Wall. "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines". In *Proc. of the ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, 1989.

7. F. Gabbay and A. Mendelson. "Speculative Execution Based on Value Prediction". Technical Report, Technion, 1997
8. J. Gonzalez and A.Gonzalez. "Speculative Execution via Address Prediction and Data Prefetching". In *Proc. of the International Symposium on Supercomputing (ICS)*, pp 196,203, 1997.
9. J. Gonzalez and A. Gonzalez. "Memory Address Prediction for Data Speculation". *In proceedings of the Europar Conference*, 1997.
10. M.S. Lam and R.P. Wilson. "Limits on Control Flow on Parallelism". In *Proc. of Int. Symp. on Computer Architecture*, pp 46-57, 1992
11. M.H. Lipasti and J.P. Shen. "Exceeding the Dataflow Limit via Value Prediction". In *Proc. of Int. Symp. on Microarchitecture*, 1996.
12. M.H. Lipasti, C.B. Wilkerson and J.P. Shen. "Value Locality and Load Value Prediction". In *Proc. of the ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, 1996.
13 P. Marcuello, A. Gonzalez and J. Tubella. "Speculative Multithreaded Processors". In *Proc. of the International Symposium on Supercomputing (ICS)*, 1998.
14. Y. Sazeides, S Vassiliadis and J.E. Smith. "The Performance Potential of Data Dependence Speculation & Collapsing". In *Proc. of Int. Symp. on Microarchitecture*, 1996.
15. Y. Sazeides and J.E. Smith. "The Predictability of Data Values". In *Proc. of Int. Symp. on Microarchitecture*, pp 248-258. 1997.
16. M.D. Smith, M. Johnson and M.A. Horowitz. "Limits on Multiple Instruction Issue". In *Proc. of the ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, 1989.
17. J.E. Smith and A.R. Pleszkun. "Implementing Precise Interrupts in Pipelined Processors". *IEEE Transaction on Computers*, 37(5), pp. 562-573, May 1988
18. G.Sohi, S.Breach and T. Vijaykumar."Multiscalar Processors". In *Proc. of Int. Symp. on Computer Architecture*, pp 414-425, 1995
19. A. Srivastava and A. Eustace. "ATOM: A system for building customized program analysis tools". In *Proc of the 1994 Conf. on Programming Languages Design and Implementation*, 1994.
20. K.B. Theobald, G.R. Gao and L.J. Hendren. "On the Limits of Program Parallelism and its Smoothability" In *Proc. of Int. Symp. on Microarchitecture*, pp 10-19, 1992.
21. D.W. Wall. "Limits of Instruction-Level Parallelism". Technical Report WRL 93/6 Digital Western Research Laboratory, 1993.
22 K. Wang and M. Franklin. "Highly Accurate Data Value Prediction using Hybrid Predictors". In *Proc. of Int. Symp. on Microarchitecture*, pp 281-290, 1997.
23. K.C. Yeager. "The MIPS R10000 Superscalar Microprocessor" *IEEE Micro*, 16(2), pp. 28-40, April 1996.

Simulating Magnetized Plasma with the Versatile Advection Code

Rony Keppens[1] and Gábor Tóth[2]

[1] FOM-Institute for Plasma-Physics Rijnhuizen,
P.O. Box 1207, 3430 BE Nieuwegein, The Netherlands,
keppens@rijnh.nl
[2] Department of Atomic Physics, Eötvös University,
Puskin u. 5-7, Budapest 1088, Hungary,
gtoth@hercules.elte.hu

Abstract. Matter in the universe mainly consists of plasma. The dynamics of plasmas is controlled by magnetic fields. To simulate the evolution of magnetized plasma, we solve the equations of magnetohydrodynamics using the Versatile Advection Code (VAC).
To demonstrate the versatility of VAC, we present calculations of the Rayleigh-Taylor instability, causing a heavy compressible gas to mix into a lighter one underneath, in an external gravitational field. Using a single source code, we can study and compare the development of this instability in two and three spatial dimensions, without and with magnetic fields. The results are visualised and analysed using IDL (Interactive Data Language) and AVS (Advanced Visual Systems).
The example calculations are performed on a Cray J90. VAC also runs on distributed memory architectures, after automatic translation to High Performance Fortran. We present performance and scaling results on a variety of architectures, including Cray T3D, Cray T3E, and IBM SP platforms.

1 MagnetoHydroDynamics

The MHD equations describe the behaviour of a perfectly conducting fluid in the presence of a magnetic field. The eight primitive variables are the density $\rho(\mathbf{r},t)$, the three components of the velocity field $\mathbf{v}(\mathbf{r},t)$, the thermal pressure $p(\mathbf{r},t)$, and the three components of the magnetic field $\mathbf{B}(\mathbf{r},t)$. When written in conservation form, the conservative variables are density ρ, momentum $\rho\mathbf{v}$, energy density $\mathcal{E}$, and the magnetic field $\mathbf{B}$. The thermal pressure p is related to the energy density as $p = (\gamma-1)(\mathcal{E} - \frac{1}{2}\rho v^2 - \frac{1}{2}B^2)$, with γ the ratio of specific heats. The eight non-linear partial differential equations express: (1) mass conservation; (2) the momentum evolution (including the Lorentz force); (3) energy conservation; and (4) the evolution of the magnetic field in an induction equation. The equations are given by

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0, \tag{1}$$

$$\frac{\partial(\rho \mathbf{v})}{\partial t} + \nabla \cdot [\rho \mathbf{v}\mathbf{v} + p_{tot} I - \mathbf{B}\mathbf{B}] = \rho \mathbf{g}, \tag{2}$$

$$\frac{\partial \mathcal{E}}{\partial t} + \nabla \cdot (\mathcal{E}\mathbf{v}) + \nabla \cdot (p_{tot}\mathbf{v}) - \nabla \cdot (\mathbf{v} \cdot \mathbf{B}\mathbf{B}) = \rho \mathbf{g} \cdot \mathbf{v} + \nabla \cdot [\mathbf{B} \times \eta(\nabla \times \mathbf{B})], \tag{3}$$

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{v}\mathbf{B} - \mathbf{B}\mathbf{v}) = -\nabla \times [\eta(\nabla \times \mathbf{B})]. \tag{4}$$

We introduced $p_{tot} = p + \frac{1}{2}B^2$ as the total pressure, I as the identity tensor, $\mathbf{g}$ as the external gravitational field, and defined magnetic units such that the magnetic permeability is unity.

Ideal MHD corresponds to a zero resistivity η and ensures that magnetic flux is conserved. In resistive MHD, field lines can reconnect. An extra constraint arises from the non-existence of magnetic monopoles, expressed by $\nabla \cdot \mathbf{B} = 0$. The ideal MHD equations allow for Alfvén and magnetoacoustic wave modes, while the induction equation prescribes that flow across the magnetic field entrails the field lines, so that field lines are 'frozen-in'. The field may, in turn, confine the plasma. The MHD description can be used to study both laboratory and astrophysical plasma phenomena. We refer the interested reader to [2] for a derivation of the MHD equations starting from a kinetic description of the plasma, while excellent treatments of MHD theory can be found in, e.g. [4, 1].

2 The Versatile Advection Code

The Versatile Advection Code (VAC) is a general purpose software package for solving a conservative system of hyperbolic partial differential equations with additional non-hyperbolic source terms [10, 11], in particular the hydrodynamic ($\mathbf{B} = 0$) and magnetohydrodynamic equations (1)-(4), with optional terms for gravity, viscosity, thermal conduction, and resistivity.

VAC is implemented in a modular way, which ensures its capacity to model several systems of conservation laws, and makes it possible to share solution algorithms among all systems. A variety of spatial and temporal discretizations are implemented for solving such systems on a finite volume structured grid. The spatial discretizations include two Flux Corrected Transport variants and four Total Variation Diminishing (TVD) algorithms (see [15]). These numerical schemes are shock-capturing and second order accurate in space and time.

Explicit time integration may exploit predictor-corrector and Runge-Kutta time stepping, while for multi-timescale problems, mixed implicit/explicit time integration is available to treat only some variables, or some terms in the governing equations implicitly [7]. Fully implicit time integration can be of interest when modeling steady-state problems. Typical astrophysical applications where semi-implicit and implicit methods are efficiently used can be found in [8, 14].

VAC runs on personal computers (Pentium PC under Linux), on a variety of workstations (DEC, Sun, HP, IBM) and has been used on SGI Power Challenge, Cray J90 and Cray C90 platforms. To run VAC on distributed memory architectures, an automatic translation to High Performance Fortran (HPF) is done

at the preprocessing phase (see [9]). We have tested the generated HPF code on several platforms, including a cluster of Sun workstations, a Cray T3D, a 16-node Connection Machine 5 (using an automatic translation to CM-Fortran), an IBM SP and a Cray T3E. Scaling and performance is discussed in section 3.

On-line manual pages, general visualization macros (for IDL, MatLab and SM), and file format transformation programs (for AVS, DX, and Gnuplot) facilitate the use of the code and aid in the subsequent data analysis.

In this manuscript, we present calculations done in two and three spatial dimensions, for both hydrodynamic and magnetohydrodynamic problems. This serves to show how VAC allows a single problem setup to be studied under various physical conditions. We have used IDL and AVS to analyse the application presented here. Our data analysis and visualization encompasses X-term animation, generating MPEG-movies, and video production.

3 Scaling results

As detailed in [9], the source code uses a limited subset of the Fortran 90 language, extended with the HPF *forall* statement and the Loop Annotation SYntax (LASY) which provides a dimension independent notation. The LASY notation [12] is translated by the VAC preprocessor according to the dimensionality of the problem. Further translation to HPF involves distributing all global non-static arrays across the processors, which is accomplished in the preprocessing stage by another Perl script.

Figure 1 summarizes timing results obtained on two vector (Cray J90 and C90) and three massively parallel platforms (Cray T3D, T3E and IBM SP). We solve the shallow water equations (1)-(2) with $\mathbf{B} = 0$ and $p = (g/2)\rho^2$ on a 104×104 grid on 1, 2, 4, 8, and 13 processors. This simple model problem is described in [13], and our solution method contains the full complexity of a real physics application. We used an explicit TVD scheme exploiting a Roe-type approximate Riemann solver. We plot the number of physical grid cell updates per second against the number of processors (solid lines). The dashed lines show the improved scaling for a larger problem of size 208×208, up to 16 processors. On all parallel platforms, we exploited the Portland Group *pghpf* compiler. We find an almost linear speedup on the Cray T3D and T3E architectures, which is rather encouraging for such small problem sizes. Note how the single node execution on the IBM SP platform is a factor of 2 to 3 faster than the Cray T3E, but the scaling results are poor. The figure indicates clearly that for this hydrodynamic application, on the order of 10 processors of the Cray T3E and IBM SP are needed to outperform a vectorized Fortran 90 run on one processor of the Cray C90. Detailed optimization strategies for all architectures shown in Figure 1 (note the Pentium PC result and the DEC Alpha workstation timing in the bottom left corner) are discussed in [13].

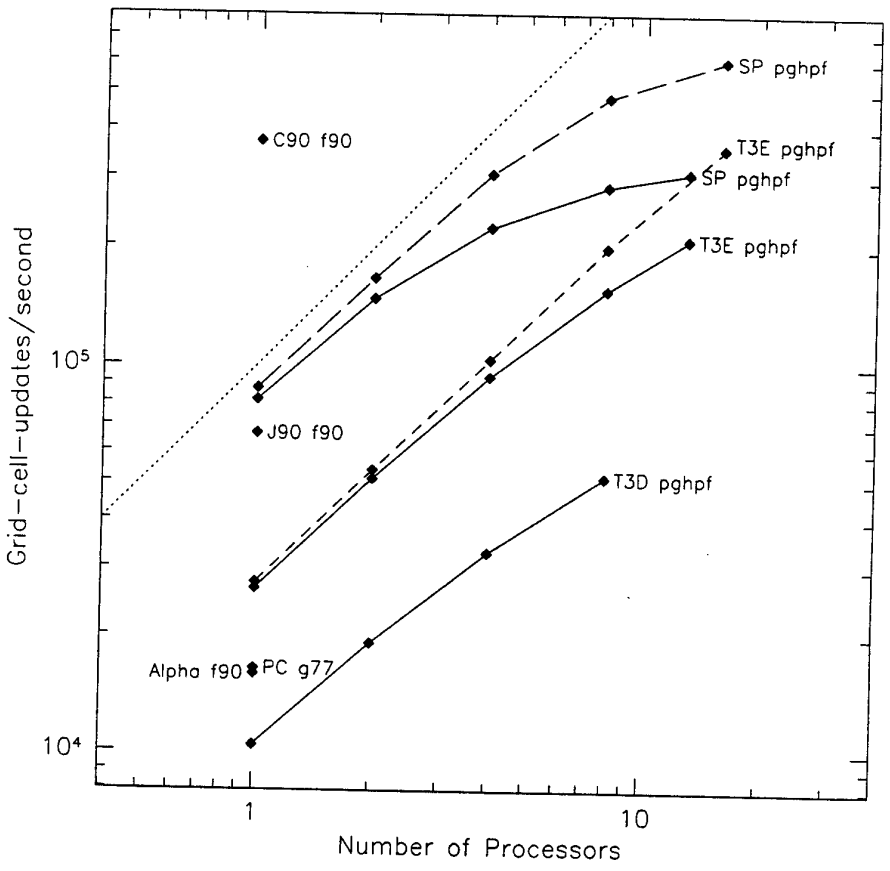


Fig. 1. Combined performance and scaling results for running the Versatile Advection Code on vector and parallel platforms. See text for details.

4 Simulating Rayleigh-Taylor instabilities

To demonstrate the advantages of having a versatile source code for simulating fluid flow, we consider what happens when a heavy compressible plasma is sitting on top of a lighter plasma in an external gravitational field. Such a situation is unstable as soon as the interface between the two is perturbed from perfect flatness. The instability is known as the Rayleigh-Taylor instability. Early analytic investigations date back to a comprehensive and detailed analysis given by Chandrasekhar [3].

The initial configuration is one where two layers of prescribed density ratio (dense to light ratio of $\rho_d/\rho_l = 10$) are left to evolve between two planes ($y = 0$ and $y = 1$), with gravity pointing downwards ($\mathbf{g} = -\hat{e}_y$ unit vector). The heavy plasma on the top is separated from the light plasma below it by the surface $y = y_0 + \epsilon \sin(k_x x) \sin(k_z z)$. Initially, both are at rest with $\mathbf{v} = 0$, and the thermal pressure is set according to the hydrostatic balance equation (centered differenced formula $dp/dy = -\rho$). Boundary conditions make top and bottom perfectly conducting solid walls, while the horizontal directions are periodic. We then exploit the options available in VAC to see how the evolution changes when going from two to three spatial dimensions, and what happens when magnetic fields are taken along. All calculations are done on a Cray J90, where we preprocess the code to Fortran 90 for single-node execution.

4.1 Two-dimensional simulations

Figure 2 shows the evolution of the density in two two-dimensional simulations without and with an initial horizontal magnetic field $\mathbf{B} = 0.1\hat{e}_x$. Both simulations are done on a uniform 100×100 square grid, and the parameters for the initial separating surface are $y_0 = 0.8$, $\epsilon = 0.05$, and $k_x = 2\pi$ (there is no z dependence in 2D). The data is readily analysed using IDL.

In both cases, the heavy plasma is redistributed in falling spikes or pillars, also termed Rayleigh-Taylor 'fingers', pushing the lighter plasma aside with pressure building up underneath the pillars. However, in the ideal MHD case, the frozen-in field lines are forced to move with the sinking material, so it gets wrapped around the pillars. The extra magnetic pressure and tension forces thereby confine the falling dense plasma and slow down the sinking and mixing process. In fact, since we took the initial displacement perpendicular to the horizontal magnetic field, we effectively maximized its stabilizing influence.

In [3], the linear phase of the Rayleigh-Taylor instability in both hydrodynamic and magnetohydrodynamic incompressible fluids is treated analytically. The stabilizing effect of the uniform horizontal magnetic field is evident from the expression of the growthrate n as a function of the wavenumber k_x

$$n^2 = g k_x \frac{\rho_d - \rho_l}{\rho_d + \rho_l} - \frac{B^2 k_x^2}{2\pi(\rho_d + \rho_l)}. \qquad (5)$$

Hence, while the shortest wavelength perturbations are the most unstable ones in hydrodynamics ($B = 0$), all wavelengths below a critical $\lambda_{crit} = B^2/g(\rho_d - \rho_l)$

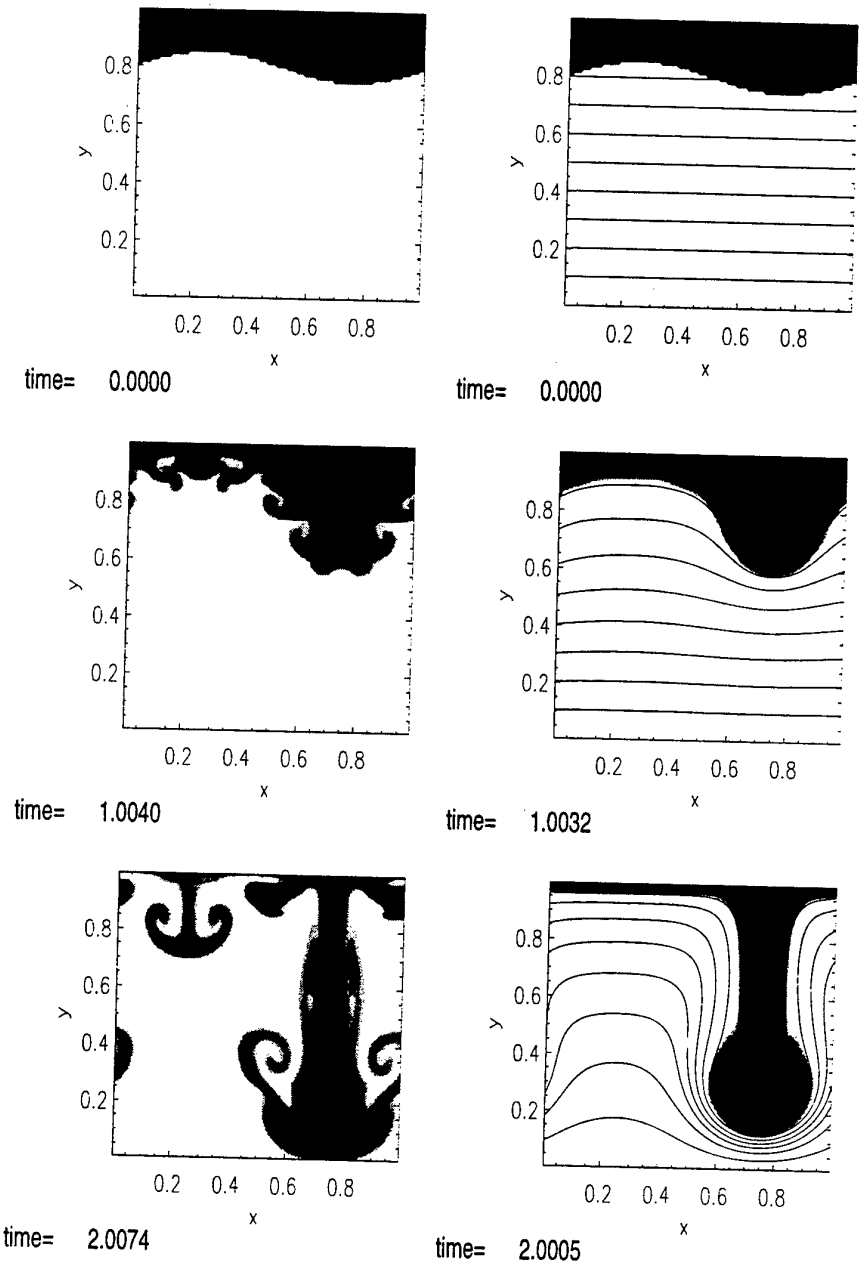


Fig. 2. Rayleigh-Taylor instability simulated in two spatial dimensions, in a hydrodynamic (left) and magnetohydrodynamic (right) case. The logarithm of the density and, in the magnetohydrodynamic case, also the magnetic field lines, are plotted.

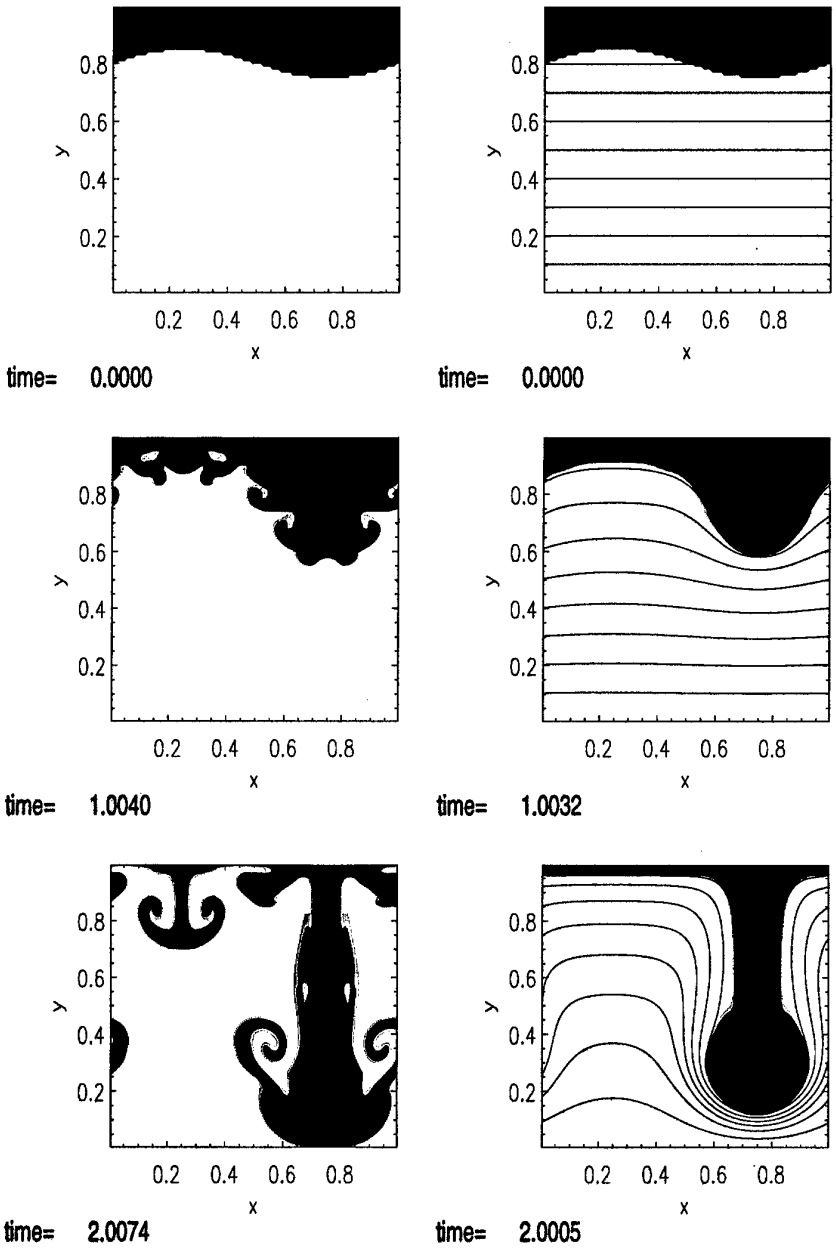


Fig. 2. Rayleigh-Taylor instability simulated in two spatial dimensions, in a hydrodynamic (left) and magnetohydrodynamic (right) case. The logarithm of the density and, in the magnetohydrodynamic case, also the magnetic field lines, are plotted.

are effectively suppressed by a horizontal magnetic field of strength B. Similarly, our initial perturbation with $\lambda = 2\pi/k_x = 1$ will be stabilized as soon as the magnetic field surpasses a critical field strength $B_{crit} = \sqrt{g\lambda(\rho_d - \rho_l)} \simeq 0.95$.

The simulations confirm and extend these analytic findings: the predicted growthrate can be checked (noting that our simulations are compressible), while the further non-linear evolution can be investigated. The discrete representation of the initial separating surface causes intricate small-scale structure to develop in the simulation at left of Figure 2. This is consistent with the fact that in a pure hydrodynamic case, the shortest wavelengths are the most unstable ones. Naturally, the simulation is influenced by numerical diffusion, while the periodic boundary conditions and the initial state select preferred wavenumbers. The suppression of short wavelength disturbances in the MHD case is immediately apparent, since no small-scale structure develops. The simulation at right has an initial plasma beta (ratio of gas to magnetic pressure forces) of about 400. For higher plasma beta yet, the MHD case will resemble the hydrodynamic simulation more closely, while a stronger magnetic field ($\mathbf{B} = \hat{e}_x$) suppresses the development of the instability entirely, as theory predicts.

Note also how the falling pillars develop a mushroom shape (left frames) as a result of another type of instability caused by the velocity shear across their edge: the Kelvin-Helmholtz instability. The lighter material is swept up in swirling patterns around the sinking spikes. In the MHD simulation (right frames) the Kelvin-Helmholtz instability does not develop due to the stabilizing effect of the magnetic field. Typically however, *both* instabilities play a crucial role in various astrophysical situations. Two dimensional MHD simulations of Rayleigh-Taylor instabilities in young supernova remnants [5] demonstrate this, and confirm the basic effects evident from Figure 2: magnetic fields get warped and amplified around the 'fingers'. General discussions of these and other hydrodynamic and magnetohydrodynamic instabilities are found in [3].

4.2 Three-dimensional simulations

In Figure 3, we present a snapshot of a hydrodynamical calculation in a 3D $50\times50\times50$ unit box, where the initial configuration has both $k_x = 2\pi$ and $k_z = 2\pi$. With gravity downwards, we look into the box from below. On two vertical cuts, we show at time $t = 2$ (i) the logarithm of the density in a color scale and (ii) the streamlines of the velocity field, colored according to the (logarithm of the) density. The cuts are chosen to intersect the initial separating surface between the heavy and the light plasma at its extremal positions where the motion is practically two-dimensional. 3D effects are readily identified by direct comparison with the two-dimensional hydrodynamic calculation. The time series of the 3D data set has been analysed using AVS (a video is made with AVS to demonstrate how density, pressure and velocity fields evolve during the mixing process).

Figure 4 shows the evolution of a three-dimensional MHD calculation at times $t = 1$ and $t = 2$. We show an isosurface of the density (at 1% above the initial value for ρ_d), colored according to the thermal pressure. A cutting plane also

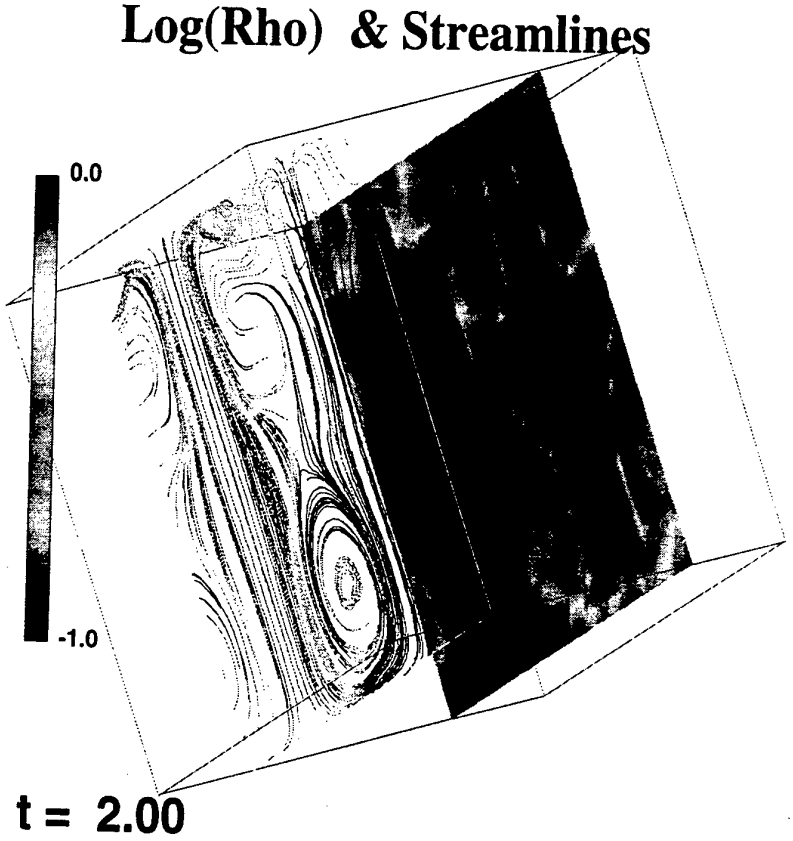


Fig. 3. Rayleigh-Taylor instability in 3D, purely hydrodynamic. We show streamlines (left) and density contours (right) in two vertical cutting planes.

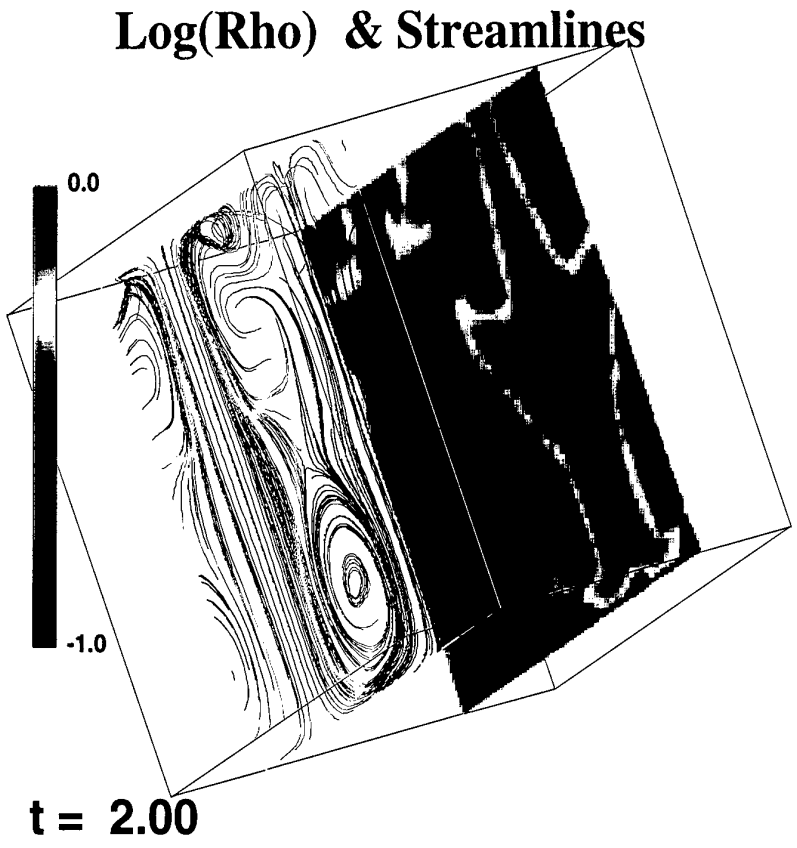


Fig. 3. Rayleigh-Taylor instability in 3D, purely hydrodynamic. We show streamlines (left) and density contours (right) in two vertical cutting planes.

shows the vertical stratification of the thermal pressure. Note the change in the initial configuration ($k_x = 6\pi$ and $k_z = 4\pi$, with $y_0 = 0.7$): more and narrower spikes are seen to grow and to split up. The AVS analysis of the full time series shows how droplets form at the tips of the falling pillars, which seem to expand horizontally to a critical size before continuing their fall. At the same time, the magnetic field gets wrapped around the falling pillars. Figure 4 nicely confirms that places where spikes branch into narrower ones correspond to places with excess pressure underneath. Similar studies of incompressible 3D ideal MHD cases are found in [6]. They confirm that strong tangential fields suppress the growth as expected from theoretical considerations, while the Rayleigh-Taylor instability acts to amplify magnetic fields locally. In such magnetic fluids, parameter regimes exist where secondary Kelvin-Helmholtz instabilities develop, just as in the hydrodynamic situation of Figure 3 (note the regions of strong vorticity in the streamlines).

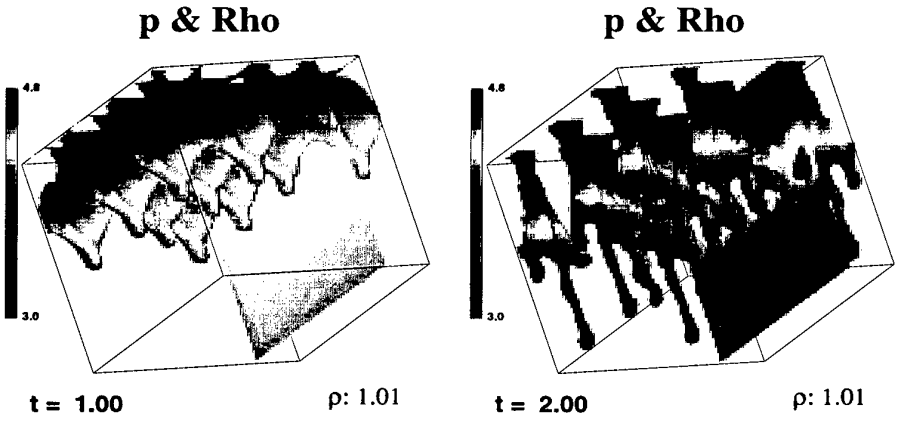


Fig. 4. 3D MHD Rayleigh-Taylor instability. At two consecutive times, an isosurface of the density is colored according to the thermal pressure. The thermal pressure is also shown in a vertical cut.

5 Conclusions

We have developed a powerful tool to simulate magnetized fluid dynamics. The Versatile Advection Code runs on many platforms, from PC's to supercomputers including distributed memory architectures. The rapidly maturing HPF compilers can yield scalable parallel performance for general fluid dynamical simulations. Clearly, the scaling and performance of VAC make high resolution 3D

simulations possible, and detailed investigations may broaden our insight in the intricate dynamics of magneto-fluids and plasmas.

We presented simulations of the Rayleigh-Taylor instability in two and three spatial dimensions, with and without magnetic fields. VAC allows one to do all these simulations with a single problem setup, since the equations to solve and the dimensionality of the problem is simply specified in a preprocessing phase. Data analysis can be done using a variety of data visualization packages, including IDL and AVS as demonstrated here. In the future, we plan to use VAC to investigate challenging astrophysical problems, like winds and jets emanating from stellar objects, magnetic loop dynamics, accretion onto black holes, etc.

Website info on the code is available at `http://www.fys.ruu.nl/~toth/` and at `http://www.fys.ruu.nl/~mpr/`. MPEG-animations of various test problems can also be found there.

Acknowledgements. The Versatile Advection Code was developed as part of the project on 'Parallel Computational Magneto-Fluid Dynamics', funded by the Dutch Science Foundation (NWO) Priority Program on Massively Parallel Computing, and coordinated by Prof. Dr. J.P. Goedbloed. Computer time on the CM-5, the Cray T3E and IBM SP machines was sponsored by the Dutch 'Stichting Nationale Computerfaciliteiten' (NCF). R.K. performed the simulations on the Cray T3D, J90, and Sun workstation cluster at the Edinburgh Parallel Computing Centre with support from the TRACS programme as part of his research at FOM. G.T. receives a postdoctoral fellowship (D 25519) from the Hungarian Science Foundation (OTKA), and is supported by the OTKA grant F 017313.

References

1. Biskamp, D.: Nonlinear Magnetohydrodynamics. Cambridge Monographs on Plasma Physics 1, Cambridge University Press, Cambridge (1993)
2. Bittencourt, J.A.: Fundamentals of Plasma Physics. Pergamon Press, Oxford (1986)
3. Chandrasekhar, S.: Hydrodynamic and Hydromagnetic stability. Oxford University Press, New York (1961)
4. Freidberg, J.P.: Ideal Magnetohydrodynamics. Plenum Press, New York (1987)
5. Jun, B.-I., Norman, M.L.: MHD simulations of Rayleigh-Taylor instability in young supernova remnants. Astrophys. and Space Science **233** (1995) 267-272
6. Jun, B.-I., Norman, M.L., Stone, J.M.: A numerical study of Rayleigh-Taylor instability in magnetic fluids. Astrophys. J. **453** (1995) 332-349
7. Keppens, R., Tóth, G., Botchev, M.A., van der Ploeg, A.: Implicit and semi-implicit schemes in the Versatile Advection Code: algorithms. Submitted for publication (1997)
8. van der Ploeg, A., Keppens, R., Tóth, G.: Block Incomplete LU-preconditioners for Implicit Solution of Advection Dominated Problems. In: Hertzberger, B., Sloot, P. (eds.): Proceedings of High Performance Computing and Networking Europe 1997. Lecture Notes in Computer Science, Vol. 1225. Springer-Verlag, Berlin Heidelberg New York (1997) 421-430
9. Tóth, G.: Preprocessor based implementation of the Versatile Advection Code for workstations, vector and parallel computers. These proceedings.

10. Tóth, G.: Versatile Advection Code. In: Hertzberger, B., Sloot, P. (eds.): Proceedings of High Performance Computing and Networking Europe 1997. Lecture Notes in Computer Science, Vol. 1225. Springer-Verlag, Berlin Heidelberg New York (1997) 253-262
11. Tóth, G.: A general code for modeling MHD flows on parallel computers: Versatile advection code. Astrophys. Lett. & Comm. **34** (1996) 245
12. Tóth, G.: The LASY Preprocessor and Its Application to general Multidimensional Codes. J. Comput. Phys. **138** (1997) 981-990
13. Tóth, G., Keppens, R.: Comparison of Different Computer Platforms for Running the Versatile Advection Code. Accepted for High Performance Computing and Networking (1998)
14. Tóth, G., Keppens, R., Botchev, M.A.: Implicit and semi-implicit schemes in the Versatile Advection Code: numerical tests. Accepted for publication in Astron. & Astrophys. (1997)
15. Tóth, G., Odstrčil, D.: Comparison of some Flux Corrected Transport and Total Variation Diminishing Numerical Schemes for Hydrodynamic and Magnetohydrodynamic Problems. J. Comput. Phys. **128** (1996) 82

Parallel Grid Manipulations in Earth Science Calculations

William Sawyer[1,2], Lawrence Takacs[1], Arlindo da Silva[1], and Peter Lyster[1,2]

[1] NASA Goddard Space Flight Center, Data Assimilation Office
Code 910.3, Greenbelt MD, 20771, USA
{sawyer, takacs, dasilva, lys}@dao.gsfc.nasa.gov
http://dao.gsfc.nasa.gov
[2] Department of Meteorology, University of Maryland at College Park
College Park MD, 20742-2425, USA
{sawyer, lys}@atmos.umd.edu

Abstract. We introduce the parallel grid manipulations needed in the Earth Science applications currently being implemented at the Data Assimilation Office (DAO) of the National Aeronautics and Space Administration (NASA). Due to real-time constraints the DAO software must run efficiently on parallel computers. Numerous grids, structured and unstructured are employed in the software.
The DAO has implemented the PILGRIM library to support multiple grids and the various grid transformations between them, e.g., interpolations, rotations, prolongations and restrictions. It allows grids to be distributed over an array of processing elements (PEs) and manipulated with high parallel efficiency. The design of PILGRIM closely follows the DAO's requirements, but it can support other applications which employ certain types of grids. New grid definitions can be written to support still others. Results illustrate that PILGRIM can solve grid manipulation problems efficiently on parallel platforms such as the Cray T3E.

1 Introduction

The need to discretize continuous models in order to solve scientific problems gives rise to finite *grids* — sets of points at which prognostic variables are sought. So prevalent is the use of grids in science that it is possible to forget that a computer-calculated solution is not the solution to the original problem but rather of a discretized representation of the original problem, and moreover is only an approximate solution, due to finite precision arithmetic. Grids are ubiquitous where analytical solutions to continuous problems are not obtainable, e.g., the solution of many differential equations.

Classically a structured grid is chosen a priori for a given problem. If the quality of the solution is not acceptable, then the grid is made finer, in order to better approximate the continuous problem.

For some time the practicality of *unstructured* grids has also been recognized. In such grids it is possible to cluster points in regions of the domain which require

higher resolution, while retaining coarse resolution in other parts of the domain. Unstructured grids are often employed in device simulation [1], computational fluid dynamics [2], and even in oceanographic models [3]. Although these grids are more difficult to lay out than structured grids, much research has been done in generating them automatically [4]. In addition, once the grid has been generated, there numerous methods and libraries are available to adaptively refine the mesh [5] to provide a more precise solution.

Furthermore, the advantages of multiple grids of varying resolutions for a given domain have been recognized. This is best known in the Multigrid technique [6] in which low frequency error components of the discrete solution are eliminated if values on a given grid are restricted to a coarser grid on which a smoother is applied. But multiple grids also find application other fields such as speeding up graph partitioning algorithms [7].

An additional level of complexity has arisen in the last few years: many contemporary scientific problems must be decomposed over an array of processing elements (or PEs) in order to obtain a solution in an expedient manner. Depending on the parallelization technique, not only the work load but also the grid itself may be distributed over the PEs, meaning that different parts of the data reside in completely different memory areas of the parallel machine. This makes the programming of such an application much more difficult for the developer.

The Goddard Earth Observing System (GEOS) Data Assimilation System (DAS) software currently being developed at the Data Assimilation Office (DAO) is no exception to the list of modern grid applications. GEOS DAS uses observational data with systematic and random errors and incomplete global coverage to estimate the complete, dynamic and constituent state of the global earth system. The GEOS DAS consists of two main components, an atmospheric General Circulation Model (GCM) [8] to predict the time evolution of the global earth system and a Physical-space Statistical Analysis Scheme (PSAS) [9] to periodically incorporate observational data.

At least three distinct grids are being employed in GEOS DAS: an *observation grid* — an unstructured grid of points where physical quantities measured by instruments or satellites are associated — a structured *geophysical grid* of points spanning the earth at uniform latitude and longitude locations where prognostic quantities are determined, and a block-structured *computational grid* which may be stretched in latitude and longitude. Each of these grids has a different structure and number of constituent points, but there are numerous interactions between them. Finally the GEOS DAS application is targeted for distributed memory architectures and employs a message-passing paradigm for the communication between PEs.

In this document we describe the design of PILGRIM (Fig. 1), a parallel library for grid manipulations, which fulfills the requirements of GEOS DAS. The design of PILGRIM is *object-oriented* [10] in the sense that it is modular, data is encapsulated in each design layer, operations can be overloaded, and different instantiations of grids can coexist simultaneously. The library is realized in Fortran 90, which allows the necessary software engineering techniques such as modules

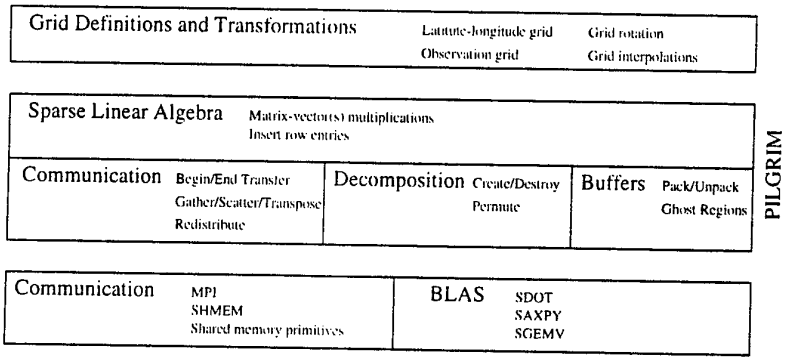


Fig. 1. PILGRIM assumes the existence of fundamental communication primitives such as the Message-Passing Interface (MPI) and optimized Basic Linear Algebra Subroutines (BLAS). PILGRIM's first layer contains routines for communication as well as for decomposing the domain and packing and unpacking sub-regions of the local domain. Above this is a sparse linear algebra layer which performs basic sparse matrix operations for grid transformations. Above PILGRIM, modules define and support different grids. Currently only the grids needed in GEOS DAS are implemented, but the further modules could be designed to support yet other grids.

and derived data types, while keeping in line with other Fortran developments at the DAO. The communication layer is implemented using MPI [11]; however, the communication interfaces defined in PILGRIM's primary layer could conceivably be implemented with other message-passing libraries such as PVM [12], or with other paradigms, e.g., Cray SHMEM [13] or with shared-memory primitives which are available on shared-memory machines like the SGI Origin or SUN Enterprise.

This document is structured in a bottom-up fashion. Reasonable design assumptions are made in Sect. 2 in order to ease the implementation. The layer for communication, decompositions, and buffer packaging is discussed in Sect. 3. The sparse linear algebra layer is specified in Sect. 4. The plug-in grid modules are defined in Sect. 5 to the degree necessary to meet the requirements of GEOS DAS. In Sect. 6 some examples and prototype benchmarks are presented for the interaction of all the components. Finally we summarize our work in Sect. 7.

2 Design Assumptions

A literature search was the first step taken in the PILGRIM design process in order to find public domain libraries which might be sufficient for the DAO's requirements [14]. Surprisingly, none of the common parallel libraries for the solution of sparse matrix problems, e.g., PETSc [15], Aztec [16], PLUMP [17], et al., was sufficient for our purposes. These libraries all try to make the parallel implementation transparent to the application. In particular, the application is not supposed to know how the data are actually distributed over the PEs.

This trend in libraries is not universally applicable for the simple reason that if an application is to be parallelized, the developers generally have a good idea of how the underlying data should be distributed and manipulated. Experience has shown us that hiding complexity often leads to poor performance, and the developer often resorts to workarounds to make the system perform in the manner she or he envisions. If the developer of a parallel program is capable of deciding on the proper data distribution and manipulation of local data, then those decisions need to be supported.

In order to minimize the scope of PILGRIM, other simplifying assumptions were made about the way the library will be used.

1. The local portion of the distributed grid array is assumed to be a contiguous section of memory. The local array can have any rank, but if the rank is greater than one, the developer must assure that no gaps are introduced into the actual data representation, for example, by packing it into a 1-D array if necessary.
2. Grid transformations are assumed to be *sparse*, i.e., each of the values on one grid is determined from a linear combination of only a few values from the other grid. The linear transformation corresponds to a sparse matrix with a predictable number of non-zero entries per row. This assumption is realistic for the localized interpolations used in GEOS DAS.
3. At a high level, the application can access data through global indices, i.e., the indices of the original undistributed problem. However, at the level where most computation is performed, the application needs to work with local indices (ranging from one to the total number of entries in the local contiguous array). The information to perform global-to-local and local-to-global mappings must be contained in the data structure defining the grid. However, it is assumed that these mappings are seldom performed, e.g., at the beginning and end of execution, and these mappings need not be efficient.
4. All decomposition-related information is replicated on all PEs.

These assumptions are significant. The first avoids the introduction of an opaque type for data and allows the application to manipulate the local data as it sees fit. The fact that the data are contained in a simple data structure generally allows higher performance than an implementation which buries the data inside a derived type. The second assumption ensures that the matrix transformation are not memory limited. The third implies that most of the calculation is performed on the data in a *local* fashion. In GEOS DAS it is fairly straightforward to run in this mode; however, it might not be the case in other applications. The last assumption assures that every PE knows about the entire data decomposition.

3 Communication and Decomposition Utilities

In this layer communication routines are isolated, and basic functionality is provided for defining and using data decompositions as well as for moving sections of data arrays to and from buffers.

The operations on data decompositions are embedded in a Fortran 90 module which also supplies a generic `DecompType` to describe a decomposition. Any instance of `DecompType` is replicated on all PEs such that every PE has access to information about the entire decomposition. The decomposition utilities consist of the following:

DecompRegular1d	Create a 1-D blockwise data decomposition
DecompRegular2d	Create a 2-D block-block data decomposition
DecompRegular3d	Create a 3-D block-block-block data decomposition
DecompIrregular	Create an irregular data decomposition
DecompCopy	Create new decomposition with contents of another
DecompPermute	Permute PE assignment in a given decomposition
DecompFree	Free a decomposition and the related memory
DecompGlobalToLocal1d	Map global 1-D index to local (pe,index)
DecompGlobalToLocal2d	Map global 2-D index to local (pe,index)
DecompLocalToGlobal1d	Map local (pe,index) to global 1-D index
DecompLocalToGlobal1d	Map local (pe,index) to global 2-D index

Using the Fortran 90 overloading feature, the routines which create new decompositions are denoted by `DecompCreate`. Similarly, the 1-D and 2-D global-to-local and local-to-global mappings are denoted by `DecompGlobalToLocal` and `DecompLocalToGlobal`, resulting in a total of five fundamental operations.

Communication primitives are confined to this layer because it may be necessary at some point to implement them with a message-passing library other than MPI such as PVM or SHMEM, or even with shared-memory primitives such as those on the SGI Origin (the principle platform at the DAO). Thus it is wise to encapsulate all message-passing into one Fortran 90 module. For brevity, only the overloaded functionality is presented:

ParInit	Initialize the parallel code segment
ParExit	Exit from the parallel code segment
ParSplit	Split parallel code segment into two groups
ParMerge	Merge two code segments
ParScatter	Scatter global array to given data decomposition
ParGather	Gather from data decomposition to global array
ParBeginTransfer	Begin asynchronous data transfer
ParEndTransfer	End asynchronous data transfer
ParExchangeVector	Transpose block-distributed vector over all PEs
ParRedistribute	Redistribute one data decomposition to another

In order to perform calculations locally on a given PE it is often necessary to "ghost" adjacent regions, that is, send boundary regions of the local domain to adjacent PEs. To this end a module has been constructed to move ghost regions to and from buffers. The buffers can be transferred to other PEs with the communication primitives such as `ParBeginTransfer` and `ParEndTransfer`. Currently the buffer module contains the following non-overloaded functionality:

BufferPackGhost2dReal	Pack a 2-D array sub-region into buffer
BufferUnpackGhost2dReal	Unpack buffer into 2-D array sub-region
BufferPackGhost3dReal	Pack a 3-D array sub-region into buffer
BufferUnpackGhost3dReal	Unpack buffer into 3-D array sub-region
BufferPackSparseReal	Pack specified entries of vector into buffer
BufferUnpackSparseReal	Unpack buffer into specified entries of vector

In this module, as in most others, the local coordinate indices are used instead of global indices. Clearly this puts responsibility on the developer to keep track of the indices which correspond to the ghost regions. In GEOS DAS this turns out to be fairly straightforward.

4 Sparse Linear Algebra

The concept of transforming one grid to another involves interpolating the values defined on one grid at grid-points on another. These values are stored as contiguous vectors with a given length, $1 \ldots N_{local}$, and distribution defined by the grid decomposition (although the vector might actually represent a multi-dimensional array at a higher level). Thus the sparse linear algebra layer fundamentally consists of a facility to perform linear transformations on distributed vectors.

As in other parallel sparse linear algebra packages, e.g., PETSc [15] and Aztec [16], the linear transformation is stored in a distributed sparse matrix format. Unlike those libraries, however, local indices are used when referring to individual matrix entries, although the mapping `DecompGlobalToLocal` can be used to translate from global to local indices. In addition, the application of the linear transformation is a matrix-vector multiplication where the matrix is not necessarily square, and the resulting vector may be distributed differently than the original.

There are many approaches to storing distributed sparse matrices and performing a the matrix-vector product. PILGRIM uses a format similar to that described in [17], which is optimal if the number of non-zero entries per row is constant.

Assumption 3 in Sect. 2 implies that the matrix definition is not time-consuming. In GEOS DAS the template of any given interpolation is initialized once, but the interpolation itself is performed repeatedly. Thus relatively little attention has been paid to the optimization of the matrix creation and definition. The basic operations for creating and storing matrix entries are:

SparseMatCreate	Create a sparse matrix
SparseMatDestroy	Destroy a sparse matrix
SparseInsertEntries	Insert entries replicated on all PEs
SparseInsertLocalEntries	Insert entries of local PE

Two scenarios for inserting entries are supported. In the first scenario, every PE inserts all matrix entries. Thus every argument of the corresponding routine,

`SparseInsertEntries`, is replicated. The local PE picks up only the data which it needs, leaving other data to the appropriate PEs. This scenario is the easiest to program if the sequential code version is used as the code base.

In the second scenario the domain is partitioned over the PEs, meaning that each PE is responsible for a disjoint subset of the matrix entries, and the matrix generation is performed in parallel. Clearly this is the more efficient scenario. The corresponding routine, `SparseInsertLocalEntries` assumes that no two PEs try to add the same matrix entry. However, it does not assume that the all matrix entries reside on the local PE, and it will perform the necessary communication to put the matrix entries in their correct locations.

The efficient application of the matrix to a vector or group of vectors is crucial to the overall performance of GEOS DAS, since the linear transformations are performed continually on assimilation runs for days or weeks at a time. The most common transformation is between three-dimensional arrays of two different grids which describe global atmospheric quantities such as wind velocity or temperature. One 3-D array might be correspond to the geophysical grid which covers the globe, while another might be the computational grid which is more appropriate for the dynamical calculation. The explicit description of such a 3-D transformation might be prohibitive in terms of memory. But fortunately, this transformation only has dependencies in two of the three dimensions as it acts on 2-D horizontal cross-sections independently.

To fulfill the assumptions in Sect. 2, a 2-D array is considered a vector x. Using this representation the transformations become parallel matrix-vector multiplications, which can be performed with one of the following two operations:

SparseMatVecMult	Perform $y \leftarrow \alpha Ax + \beta y$
SparseMatTransVecMult	Perform $y \leftarrow \alpha A^T x + \beta y$

In order to transform several arrays simultaneously, the arrays are grouped into multiple vectors, that is, into a $n \times m$ matrix where n is the length of the vector (number of values in the 2-D array), and m is the number of vectors. The following matrix-matrix and matrix-transpose-matrix multiplications can group messages in such a way as to drastically minimize latencies and utilize BLAS-2 operations instead of BLAS-1:

SparseMatMatMult	Perform $Y \leftarrow \alpha AX + \beta Y$
SparseMatTransMatMult	Perform $Y \leftarrow \alpha A^T X + \beta Y$

The distributed representation of the matrix contains, in addition to the matrix information itself, space for the *communication pattern*. Upon entering any one of the four matrix operations, the the matrix is checked for new entries which may have been added since its last application. If the matrix has been modified, the operation first generates the communication pattern — an optimal map of the information which has to be exchanged between PEs — before performing the matrix multiplication. This is a fairly expensive operation, but in GEOS DAS it only needs to be done once when the matrix is first defined. Subsequently, the

matrix multiplication can be performed repetitively in the most efficient manner possible.

5 Supported Grids

The *grid data structure* describes a set of *grid-points* and their decomposition over a group of PEs as well as other information, such as the size of the domain. The grid data structure itself does not contain actual data and can be replicated on all PEs due to its minimal memory requirements. The data reside in arrays distributed over the PEs and given meaning by the information in the grid data structure. There is no limitation on how the application accesses and manipulates the local data arrays. Two types of grids employed in GEOS DAS are described here, but others are conceivable and could be supported by PILGRIM without modifications to the library.

The latitude-longitude grid defines a *lat-lon* coordinate system — a regular grid spanning the earth with all points in one row having a given latitude and all points in a column a given longitude. The grid encompasses the entire earth from $-\pi$ to π longitudinally and from $-\pi/2$ to $\pi/2$ in latitude.

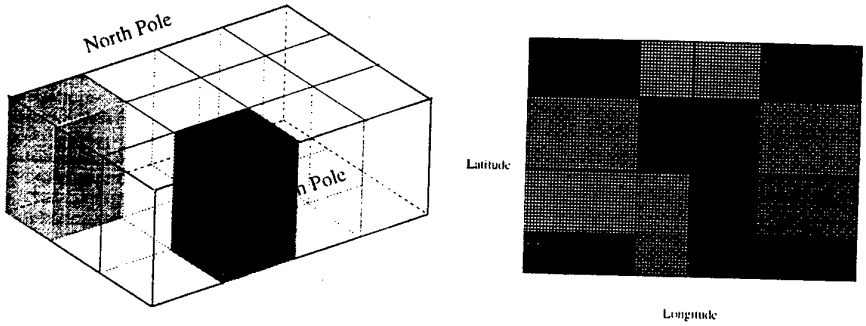


Fig. 2. GEOS DAS uses a column decomposition of data (left), also termed a "checkerboard" decomposition due to the distribution of any given horizontal cross-section (right). The width and breadth of a column can be variable, although generally an approximately equal number of points are assigned to every PE.

The decomposition of this grid is a "checkerboard" (Fig. 2), because the underlying three-dimensional data array comprises all levels of the column designated by the 2-D decomposition of the horizontal cross-section. This decomposition can contain a variable-sized rectangle of points — it is not necessary for each PE to be assigned an equal number — and thus some freedom for load balancing exists. The basic data structure for the lat-lon grid is defined below in its Fortran 90 syntax:

```
TYPE LatLonGridType
  TYPE (DecompType)    :: Decomp      ! Decomposition
  INTEGER              :: ImGlobal    ! Global Size in X
  INTEGER              :: JmGlobal    ! Global Size in Y
  REAL                 :: Tilt        ! Tilt of remapped NP
  REAL                 :: Rotation    ! Rotation of remapped NP
  REAL                 :: Precession  ! Precession of remapped NP
  REAL,POINTER         :: dLat(:)     ! Latitudes
  REAL,POINTER         :: dLon(:)     ! Longitudes
END TYPE LatLonGridType
```

This grid suffices to describe both the GEOS DAS computational grid used for dynamical calculations and the geophysical grid in which the prognostic variables are sought. The former makes use of the parameters **Tilt**, **Rotation** and **Precession** to describe its view of the earth (Fig. 3), and the **dLat** and **dLon** grid box sizes to describe the grid stretching. The latter is defined by the normal geophysical values for **Tilt**, **Rotation** and **Precession** $= (\frac{\pi}{2}, 0, 0)$ and uniform **dLat** and **dLon**.

The observation grid data structure describes observation points over the globe, as described by their lat-lon coordinates. In contrast to the lat-lon grid, the point grid decomposition is inherently *one-dimensional* since there no structure to the grid.

```
TYPE ObsGridType
  TYPE (DecompType)    :: Decomp         ! Decomposition
  INTEGER              :: Nobservations  ! Total points
END TYPE ObsGridType
```

The data corresponding to this grid data structure is a set of vectors, one for the observation values and several for *attributes* of those values, such as the latitude, longitude and level at which an observation was taken.

6 Results

An example of a non-trivial transformation employed in atmospheric science applications is grid rotation [18]. Computational instabilities from finite difference schemes can arise in the polar regions of the geophysical grid when a strong cross-polar flow occurs. By placing the pole of the computational grid to the geographic equator, however, the instability near the geographic pole is removed due to the vanishing Coriolis term.

It is generally accepted that the physical processes such as those related to long- and short-wave radiation can be calculated directly on the geophysical grid. Dynamics, where the numerical instability occurs, needs to be calculated on the computational grid. An additional refinement involves calculating the dynamics on a rotated *stretched* grid, in which the grid-points are not uniform in latitude and longitude. The **LatLonGridType** allows for both variable lat-lon coordinates as well as the description of any lat-lon view of the world where the poles are

assigned to a new geographical location. The grid rotation (without stretching) is depicted in Fig. 3.

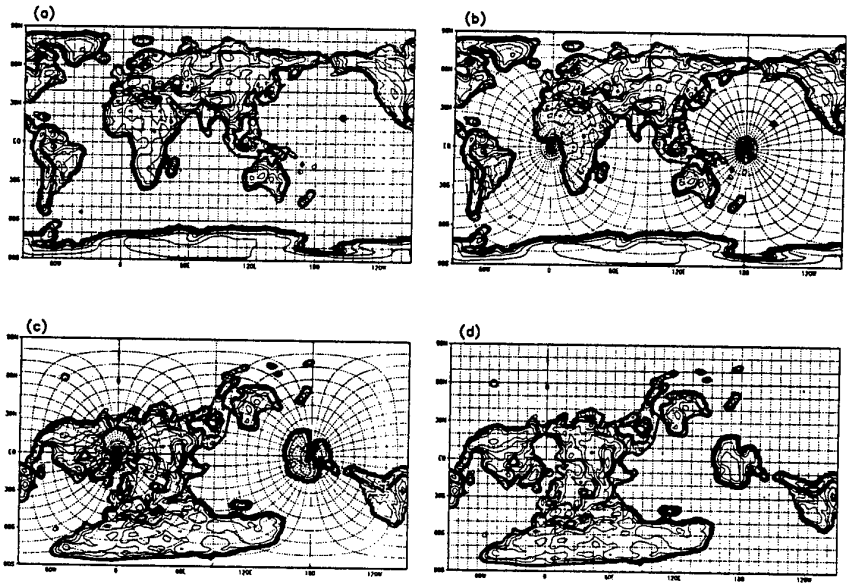


Fig. 3. The use of the latitude-longitude grid (a) and (c) as the computational grid results in instabilities at the poles due to the Coriolis term. The instabilities vanish with on a grid (b) where the pole has been rotated to the equator. The computational grid is therefore a lat-lon grid (d) where the "poles" on the top and bottom are in the Pacific and Atlantic Oceans, respectively.

It would be natural to use the same decomposition for both the geophysical and computational grids. It turns out, however, that this approach disturbs data locality inherent to this transformation (Fig. 4). If the application could have unlimited freedom to choose the decomposition of the computational grid, the forward and reverse grid rotations could exhibit excellent data locality, and the matrix application would be much more efficient.[1] Unfortunately, practicality limits the decomposition of both the geophysical and computational grids to be a checkerboard decomposition.

However, there are still several degrees of freedom in the decomposition, namely the number of points on each PE and the assignment of local regions to PEs. While an approximately uniform number of points per PE is generally best for the dynamics calculation, the assignment of PEs is arbitrary. The following optimization is therefore applied: the potential communication pattern of a naive

[1] A simply connected region in one domain will map to at most two simply connected regions in the other.

Unpermuted Communication Matrix

0	219	5905	172	0	97	507	12
53	5731	690	2	1	303	132	0
3	477	136	0	53	5727	516	0
0	97	335	4	3	366	5942	165
516	0	53	5727	136	0	3	477
5967	166	2	341	371	4	0	61
543	12	0	61	5941	172	0	183
133	0	1	302	760	1	54	5661

Permuted Communication Matrix

5967	166	2	341	371	4	0	61
53	5731	690	2	1	303	132	0
0	219	5905	172	0	97	507	12
516	0	53	5727	136	0	3	477
543	12	0	61	5941	172	0	183
3	477	136	0	53	5727	516	0
0	97	335	4	3	366	5942	165
133	0	1	302	760	1	54	5661

Fig. 4. The above matrices represent the number of vector entries requested by a PE (column index) from another PE (row index) to perform a grid rotation for one 72×48 horizontal plane (i.e., one matrix-vector multiplication) on a total of eight PEs. The unpermuted communication matrix reflects the naive use of the geophysical grid decomposition and PE assignment for the computational grid. The permuted communication matrix uses the same decomposition, except the assignment of local regions to PEs is permuted. The diagonal entries denote data local to the PE and represent work which can be overlapped with the asynchronous communication involved in fetching the non-local data. The diagonal dominance of the communication matrix on the right translates into a considerable performance improvement.

computational grid decomposition is analyzed by adopting the decomposition of the geophysical grid. With a heuristic method, this analysis leads to a *permutation* of PEs for the computational grid which reduces communication (Fig. 4). The decomposition of the computational grid is then defined as a permuted version of the geophysical grid. Only then is the grid rotation matrix defined. An outline of the code is as given in Algorithm 1.

Algorithm 1 (Optimized Grid Rotation) *Given the geophysical grid decomposition, find a permutation of the PEs which will maximize the data locality of the geophysical-to-computational grid transformation, create and permute the computation grid decomposition, and define the transformation in both directions.*

```
SparseMatrixCreate( ..., GeoToComp )
SparseMatrixCreate( ..., CompToGeo )
DecompCreate( ..., GeoPhysDecomp )
LatLonCreate( GeoPhysDecomp, ...., GeoPhysGrid )
AnalyzeGridTransform( GeoPhysDecomp, ...., Permutation )
DecompCopy( GeoPhysDecomp, CompDecomp )
DecompPermute( Permutation, CompDecomp )
LatLonCreate( CompDecomp, ...., CompGrid )
GridTransform( GeoPhysGrid, CompGrid, GeoToComp )
GridTransform( CompGrid, GeoPhysGrid, CompToGeo )
```

In `GridTransform` the coordinates of one lat-lon grid are mapped to another. Interpolation coefficients are determined by the proximity of rotated grid-points to grid-points on the other grid (Fig. 3). Various interpolation schemes can be employed including bi-linear or bi-cubic; the latter is employed in GEOS DAS.

The transformation matrix can be completely defined by the two grids — the values on those grids are not necessary.

Once the transformation matrix is defined, sets of grid values, such as individual levels or planes of atmospheric data, can be transformed ad infinitum using a matrix-vector multiplication.

```
DO L = 1, GLOBAL_Z
  CALL SparseMatVecMult(GeoToComp, 1.0, In(1,1,L), 0.0, Out1(1,1,L))
END DO
```

Alternatively, it the transformation of the entire 3-D data set can be performed with one matrix-matrix product:

```
CALL SparseMatMatMult( GeoToComp, GLOBAL_Z, 1.0, In, 0.0, Out2 )
```

Note that the pole rotation is trivial (embarrassingly parallel) if any given plane resides entirely on one PE, i.e., if the 3-D array is decomposed in the z-dimension. Unfortunately, there are compelling reasons to distribute the data in vertical columns with the checkerboard decomposition.

Fig. 5 compares the performance of the unpermuted rotation with that of the permuted rotation on the Cray T3E. A further optimization is performed by replacing the non-blocking MPI primitives used in `ParBeginTransform` by faster Cray SHMEM primitives. The result of these optimizations is the improvement in scalability from tens of PEs to hundreds of PEs. The absolute performance in GFlop/s is presented in Fig. 6.

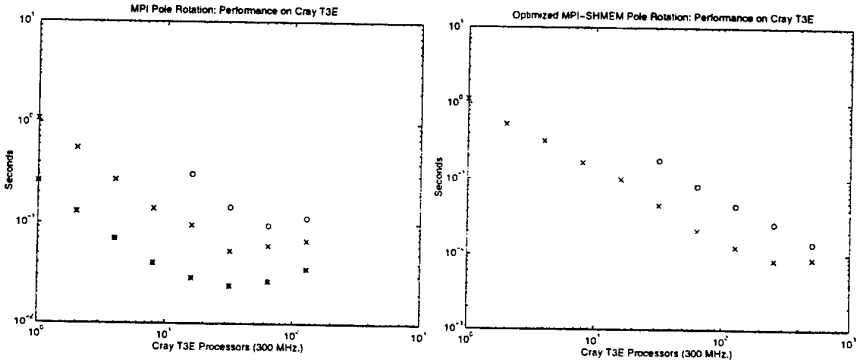


Fig. 5. With a naive decomposition of both the geophysical and computational grids and a straightforward MPI implementation, the performances at the left for the $72 \times 46 \times 70$ (*), $144 \times 91 \times 70$ (x), and $288 \times 181 \times 70$ (o) resolutions yield good scalability only to 10-50 processors. The optimized MPI-SHMEM hybrid version on the right scales to nearly the entire extent of the machine (512 processors).

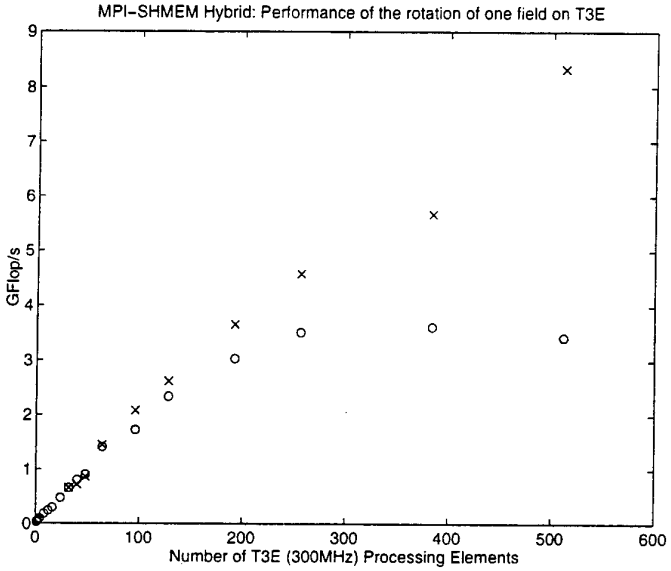


Fig. 6. The GFlop/s performances of the grid rotation on grids with $144 \times 91 \times 70$ (o), and $288 \times 181 \times 70$ (x) resolutions is depicted. These results are an indication that the grid rotation will not represent a bottleneck for the overall GEOS DAS system.

7 Summary

We have introduced the parallel grid manipulations needed by GEOS DAS and the PILGRIM library to support them. PILGRIM is modular and extensible, allowing us to support various types of grid manipulations. Results from the grid rotation problem were presented, indicating scalable performance on state-of-the-art parallel computers with a large number (> 100) of processors.

We are hoping to extend the usage of PILGRIM in GEOS DAS to the interface between the forecast model and the statistical analysis, to perform further optimizations on the library, and to offer the library to the public domain.

Acknowledgments

We would like to thank Jay Larson, Rob Lucchesi, Max Suarez, and Dan Schaffer for their valuable suggestions. The work of Will Sawyer and Peter Lyster at the Data Assimilation Office was funded by the High Performance Computing and Communications Initiative (HPCC) Earth and Space Science (ESS) program.

References

[1] G. Heiser, C. Pommerell, J. Weis, and W. Fichtner. Three dimensional numerical semiconductor device simulation: Algorithms, architectures, results. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(10):1218–1230, 1991.

[2] A. Ecer, J. Hauser, P. Leca, and J. Périaux. *Parallel Computational Fluid Dynamics*. Elsevier Science Publishers B.V. (North–Holland), Amsterdam, 1995.

[3] H.-P. Kersken, B. Fritzsch, O. Schenk, W. Hiller, J. Behrens, and E. Kraube. Parallelization of large scale ocean models by data decomposition. *Lecture Notes in Computer Science*, 796:323–336, 1994.

[4] P. Knupp and S. Steinberg. *Fundamentals of Grid Generation*. CRC Press, Boca Raton, FL, 1994.

[5] M. T. Jones and P. E. Plassmann. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. In IEEE, editor, *Proceedings of the Scalable High-Performance Computing Conference, May 23–25, 1994, Knoxville, Tennessee*, pages 478–485. IEEE Computer Society Press, 1994.

[6] W. L. Briggs. *A Multigrid Tutorial*. SIAM, 1987.

[7] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. Technical Report RNR-092-033, NASA Ames Research Center, 1992.

[8] L. L. Takacs, A. Molod, and T. Wang. Documentation of the Goddard Earth Observing System (GEOS) General Circulation Model — Version 1. NASA Technical Memorandum 104606, NASA, 1994.

[9] A. da Silva and J. Guo. Documentation of the Physical-space Statistical Analysis System (PSAS). Part 1: The Conjugate Gradient Solver, Version PSAS 1.00. DAO Office Note 96-02, Data Assimilation Office, NASA, 1996.

[10] T. Budd. *Object-Oriented Programming*. Addison-Wesley, New York, N.Y., 1991.

[11] MPIF (Message Passing Interface Forum). MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3&4):157–416, 1994.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[13] Cray Research. CRAY T3E Applications Programming, 1997, 1997.

[14] DAO Staff. GEOS-3 Primary System Requirements Document. Internal document, available on request., 1996.

[15] L. C. McInnes and B. F. Smith. Petsc 2.0: A case study of using mpi to develop numerical software libraries. In *1995 MPI Developers' Conference*, 1995.

[16] S. A. Hutchinson, J. A. Shadid, and R.S. Tuminaro. *The Aztec User's Guide - Version 1.0*, 1995.

[17] O. Bröker, V. Deshpande, P. Messmer, and W. Sawyer. Parallel library for unstructured mesh problems. Tech. Report CSCS-TR-96-15, Centro Svizzero di Calcolo Scientifico, 1996.

[18] M. J. Suarez and L. L. Takacs. Documentation of the ARIES/GEOS Dynamical Core: Version 2. NASA Technical Memorandum 104606, NASA, 1995.

Molecular Dynamics as a Natural Solver

W. Dzwinel[1], J.Kitowski[1,2], J.Mościński[1,2], and D. Yuen[3]

[1]Institute of Computer Science AGH, Al. Mickiewicza 30, 30-059 Kraków, Poland
[2] ACK CYFRONET, ul. Nawojki 11, 30-950 Kraków, Poland
[3]Minnesota Supercomputing Institute, University of Minnesota, Minneapolis, USA

Abstract. A universal character of molecular dynamics (MD) method is discussed. Contrary to the classical area of MD applications in microscopic world investigations, MD simulation of mesoscopic phenomena is considered. Sample results of MD simulations of the Rayleigh-Taylor instability are shown and discussed briefly. To cover the larger time-and-space scale either simplified MD model or more sophisticated particle based algorithms can be used. In the first case MD method can be directly applied as a predictive display in computer animation. In the second, MD code can be a "backbone" of efficient computer realization of such particle based methods as dissipative particle dynamics and smoothed particle hydrodynamics. Applications of MD approach in global optimization problems are discussed also. It is emphasized that inherent parallelism of MD method resulting in efficient realization on MPP systems together with its universal properties makes the method a powerful natural solver.

1 Introduction

According to physics, particles interact one with another through exchange of virtual objects, e.g., photons in electromagnetics. Changes in physical states of particles, i.e., their positions, momenta, spins *etc.* result from their interactions. This atomistic approach reflects an important principle of nature and human logic, i.e., construction of complex models from simple elements and rules via their mutual "interactions", or in other terms, information exchange.

Virtual particle (VIP) [1,2] is a base element of the particle based computational model. VIP can be defined on different levels of abstraction [2] e.g. as: atom, particle, cluster of particles, vehicle-target-obstacle, genotype, multidimensional point, UNIX process, single processor, *etc.* For example, taking into account that UNIX processes can "interact" via sending and receiving messages we can think about direct transformation of the VIP model into the message-passing model of parallel computations. This involves the change of the the VIP level of abstraction from the particles to the processes exchanging messages. It is relatively easy, due to flexibility of VIP model and its self-consistency.

The main suggestion put forward in [1,2] consists in the elaboration of a new strategy of parallel realization of an application using two stages of mapping (see Fig.1). At first, a problem is transformed into one of the natural solvers (or their hybrid) and virtual particles are defined. Then the method is realized on a multicomputer system through the transformation of virtual particles onto a virtual parallel machine model [1]. Several widely used natural solvers such as: Boltzmann lattice gas, lattice gas, simulated annealing, direct Monte-Carlo, cellular automata,

genetic algorithms, neural networks and others, having more limited scope of use such as: diffusion limited aggregation (DLA), percolation *etc.*, can be treated as particles based techniques in accordance with the definition presented in [2]. All these techniques, have been used in physics, chemistry and biology for many years. Therefore, the second stage of mapping (i.e., its implementation on a multiprocessor architecture) often allows us to exploit ready to use parallel algorithms or at least existing knowledge about the ways of parallelization of the particle based methods. In the authors opinion, successful mapping of a problem into a solver is crucial. This sort of mapping needs a creative and abstract way of thinking impossible to mimic by current and future generations of computer systems.

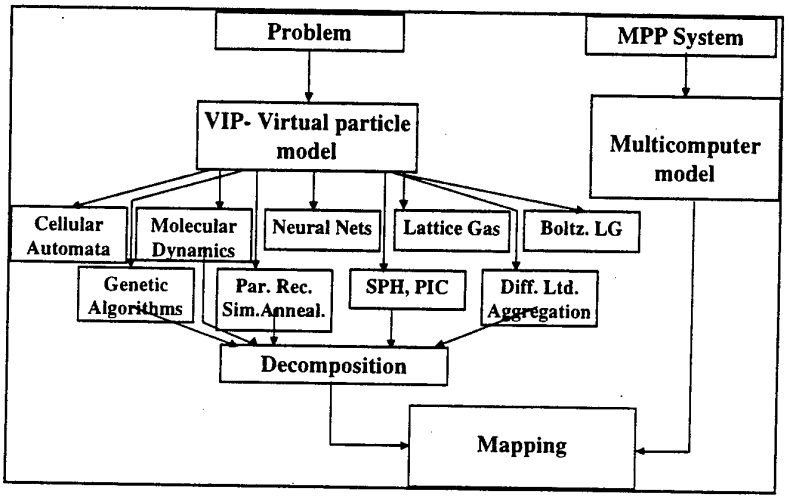


Fig.1. Problem mapping onto multiprocessor model through its transformation into a natural solver [1].

Molecular dynamics method (MD) (a well known technique of computational physics and one of the Grand Challenges of Science [3] problems) can be taken as a pure particle paradigm. The goal of this paper is to show that MD can be treated as a natural solver, i.e., a universal paradigm, which principles come from nature and which can be used as a solver in various fields of science and engineering. MD and other natural solvers like: simulated annealing, genetic algorithms, neural networks, cellular automata, *etc.*, due to their inherent parallelism, constitute the class of powerful computational tools when empowered by a parallel system. Increasing interest in implementation of these techniques on multiprocessor systems constitutes the natural consequence of this property.

At the beginning of the paper the mathematical background and computer realization of MD method are discussed briefly. Then sample results of MD applications in large-scale computational experiments concerning investigations of Rayleigh-Taylor instability are presented. In the following section it is shown that simplified computer realization of the MD method can be used as an efficient

animation technique based on the principal physical laws. Since the visual impression of movement plays the principal role in animation, physical details can be hidden from the observer and then substantially simplified. Other advantages of MD applications for computer animation are also discussed. The role of simulation using particles as a new technique of global minimum search is introduced. The visual clustering problem is considered as an example. Based on the results, conclusions are formulated at the end of paper.

2 MD principles

Molecular dynamics is a computational technique, widely used in physics, chemistry and biology for almost 35 years (e.g. [4]). Its basic principles are shown in Fig.2.

Each particle i interacts with all others located in sphere with R_{cut} radius according to potential energy of interactions. In the simplest case two body pair radial potential function $\phi(r_{ij.})$ depends on the distance r_{ij} between the particles. For more complex molecules, the potential function can be more sophisticated. Let the pair force $\mathbf{f}_{ij} = -\nabla\phi\,(r_{ij.})$, while the total force $\mathbf{F}_i$, which acts on a single particle i, is the sum of pair forces $\mathbf{f}_{ij}$ of its neighbour particles within R_{cut} sphere.

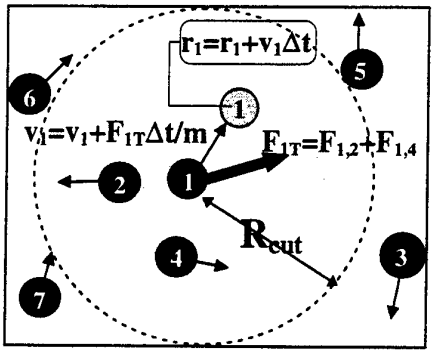


Fig.2. Basic principles of MD paradigm.

Time evolution of particles, i=1,...,M., is defined by the Newtonian equations of motion:.

$$m_i \frac{d\mathbf{v}_i}{dt} = \sum_{j \in S(i,Rcut)} \mathbf{f}_{ij}\,, \qquad \frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i \tag{1}$$

where: $\mathbf{v}_i$ and $\mathbf{r}_i$ - represent velocity and coordinates of particle i, respectively. The computer implementation of MD techniques consists of subsequent calculation of forces and particle movements for each time step.

A set of simulated particles is confined (in the most cases) in a rectangular box with periodic boundary conditions (PBC) implied. This assumption is important to obtain valuable simulation results. The number of particles, M, is limited by the computational power of computers (M=10^9 on the fastest parallel system [5]). In the

real world, one mole of liquid contains 10^{26} molecules. PBC enables to mimic infinity of a medium using limited number of molecules. However, this assumption works well only for time scale limited by the size of computational box divided by sound speed in a medium simulated. Because the former one depends on M, to get more accurate results of phenomena under investigation, larger samples of molecules should be taken into account. Assuming that a molecule may consist of hundred and thousands of atoms (particles) and its simulation is much more slower than for a simple molecule in liquid Argon for example, the evolution of large number of particles simulated in longer and longer time scales becomes the great challenge for the fastest computer systems ever constructed. Therefore, the serious research has been going on for years now to implement MD codes on the top performance computer systems [6]. For parallel implementation of MD method, geometric decomposition is usually used. In Fig.3 we can see typical decomposition of the computational box for distributed computations on the ring of workstations (Fig.3a) and for parallel processing on MPP tightly coupled architectures (Fig. 3b).

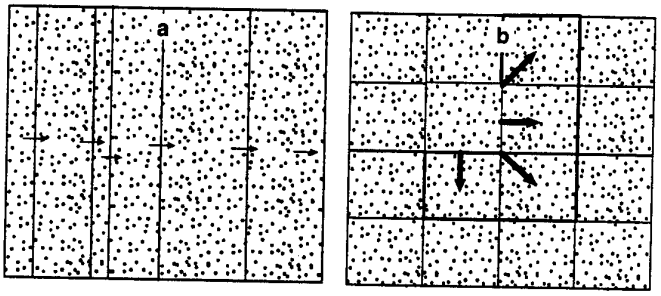


Fig.3. Two approaches for MD domain parallelism. The arrows show directions of information exchange between a domain (shaded) and its neighborhood. For (a) the load balancing is realized changing the strips width while for (b) it is more fine grained though complicated.

As is shown in [6], the progress in hardware and software development lets to increase the number of atoms simulated using MD codes from hundreds in late seventies to billions in the middle of nineties. The parallel MD codes reach 95% efficiency on hundreds of processors. A vast amount of literature and MD software for the full spectrum of vector and multiprocessor architectures are available. From this point of view, the MD method fulfills the important condition which the natural solver should posses. However, the most relevant feature of natural solvers consists in their universality.

3 Large-scale MD simulations of physical phenomena

The classical field of interest of MD simulations covers the microscopic, short-time phenomena in liquids and solids. Due to time and space averaging of stochastic functions and variables one can obtain integral and/or differential parameters of a medium investigated. Fitting simulation results to the experimental and theoretical values, one can find the proper model of molecules and/or potential energy of the

interacting particles. Moreover, it is possible to observe reactions of separate molecules and the whole system on the external stimulus. Nevertheless, all these phenomena occur in abstract microscopic world, which (as seems to be) limits the field of MD approach application.

The first MD experiments [7,8] in which not statistical fluctuations but rather collective movement of simple Lennard-Jones particle ensembles were investigated, show that even for relatively small number of particles in short-time simulations it is possible to observe the striking resemblance of patterns created in microscopic and macroscopic worlds. Increasing the number of particles to millions it is possible to simulate the phenomena in mesoscale (i.e., where the size of samples is 1μm of order and simulation time is tens of nanoseconds), e.g., fluid flows [8,9], crack formations [10], hydrodynamical instabilities creation [11,12]. Such investigations are important while classical models based on continuous matter and momenta equations (e.g. Navier-Stokes formulae in hydrodynamics) are insufficient and the assumptions of continuity are not valid any longer. The same concerns description of phenomena having their origins in microscale and resolving in macroscale. To simulate them using classical continuous models, artificial fluctuations are introduced. This results in the lack of any information about the beginning stage of mixing process, its causality and start up time.

The first results of simulations of the Rayleigh-Taylor instability using pure MD parallel code are presented in [12]. The computer experiment consists in simulation of mixing of two particle layers. The first layer consists of heavy particles and the second one - placed below - is made of light particles. The gravitational field directed from the heavy layer to the lighter one makes the system unstable. Due to statistical fluctuations two fluids begin to mix. This sort of instability belongs to the hardest case for simulation using classical hydrocodes. Especially its initialization is not investigated yet in details because of the lack of causality factor in the classical equations of fluid dynamics. As one can see in Fig.4, the evolution of mixing process using MD code is similar to this observed in experiment and those obtained from simulations which use classical hydrocodes. Unlike in simulations which use hydrocodes, however, the process is spontaneous, i.e., not initialized artificially. The fluctuations represent the real causality factor lacking in the former models. Due to this advantage it is possible to investigate more thoroughly time evolution of mixing layer not only for infinitely thick liquid layers but also for the layers with free surface (see Fig.4). For example, as one can see in Fig.5, two mixing regimes can be distinguished. The first one is observed at the beginning of process when only thin boundary layers of two liquids take part in mixing. While the sound wave - caused by turn on of the acceleration field - reflects from the bottom of computational box, the process changes in character and mixing gets faster.

The resemblance of the simulation results of similar processes in micro and macroscales inclines to the conclusion that by rescaling, changing the definition of a particle and interparticle potential we can use the MD model for simulation of physical phenomena in macroscale [13].

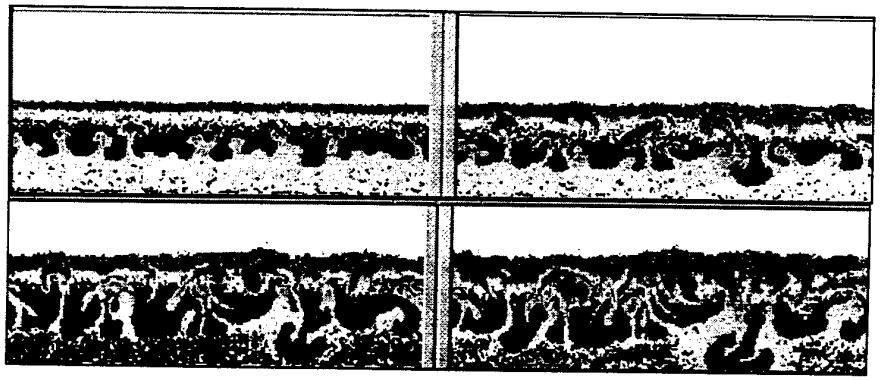


Fig.4. The snapshots of the Rayleigh-Taylor instability simulation using a million of particles for 300.000 timesteps in MD experiment The colors show the particles density. Simulation was performed using MD parallel code in PVM environment on 48 processors of Cray T3E system.

The advantages of particle approach over the computational methods, which use finite elements or finite differences, are evident. The most important factors are as follows:

- the lack of any grid,
- simple and flexible computational model,
- simple definition of discontinuities,
- efficient parallel codes,
- minor problems with complicated boundaries and inhomogenities.

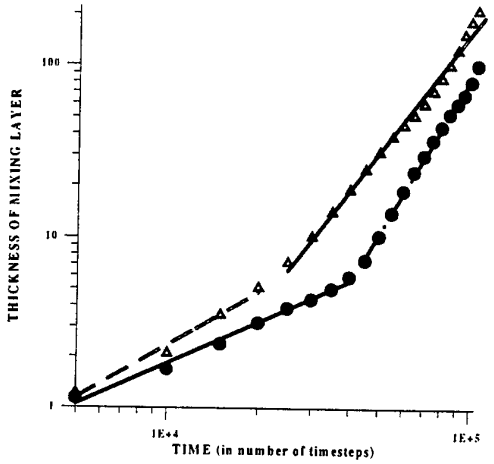


Fig.5. The growth of mixing layer for two different simulations (different thickness of the heavy layer assumed).

The problems with interparticle potential definition can be overcome using models for, so called, dissipative particle dynamics method [14] or deriving it directly from

the particle formulation of the Navier-Stokes equations using smoothed particle hydrodynamics method [15]. Another approach is used for granular media investigations (e.g. [16]) where the particles have different shapes and interaction potential is very sophisticated. Nevertheless, the "backbone" of all these models is based on the pure MD formulation and their parallel realization on MD parallel algorithms and methods.

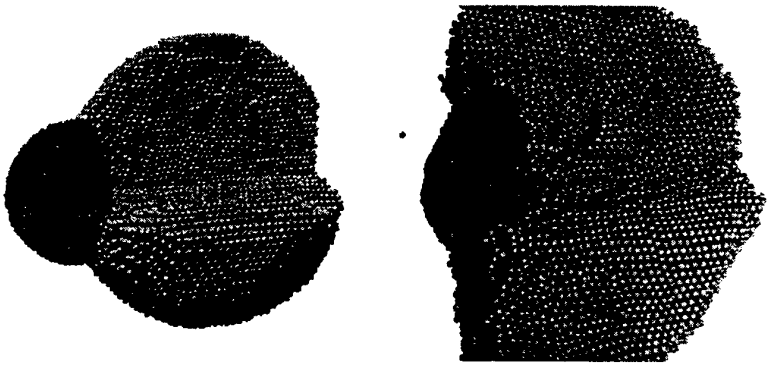


Fig.6. Two balls made of particles hitting one another. MD 3-D simulation.

We can expect, of course, that making the model more exact (e.g. due to more realistic potentials applied) thus more complicated, one can obtain eventually the results of MD simulations, which are in good quantitative agreement with an experiment. However, the fact that even for the simplest implementation of the MD method the quality of results obtained is astonished emphasizes the universal character of MD approach. For example, some effects in granular dynamics, similar to these observed in the reality can also be simulated using the simplest "soft balls" MD algorithms (see Fig.6). This fact can be exploited for animation purposes.

4 Method of particles as a predictive display

In some situations detailed physics, which stays behind phenomena under consideration, is not crucial. In animation methods, which assume some level of agreement with physical laws (so called, predictive display) more important is visual impression, than accurate quantitative agreement with the reality.

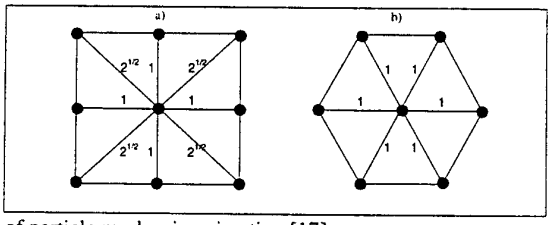


Fig.7. Two types of particle meshes in animation [17].

Assume that we are going to animate a thin flexible surface. This is a very complicated task in fact. As was shown in [18], such animation in real time is impossible due to complicated mathematics models laying behind a fabric dynamics. Moreover, the simulation needs supercomputer power when a typical FEM algorithm is involved.

Imagine that the fabric is made of particles. At the beginning of simulation the particles are placed in the nodes of hexagonal or rectangular grid (see Fig.7).

Each particle interacts with its neighbors via a semi-harmonic potential (for more details see[19]). Let us introduce gravitation and friction forces in Eqs.(1). Using *leap-frog* numerical scheme to the Newton equations (1) we obtain:

$$\mathbf{v}_i^{n+1/2} = \frac{(1-\varphi)}{(1+\varphi)} \cdot \mathbf{v}_i^{n-1/2} + \frac{\alpha \Delta t}{(1+\varphi)} \cdot \left\{ \sum_{j=1}^{K} (r_{ij}^{n\,2} - a_{ij}^2)\mathbf{r}_{ij}^n + \frac{g}{\alpha}\mathbf{i}_z \right\}, \quad \mathbf{r}_i^{n+1} = \mathbf{r}_i^n + \mathbf{v}_i^{n+1/2} \cdot \Delta t \qquad (2)$$

assuming that the friction force is:

$$\mathbf{F}_i = -\lambda \cdot \mathbf{v}_i \qquad \text{and} \qquad \alpha = \frac{k}{m}, \quad \varphi = \frac{\lambda}{2m} \cdot \Delta t$$

r_{ij} - current distance between particles i and j,
a_{ij} - initial distance between i and its neighbours on the mesh at the beginning of simulation,
m - particle mass,
k - a parameter of the semi-harmonic interparticle potential assumed,
Δt - time step.

Using MD code modified in such a way, realistic pictures of the fabric dynamics can be obtained during on-line animation on a standard Pentium II based PC (see Fig.8 for example, see also [17,19]).

Next, assume that several moving objects are animated. For very simple objects (see Fig.9) it can be done easily using the MD code on a PC computer. However, when the objects are more complicated and each consists of about 10.000 particles (e.g. the fiber in Fig.8) the fluent on-line animation is possible using a parallel machine.

As shown in [20], objects-to-processor mapping can be used. More than one object on a single processor is recommended. Additionally, two processors are used for graphical service and animation supervision (master processor) respectively. Load balancing is organized in such a way, that two colliding objects are moved to a single processor. If the number of objects taking part in collision is larger than 2 the number of processors used for simulation of this event is increased. The processors which are used in simulation of dynamics of the remaining objects communicate only with master processor to check collision conditions. As shown in Fig.10, for four colliding objects the optimal number of slaves is 2 (plus master and visualization processors).

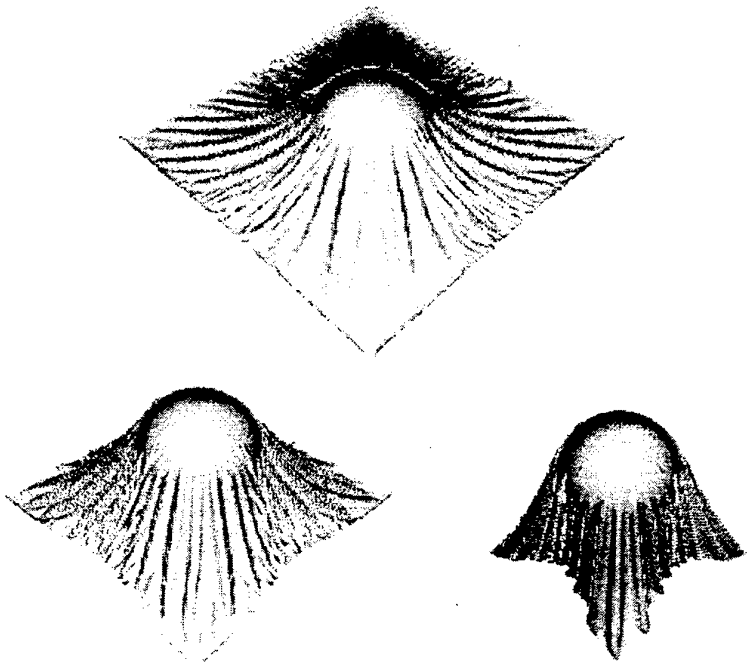


Fig.8. Snapshots of animation of the flexible surface using MD code.

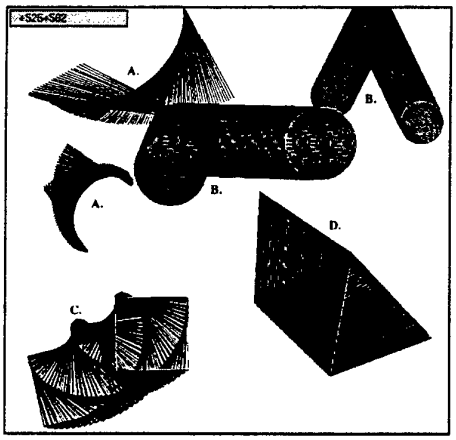


Fig.9. Fragments of trajectories of the simple objects animated using MD approach. The scene consists of: 2 sticks (A), 2 circles of various radiuses (B), a square (C) and a triangle (D). One can see the collisions between the objects and the square rotating after collision against the wall.

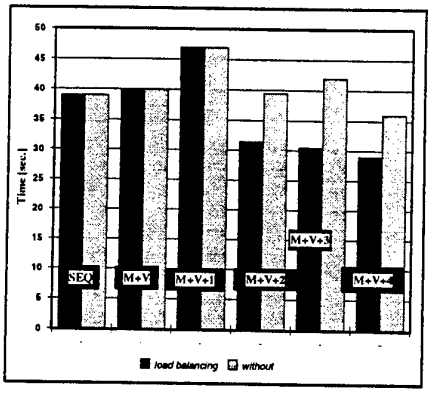


Fig.10. Timings for animation of a scene (with and without of load balancing), which consists of four moving cubes (2000 particles each). SEQ - sequential version, M - master processor, V - processor for visualization, K - slaves.

5 MD in global optimization problems

The change of particle abstraction level and interpretation of interparticle forces makes possible MD code application solving problems of a vehicle navigation between obstacles and search of global minimum of multidimensional functions.

In the first case the shortest or the most feasible path of a moving vehicle from a starting point to a target is looked for in presence both of static and dynamic obstacles. The application of the MD model for solving this problem is straightforward. Let us assume that the vehicle represented by a particle is attracted by the target. The obstacles are made of static particles, which repel the moving object. Then the object moves in accordance with Newton laws.

An MD approach to the navigation problem [21] differs from the classical navigation algorithms. This difference concerns a dynamic layer of the problem considered, i.e. the movement scenario, which is directly connected by physical laws with the vehicle-environment (obstacles and terrain) interactions. This makes the algorithm more flexible and open for verifications and improvements. Unlike graph theory algorithms both static and moving obstacles can be considered. An example of the vehicle paths are shown in Fig.11, assuming the presence of static obstacles only. Even for more complicated scenario the parallel realization of MD algorithm is not necessary because only local interaction between the object and obstacle are considered. While moving obstacles are taken into account, the parallel algorithm can be similar to that described earlier for animation purposes.

The problem of global optimization in a multidimensional space of a multimodal function is one of the most important and complex goals in many branches of science and engineering. Because, in general, the problem is unresolved using deterministic approaches many stochastic and heuristic methods were constructed in search of "immune" (problem independent) optimizer. According to our best knowledge such a method does not exist, though success of approaches such as genetic algorithms and

simulated annealing is out of question. MD, alike both of these heuristics, bases on the principles which come from nature. Let us assume that in Eqs.(1) a small dissipative factor is introduced. After some time, when kinetic energy of the particle system is removed, the particles stop moving and a minimum of the total potential energy of the system is gained. When dissipation of the kinetic energy is sufficiently slow, the global minimum is achieved.

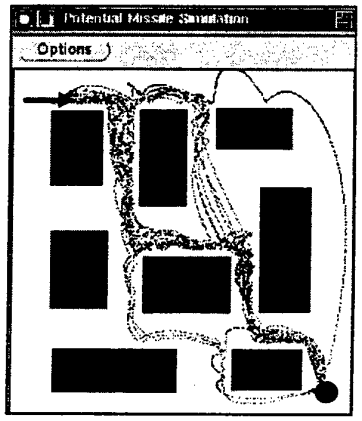


Fig.11. The paths from starting point to the target for different initial velocities of a vehicle. The most feasible path is the shortest one.

In Fig.12 one can see a realization of this idea. A global minimum of a multimodal and multidimensional function f(**x**) is searched. Initially the particles are scattered randomly in the function domain. The particles, which coordinates are $\mathbf{x}_i$ (i=1,...,*M*), interact via two-body, one-directional forces. Only particle representing lower f(**x**) value attracts the other one. A particle which gives the lowest function value for a current simulation step is stopped. The force between two particles *i* and *j* is dependent on the difference between the function values in $\mathbf{x}_i$ and $\mathbf{x}_j$, i.e., $|f(\mathbf{x}_i)-f(\mathbf{x}_j)|$. As one can see in Fig.12 the right solution is found for relatively small number of particles and without f(**x**) gradient calculation.

MD approach to global optimization was successfully applied in, so called, visual clustering and non-linear mapping problems [22]. The principal goal of non-linear mapping algorithms, consists in such a generation of points in 2(3)-dimensional space that the distances between them approximate the distances between respective N-dimensional points, which represents the measurement data. The method lets to visualize the multidimensional forms in 2(3)-dimensional space. This is accomplished by minimizing the criterion function

$$E = \sum_i \sum_j V_{ij}\left(D_{ij}, r_{ij}\right) \qquad (3)$$

The criterion (3) is the generalized case of the well known Sammon's criterion

$$E = \sum_i \sum_j D_{ij}^{2 \cdot w \cdot m} \left(D_{ij}^2 - r_{ij}^2 \right)^m \tag{4}$$

where: D_{ij}^2 – is squared distance between points i and j in N-dimensional space, r_{ij}^2 – is squared distance between respective i and j points in 2(3)-D Euclidean space, w and m - parameters (m>1 and $w \in \{-1,0,1\}$).

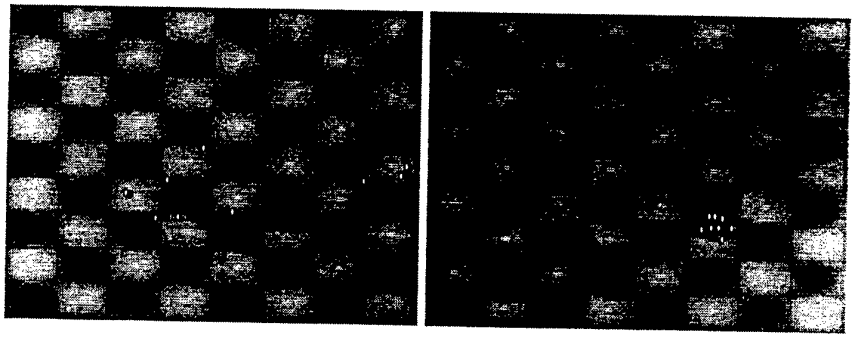


Fig.12. The application of MD paradigm in search for global minimum of multimodal and multidimensional (10-D) test function.

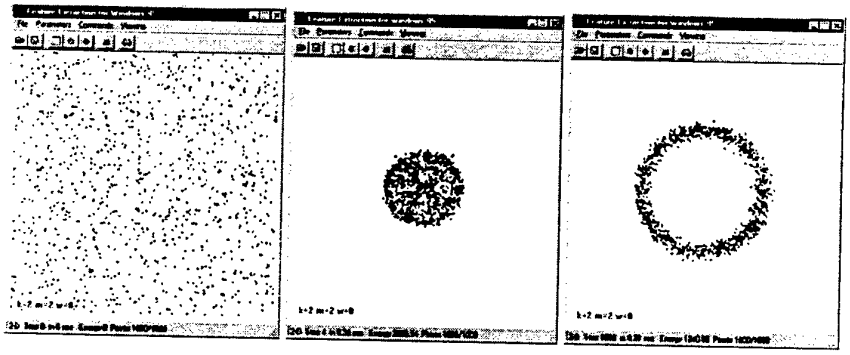


Fig.13. The snapshots of MD mapping process of 100-dimensional data placed on the sphere.

A new method proposed in [2,22], uses MD for minimization of the criteria (3,4). It is assumed that in 2(3)-D M particles are scattered randomly. Each particle corresponds to the respective N-dimensional data point. The particles interact one with another via two-body potential dependent on D_{ij} and r_{ij} and equal to $V_{ij}(D_{ij}, r_{ij})$. The particles move according to Newton's laws of motion. The friction force assumed removes the kinetic energy from the particle system, which stops moving eventually when the potential energy (1) reaches global minimum. The positions of particles reflect the final result of mapping (see Fig.11).

6 Conclusions

In the paper it is shown that the MD model can be treated as a natural solver which has broad scope of use in different fields. The application of MD simulation in mesoscopic scales for studies of collective movement of particles can be a valuable supplement for classical, continuous models. For studies of nonlinear phenomena such as Rayleigh-Taylor instability, which has their origins in microscale, MD can be treated as an unique tool for simulation of initial phase of mixing and observation of instabilities evolution. Moreover, MD algorithms yield a simple and effective parallel computational code, which can be treated as a "backbone" for other more sophisticated particle based methods such as dissipative particle dynamics and smoothed particle hydrodynamics used in simulations of the macroscopic world phenomena. The change of definition of a particle from single atom to the cloud of matter and changes in the interaction potentials assumed, does not affect the structure of the parallel codes used for pure MD formulation. The MD model can be also applied for animation purposes of macroscopic objects giving an impression that the objects dynamics is in good agreement with physical laws, though detailed physics may be considerably simplified.

The encouraging results of tests of MD applications in global optimization problems such as vehicle navigation problem and search of global minimum of multimodal and multidimensional functions show that miscellaneous branches of science are subordinated to the similar, general and universal rules, while the computer science plays the important role in their extraction and dissemination.

Acknowledgments

Thanks are due to Dr W.Alda, M.Sc. R.Wcisło and Mr G.Popiela whose contribution to some results used in this work is appreciated. Special acknownledgment is directed to Dr M.Bubak and M.Pogoda for for suupporting us with the MD parallel code. The work is supported by AGH funds No.11.11.120.16 and 10.120.20.

References

1. Sloot, P.,M.,A., Kaandrop, J., A., Schoneveld, A.: Dynamic Complex Systems (DCS): a new approach to parallel computing in physics. Technical Report of Department of Computer Science, University of Amsterdam, CS-95-08 (1995).
2. Dzwinel, W.: Virtual Particles and Search for Global Minimum. Future Generation Computer Systems, 12, (1997) 371-389.
3. in: Gather/Scatter Newsletter, 10, 3, (1994).
4. Haile, J.,M.: Molecular Dynamics Simulation. John Wiley&Sons Inc., New York, (1992).
5. in: IEEE Computational Science and Engineering, (1995), 78.
6. Beazley, D.,M., Lomdahl, P.,S., Jensen, N.,G., Giles, R., Tomayo, P.: Parallel Algorithms for Short-Range Molecular Dynamics. World Scientifics Annual Reviews in Computational Physics, 3, (1995).

7. Rapaport, D.,C.: Eddy Formation in Obstructed Fluid Flow: a Molecular Dynamics Study. Phys.Rev.Lett., 57, (1986), 695.
8. Rapaport, D.,C..: Microscale Hydrodynamics: Discrete-particle Simulation of Evolving Flow Patterns. Phys. Rev., A36, 7, (1987), 3288.
9. Cui, S.T., Evans, D.J.: Molecular Dynamics Simulation of Two Dimensional Flow Past a Plate. Molecular Simulation, 9, (1992), 179.
10. Holian, B.,L., and Ravelo, R.: Fracture Simulation Using Large-Scale Molecular Dynamics. Phys.Rev B., 51, 17, 1995, 11275.
11. Rapaport, D., C.: Molecular-Dynamics Study of Rayleigh-Benárd Convection. Phys. Rev. Let., 60, 24, (1988), 2480.
12. Mościński, J., Alda, W., Bubak, M., Dzwinel, W., Kitowski, J., Pogoda, M., and Yuen, D.: Molecular Dynamics Simulations of Rayleigh-Taylor Instability, Annual Reviews of Computational Physics 5, (1997), 96-136.
13. Dzwinel, W., Alda, W., Kitowski, J., Mościński, J., Wcisło, R., and Yuen, D.: Macro Scale Simulations Using Molecular Dynamics Method. Molecular Simulation, 15, (1995), 343.
14. Koelman, J.M.V.A. and Hoogerbrugge, P.,V.: Dynamic simulation of hard-sphere suspensions under steady shear. Europhysics Lett. 21, (1993), 363.
15. Peschek,A.,G.and Libersky,L.,D.: Cylindrical Smoothed Particle Hydrodynamics. Journal of Computational Physics, 109, 1, (1993), 76.
16. Form, W., Kohring, G.A., Melin, S., Puhl, H., and Tillemans, H.,J.: Computer Simulation of Critical, Non-Stationary Granular Flow in a Hopper. *KFA-Juelich Hochstleistung-srechenzentrum HLRZ Preprint,* 75/93, (1993).
17. Wcisło, R., Dzwinel, W., Kitowski, J., and Mościński, J.: Molecular Dynamics for Real World Phenomenon Animation. *CCP5 Information Quarterly, Darresbury Labolatory, Warrington, U.K*, Sierpień 1993, **38**, (1993), 25.
18. Wang B., Wu Z., Sun Q., and Yuen, M.,M.,F.: A deformation model of thin flexible surfaces. 6 Int. Conf. in Cent. Eur. on Computer Graphics and Visualization, Plzen, Czech Republic, February 9-13, (1998).
19. Wcisło, R., Dzwinel, W., Kitowski, J., and Mościński, J.: Real-time Animation Using Molecular Dynamics Methods. Machine Graphics&Vision, 3(1/2), 1994, 203-210.
20. Wcisło, R., Kitowski, J., Mościński, J.: Parallelization of a code for animation of multi-object system" in: Waśniewski, J., Dongarra, J., Madsen, K., and Olesen, D., (eds.), Applied parallel computing - industrial computation and optimization. Lecture Notes in Computer Science 1184 , 697-709, Springer, (1996).
21. Mościński, J., and Dzwinel, W.: Simulation Using Particles in Robot Path Planning. Proc. of Int. Conf. Methods and Models in Automation and Robotics, 10-13 September 1996, Międzyzdroje, Poland, 3, (1996), 1000-1110.
22. Dzwinel, W.: On Search for the Global Minimum in Problems of Features Extraction and Selection. Proceedings of the Third European Congress on Inteligent Techniques and Soft Computing, EUFIT'95, 28-31 August 1995, Aachen, Germany, 3, 1326-1330, 1995

Co-design Decisions for High Performance Parallel Architectures

J. C. Moreno, A. Alcolea

Department of Electronics and Communications Engineering
University of Zaragoza

María de Luna 3, 50015 Zaragoza, Spain
Phone 34 (76) 761943
e-mail jcmoreno@posta.unizar.es

Abstract. The goal of this paper is to propose cost-performance criteria which can be used to take co-design decisions. The criteria are simplified with some assumptions, and are used to modify the hardware design of a fine grain multiprocessor architecture. The modifications optimize the execution time of the elemental operations (addition, substraction, comparison and product). The criteria are a trade-off measure between the hardware complexity and the execution time of the elemental operations. The modifications improve the system efficiency while the cost is maintained.

1 Introduction.

When some modifications should be done in a hardware design, and the cost of the system is important too, one main question is: the performance increase justifies the cost increase?. However, parallel architectures allow the interchange between the processor element complexity and the number of processor elements of the system while the total cost of the system is maintained. This means that, for the same total cost, we can have more complex processor elements, but a lower number of them, or we can have less complex processor elements, but a higher number of them. It is obvious that there will exist a trade off between the processor element complexity (unitary cost) and the system size that makes maximum the system performance for a given cost. So, the new question is: the hardware modification increases the system performance while maintaining the total cost?. It is clear that if the answer is yes, the modification can be immediately accepted, otherwise the modification will be accepted or not depending on the cost goal.

This paper proposes cost-performance criteria that allow to decide if a modification can be immediatelly accepted or not. The criteria are used to evaluate hardware modifications which try to decrease the execution time of the software instructions for elemental operations.

But, what was the problem that led us to this point?. Some time ago, we designed a vision oriented SIMD architecture [1], but it is well known the saturation effect that SIMD architectures show: in most cases, the slope of the performance

function decreases as the number of the processor elements increases for intermediate and high level vision algorithms. We have demonstrated in previous works [1] that the reconfiguration of the datapath width palliates this problem.

The reconfiguration consists in the interchange between the number of processor elements of the system and their datapath width. So, we can have a system integrated by n processor elements with 1-bit datapath width and we can reconfigure it to a system integrated by n/B processor elements with B-bit datapath width. The problem arised when we evaluated the speed of the hardware for elemental operations in reconfigurated mode. This speed was low, and hardware modifications became necessary for a high performance in reconfigurated mode.

Then, in order to have objective parameters to measure the convenience of a hardware modification, we proposed the cost-performance criteria which are explained in this paper.

Other works have been developed in the literature about this theme. References [2], [3] give general ideas about the hardware-software co-design. However, only general criteria are shown in [4] and [5]. In [4] are presented optimization criteria which can be applied to architectures that show a linear cost in their communication network (i.e. a processor element can always communicate with the same processor elements for all system sizes). In [5] the criteria take into account a non-linear dependence on the cost with the interconnection network and can be applied to more complex connection patterns.

2 Cost-performance criteria.

The total cost of a system may be very difficult to model: hardware, software and peripherical circuitry, among others, are different parts of the cost. In order to obtain reliable models, [4] and [5] take into account the hardware cost due to the silicon area, which is the most important in most cases.

We have used the criteria described in [4] because in our SIMD architecture every processor element can communicate with the same neighbours (North, East, South, West) without dependence on the system size. Reference [4] gets the condition which a modification has to verify:

$$\frac{A_i}{A_f} > \left[1 + t_{oop}(A_i) \times (R-1)\right] \times \left[\frac{E(N_i, A_i)}{E(N_f, A_f)}\right]. \quad (1)$$

$$R = \frac{T_{poop}(A_f)}{T_{poop}(A_i)}. \quad (2)$$

$$t_{oop}(A_i) = \frac{T_{oop}(A_i)}{T_{noop} + T_{oop}(A_i)}. \quad (3)$$

Where:

$A_{i/f}$ = Initial/final area, before/after the modification.

$N_{i/f}$ = Initial/final number of processor elements.

$E(N_{i/f}, A_{i/f})$ = Initial/final system efficiency.

$T_{poop}(A_{i/f})$ = Time per optimized operation in the initial/final conditions.

$T_{noop/oop}$ = Time which is needed by a processor element to execute the non optimized/optimized operations of the task.

If the modification implies a higher area for the processor element, then normally $E(N_i, A_i)/E(N_f, A_f) > 1$ and a harder condition, which is easier to verify, is:

$$\frac{A_i}{A_f} > \left[1 + t_{oop}(A_i) \times (R-1)\right] . \qquad (4)$$

The simplified procedure to evaluate the convenience of a modification is the following (we suppose that initial conditions are known):

a) Calculate the final area A_f.

b) Obtain the final time per optimized operation $T_{poop}(A_f)$.

c) Get the reduction factor R.

d) Find the time relation between the optimized operation and the total task in the initial conditions $t_{oop}(A_i)$.

e) Check the eq. (4). If it is verified and the modification has increased the processor element area, then the modification can be accepted, else it is necessary to evaluate the final efficiency $E(N_f, A_f)$ and to check the eq. (1).

3 Criteria application to the addition operation.

Figure 1 shows an addition example the data 1 is added to the data 2 and the result is obtained. This type of addition (reconfigurated mode) presents two main problems:

a) The carry generated by the most significant processor element should be communicated to the least significant processor element. Besides, the communication path depends on the number of processor elements rows that integrate a multibit processor (see fig. 2). For an even number of rows, it is necessary a horizontal communication followed by a vertical one, while for an odd number of rows, it is only necessary one vertical communication.

b) The least significant processor element receives zero in its ALU carry input for the first sum, and for long data (more than one word), it receives the carry from the most significant processor.

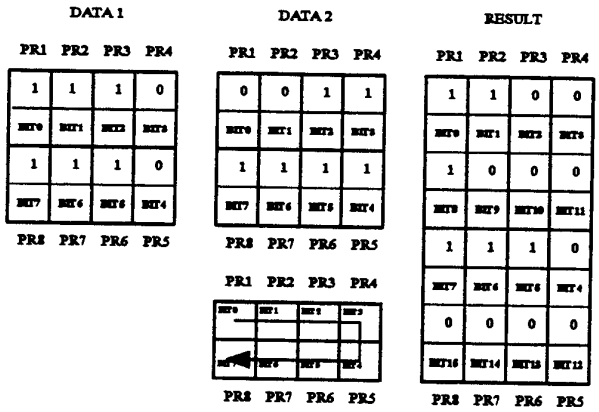


Figure 1. Multibit addition example.

These and other considerations makes the multibit addition no efficient. It is clear that for a 100% of efficiency these two terms should be equal:

a) Number of clock cycles to execute one monobit addition.
b) Number of clock cycles to execute B multibit additions. Remember that B is the datapath width in the reconfigurated work mode.

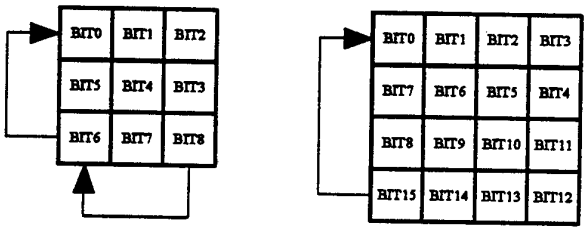


Figure 2. Communication carry path depending on the number of processor element rows.

Actually, a multibit processor is integrated by B processor elements, so a fair comparison is to evaluate the clock cycles for the same number of operations in both work modes (monobit and reconfigurated). This implies the previous equality because B additions are executed in parallel in monobit mode, and their time cost is the number of clock cycles for one monobit addition, so B additions should be executed in multibit mode. It is clear that because of the bit paralellism and for 100% efficiency, every multibit addition should execute in 1/B times the number of clock

cycles of one monobit addition. However, due to the hardware design and the difference between the datalength and the datapath width of the architecture, the efficiency will be lower than 100%.

Figure 3 shows the efficiency for the addition operation with the initial hardware design. In order to increse its efficiency we have modificated the hardware design. The modification allows the carry communication between the most significant processor element and the least significant processor in a single clock cycle.

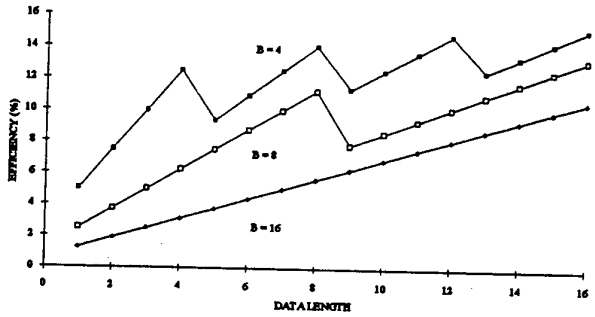


Figure 3. Efficiency (%) for the multibit addition respect to the monobit addition without hardware modification.

The hardware modification adds one input to the output multiplexer and to the ALU carry input multiplexer. Figure 4 shows the efficiency with the hardware modification included in the design. Note that the efficiency has been duplicated. This means that the execution time per multibit addition has been reduced to half.

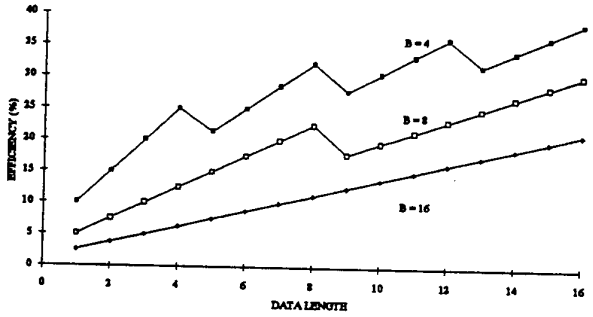


Figure 4. Efficiency (%) for the multibit addition respect to the monobit addition with hardware modification.

The increase on the processor element area due to the modification is 2% using ES2 library for 0,7μm double metal CMOS technology.

Once we have the time relation and the area relation, we can evaluate the eq. 4. In this case: $A_i / A_f = 0.98$ and $R \equiv 0.5$.

So, from eq. 4, $t_{oop}(A_i) \geq 3.9\%$. This means that, for the modification acceptance, at least the 3.9% of the total execution time of the task, in the initial conditions, should be dedicated to addition operations in reconfigurated mode.

A global vision task is normally divided into different subtasks. Every subtask may have part of the object code that is executed in monobit mode, and other part executed in reconfigurated mode. Besides, not all operations are additions in reconfigurated mode. So, depending on the vision task, the hardware modification will be or not accepted.

4 Conclusions.

Cost-performance criteria have been proposed in this paper that can be applied to multiprocessor architectures with no cost dependence on the interconnection network (the number of interconnections per processor element does not depend on system size). The criteria have been simplified to make the equations easier to evaluate, and one example has been explained.

The example demonstrates that the criteria can be extended to other hardware modifications. The criteria measure the interchange between the processor element complexity and its unitary cost, while the total cost of the system is maintained. However, this interchange allows to maximize the system performance. This means that for a given total cost, we can obtain the processor element design that maximizes the system performance.

References.

1. J. C. Moreno, A. Alcolea. "SIMD Architecture with Reconfigurable Datapath Width Efficiently Adaptable to Different Computer Vision Levels". *Proc. of the Tenth Int. Conf. on Systems and Integrated Circuits Design (SICD)*, November 1995.

2. N. Woo, A. Dunlop, W. Wolf. "Codesign from Cospecification". *Computer*, January 1994.

3. D. Thomas, J. Adams, H. Schmit. "A Model and Methodology for Hardware-Software Codesign". *IEEE Design & Test of Computers*, September 1993.

4. J. C. Moreno, A. Alcolea. "Architectural Optimization Via Cost-Performance Criteria". *Proc. of the Eleventh Int. Conf. on Systems and Integrated Circuits Design (SICD)*, November 1996.

5. D. Sarkar. "Cost and Time-Cost Effectiviness of Multiprocessing". *IEEE Transactions on Parallel and Distributed Systems*, June 1993.

Achieving Data Availability on Parallel and Distributed File Systems

Francisco Rosales[1] and Raimundo Vega[2] *

[1] frosal@fi.upm.es Facultad de Informática, Universidad Politécnica de Madrid
[2] rvega@uach.cl Facultad de Ingeniería, Universidad Austral de Chile

Abstract. We present an enhanced data availability I/O Subsystem model for ParFiSys, a Distributed and Parallel File System. We evaluate the application of data redundancy at the different levels of the I/O hierarchy. A virtual distributed and redundant device, known as VRAID, is used as the basis to achieve both I/O accesses parallelism and better fault tolerance.
Keywords: Parallel, file system, data availability, redundancy.

Introduction

ParFiSys [2] is a Distributed and Parallel File System [1] devoted to exploit as much as possible the I/O Subsystem on architectures where several I/O nodes are interconnected by a high performance network. ParFiSys early design was focused on improving I/O performance, and data availability problems due to a large number of underlying devices [9] were not taken into account.

In this paper, we describe a new redundant I/O Subsystem model for ParFiSys that should be able to offer data availability even on underlying device failures. We detail the algorithms used to improve performance by minimizing both, the impact of redundancy management on communications, and the reconstruction phase overhead. We evaluate the model over a massively parallel architecture simulator that has also been developed [10, 12, 13].

1 I/O Subsystem Model

The I/O Subsystem (Fig. 1 is built on the I/O hardware of a massively parallel machine with a high performance interconnection network. The physical storage devices are distributed over several I/O network nodes. Additionally, two logical storage devices are defined, one per I/O node server (SERV), that manages remote accesses to any other storage device of the node, and a single virtual redundant storage device known as VRAID, that distributes the data all over the SERV devices of the whole system.

* Thanks to Professor De Miguel for his technical advice.
[1] ParFiSys was developed at the Polytechnical University of Madrid, under the ESPRIT project P5404 funded by European Union.

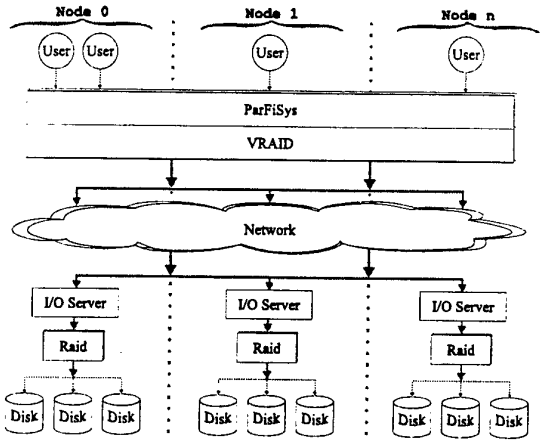


Fig. 1. I/O Subsystem Architecture

The Raids and VRAID can be configured as level 0, 4 or 5 [4, 3]. Usually the redundancy unit is known as stripe–unit, and is composed of one storage unit of each underlying device, one of which (the parity unit) contains the exclusive–OR calculation of all the others. It is important to note here that at any time the parity unit contents must be consistent with the rest of the information stored in the stripe, so a locking mechanism must be used to organize concurrent accesses involving parity units. This means that we will need to use locks at every access but when reading a free of fault device.

VRAID Distributed Lock Management In the VRAID, the parity calculation is done at the node that makes the I/O request, so a lock mechanism is required to ensure the correct order between any number of parallel remote accesses.

We have chosen to locate a lock service at SERV, and to lock only the parity units involved. Therefore, the distribution of locks will follow the same mapping as those of parity units. This means three things: a) this distributed consensus will ensure per stripe–unit consistency, b) this will not suppose a bigger bottleneck than the access to the parity unit itself and c) there will also be a unified distributed consensus on the new lock server to use in case that the device goes to degraded state.

Improving Performance Depending on its size, an I/O action could correspond to a huge number of subactions over a (possibly sparse) set of individual storage units of the underlying devices (i.e. Fig. 2). In order to reduce the amount of individual subactions and to optimize underlying device access, this set is reordered by joining subactions that are logically contiguous: 1) they refer to the same underlying storage device, 2) they are of the same action type (lock, read, xor, write or unlock) and 3) they concern to a set of contiguous units.

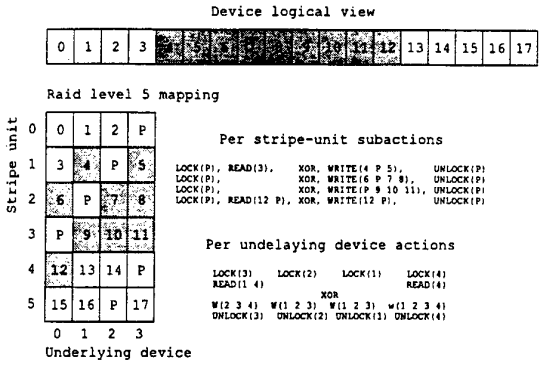


Fig. 2. Raid level 5. Write from 4 to 12 decomposition

The resultant set of actions ordered is processed running in parallel actions for each device, but doing it in the following order: all locks, all reads, the internal xor calculation, all writes and finally the unlocks. This method has the following properties: a) ensures consistency between data and parity of each concerned stripe–unit, b) minimizes the final number of actions and therefore, (in the case of VRAID) the network traffic, b) the final per device action is more compact and could be done faster.

2 System and Workload Characterization

All the performance analyses in this paper have been made over a simulation of a massively parallel machine characterized as shown in table 1. The File System is feed by workers distributed over the nodes in a round robin way. Each worker executes I/O operations continuously from the selected synthetic workload (Tab. 2). We use enough workers to make the system to perform at its limit.

We have done experiments in order to determine the system scalability and its behavior on different combinations of redundancy levels and VRAID states (fault–free, degradated and during the reconstruction phase).

Table 1. Systems Evaluation Parameters

Network	crossbar topology with 100 MB/s links
Nodes	2 to 32 (plus one for VRAID type 4 or 5)
VRAID	Levels 0, 4 and 5. Unit of 64KB or 4KB for OLPT
RAID	Levels 0, 4 and 5. Unit of 4KB.
	With 4 disks (5 for levels 4 and 5)
Disks	"Seagate Elite3", 2627 cylinders * 21 tracks * 99 sectors
	5400 RPM and seek times 1.7 min., 11.0 avr. and 22.5 max. (ms)

Table 2. Synthetic Workloads Parameters

OLPT	SSIM
Online Transaction Processing [6, 11]. 80% reads of 4KB, 16% writes of 4KB, 2% reads of 24KB, 2% writes 24KB. All uniformly distributed.	Scientific Simulation. 50% sequential 1MB accesses to one 100MB file (90% reads, 10% writes) 50% uniform 512KB accesses to 10 5MB files (10% reads, 90% writes)

3 Results Analysis

In Fig. 3 we show comparative performance for different system sizes running with VRAID level 5 in fault–free, degradated and recovery states.

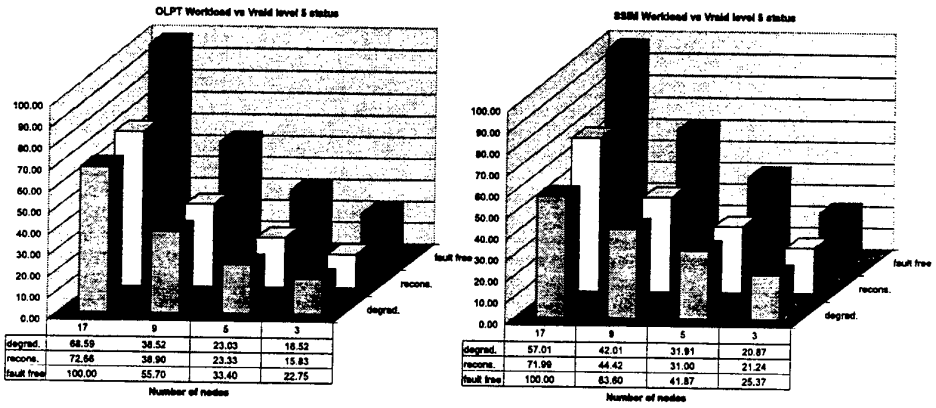


Fig. 3. Performance in Different VRAID States

We observe that the performance in degraded state shows a better scalability for OLPT than for SSIM. Whereas the overhead of degraded accesses grows with the number of involved nodes, the probability that an OLPT operation does not concerns the failed node also grows. This is not true for SSIM accesses, that affect all nodes, so for each write, a previous read of the parity information is needed.

During the reconstruction phase one special worker recovers the failed device. This implies an added overhead. To improve performance recovery is done in chunks which are put to normal service as soon as recovered. As Fig. 3 shows the mean bandwidth during recovery phase is improved over the degraded one.

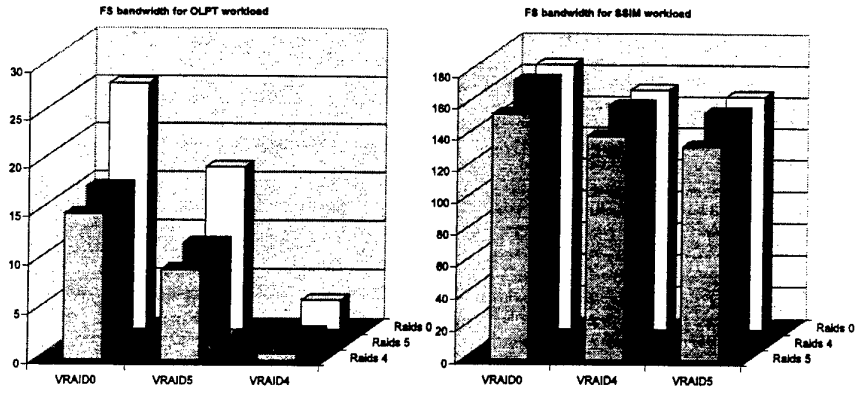


Fig. 4. Comparative Performance for Different Redundancy Models

In Fig. 4 we show a 32 node system with different models of redundancy both for VRAID and Raids.

Our results show that SSIM workload gives around 160MB/s peek bandwidth whereas OLPT gives 25MB/s. SSIM is not affected very much by the redundancy model, because large operations involving contiguous blocks on all disks. are done much more efficiently. Obviously VRAID level 0 gives the best bandwidth, but does not protect us from a node failure, it is given for comparison.

The OLPT workload involves very small size operations (4KB and 24KB), so the redundancy management overhead is more significant than in SSIM. Nevertheless combination VRAID 5 – Raids 4 has very similar performance than VRAID 5 – Raids 5.

4 Conclusions and Future Work

Given the observed system behavior, we can conclude that the systems scales very well, and systems of 128 nodes or more are possible. For small systems (32 nodes or so) we suggest configurations with VRAID level 5 and Raids level 0, this allows for the same recovery procedure from a node or a disk failure.

The recovery time for a disk failure using the VRAID redundancy at is impractical in larger systems. Therefore, we suggest the use of level 5 redundancy at both VRAID and Raids levels.

We are now including the effect of different caching alternatives on the above results.

References

1. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, John K. Ousterhout: Measurements of a Distributed File System. SOSP (1991)
   `ftp://ftp.cs.berkeley.edu/ucb/sprite/papers/measureSOSP91.ps`
2. J. Carretero, F. Pérez, P. de Miguel, F. García, L. Alonso: ParFiSys: A Parallel File System for MPP. ACM SIGOPS 30, (1996) 74–80
3. Peter M. Chen, Edward K. Lee: Striping in a RAID Level 5 Disk Array. Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems
4. P. Chen, E. Lee, G. Gibson, R. Katz, D. Patterson: RAID: High-Performance, reliable Secondary Storage. Acm Computing Survey, Vol 26, N? 2, (June 1994)
5. Drog G. Feitelson, Peter F. Corbett, Sandra Johnson Baylor, Yarsun Hsu: Parallel I/O Subsystems in Massively Parallel Supercomputers. IEEE Parallel & Distributed Technology 3(3), (1995) 33–47
6. Mark Calvin Holland: On-Line Data reconstruction In redundat Disk Array. Dept. of Electrical and Computer Engineering, Carnigie Mellon University (1994)
7. John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michel J. West: Scale and Performance in a Distributed File System. ACM Transaction on Computer Systems. (February 1988)
   `http://das-www.harvard.edu/cs/academics/courses/cs261/readings/howard-1988.html`
8. Nils Nieuwejaar, David Kotz: Low-level Interfaces for High-level Parallel I/O. Dartmouth PCS-TR95-253 (1995)
   `ftp://ftp.cs.dartmouth.edu/TR/TR95-253.ps.Z`
9. Q. M. Malluhi, W. E. Johnston: Techniques For Availability And Reliability of Distributed Parallel Storage Systems. Proceeding of the Lasted International Conference Parallel and Distributed System-Euro-PDS'97 (June 1997) Barcelona Spain.
10. Raj Jain: The Art of Computer Systems Performance Analysis. Wiley Professional Computing. ISBN 0-471-50336-3
11. Apratim Purakayastha, Carla Ellis, David Kotz, Nils Nieuwejaar, and Michael Best: Characterising Parallel File-Access Patterns on a Large-Scale Multiprocessor. Technical Report CS-1994-33, (Oct. 1994) Presented at IPPS95.
    `http://www.cs.duke.edu/~ carla/ap.ps`
12. Chris Ruemmler and John Wilkes: An introduction to disk drive modelling. IEEE Computer 27(3), (March 1994) 17–29
    `http://www.hpl.hp.com/personal/John_Wilkes/papers/IEEEComputer.DiskModel.ps.Z`
13. Chandramohan A. Thekkath, John Wilkes and Edward D. Lazowska: Techniques for File System Simulation. SPE 24(11), 981–999 (November 1994)
    `http://columbus.cs.nott.ac.uk/cgi-bin/getpaper?paper=spe922ct.pdf`

PC and DSP based AC motor adaptive vector control system

David Bedford Guaus, Antoni Arias Pujol, Emiliano Aldabas Rubira,
and José Luis Romeral Martínez

Electronic Engineering Department, Polytechnic University of Catalonia, c) Colom, 1,
08222 Terrassa (Barcelona), Spain
Bedford@Eel.upc.es

Abstract. In this paper we propose a system with an architecture capable of parallel processing. Also, due to its computational power, the system is able to handle complex algorithms. This structure is applied to an AC motor vector control system, formed by two control loops which are running simultaneously: a speed control loop and a motor model parameters (needed by the speed controller) identification loop. This architecture allows the experimentation of new control algorithms in this field. Some results are presented that show the system's performance.

1 Introduction

The new control algorithms experimentation requires the availability of a system with an architecture that allows the easy reprogramming of their elements separately and the execution of complex algorithms, which can be executed simultaneously. Also, many industrial controls are based on a multi processor architecture, that use two or more low cost processors instead of one complex (expensive) processor. In this paper we present an architecture based in two processors that will allow the experimentation of the control techniques that we have previously studied analytically and/or simulated, which will be afterwards implemented in a multiprocessor configuration.

2 Proposed system architecture

The implemented system architecture is shown in figure 1. The system has two processors: a 486 (PC) and a Digital Signal Processor (32 bits floating point DSP). The DSP is placed in a PC ISA bus slot, which acts as the physical interface. The data exchange between the two processors is done using a Dual Port RAM (DPRAM), which can be accessed simultaneously by both processors. The DPRAM allows fast information exchange between the PC and DSP without disrupting the

processing of either device. If it is necessary, the DSP is able to interrupt the PC by means of the IRQ3 line; also the PC can interrupt the DSP using one of its four interrupt lines (INT3). The PC is able to control and monitor the DSP by means of an I/O mapped interface. The communication of the system with the external world is done by means of the following devices, which are connected to the DSP: an A/D converter module with four 16 bit channels, with a maximum sampling speed of 50 kHz, and a digital I/O board, with 32 user configurable I/O channels. This configuration is clearly being used in many fields [1]. The PC is programmed using C language (Borland C). The DSP is programmed using either Assembler and C language. In the latter, the routines that are time critical are programmed using Assembler to control precisely the execution time.

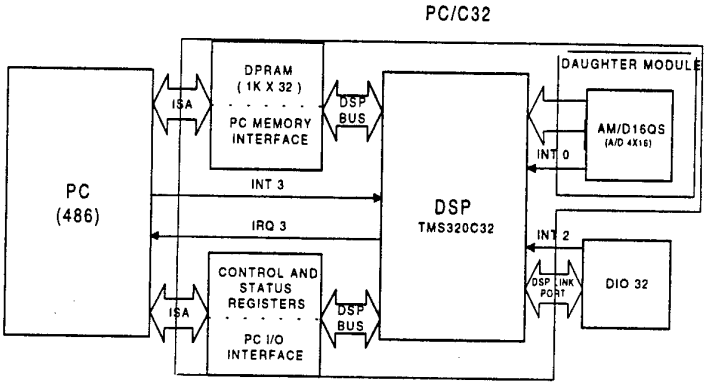


Fig. 1. System architecture

3 AC motor control system

The block diagram of the AC motor adaptive vector control system that we have implemented is shown in figure 2. This control system has two loops which are running simultaneously: the speed control loop, that actually controls the motor speed, and the parameters identification loop that tunes the FAM controller parameters.

3.1 Speed control loop

This loop controls the AC motor speed. It has the following elements:

Speed Controller. It computes the torque setpoint (T) from the speed error (Ew). This controller algorithm has been implemented using fuzzy logic, due to its major robustness faced by system changes (inertia, load) [2].

FAM Controller. It computes the voltage (V) in amplitude, phase and frequency that has to be applied to the AC motor from the torque setpoint (T). It uses the Field Acceleration Method, that maintains the motor magnetising flux constant, thus avoiding electromagnetic transients. To achieve this it is necessary to tune the FAM controller parameters precisely in accordance with the AC motor [3], which is performed by the other loop.

Inverter Controller. It generates every 100 μs the control signals for the inverter gates from the desired voltage (amplitude, phase, frequency). Is based in a vector modulation algorithm that takes into account the necessary inverter dead times, and it uses an accumulated error algorithm to improve its performance (harmonic distortion).

Inverter. It is the power device that supplies the voltage and current consumed by the AC motor. This device includes the logic necessary to protect it from overvoltages and overcurrents.

AC motor. It is the machine whose speed (and torque) we control.

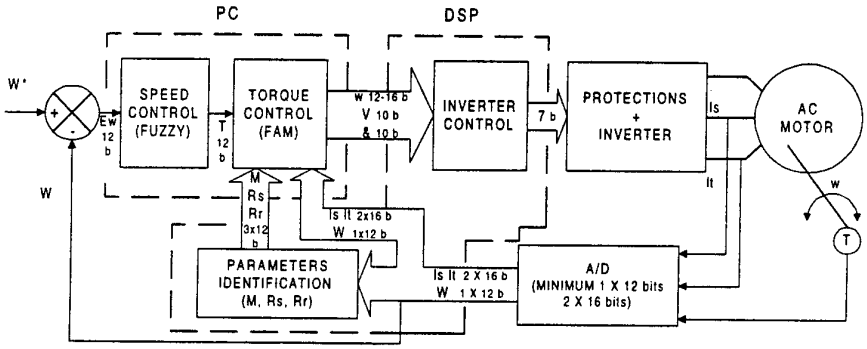


Fig. 2. AC motor control system, formed by two loops

3.2 Parameters identification loop

This loop modifies the FAM controller parameters. It is formed basically by the Model Reference Adaptive Controller (MRAC), which is the block that performs the parameters identification that the FAM controller needs, by means of an algorithm programmed using fuzzy logic. To perform this task the MRAC controller compares the intensity that the AC motor consumes with that estimated by the FAM model; as a result of the comparison, an amplitude and phase error are obtained, from which the MRAC algorithm calculates the parameters' new values. This algorithm has been programmed from the study of the parameters variation effect over the amplitude and phase intensity consumed by the motor.

3.3 Tasks assignment

The tasks assignment is presented in figure 2. As can be seen, the DSP executes the inverter controller and the MRAC controller algorithms. The DSP main task is the MRAC controller, and is interrupted every 100 μs by the inverter controller algorithm, whose output signals can't be delayed. The 486 executes the speed controller and the FAM controller algorithms, monitors all the system and stores system variables (speed, torque, voltage,...). The AC motor speed and the current consumed are acquired using the A/D acquisition board. The control signals for the inverter gates are generated using 7 lines (6 gates, 1 enable) of the digital I/O board. With this task assignment, the 486 discharges the DSP computing load, allowing the experimentation of more complex algorithms.

3.4 Data exchange

The data exchange can be easily made by means of the DPRAM. The DSP provides the PC with the motor speed acquired by the A/D converter module, and the new AC motor parameters obtained by the MRAC controller. The PC provides the DSP with the desired voltage (amplitude, phase, frequency) that has to be applied to the motor. As one processor writes to the DPRAM without interrupting the other, this data exchange is made with no interaction between them.

4 Results: discussion of performance

To demonstrate system's performance, we have studied the control system's response to a ramp, using the FAM controller with its parameters not properly tuned (stator and rotor resistance, Rs and Rr respectively). In these experiments, the parameter identification loop (MRAC controller) is tuning the model parameters (_Rs, _Rr) that the FAM controller uses, meanwhile the speed control loop is controlling the motor speed. As we can see from the graphical results (figure 3), the MRAC controller tunes the model parameters ($\_R_s$, $\_R_r$) to the real ones (R_s, R_r) in a few seconds. It works properly even during transients in the speed control system. Furthermore, the parameters identification loop improves the system's performance, because it obtains the real AC motor parameters that the FAM controller needs. As we mentioned before, the parameters identification algorithm compares the real current consumed by the AC motor with that one estimated using the model. In order to measure the phase of the real intensity, a zero-pass detection circuit is used, which interrupts the system every cycle. This means that, at most, is possible to execute an identification cycle each period of the power supply signal. If we use a single processor, the system won't be able to execute the parameters identification algorithm so often, while is executing all the other control routines (that have to be executed to avoid the degradation of the control system performance), and the

identification time will be longer compared with that one of a parallel processing system (figure 4).

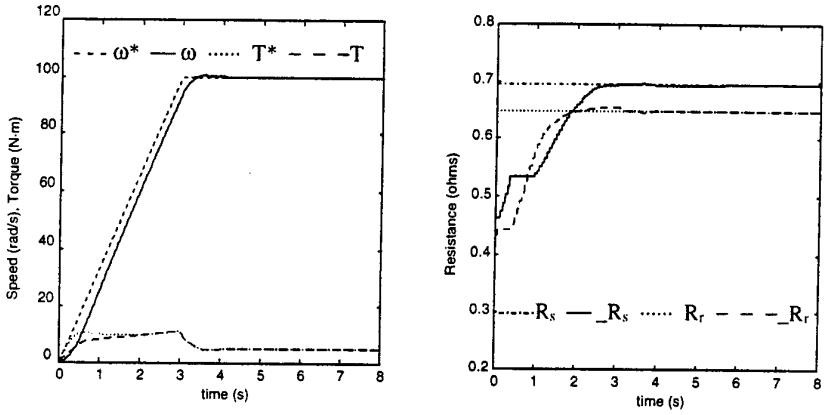


Fig. 3. Speed (ω) and torque (T) control system response to a ramp during parameters identification (R_s, R_r) with two processors

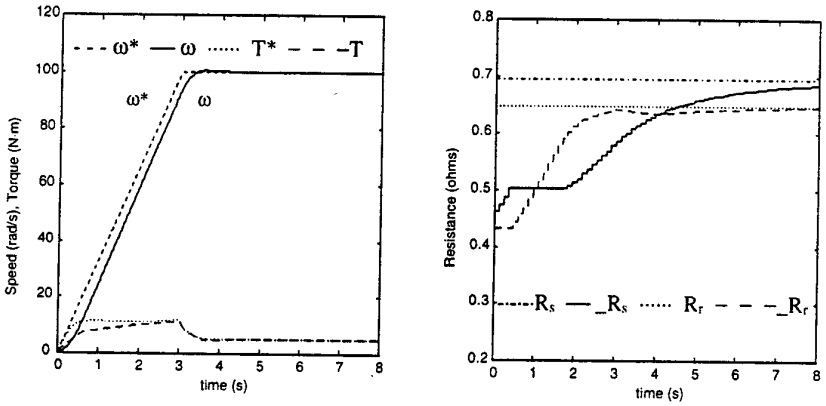


Fig. 4. Speed (ω) and torque (T) control system response to a ramp during parameters identification (R_s, R_r) with one processor

5 Conclusions

We have presented a parallel processing architecture with two processors running simultaneously: a 486 (PC) and a DSP. The latter is placed in the ISA bus, giving an interface with enough immunity to conducted and radiated interferences. This

architecture solves the data exchange between processors and allows the experimentation of an AC motor control system with two loops that have to be executed simultaneously: the speed control loop and the parameter identification loop. The main advantatge of this system is that we can reprogram the algorithms that one processor executes without changing the ones executed by the other processor. Also the system is capable of acquiring external signals (current consumed by the AC motor, DC bus voltage) and generating digital output signals (inverter control). The results presented show that the system formed by the two processors is able to control the AC motor speed and, simultaneously, tune the motor model parameters used by the FAM controller.

References

1. Chi-KwongLuk, P., Drissi El Khamlichi, D.: "An innovative DSP-based teaching module for electrical machine drives", IEEE Transactions on Education, vol. 39, N° 2, May 1996, pp 158-164.
2. Romeral, J.L., Bordonau, J., Bedford, D., Aldabas, E.: "Adaptive fuzzy speed controller for an AC drive", EPMC'96.
3. Romeral, J.L: "Optimización de modelos de control digital para motores AC", Doctoral Thesis, Electronic Engineering Department, UPC. June 15, 1995.
4. Papamichalis, P.E.: "Digital signal processing applications", Prentice Hall, 1990.
5. Yamamura, S.: "AC motors for high-performance applications. Analysis and control", Marcel Dekka, Inc., 1986.
6. Novotny, D.W., Lipo, T.A.: "Vector control and dynamics of AC drives", Clarendon Press, 1996.
7. Bose, B.K.: "Power electronics and AC drives", Prentice Hall, 1986, pp. 266-280.
8. Harris, C.J., Billings, S.A.: "Self-tuning and adaptive control: theory and applications", IEE Control Engineering Series 15, Peter Peregrinus Ltd, 1985.

Parallel Optimisation for Optical Lens Design

Enric Fontdecaba Baig *, José M. Cela Espín, and Juan C. Dürsteler Lopez

[1] Universitat Politecnica de Catalunya enricf@ac.upc.es
[2] Universitat Politecnica de Catalunya cela@ac.upc.es
[3] Industrias de Óptica S.A. Dus@indo.es

Abstract. This paper presents the parallelization of a non linear non constrained optimization code used in a industrial design, two different approaches are presented and the results of the comparison is shown.
Keywords: Non linear optimisation, Parallel Algorithms, Lens Design, Parallel Linear Solvers.

1 Introduction

In this paper we will discuss an industrial design problem, we will show the difficulties encountered and why a parallel approach was needed. Furthermore the parallel algorithm will be described, and the performance obtained also will be presented.

Industrias de Optica S.A. is the biggest Spanish lens manufacturer, the flagship product of the company is the progressive lens. This kind of lens is used to compensate the presbiopya, resulting from the aging of the eye. This product is growing its market share.

A progressive lens has three different vision zones, in one of them the user can see distant objects, in the second (intermediate vision zone) a progressive change of optical power is made in order to allow the wearer see all distances. The last zone is used in near vision. It is known that there is no analytical solution that gives the best possible progressive lens, so it is mandatory to use an optimization algorithm. [2]

In addition to these three zones, used in phoveal vision, there is a fourth zone, the lateral zone. All the effort in the optimisation process is devoted in reducing the astigmatism in this zone, improving the overall lens performance. In figure 1 the different zones can be observed.

In the Progressive Addition Lens design process, it is necessary to optimize the lens surface in every performed trial. This being an iterative process, it is very important to use the fastest possible algorithm. This is the motive that led us to a parallel approach.

* Also in Industrias de Óptica S.A.

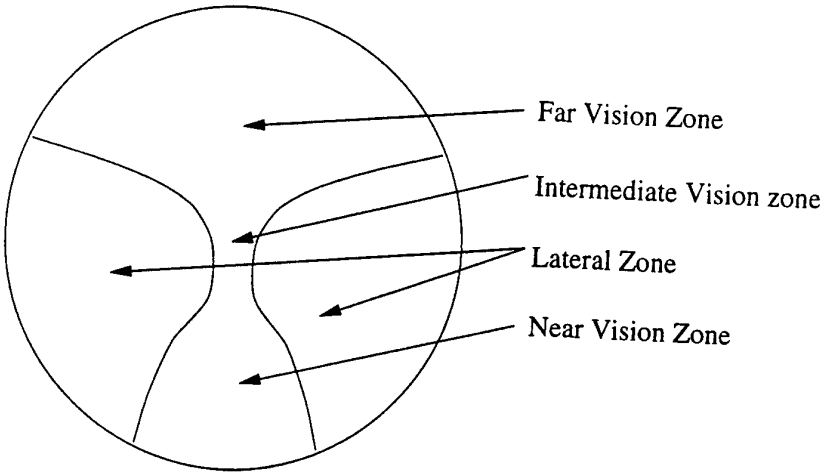


Fig. 1. Progressive Addition Lens vision zones

2 Mathematical approach

2.1 Lens Modeling

The issue in this optimisation problem is to build an appropiate surface, thus a surface modeling tool is needed. We use B-Splines as the basis to model the optical surface. With this basis, two interesting properties are achieved:

Local control. When moving a control point of a B-Splines model, only the neighbouring zones are modified.

C^2 **class surface.** The B-Splines are twice derivable. This property is needed in order to calculate the surface curvature. The optical power and the astigmatism, the lens properties measured, are related to the surface curvature.

This technique allows enough flexibility. Also, as our algorithms improve, it is possible to obtain a more accurate model increasing the number of control points.[3]

2.2 Optimization Algorithm

After a complete review of the different non linear non constrained optimisation algorithms available, the Newton algorithm was selected. The algorithms reviewed were, Polytope Method, Steepest Descent, Conjugate Gradient, Quasi-Newton and Newton.

In the numerical test it was clear the enormous possible gain we could achieve with an analytical derivative form of the cost function. We managed to obtain the derivative with exact formulas and had those analytical expressions programmed.

With this modification the calculation time for a Hessian was reduced enough to make the Newton algorithm preferable to a Quasi-Newton approach. [6] [4]

3 Parallelization approaches

The targeted platform was a workstation cluster, so we choose PVM as the message passing environment for the new application. [8]
In order to start the parallelization a profile of the algorithm sequential version was performed. As a result of this profile it was clear that the biggest part of the CPU time was spent on building the Hessian. Those routines where the first ones to be parallelized. With the first parallel program, performance measurements were done to study its behaviour. We used the analysis tools available on the CEPBA (European Center for Parallel Computing of Barcelona) [7], the Dimemas and Paraver tools, to perform those tests.

3.1 Objective Function Parallelization

The numerical test revealed that a very important part of the calculation time was spent in computing the objective function. Furthermore, the most important part is the Hessian computation. So, the first parallel approach faced the reduction of this time.

The Hessian is computed by finite differences of the gradient. In order to improve the performance, an analytical gradient routine was implemented. It is notable that mathematical packages like Mathematica or Maple failed to compute this analytic derivative.

In order to obtain a finite difference Hessian approach, it is necessary to calculate $n+1$ (n is the problem dimension) function gradients. Those calculations are independent, so they are splitt among the different available processors. A master-slave approach is used. The other computations needed by the algorithm, the linear search and the linear equations system, are computed by the master. In table 1 the speed-up results of different problem sizes are shown. The tests were performed for 2,4,8,12 and 16 processors in order to study the algorithm scalability.

Studying the code and profiles, it was clear that the algorithm bottleneck was the linear solver. The traces obtained in our performance analysis tool corroborate this conclusion. In order to improve the scalability, the parallelization of the linear system solver was decided upon.

3.2 Linear Solver Parallelization

In order to achieve a better scalability we parallelized the linear solver. We used preconditioned Krylov subspace iterative methods as linear solvers (Conjugate Gradient and GMRES(m)). The selected preconditioners are a set of different Incomplete Factorizations. The parallelization of the linear solvers is based on a

Table 1. Parallel Speed-Up with Function Parallelisation

Dimension	2 Proc	4 Proc	8 Proc	12 Proc	16 Proc
70	1.69	2.63	3.75	3.38	3.95
140	1.85	3.16	5.00	6.13	6.96
390	1.92	3.56	6.12	8.10	9.18
1390	1.95	3.78	6.90	9.08	12.39

Domain Decomposition data distribution. [1] The main bottleneck of the linear solver is the solution of the sparse triangular linear system arising from the pre-conditioner. The communication requirements of this operation depend on the block structure of the triangular factors. In order to minimize this bottleneck two strategies are used:

1. Control the fill-in at the block level with a different criteria than at the element level.
2. Perform a coloring of the domains which minimizes the fill-in at the block level and ensures the maximum parallelism.

Because the granularity of the Hessian assembly and the linear solver is quite different, we use a different number of processes in each phase. This means that additional communications are required to redistribute the data before and after the linear system solution phase. We must find for each problem size the optimum number of processes of each part in order to obtain the minimum execution time. In this way we can improve the scalability of the whole application.

The results are shown in table 2 and table 3. The results with the smaller data sets are not shown because due to their size they did not achieve any reasonable speed-up.

Table 2. Parallel Speed-Up with Function and Linear Solver Parallelisation. Using 2 processors in the Linear Solver.

Dimension	2 Proc	4 Proc	8 Proc	12 Proc	16 Proc
390	1.84	2.91	4.08	4.63	4.20
1390	1.97	3.58	6.06	7.88	9.27

Surprisingly, we achieve no increases in speed in parallelising the linear solver. Analysing the results and the code, we find two reasons for this behaviour:

Table 3. Parallel Speed-Up with Function and Linear Solver Parallelisation. Using 4 processors in the Linear Solver.

Dimension	2 Proc	4 Proc	8 Proc	12 Proc	16 Proc
390	1.73	2.58	3.45	4.02	3.34
1390	*No convergence*				

- As we use an iterative method, the number of iterations needed in order to solve the linear system is a key parameter. The parallelisation, involving a matrix reordering increased the number of iterations. Futhermore, in the bigger case (when we expected some performance improvements), the reordering affected the algorithm convergence in such a way that made it diverge.
- With the solver parallelisation, the number of communications is greatly increased. In the Hessian parallelisation there are two communications, at the beginning and the end of the parallel phase. With the linear solver, there is comunication in each linear solver iteration.

Summarising, the linear system involved in the optimisation algorithm is too small and too badly conditioned to be solved with a parallel iterative method.

4 Conclusion and Future Work

The speed-ups obtained are satisfactory for the industrial process. It is not expected to use more than 12 machines at the same time. In fact INDO is installing a network of 6 DEC Alpha workstation with a Fast Ethernet switch. Taking the previous results into account, with the targeted platform, the first parallel approach is the most suitable for the company.

It is also interesting to remark that the problems with the parallel linear solver. In our previous experience with linear systems from numerical simulations we have never found such a bad conditioned problem. In order to overcome this behaviour we are thinking about new reordering methods.

The future work includes an upgrade of the basic sequential algorithm, and the changes needed by this improved approach. We also want to study the possibilities of Quasi-Newton approaches to our problem.

5 Acknockledgements

We would like to thanks all the R & D staff of INDO for their support in the development of this project. Specially to Roberto Villuela for their contribution to the tests and in developing key parts of the code.

References

1. Cela, José M.; Alfonso, J. M.; Labarta, J.: *PLS: A Parallel Linear Solvers library for domain decomposition methods*: EUROPVM'96, Lecture Notes in Computer Science 1156, Springer-Verlag, 1996.
2. Dürsteler, Juan Carlos.: *Sistemas de Diseño de Lentes Progresivas Asistido por Ordenador.*: PhD Thesis. Universitat Politecnica de Catalunya, 1991.
3. Farin, Gerald.: *Curves and Surfaces for Computer Aided Geometric Design. A Practical Guide.*: Second Edition. Academic Press, 1990.
4. Gill, Philip E.; Murray, Walter & Wright, Margaret H.: *Practical Optimization.*: Academic Press, 1981.
5. Dennis & Schnabel.: *Numerical Methods for Unconstrained Optimization and Non Linear Equations.*: Prentice-Hall, 1983.
6. Nemhauser, G.L.; Rinnooy Kan, A.H.G. & Todd, M.J.: *Optimization.*: Elsevier Science Publishers B.V. 1989.
7. Labarta, J., Girona, S., Pillet, V., Cortes, T., Cela, J.M.: *A Parallel Program Developement Environment*: CEPBA/UPC Report No. RR-95/02 (1995)
8. Gueist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchel, R., Sunderam, V.: *PVM 3 User's Guide and Reference Manual*: Oak Ridge National Laboratory, TM-12187 (May 1994)

Supercomputer Optimized Microwave Domestic Oven Design via FD-TD

Gaetano BELLANCA [1], Paolo BASSI [1],
Giovanni ERBACCI [2], Gianni DE FABRITIIS [2] and Ruggero ROCCARI [3]

[1] Dipartimento di Elettronica Informatica e Sistemistica (D.E.I.S.)
University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy
[2] CINECA, Interuniversity Computing Center
Via Magnanelli 6/3, 40033 Casalecchio di Reno, Bologna, Italy
[3] De' Longhi Italia, Treviso, Italy

Abstract. Finite Difference Time Domain (FD-TD) is a numerical technique widely used to evaluate the electromagnetic field distribution in geometrically complicated devices. The explicit formulation and the intrinsic parallel structure of the FD-TD algorithm suggest the possibility to increase the code performance, particularly in terms of computation time reduction, using parallel architectures. In this paper, advantages in the design process of domestic microwave ovens via FD-TD on massively parallel computers are described and commented. Comparisons between the simulation times required using different workstations and the Cray-T3D parallel computer are finally reported.

1 Introduction

In the design of microwave ovens, overall performances in terms of heating uniformity of the load and energy conversion efficiency, user's safety and device cost reduction must be taken into account and optimized. The availability of a CAD tool is fundamental for oven designers. In fact, this allows not only to obtain improvements in heating uniformity and efficiency, but also to prevent possible microwave leakage and abnormal heating or arcing in the feeding system.
The Finite Difference Time Domain (FD-TD) method is a numerical technique that can be profitably used to investigate the electromagnetic (e.m.) behavior of a microwave heating applicator [1] [2]. Because of the complexity of the overall equations, and also the generally complicated geometry of the heating devices, the determination of the e.m. field distribution inside the oven could require many days of simulation on ordinary Personal Computers or Workstations [3]. To reduce the mathematical dimensions of the problem, some approximations can be taken into account, but this could introduce unacceptable loss of accuracy.
This bottleneck can be overcome using modern parallel computers. However, to obtain the best results from this architecture, the simulation code must converted in parallel form and correctly optimized. The FD-TD approach [4], being based on

explicit formulation with an intrinsic parallel structure of the solving equations, is well suited to take full advantage on this kind of architecture.
In the following, after a short introduction to the algorithm, the code parallellization will be discussed and its performances presented, showing the computation time reduction obtained on the CINECA's 128 processors Cray-T3D system.
This program will be used as the basic kernel for an European Community HPCN project, a demonstration action devoted to the introduction of High Performance Parallel Computers in the design process of domestic microwave ovens. The project is named POPCORN (Production Of Parallel Computer Optimized micRowave oveNs) and is managed by a consortium composed by De' Longhi, CINECA and D.E.I.S.

2 The numerical approach

The electromagnetic field inside a metallic microwave cavity representing the oven has been described by the Time Domain Maxwell's curl Equations. Differential operators have been written in difference form following the Yee's scheme [4]. The resulting equations for all the 6 field components (electric and magnetic) have the same form and differ only from the values of the multiplication coefficients, that are evaluated according to the dielectric properties of materials in each cell of the computational domain. As an example, the equation of the E_x field component can be written as:

$$\begin{aligned} E_x^{n+1}(i,j,k) = {} & C_1(i,j,k)E_x^n(i,j,k) + \\ & C_2(i,j,k)\left[H_z^{n+1/2}\left(i+\tfrac{1}{2},j+\tfrac{1}{2},k\right) - H_z^{n+1/2}\left(i+\tfrac{1}{2},j-\tfrac{1}{2},k\right)\right] + \\ & C_3(i,j,k)\left[H_y^{n+1/2}\left(i+\tfrac{1}{2},j,k+\tfrac{1}{2}\right) - H_y^{n+1/2}\left(i+\tfrac{1}{2},j,k-\tfrac{1}{2}\right)\right] \end{aligned} \qquad (1)$$

where n is the iteration time step, (i, j, k) represents the generic node of the discrete computational domain and the coefficients C_1, C_2 and C_3 are functions of both the local values of the dielectric properties and the spatial step increments along y and z.
These equations are well suited to be solved on a parallel computer. In fact, as it is easy to observe, the three electric field components do not depend from each other, but are only functions of the previous value of themselves in the same cell and of the magnetic field components in the surrounding cells. A similar result holds also for all the three magnetic field equations.

3 The parallel implementation

The Cray-T3D is a massively parallel system that integrates commodity microprocessors with a proprietary system interconnection network and high-speed synchronization mechanisms. Each Processing Element (PE) consists of a processor, the associated logic and a connection to the interprocessor communication network. The processor is a DEC Alpha chip 21064, a 64-bit RISC architecture with dual–issue, pipelining instruction stream, that provides 150 Mflop/s peak performance. Each PE is equipped with a direct-mapped cache of 8 Kbyte for the data, and with a

DRAM local memory of 8 Mwords (64-bit words). The global memory subsystem is a directly connected shared distributed memory architecture in which memory is globally addressable but physically distributed. The interconnection network is a 3D torus which operates asynchronously and independently from the PEs to access and redistribute global data. The 3D torus topology ensures short connection paths and high bisection bandwidth(300 Mbytes/s in every direction).
The original FD-TD code has been parallelized on the Cray-T3D using the *CRAFT work sharing* paradigm. A preliminary version of the parallelization scheme is reported in [5]. *CRAFT* is a Cray proprietary parallel programming model, similar to *HPF*, that allows the use of a global address space and supports the *SPMD* (Single Program Multiple Data) programming style. The same program is loaded and executed in all the PEs, but controlled by processor number and data. *CRAFT* is based on directives to the Fortran compiler, to express data and work distribution among the PEs, and it is efficient and easy to use. Unfortunately the portability is restricted only to the Cray-T3D massively parallel systems [5]. One of the tasks of the POPCORN Consortium is to overcome this limitation. In order to accomplish this task, the FD-TD code will be parallelized also using the *MPI message passing* paradigm, a more general and portable parallel programming model than the *work sharing* one. In this way the program will be ported on different parallel architectures to investigate the performances that can be reached even on a cluster of PC's, thus making this tool practically useful for the Research and Development division of an industry.

4 Results

In the structure of the developed FD-TD simulator three main sections can be identified: *pre-processing*, *field evaluation* and *data output*.
Data input and initialization of all variables are the activities of the first section. Information related to the physical structure of the computational domain (dimensions, e.m. properties of the considered materials, used mesh, etc.) are obtained reading a binary file produced by an external program used for the modeling. Then, once all the dielectric properties of each mesh point are known, values of all the variables used for e.m. field evaluation can be prepared. The second section contains the field computation procedures, based on the Yee's algorithm for the inner domain and boundary conditions for the outer faces. Also field excitations is performed in this section. Output binary files are used for final post-processing procedures.
As an example, the FD-TD approach has been used to simulate the behavior of a domestic microwave oven represented by 32 x 32 x 32 cells and for a temporal evolution of 1000 time steps. Simply adapting the existing code to the parallel machine, we have observed that the simulation times in all the parallel regions scale very well with the number of the used processors. However the global performances are always limited by the unoptimised sequential I/O procedures, which shown an almost random contribution to the overall simulation time. The solution to this problem has been obtained modifying the I/O routines, increasing the number of data associated to each I/O request (Fig. 1).

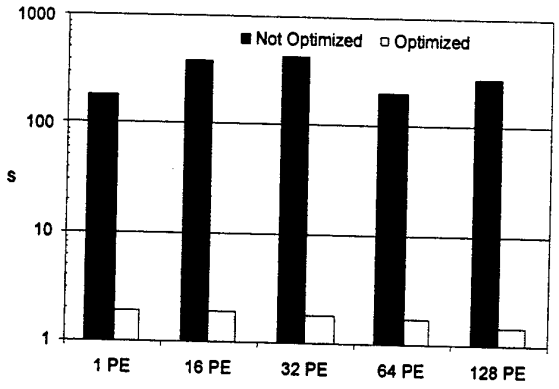


Fig. 1. Comparisons between the computation times (Log scale) required by the I/O procedures of the parallel FD-TD simulator before and after the optimizations.

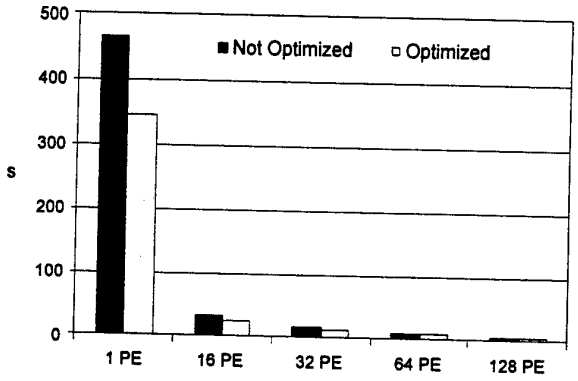


Fig. 2. Computation times before and after the optimizations of the field evaluation section using different number of PEs.

Other optimizations have also been introduced to increase the computation speed on the Cray-T3D parallel computer. This has been done modifying the data structure of the coefficients used in the Yee's field equations, avoiding the so called *cache miss* phenomenon. With this solution we have doubled, in terms of Mflop/s, the performances of each PE. For the main computational part of the code the improvements shown in Fig. 2 have been obtained.

The speed-Up of each section of the code as a function of the used PEs is reported in Fig. 3. This speed-up has been evaluated as the ratio between the simulation time required to perform a given procedure on a single PE and the time required to perform the same part of the code in parallel. As it is possible to see, parallel procedures (Yee's coefficient preparation (PrepcY) and field computations (Calc)) scale accordingly with the number of used PEs, confirming the good implementation of the

code. Loss of efficiency results for operation defined over data subspaces (as, for example, those related to preparation of coefficients for the boundary conditions, indicated as PrepcB, and those related to field excitation and boundary field evaluation, which influence the behavior of the Calc procedures). The resulting performance, however, can be considered satisfactory.

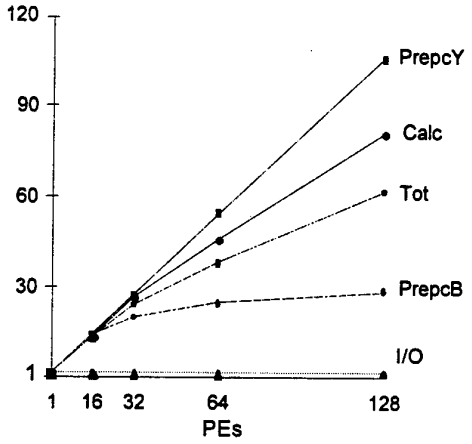


Fig. 3. Speed-Up of the different procedures of the FD-TD simulator vs the used PEs.

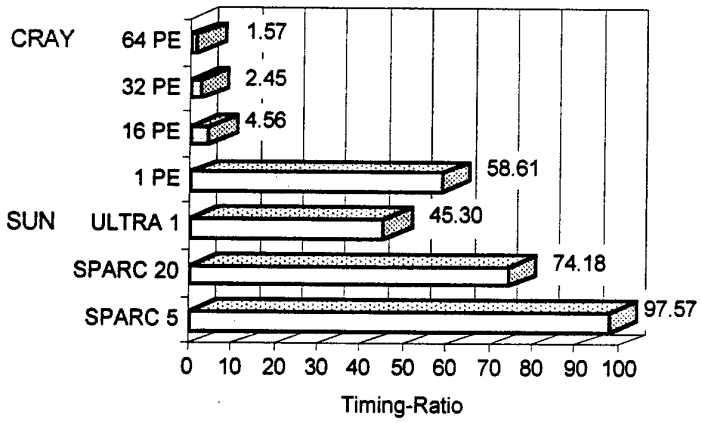


Fig. 4. Timing ratio of the 128 PEs Cray-T3D system respect the same system with different number of PE and some SUN workstations.

Using this code, the behavior of a more complicated microwave domestic oven with different load situations has been simulated [6]. For a 7500 time step run of a

64 x 64 x 64 mesh and 128 PEs, the overall CPU time has been reduced to 215 s respect to the about 9 h and 3.5 h required respectively by a Sun SPARCStation 20 and a Sun ULTRA 1 workstations. Obtained Speed-up are reported in Fig. 4.

5 Conclusions

In this paper, advantages in the design process of domestic microwave ovens using massively parallel computers have been described and commented. Comparisons between simulation time required by different workstations and the Cray-T3D parallel computer have been reported, to show the obtained performance increments. Speed-up of 59 and 154 have been shown comparing Cray-T3D 128 PE and Sun's ULTRA1 and SPARC20 workstation's results. This code will be used as the basic kernel for the POPCORN European Community project. The FD-TD simulator will be ported to the new CINECA's Cray-T3E parallel computer and on a PC cluster using the *message passing* paradigm (*MPI*), to investigate the level of performance that can be reached and to make this tool available for industrial Research and Development divisions engaged, for example, in domestic microwave oven design.

Acknowledgments

This work was supported by the "POPCORN" ESPRIT Project n° EP 27213.

References

1. Sundberg M., Risman P.O., Kildal P-S, Ohlsson T.: Analysis and Design of Industrial Microwave Ovens using the Finite Difference Time Domain Method. Journal of Microwave Power and Electromagnetic Energy, Vol. 31, No. 3, (1996), 142-157.
2. Bellanca G.: Microwave Domestic Ovens Performances evaluated by FD-TD Simulations. Rapporto '95, Scienza e Supercalcolo al CINECA, (1995), 361-368.
3. Bellanca G., Botti S., Erbacci G., Ansaloni R.: Advances in Microwave Ovens design via FD-TD on Supercomputers. VI International Conference "Microwave and High Frequency Heating", Fermo (AP), 9-13 Sept. 1997, 7-10.
4. Yee K.S.: Numerical Solution of initial boundary value problems involving Maxwell's equations in isotropic media. IEEE Trans. Ant. Prop., Vol. 14, No. 5 (1966), 302-307.
5. Erbacci G., Bellanca G., Botti S., Ansaloni R.: Supercomputing based Microwave Ovens Design via FD-TD, II Italian-Latinamerican Conference on Applied and Industrial Mathematics (ITLA '97), Rome, Jan. 27-31, CNR/GNFM n° 53 (1997), 90-93.
6. Pase D. M., MacDonald T., Meltzer A.: The CRAFT Fortran Programming Model", Scientific Programming, Vol. 3, (1994), 227-253.
7. Bellanca G., Botti S., Bassi P., Falciasecca G.: Sensitivity of FD-TD simulations to Small Mesh Modifications in Microwave Ovens design. VI International Conference "Microwave and High Frequency Heating", Fermo (AP), 9-13 Sept. 1997, 60-63.

Debugging Message Passing Parallel Applications: a General Tool

Ana Paula Cláudio[1], João Duarte Cunha[2], Maria Beatriz Carmo[1]

[1]Faculdade de Ciências da Universidade de Lisboa
Departamento de Informática - Campo Grande - Edifício C5 -Piso 1- 1700 LISBOA - Portugal
{apc, bc}@di.fc.ul.pt
[2]Laboratório Nacional de Engenharia Civil
Av. do Brasil, nº 101, 1799 LISBOA CODEX - Portugal
jdc@lnec.pt

Abstract. The paper describes a general purpose tool for the debugging of message passing parallel applications. The basic components of this tool are the trace/replay mechanism, the graphical user interface and the central component, called visualization engine. The engine, which plays the central role during the replay phase, can be used with different message passing environments and different graphical interfaces. This is a significant step to ensure a wider range of usability. Also relevant is the fact that this engine is able to learn how to detect predicates.

1 Introduction

Debugging sequential programs is not an easy task and it is common knowledge that the insertion of print statements is one of the most popular debugging techniques.

Henry Lieberman calls debugging "the dirty little secret of computer science" and concludes that it is still, largely, a matter of trial and error [10]. The fact that the April 97 issue of "Communications of the ACM" is entirely dedicated to debugging, proves how relevant the subject is. The debugging problem has largely been ignored what contrasts sharply with the remarkable progress in software development over the last thirty years [3].

Debugging parallel applications is even more difficult than debugging sequential programs due to non-determinism caused by race conditions. These conditions happen since processes in a parallel application must communicate with one another.

That is why our tool focuses on communication events. The tool includes a replay mechanism and a graphical interface. Between these two components, a central component, the visualization engine, makes the tool easily adaptable to different message passing mechanisms and different graphical environments.

2 Comparing Similar Tools

In November 1993 a group named *Parallel Tools Consortium* [1] was established, whose "mission is to take a leadership role in defining, developing, and promoting parallel tools that meet the specific requirements of users who develop scalable applications on a variety of platforms". According to this consortium, parallel program debuggers, execution trace visualizers, and tools for performance tuning, are subgroups that form a larger group named *Execution Analyzers*. Besides this, there are two more groups: *Source code analyzers* which are used to analyze and convert serial programs to parallel code and *Parallel languages and libraries.*

The usage of execution analysis tools is mandatory for programmers to obtain correct and tuned parallel programs and it takes place after the usage of any tool from the other groups. Among those, debuggers have to be used before execution trace visualizers and tools for performance tuning. There are myriades of tools of these sorts, therefore, one can only mention a limited number of them.

Among execution trace visualizers and tools for performance tuning we can mention AIMS[2], mp2sddf[2], ntv[2], Pablo[2], VT[2], Paragraph [6], Forge[2], XProfiler[2], Paradyn[3], PATOP[3], Poet [7]. The following belong to the group of debugging tools: xpdbx[2], TotalView[2], DETOP[3], Xmdb[3].

Our tool is intended to be independent of the message passing software. However, it is being tested for PVM applications so, it makes sense to mention execution analizers exclusively applicable to this message passing system: Xpvm[2], Hence[4], PVaniM [11], Xab3[3], DBPVM[3], TAPE/PVM[3], DDBG [4] and TOOL-SET [12]. The last one comprises a set of integrated tools, among them the debugger DETOP and the performance analyser PATOP, previously mentioned.

A complete description of all these tools and a detailed comparison with the one described here, is outside the scope of this paper. Nevertheless, it is possible to identify two of its distinctive features. First, it incorporates both a replay mechanism and a graphical representation, and second, its basic component, the visualization engine, builds an object-oriented model of the message passing application. Taking full advantage of inheritance and polymorphism, the tool becomes easily adaptable to different message passing softwares and/or to different graphical representations or graphical softwares.

Besides, due to the adoption of the object-oriented paradigm, the tool is flexible enough to acquire an important additional skill: predicate detection.

---

[1] http://www.ptools.org

[2] Links to a site containing information about this tool can be obtained in http://www.tc.cornell.edu/Parallel.Tools/exec-analysis-tools.html

[3] Links to a site containing information about this tool can be obtained in http://www.cse.ogi.edu/DISC/projects/mist/related-work/monitoring.html

[4] Links to a site containing information about this tool can be obtained in http://www.henceedp.com/

3 Our Tool

As explained before, the tool includes three components: a replay mechanism, a graphical interface and a central component named visualization engine.

The replay mechanism makes a particular execution repeatable, allowing cyclic debugging, a frequently used technique in sequential programs. The replay mechanism adopted is similar to the one described in [9] for applications based on the shared memory paradigm. Assuming that the individual processes in the parallel application do not contain nondeterministic statements, this mechanism is based in the principle that if each process is supplied with the same input values, in the same order, during successive executions, it will exhibit the same behaviour each time. The mechanism includes two distinct phases: trace phase and replay phase. In the trace phase, minimal information is stored in order to minimize the probe effect. Although minimal, the stored information is enough to assure that, during the replay phase each process will consume the same messages, in the same order.

It should be emphasized that it is not necessary to modify the code of a parallel application to use this debugging tool. The monitoring code is inserted in the standard libraries of the message passing software, which should not be modified by the common user. In the trace phase, the application under study must be linked with one modified library (trace library); for the replay phase, it must be linked with a second modified library (replay library).

During the replay phase, the visualization engine builds an object-oriented model of the application. The model provides the necessary semantic feedback to answer most of the questions the user may ask about the application, during and after replay.

The engine contains two sorts of classes: classes that define the building blocks of the model (namely, class *Process* and class *Message*) and management classes. In this last group, the most important classes are class *Manager* and class *Agent*.

There is one *Agent* executing in each machine that is running processes of the replaying application. Each *Agent* receives information from the local processes and sends it to the object in charge of building the object-oriented model of the application and maintaining its coherence along the replay. This object is an instance of a class derived from *Manager*.

In order to support different graphical representations or different graphical environments we take profit of inheritance, a major property of object-oriented models. A Graphical Interface Manager (*GIManager*), derived from *Manager*, contains the knowledge necessary to deal with the graphical interface. Similarly, the model contains classes *GIProcess*, derived from *Process*, *GIMessage*, derived from *Message* and so on.

In this way, data and code that depend on the graphical interface are encapsulated inside GI classes. On the other hand, everything that depends on the message passing software used by the parallel application, is encapsulated inside class Agent. Agents must be able to understand the message passing "dialect" .

Inheritance will be adopted again, this time to teach the model how to detect predicates [2]. In order to achieve this feature, for each specific predicate new specific classes, subclasses of the classes in the model, will be defined.

These classes inherit the behaviour of their superclass(es) and additionally know how to detect that predicate. For each predicate, particular information has to be collected and processed. Therefore, those classes must contain specific attributes and methods. Some of these methods will be overridden methods giving rise to polymorphic behaviour.

Two granularity levels for the observation of a parallel application are defined:
Level 1: external events level
External events, that is, communication events, are observable.
Level 2: internal events level
Internal events, concerning each individual process, are observable, together with communication events.

Our tool directly supports level 1. However, it is prepared to support level 2, as long as a sequential debugger is integrated. This kind of integration has been accomplished in similar tools [4].

A message has a source, one or several destinations[5], a tag and a body. Accordingly, the sort of bugs that an user is able to detect, using a tool which supports level 1, are:

- Bugs concerning one message
  - On the source side
    wrong destination;
    wrong tag;
    wrong body.
  - On the destination side
    wrong source;
    wrong tag.
- Bugs concerning all messages
  race conditions.
- Bugs concerning communication primitives
  wrong type of primitive.

Each of the following examples illustrates one of the previous sort of bugs: a process disturbs the application's expected behaviour because it has sent a message to the wrong destination (this one is a bug concerning one message, on the source side); a process waits for a message that will never arrive, meanwhile the correct message has arrived and will not be consumed (this is a bug concerning one message, on the destination side); the programmer intended to develop a race-free application but, in fact, he did not (this is a bug concerning all messages); the user intended to use a blocking receive and instead used a non-blocking one (this is a bug concerning communication primitives).

[5] A source or a destination is a process identity.

With level 1 tools, detectable predicates are those properties that depend exclusively on variables associated with communication events. For instance, suppose that process P1 processes a n-dimensional matrix, and after having processed each line sends it to process P2; the property " has process P2 received exactly n messages from process P1?" is a detectable one.

General predicates will be detectable as long as the level 2 of granularity observation is guaranteed.

The tool has been tested with PVM applications [1] (PVM [5] supports message-passing paradigm); C++ was used to develop the visualization engine and OSF-Motif for the graphical interface.

Although our debugging tool is easily adaptable to different graphical interfaces, we have started with a rather simple representation, the time space-diagram [8]. We made this choice because we think that a complex representation disturbs user's attention. He spends more time trying to understand all the symbols than focusing his mind in what really matters: the parallel application.

REFERENCES

1. Cláudio, A., Cunha, J.: Monitoring and Debugging of Message Passing Parallel Applications. In Proceedings of the 5th International Conference on Educational, Practice and Promotion of Computational Methods in Engineering Using Small Computers, Macau, August 1995
2. Cooper, R., Marzullo, K.: Consistent Detection of Global Predicates. In: ACM/ONR Workshop on Parallel and Distributed Debugging- ACM Press Sigplan Notices, Vol. 26, No. 12, 1991
3. Crawford, D.: Editorial Pointers. In: Communications of the ACM, April 1997, Vol. 40, No. 4, page 5
4. Cunha, J., Lourenço, J., Antão, T.: An Experiment in Tool Integration: The DDBG Parallel and Distributed Debugger. Submitted to Euromicro Journal of Systems Architecture
5. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM: Parallel Virtual Machine. MIT Press, 1994
6. Heath, M., Etherridge, J.: Visualizing the Performance of Parallel Programs. IEEE Software, September 1991
7. Kunz , T., Black, J., Taylor, D., Basten, T.: Poet: Target-System-Independent Visualisations of Complex Distributed Appliaction Executions. In:Computer Journal, Vol. 40, No. 8, February. 1998
8. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. In: Communications of the ACM, July 1978, Vol. 21, No. 7, 558-565
9. Leblanc, T., Mellor-Crummey, J.: Debugging Parallel Programs with Instant Replay. In: IEEE Transactions on Computers, Vol C-36, No. 4, April 1987
10. Lieberman, H.: The Debugging Scandal and what to do about it. In: Communications of the ACM, April 1997, Vol. 40, No. 4, 26-29

11. Topol, B., Stasko, J.: PVaniM 2.0: Online and Postmortem Visualization Support for PVM, June, 1996
12. Wismüller, R., Ludwig, T., Bode, A., Borgeest, R., Lamberts, S., Oberhuber, M., Röder, C., Stellner, G.: The TOOL-SET Project: Towards an Integrated Tool Environments for Parallel Programming. In: Proc. of the 2nd Sino-German Workshop on Advanced Parallel Processing Technologies, Koblenz, Germany, September 1997

Parallel *Ensemble-Averaged* Molecular Dynamics Simulation of Shock Wave on Distributed Memory Multicomputers

Sergey V.Zybin *

High Energy Density Research Center, Institute for High Temperatures, Russian Academy of Sciences, Izhorskaya 13/19, Moscow 127412, Russia.

Abstract. In this paper we present a simple parallel algorithm for *ensemble-averaged* molecular dynamics simulation of non-stationary transport processes in Lennard-Jones systems on distributed memory MIMD multicomputers. This algorithm has been used for simulation of shock wave in two- and three-dimensional solids and calculations of ensemble-averaged particle distribution functions of kinetic and potential energy as well as the pair correlation functions for several cross sections within the shock layer. The algorithm is based on parallel simulation of independent systems from a canonical ensemble on different processors allowing a computation of the ensemble-averaged structural and thermodynamic properties. We have implemented the algorithm in the PVM programming environment and performed simulations on various multicomputers.

Keywords: parallel computing, molecular dynamics, shock wave, PVM.

1 Introduction

The molecular dynamics (MD) is a powerful simulation tool for studing structural and dynamical properties of liquids and solids. Recently, more attention has been focused on understanding the molecular mechanisms of nonstationary macroscopic processes such as shock wave [1, 6], detonation [8], fracture and failure [3], partly due to the advent of massively parallel computers.

In the present work we apply the MD method for simulation of a planar shock wave in Lennard-Jones solid. The principal limitation to such simulation is that the shock layer properties can vary significantly within a few lattice spacings. In the most general case, both the space- and time-dependences of all the dynamical quantities need to be considered. Thus, sufficiently large cross-sectional area is required to reduce large nonphysical fluctuations. Up to now, the number of atoms per transverse plane was typically $10^2 - 10^3$ which is not sufficient for reducing the fluctuations considerably. Owing to these fluctuations, important characteristics of the shock layer, such as the evolution of velocity distribution function across the layer, have not been well studied. One way to improve the

* Present address: Universidade Federal de Sergipe, DEI/CCET, 49100-000, São-Cristóvão - SE, Brazil, e-mail: zybin@sergipe.ufs.br. This research was supported by Russian Foundation for Basic Research, grant 96-01-01901.

quality of simulation is to take a time average, but it is possible for modeling only steady shock waves. It has been employed in [1] through the use of special potential configuration, which makes it possible to generate a steady shock wave at rest in the laboratory frame. Recent advances in parallel computers provide a means for multi-million atoms simulations of such nonstationary processes [2, 3], which enables one to extend considerably the cross-sectional area. However, the implementation of message-passing multi-cell MD on massively parallel computers [2, 9] usually involves intensive interprocessor communications on each time step and possible non-uniform workload of processors. It complicates the implementation of spatial-decomposition technique on less sophisticated and cheap heterogeneous multicomputers such as network-connected clusters of workstation or PC-clones coupled with free PVM/Linux software.

Here we implement an alternative ensemble-decomposition approach that consists in taking a statistical average over canonical ensemble by repeating the shock wave simulations with different initial conditions. An advantage of this approach is a straightforward implementation on parallel computers with virtually no interprocessor communications, where each processor is responsible for independent simulation. It has been applied in modeling a shock wave in Lennard-Jones crystal with $10^2 - 10^3$ atoms in the cross-sectional area and 10^2 simulation runs. The time-dependent profiles for density, velocity, mean square fluctuations of the longitudinal and transverse velocity components, internal energy and pressure tensor were obtained. We also measured the velocity distribution functions, the probability density for the potential energy and the pair correlation functions in several transverse planes within the shock layer.

2 Parallel ensemble-averaged MD algorithm

We have developed a parallel algorithm of ensemble-averaged MD method in the PVM programming environment for simulation of shock wave in the fcc lattice composed of atoms interacting via Lennard-Jones (6-12) potential $U(r) = 4\varepsilon[(\sigma/r)^{12} - (\sigma/r)^6]$. The program was initially developed in the PVM on distributed shared memories machine CONVEX SPP-1000 and then adapted on IBM SP2 RS/6000 and the network-connected PC-clone. The algorithm has the "master-slave" parallel structure presented by the following scheme

Master
1.Initialization: Compute initial data and send to K **Slaves**
2.The beginning of parallel computations
for n=1 to number of simulation steps **pardo**
receive from **Slaves** binned profiles of variable α_k, $k = 1, ..., K$
compute ensemble average $\langle \alpha; f \rangle = \frac{1}{K} \sum_k \alpha_k$ (or by Metropolis procedure)
end pardo
3.Repeat the step 2 if required
4.Kill Slaves and finish the computations

Slaves

1.Initialization: Receive from **Master** initial data
2.Computations in **Slave k** (k=1,...,K)
  for n=1 **to** number of simulation steps **do**
    compute forces $f(\mathbf{r}_i^{n-1}) = -\sum_{j \neq i} \frac{\partial U(r_{ij})}{\partial r_{ij}}$, **move** atoms to new positions $\mathbf{r}_i^n$
    compute binned profile of dynamical variable α_k
    send α_k to **Master**
  end do
3. Repeat the step 2 if required

The algorithm consists in concurrent simulations of different systems from a canonical ensemble generated by randomization of the initial velocities of atoms. From time to time, the binned spatial profiles of a dynamical variable α are calculated in each simulation subtask. Then the averaging over K systems of ensemble is performed yielding the expectation value $\langle \alpha; f \rangle$ for a distribution function f. The theoretical speed-up for the algorithm presented above is

$$\text{Speed-up} = (\tau_{comp} K)/(\tau_{comp} + \tau_{comm}/Nx),$$

where Nx is the number of atoms in cross-sectional area, τ_{comp}, τ_{comm} – the parameters responsible for computation and communication time. As the computational experiments show, the communication time is negligible small in comparison to the computation time (see Figure 1).

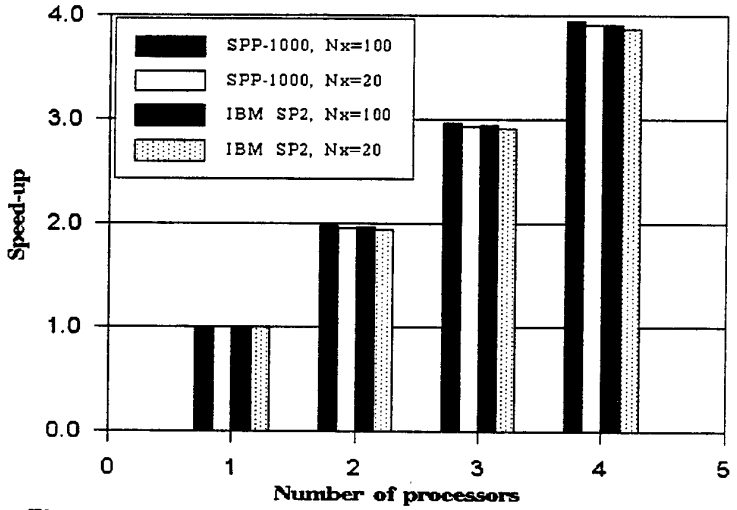


Fig. 1. The speedup obtained on different computer architectures: (a) single hypernode of the CONVEX SPP-1000 (4 processors), (b) 4 IBM RS/6000 POWER2 networked workstations (for different numbers Nx of atoms in cross-section).

It should be noted that any standard MD program optimized for sequential execution can be readily implemented in this algorithm with minor changes. However, there are two cases where its use becomes impracticable: when it is necessary to use a large system size (due to the memory limitations on the number of atoms) and when the simulation time is longer than can be realistically achieved using a single processor (due to the large update time per atom). In such situations the best approach is the parallel spatial-decomposition MD technique.

3 Simulation results

The algorithm has been used for a simulation of shock wave in a lattice composed of argon atoms ($m = 40$ $a.u.$, $\sigma = 3.4$Å, $\varepsilon/k_B = 120$ °K). The rectangular simulation cell had the length of $100-150$ unit cells ($200-300$ planes of atoms) in the z direction of shock propagation. The transverse x dimension was usually $50-100$ unit cells with periodic boundary conditions imposed along the x axis. The initial density n_0 was chosen to be $0.93-1.03$ and the temperature $T_0 = 0.1$.

A planar shock wave is initiated by causing a few atom planes to move with a constant piston velocity u_p in the z direction. During a simulation the piston atoms are constrained to remain at their moving lattice sites. The time-dependent profiles for velocity, density, mean square fluctuations of the longitudinal and transverse components of atom velocity ("kinetic temperature" components), internal energy, and pressure tensor were obtained. We also measured the pair correlation functions, the distribution functions of the velocity components and the probability density for the potential energy in several planes $z = const$ within the shock layer at different times for describing the evolution of the lattice structure during the shock compression.

The simulation cell is divided into bins along the z direction to obtain the shock-wave profiles. Typically the number of bins was equal to twice the number of unit cells in uncompressed lattice, giving a bin width $0.87\sigma-0.96\sigma$. The local properties at a point are obtained by taking a spatial average over a bin around point and an average over the systems of an ensemble. We have followed the approach [4] based on the formulas given in [7] for the expectation value $\langle \alpha; f \rangle$ of dynamical variable α over an ensemble having distribution function f. It is assumed that a local property dependent directly on atomic position, such as the mass density, is given by

$$n(\mathbf{r},t) = \sum_i \langle m_i \, \Delta(\mathbf{r}_i - \mathbf{r}); f \rangle, \quad \Delta(\mathbf{r}_i - \mathbf{r}) = \begin{cases} 1/(Sd), & \text{if } z - \frac{1}{2}d < z_i < z + \frac{1}{2}d, \\ 0 & \text{,otherwise,} \end{cases}$$

where d is the bin width, S is the area of cross section of MD cell, and $\Delta(\mathbf{r}_i - \mathbf{r})$ is the localization function (in [7] the Dirac's δ-function was used). For a local property dependent on interatomic separation $\mathbf{r}_{ij}$, such as the stress tensor, the interaction of atoms on the opposite sides of S are taken into consideration

$$\sigma(\mathbf{r},t) = -\sum_i \langle m_i \, (\mathbf{v}_i - \mathbf{u})(\mathbf{v}_i - \mathbf{u}) \Delta(\mathbf{r}_i - \mathbf{r}); f \rangle + \frac{1}{2} \sum_{\substack{i,j \\ i \neq j}} \left\langle \frac{\mathbf{r}_{ij}\mathbf{r}_{ij}}{r_{ij}} \frac{\partial U(r_{ij})}{\partial r_{ij}} \Delta(\mathbf{r}_i - \mathbf{r}); f \right\rangle$$

where $\mathbf{u}(\mathbf{r},t)$ is the mean velocity in the bin centered about $\mathbf{r}$. All the generated systems are accepted for taking ensemble averages, implying the constant distribution function. The fluctuations of internal energy measured for different systems of an ensemble were sufficiently small in the simulation. Besides, one can use the canonical distribution function by introducing a Metropolis procedure to accept or reject the new realization at a given time t as in [5].

Figure 2 shows some simulation results of typical example of shock wave in 2D lattice with 100 atoms in the cross section. The parameters of simulation (piston velocity $u_p/c_0 = 0.7$, where c_0 - longitudinal zero-temperature sound speed, Mach number $M = 3$, compression $n_1/n_0 \approx 30\%$) are representative for rather strong steady shock wave. The simulation makes it apparent that the fluctuation of the longitudinal velocity component, $T_n \sim \sum_i (v_{iz} - u_z)^2$, grows faster than the fluctuation of the transverse component $T_t \sim \sum_i (v_{ix} - u_x)^2$. A similar phenomenon has been observed previously [1, 6]. The difference between T_n and T_t leads to the anisotropy of pressure within the shock layer and to the effect similar to the surface tension [1]. The evolution of the velocity component v_z distribution function across the shock layer reveals significant deviation not only from the Maxwellian equilibrium distribution but also from the corresponding bimodal distribution. The virial terms of normal P_n' and tangent P_t' components, and the difference between them are also presented as well as the evolution of potential energy distribution function across several planes $z = const$ within the shock layer. The simulation results were obtained for 200 systems from an ensemble showing a considerable reduction in statistical fluctuations.

The experiments on network-connected multicomputers in PVM environment confirm an efficiency of the algorithm for obtaining ensemble averages of the time-dependent dynamical variables. The three-dimensional ensemble-averaged MD simulation of a shock wave in solid states are currently in progress.

The computational resources were provided by the Keldysh Institute of Applied Mathematics of Russian Academy of Sciences and the National Center of Supercomputing of the Federal University of Rio Grande do Sul. I would like to thank S.I.Anisimov and V.V.Zhakhovskii for encourangement, support and many useful discussions concerning this work.

References

1. S.I. Anisimov, V.V. Zhakhovskii, and V.E. Fortov. JETP Lett. (1997).
2. D.M. Beazley and P.S. Lomdahl. *Paral. Comput.* **20**, 173 (1994).
3. P. Gumbsch, S.J. Zhou, and B.L. Holian. *Phys. Rev. B* **55**, 3445-3455 (1997).
4. R.J. Hardy. *J. Chem. Phys.* **76**(1), 622-628 (1982).
5. D.W.Heerman, P.Nielaba, and M.Rovere. *Comp. Phys. Comm.* **60**, 311-318 (1990).
6. B.L. Holian. *Phys. Rev. A* **37**, 2562-2568 (1988).
7. J.H. Irving and J.G.Kirkwood. *J. Chem. Phys.* **18**, 817-829 (1950).
8. J.W. Mintmire, D.H. Robertson, and C.T. White. *Phys. Rev. B* **49**, 14859 (1994).
9. S. Plimpton. *J. of Comput. Physics* **117**, 1-19 (1995).
10. N.J. Wagner, B.L. Holian, and A.F. Voter. *Phys. Rev. A* **45**, 8457-8470 (1992).

This article was processed using the LaTeX macro package with LLNCS style

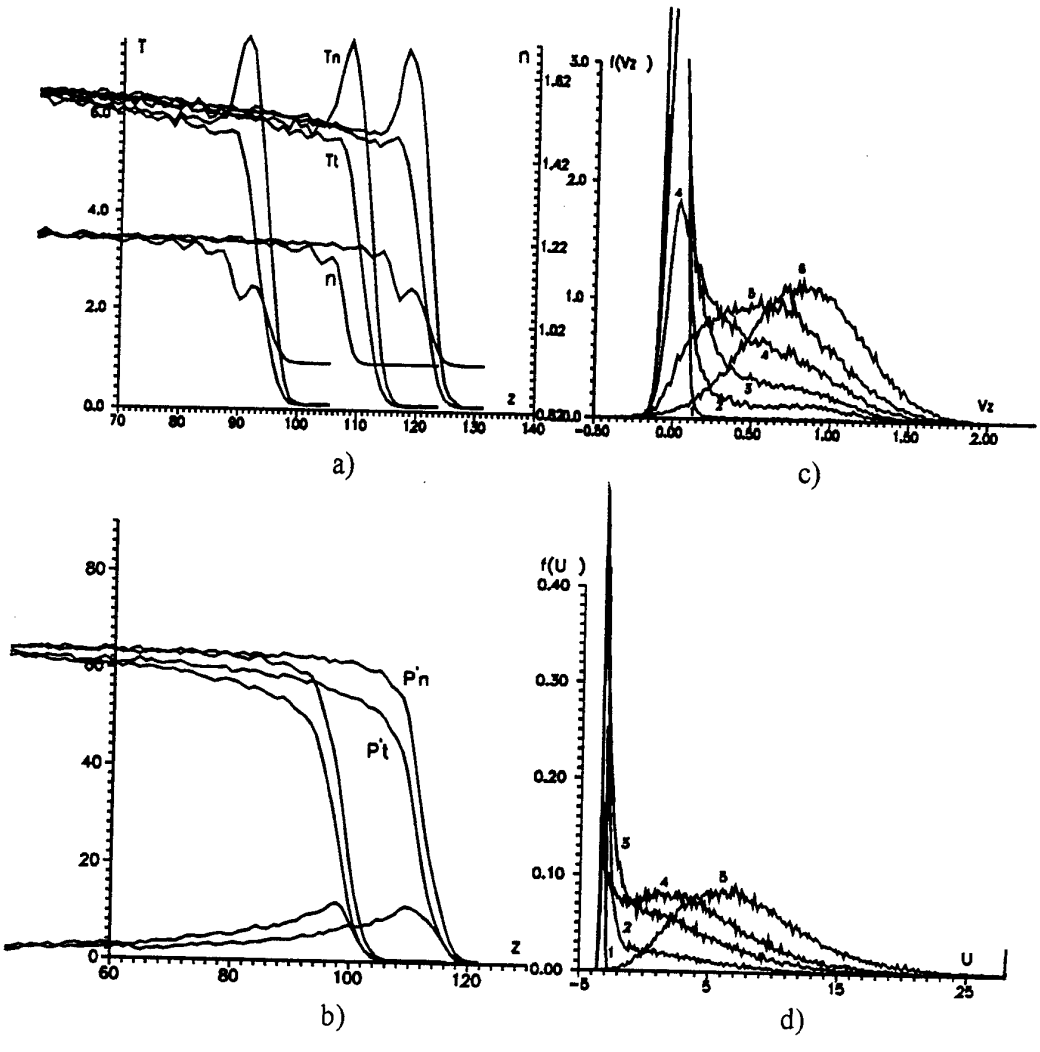


Fig. 2. a) Spatial profiles of mean-square fluctuations of the longitudinal T_n and transverse T_t velocity components with corresponding profiles of the density n. b) Spatial profiles of normal (P_n') and tangent (P_t') components of potential contribution to the pressure tensor. Distance z from the piston is given in σ units. c) Distribution functions of the longitudinal v_z velocity component in different layers normal to z-axis d) Distribution functions of the potential energy in different layers normal to z-axis. Layers are numbered from upstream to downstream. Piston velocity $u_p = 0.7c_0$ (c_0 – longitudinal zero-temperature sound speed). Shock velocity is $3.0\,c_0$, compression n_1/n_0 is 30%. The data were averaged over 200 systems from an ensemble.

THE INFLUENCE OF COMMUNICATION PATTERNS IN THE *h*-RELATION HYPOTHESIS IN THE IBM SP2

Roda, J.L, Rodríguez C., Almeida F., Morales D.G.

Dpto. Estadística, Investigación Operativa y Computación,
Universidad de La Laguna, Tenerife, Spain
roda@ull.es

Abstract. The Bulk Synchronous Parallel Model, BSP has been proposed by Valiant to predict the performance of current parallel systems. In the BSP model the computation is divided in supersteps. The fundamental assumption of the BSP model is the *h*-relation hypothesis. This states that the communication time of a given superstep is proportional to the maximum number *h* of packets communicated by any processor. This paper makes a brief survey of the BSP parallel computational model and studies the validity of the *h*-relation hypothesis using current standard message passing parallel software and current standard network technology. We measure the influence of the communication pattern on the time invested in an *h*-relation. The conclusion is that a linear model based in the *h*-relation hypothesis can be used to predict the execution time for a wide set of algorithms written using Standard Message Passing Libraries.

1 Introduction

Among the plethora of parallel computational models proposed, PRAM, Networks, BSP and LogP are the most popular. The PRAM model [3] has been widely used to represent the complexity of parallel algorithms. The model is simple and useful for a gross classification of parallel algorithms but is unrealistic because all processors work synchronously and inter-processor communication is free. It assumes a single shared memory where each processor can access any cell in unit time and neglects contention caused by concurrent access to different cells within the same memory module. In a Network Model [6], communications are only allowed between directly connected processors; other communications are explicitly forwarded through intermediate nodes. Many algorithms have been created which are perfectly matched to the structure of a particular network. However these elegant algorithms lack robustness, as they usually do not map with equal efficiency onto interconnection structures different from those for which they were designed.

Many of current parallel computers consist of a collection of complete computers connected through a network interface to a multistage interconnection network. Culler et al. [2] believe that this hardware organization is going to dominate commercial

Massively Parallel Computers in the near future. The LogP Model, [2] characterizes a parallel hardware/software platform by four parameters: the number of processors (P), the gap (g), the latency (L) and the communication overhead o. The model also assumes that if a processor attempts to transmit more than [L/g] not consumed messages, it will stall until the message can be sent without exceeding the limit. Although the model encourages the careful scheduling of communication and overlapping of communications and computations, there is a concern that a complete LogP analysis for non-trivial algorithms is in not few cases almost unfeasible.

Section 2 introduces the BSP model. Section 3 measures the influence of the communication pattern on the time invested in an h-relation. Section 4 concludes that the linear model approach proposed in section 3, can be used to predict the performance of PVM [4] and MPI [11] bulk synchronous programs.

2 The Bulk Synchronous Parallel Model.

The BSP model [12] tries to provide a simple but accurate interface between the domains of parallel architectures and algorithms. In the BSP model, a parallel machine consists of a set of processors, each with its own private memory, and an interconnection network that can route packets of some fixed size between processors. The computation is divided in supersteps. In each superstep, a processor can perform operations on local data, send packets, and receive packets. This local computation must depend only on data present in the local memory of the processor at the beginning of the superstep. A packet sent in one superstep is guaranteed to be delivered to the destination processor at the beginning of the next superstep. Consecutive supersteps are separated by a global synchronization of all processors.

The two basic BSP parameters that model a parallel machine are: the gap **g**, which reflects per-processor network bandwidth, and the minimum duration of a superstep **L**, which reflects the latency to send a packet through the network as well as the overhead to perform a global synchronization. Let be h the maximum number of packets a processor communicates (the sum of the packets received and sent) in a superstep (such a communication pattern is called an h-relation). The fundamental of the BSP model lays on the ***h*-relation hypothesis** introduced by Valiant. It states that the communication time spent on an h-relation is given by

$$\textit{Communication Time} = g\,h \tag{1}$$

Let denote by W the maximum time spent in local computation by any processor during the superstep. The BSP model guess that the running time of a superstep is bounded by the formula:

$$\textit{Time Superstep} = W + g\,h + L \tag{2}$$

In consequence, the design of algorithms under the BSP model tries to minimize the number of supersteps, the maximum number of operations performed by any processor W and the maximum number h of packets communicated. A virtue in BSP of having barriers available as a primitive is that analysis is simplified by assuming the processors exit the barrier in synchrony. Special libraries have been built to support the BSP style of programming [8]. However, such software is not still widely

extended. There is no doubt that MPI and PVM constitute the facto current standards for distributed computers.

3 Checking the Validity of the *h*-relations Hypothesis

The experiments were done in the IBM Scalable POWERparallel SP2 [1]. In this distributed-memory parallel computer, processors or nodes are interconnected through a High Performance Switch (HPS). The HPS is a bi-directional multistage interconnection network. The computing nodes are Thin2, each powered by a 66MHz Power2 RISC System/6000 processor. All the algorithms were implemented in PVMe [5], the improved version of the message-passing software PVM.

The *h*-relation hypothesis does not consider the influence of communication patterns. For example, independently of the number of pairs, processors communicating according to a PingPong algorithm fall in the same *h*-relation class. That is $h = n$, where n is the size of the outgoing packets. Their cost under the BSP model matches the cost of a single couple of communicating processors: $g*n$. This *h*-relation class appears for the exchange pattern for packets of size $n/2$. When p-processors are involved, the personalized OneToAll and AllToOne communication patterns fall into the same former class of *h*-relations for packets of size $m = n/(p-1)$. The same *h*-relation appears under the personalized AllToAll communication pattern when the size of the outgoing messages is $m = n/2*(p-1)$. Each processor sends $(p-1)*m$ packets and receives the same number $(p-1)*m$. The number of communications performed by any processor is $2*(p-1)*m = n = h$. The actual times spent on these five patterns for their respective packet sizes have to be similar if the *h*-relation hypothesis holds.

Table I shows the influence of the communication pattern in the time spent in an *h*-relation. Experiments were carried out for each pattern with the *h*-relation size between 420 and 13762560 bytes and the number of processors between 2 and 8. For the PingPong and Exchange, 2, 4, 6 and 8 communicating couples were used. For the others, experiences involved 4, 6 and 8 processors. For each fixed number of processors, 500 experiences were performed. The entry in each column shows the average time in seconds. The Exchange pattern is the fastest due to the maximum parallelism it achieves. On the Exchange pattern the two processors in each couple simultaneously send their messages. On the other extreme, the PingPong communication pattern is the slowest since it implies the most sequential case of sending (receiving) by one processor the h bytes implied in the *h*-relation. The time for any other pattern is in the range between these two. An straightforward implementation of the personalized OneToAll is to consecutively send the whole message to each of the other processors. The policy we propose is to divide the message in packets and proceed to apply to each packet the former algorithm. This policy is optimal using a packet size of 32KB. The best policy for the AllToAll pattern for *h*-relations under 430080 Bytes is to start sending all the messages according to a processor permutation. From this size on, the network becomes saturated and it is better to consume the incoming messages. Although the values do not appear in Table I, for all the patterns, the dependency of times in the number of processors was negligible (under 0.2% for *h*-relations larger than 215040 bytes). Observe that, the times for Exchange, OneToAll, AllToOne and AllToAll keep closer

among them than the PingPong time. The maximum difference percentage $\max_h$ {(max($t_i(h)$)-min $t_j(h)$)/min{$time_k(h)$} / i,j,k ≠ PingPong} is 21%, reached for h = 13762560 bytes.

	PingPong	Exchange	OneToAll	AllToOne	AllToAll	AvErr	MaxErr
420	0.000114	0.000114	0.000269	0.000157	0.000352	40.30	202.94
840	0.000139	0.000121	0.000286	0.000169	0.000362	37.51	188.03
1680	0.000191	0.000153	0.000313	0.000206	0.000380	34.19	141.69
3360	0.000259	0.000217	0.000356	0.000265	0.000439	27.89	100.63
6720	0.000394	0.000306	0.000442	0.000377	0.000527	17.57	62.21
13440	0.000659	0.000470	0.000659	0.000596	0.000689	7.44	25.47
26880	0.001168	0.000855	0.001081	0.000993	0.001088	0.46	20.75
53760	0.002239	0.001553	0.001939	0.001822	0.001887	-3.73	26.11
107520	0.004820	0.003105	0.003745	0.003562	0.003492	-1.79	32.48
215040	0.009418	0.006418	0.007508	0.007080	0.006883	-0.75	29.61
430080	0.018768	0.012684	0.015016	0.014222	0.013685	-0.36	30.27
860160	0.037199	0.025448	0.030131	0.028558	0.026845	-0.39	29.26
1720320	0.074260	0.050447	0.060469	0.057548	0.053980	-0.10	29.46
3440640	0.148477	0.100577	0.121139	0.115967	0.106736	-0.09	29.62
6881280	0.297393	0.201113	0.241496	0.232967	0.212628	-0.06	29.89
13762560	0.593549	0.402237	0.487938	0.467001	0.422079	0.03	29.61

Table 1. Pattern Communication Times and Error Percentage for different *h*-relation sizes.

To obtain the general linear approach to the h-relation time we have computed the least square fit of the average times of the five patterns. This gives L = $1.06*10^{-4}$ and g = $3.45*10^{-8}$. Compare these BSP-PVM values with the obtained using the Oxford BSP library: g' = $35*10^{-8}$, L' = $4.62*10^{-4}$ for the same machine [7]. Columns labeled Av. Err. and Max. Err. respectively show the average and maximum errors defined as:

$$AvErr(h) = 100\{(\sum_i T_i(h)/5)-(gh+L)\}/(\sum_i T_i(h)/5)\text{: } i \text{ in the set of patterns}\}. \quad (3)$$

$$MaxErr(h) = 100\{\max_i |T_i(h)-(gh+L)| / \min_j T_j(h)\text{: } i, j \text{ in the set of patterns}\}.$$

Negative numbers in the Average Error column correspond to cases in which the model time is larger than the actual time. For h larger than 26880, the Average Error is under 4%. From 13440 on, the Maximum Error keeps almost constant around 30%.

4 Conclusions.

The collective computation provided by MPI fits the Bulk Synchronous Programming Methodology. Extensions of PVM like La Laguna C [9] make PVM a tool suitable for the expression of BSP algorithms. Based in the h-relation hypothesis, a linear model approach to predict the performance of PVM/MPI bulk synchronous programs has been presented. The maximum error incurred by neglecting the influence of

communication patterns is under 30% for medium and large *h*-relation sizes. A more accurate prediction can be achieved by using the values for g and L obtained for each pattern [9].

Acknowledgments

Part of this research has been done using resources at the Centre de Computació i Comunicacions de Catalunya (CESCA-CEPBA).

References

1. Arruabarrena J.M., Arruabarrena A., Beivide R., Gregorio J.A. *Assesing the Performance of the New IBM-SP2 Communication Subsystem.* IEEE Parallel and Distributed Technology. pp. 12-22. 1996.
2. Culler D., Karp Richard, Patterson D., Sahay A., Schauser K.E., Santos E., Subramonian R., von Eicken T.. *LogP: Towards a Realistic Model of Parallel Computation.* Proceedings of the 4th ACM SIGPLAN, Sym. Principles and Practice of Parallel Programming. May 1993.
3. Fortune, S. Wyllie, J. *Parallelism in Randomized Machines.* Proceedings of STOC., pp. 114-118. 1978
4. Geist A., Beguelin A., Dongarra J., Jiang W., Mancheck R., Sunderam V.. *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing*. MIT Press. 1994.
5. IBM *PVMe for AIX User's Guide and Subroutine Reference* Version 2, Release 1. Document number GC23-3884-00. IBM Corp. 1995.
6. Leighton, T. Introduction to Parallel Architectures; Arrays, Trees, Hypercubes, Morgan Kaufmann, San Mateo, CA. 1992.
7. Marín, J. Martínez, A. *Testing PVM versus BSP Programming*, VIII Jornadas de Paralelismo. pp 153-160. Sept 1997
8. Miller, R. Reed, J.L. *The Oxford BSP Library Users' Guide. Technical Report*, Programming Research Group, University of Oxford. 1993.
9. Roda J., Rodríguez C., Almeida F., Morales D.. *Predicting the Performance of Injection Communication Patterns on PVM.* 4th European PVM/MPI Meeting Group. Cracow, Poland. Springer-Verlag. Nov-1997.
10. Rodríguez, C., Sande F., León C., García L. *Providing Nested Parallelism and Load Balancing on Message Passing Libraries.* 6th IEEE Euromicro Workshop on Parallel and Distributed Processing. 1998.
11. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J. *MPI: The complete Reference.* Cambridge, MA: MIT Press. 1996.
12. Valiant L.G.. *A Bridging Model for Parallel Computation.* Communications of the ACM, 33(8): 103-111, August 1990.

One-sided Block Jacobi Methods for the Symmetric Eigenvalue Problem *

D. Giménez[1], J. Cuenca[1], R. M. Ralha[2] and A. J. Viamonte[3]

[1] Departamento de Informática y Sistemas. Univ de Murcia.
Aptdo 4021. 30001 Murcia, Spain. ({domingo,jcuenca}@dif.um.es)
[2] Departamento de Matemática. Univ do Minho. Campus Gualtar.
4710 Braga, Portugal. (r_ralha@math.uminho.pt)
[3] CEMPA. Univ Portucalense. R. Dr. António B. de Almeida, 541-619.
4200 Porto, Portugal. (ajs@uportu.pt)

Abstract. In this paper we study theoretically two different one-sided block Jacobi algorithms for solving the Symmetric Eigenvalue Problem. Sequential and parallel versions of the algorithms are analized and compared with a two-sided block Jacobi algorithm. The main advantage of the one-sided algorithms is that they are better suited to parallel computers, and when computing eigenvalues and eigenvectors on multicomputers a more reduced execution time is predicted for the one-sided algorithms than for the two-sided algorithm.

1 Introduction

In this work we studied the design of two one-sided block Jacobi algorithms for the Symmetric Eigenvalue Problem. The algorithms are designed using as a basic the two one-sided Jacobi algorithms proposed in [1].

We begin by explaining how a two-sided block Jacobi method works [2], and after that the two-sided method will be compared with two one-sided block Jacobi methods. The main goal of the comparison is to conclude if one-sided block Jacobi algorithms can be designed maintaining the high degree of parallelism of the algorithms not working by blocks, and the one-sided methods can be competitive with two-sided block algorithms.

2 A two-sided block Jacobi algorithm

The method works over two matrices: the matrix A and a matrix V where the rotations are accumulated. Matrix V is initially the identity matrix. Both matrices A and V are divided into columns and rows of square blocks of size $s \times s$, and these blocks are grouped to obtain bigger blocks of size $2s \times 2s$.

* Partially supported by Comisión Interministerial de Ciencia y Tecnología, project TIC96-1062-C03-02; Consejería de Cultura y Educación de Murcia, Dirección General de Universidades, project COM-18/96 MAT; and Acción Integrada Hispano-Lusa HP1996-0007.

Jacobi methods work by constructing a matrix sequence $\{A_l\}$ by means of $A_{l+1} = Q_l A_l Q_l^t$, $l = 1, 2, \ldots$, where $A_1 = A$. In a non block version of the method, Q_l represents a plane rotation and each product $Q_l A_l Q_l^t$ annihilates a pair of nondiagonal elements, a_{ij} and a_{ji}, of matrix A_l, but in a block version, each Q_l represents a set of rotations that nullify elements in a block of A_l. In each block the algorithm works by making a sweep over the elements in the block. The subdiagonal elements belonging to diagonal blocks will not be zeroed. To correct this, blocks corresponding to the first Jacobi set are considered to be of size $2s \times 2s$, adding to each block the two adjacent diagonal blocks and the symmetrical block. The work over each block can be performed using level-1 BLAS. The corresponding rotations are accumulated to form a matrix Q of size $2s \times 2s$. Finally, the corresponding columns and rows of blocks of size $2s \times 2s$ of matrix A and the rows of blocks of matrix V are updated using Q. These matrix-matrix multiplications can be effected using level-3 BLAS.

After completing a set of blocked rotations, a swap of column and row blocks is performed, according to the order we are using. The odd-even order will be used, [3], because it simplifies a block based implementation of the sequential algorithm, and allows parallelization. If $n = 8$, numbering indices from 1 to 8, and initially grouping the indices in pairs $\{(1,2),(3,4),(5,6),(7,8)\}$, the sets of pairs of indices are obtained as follows: $\{(1,2),(3,4),(5,6),(7,8)\}$, $\{2,(1,4),(3,6),(5,8),7\}$, $\{(2,4),(1,6),(3,8),(5,7)\}, \ldots$.

This data movement brings the next blocks of size $s \times s$ to be zeroed to the subdiagonal, and the process continues similarly to operations performed in the first step. However, in this case the elements to be nullified are in square blocks of size $s \times s$ inside diagonal blocks of size $2s \times 2s$. This data movement will imply data transferences in the parallel version of the algorithm.

The cost per sweep is:

$$8k_3n^3 + (12k_1 - 16k_3)\, n^2 s + 8k_3 n s^2 \quad flops, \tag{1}$$

where k_1 and k_3 represent the cost of an arithmetic operation performed using BLAS 1 or BLAS 3, respectively.

2.1 A parallel algorithm

It is possible to obtain a balanced algorithm for a ring. Grouping blocks of size $2s \times 2s$ of matrix A and V in bigger blocks A_{ij} and V_{ij} of size $2sk \times 2sk$, we assign to each processor P_i, with $p = \frac{q}{2}$ and $\frac{n}{2sk} = q$, rows of blocks i and $q-1-i$ of matrices A and V. Therefore, each processor P_i contains blocks A_{ij} and $A_{q-1-i,j}$, with $0 \leq j \leq i$, and V_{ij} and $V_{q-1-i,j}$, with $0 \leq j < q$.

Due to the data movement between odd and even steps, it is necessary to reserve some additional memory, and $(2sk+s)(2n+2sk+2s)$ positions of memory are reserved on each processor.

The arithmetic cost per sweep when computing eigenvalues and eigenvectors is:

$$8k_3\frac{n^3}{p} + (12k_1 - 8k_3)\frac{n^2 s}{p} + 12k_1\frac{ns^2}{p} \quad flops. \tag{2}$$

And the cost per sweep of the communications is:

$$\beta(p+3)\frac{n}{s} + \tau\left(8n^2 + 2ns - \frac{2n^2}{p}\right) \quad , \tag{3}$$

where β and τ represent the start-up and the word-sending time, respectively.

3 A one-sided block Jacobi algorithm. First version

We analized the first one-sided Jacobi algorithm in the paper [1]. The algorithm works on matrices $B_0 = A$ and $W_0 = I$, obtaining $B_{r+1} = V_r B_r$, $W_{r+1} = V_r W_r$, with V_r the rotation matrix nullifying a non-diagonal element of matrix $A_r = B_r W_r^t = V_{r-1} V_{r-2} \ldots V_0 B_0 W_0^t V_0^t \ldots V_{r-1}^t$.

To nullify a_{ij} it is necessary to compute a_{ii}, a_{jj} and a_{ij}, because the algorithm works on matrices B_r and W_r, and not on matrix A_r. These elements are obtained with three dot products. After that, rows i and j of B_r and W_r are updated. If the diagonal elements are stored in an auxiliary vector, it is not necessary to compute a_{ii} and a_{jj} every time, and the cost per sweep is:

$$7n^3 - \frac{15}{2}n^2 + \frac{n}{2} \quad flops. \tag{4}$$

We propose a one-sided block Jacobi algorithm by combining the ideas of the two-sided block algorithm and the ideas of the one-sided algorithm.

Matrices B and W, of size $n \times n$, are divided in blocks of size $s \times n$, and blocks of $A = BW$ are treated using the odd-even ordering.

Initially the $\frac{n}{2s}$ blocks corresponding to the first Jacobi set are treated, making a two-sided sweep on blocks of size $2s \times 2s$ of matrix A and accumulating rotations. These operations are done using BLAS 1.

After that, matrices B and W are updated multiplying the rotation matrices, of size $2s \times 2s$, by the corresponding blocks of B and W, of size $2s \times n$. In this case matrices B and W are not symmetric.

In the two-sided algorithm a movement of rows and columns of blocks is performed in order to have the blocks grouped according to the next Jacobi set. This movement can be include in the updating of the matrix if it is done on the rotation matrix before updating A. In the one-sided algorithm the movement of rows of blocks of B and W can be done in the same way (figure 1).

In successive steps it is necessary to compute A_{ii}, A_{jj} and A_{ij}, because the work is not done directly with matrix A. If we call B_i and W_i the i-th row of blocks of B and W in figure 1.a), $A_{ii} = B_i W_i^t$, $A_{jj} = B_j W_j^t$ and $A_{ij} = B_i W_j^t$. If the diagonal blocks are stored it is not necessary to compute A_{ii} and A_{jj}.

After the blocks A_{ii}, A_{jj} and A_{ij} are computed, a matrix of size $2s \times 2s$ is formed, and a two-sided sweep is performed on this matrix, accumulating the rotations.

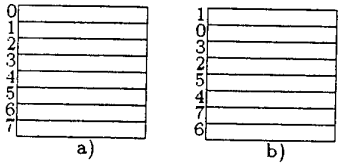


Fig. 1. Distribution of matrices B and W on the first one-sided block algorithm: a) initially, b) after application of the first set of rotations.

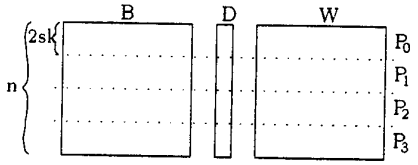


Fig. 2. Initial distribution of matrices B, D and W in the system of processors for the first one-sided parallel block algorithm.

The cost per sweep is:

$$9k_3n^3 + (12k_1 - 9k_3)\, n^2 s \quad flops. \tag{5}$$

3.1 A parallel algorithm

It is possible to assign to each processor k consecutive blocks of size $2s \times n$, with $n = 2skp$, of matrices B and W (figure 2). In the figure the distribution of the matrices is shown, but also in this case it is necessary to reserve some additional memory to store data in sucessive steps of the algorithm. The quantity of memory reserved in each processor is $(2k+1)sn$ to store elements of B, the same quantity to store elements of W, and $(2k+1)s^2$ to store elements of D.

The arithmetic cost per sweep is:

$$9k_3\frac{n^3}{p} + 12k_1\frac{n^2t}{p} + 12k_1\frac{ns^2}{p} \quad flops. \tag{6}$$

It is not necessary to broadcast the rotation matrices because each processor updates the rows of blocks it contains. The only communications are those between steps to group data according to the next Jacobi set. In odd steps blocks of size $s \times n$ of B, and W, and a diagonal block of size $s \times s$ are sent from P_i to P_{i-1}, with $i = 1, 2, \ldots, p-1$, and in even steps the same communications are done from P_{i-1} to P_i. Therefore, the cost per sweep of communications is:

$$2\frac{n}{s}\beta + (4n^2 + 2ns)\,\tau \quad . \tag{7}$$

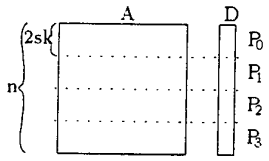


Fig. 3. Initial distribution of matrices A and D in the system of processor for the second one-sided parallel block algorithm.

4 A one-sided Jacobi algorithm. Second version

The second one-sided Jacobi algorithm proposed in [1] has the advantage of a lower execution time, but also has the disadvantage of a worse precision [4].

The method works by diagonalizing matrix $B = A^2$ but without explicitly form B. Rotations V nullifying elements b_{ij} of B are applied to A. If initially $A_1 = A$ and $B_1 = A_1 A_1^t$, we will have $A_{r+1} = V_r A_r$, and A must be updated only by one side. Because $B_r = A_r A_r^t$, it is necessary to perform dot products to obtain b_{ii}, b_{jj} and b_{ij}, which are needed to obtain the next Jacobi rotation.

The cost per sweep is approximately $4n^3$ flops if the elements of the diagonal are stored.

The method has some problems derived from the fact that the eigenvalues computed are those of A^2, but not the eigenvalues of A [1, 4].

To design an algorithm by blocks matrix A is divided into consecutive blocks of size $s \times n$.

Before each subsweep on a block B_{ii}, B_{jj} and B_{ij}, are computed (or only B_{ij} if the diagonal blocks are stored). Even if the diagonal blocks are stored, in the first step all the blocks must be computed, because the algorithm works with A and not with B.

The cost per sweep is:

$$5k_3 n^3 + (12k_1 - 5k_3)\, n^2 s \quad flops. \tag{8}$$

4.1 A parallel algorithm

The distribution of matrix A and matrix D, where the diagonal blocks are stored, can be that shown in figure 3. Also in this case it is necessary to reserve some additional memory. The size of memory reserved on each processor to store data from matrix A is $(2k+1)sn$ and to store data from matrix D is $(2k+1)s^2$.

The arithmetic cost per sweep is:

$$5k_3 \frac{n^3}{p} + 12k_1 \frac{n^2 s}{p} + 12k_1 \frac{n s^2}{p} \quad flops. \tag{9}$$

Table 1. Predicted execution time of different parallel block Jacobi algorithms.

$n = 1024$	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
$two - sided$	412.3	217.5	108.9	54.6	27.5	15.9	9.4	
$one - sided, version1$	463.8	246.5	123.6	62.1	31.3	16.0	9.3	5.2
$one - sided, version2$	257.6	143.1	71.7	36.0	18.1	9.2	5.2	2.9

The only communications are those produced by the data movements between steps. In odd steps $s(n+s)$ elements are sent from P_i to P_{i-1}, and in even steps the same quantity is sent from P_i to P_{i+1}. The cost per sweep of communications is:

$$\frac{2n}{s}\beta + (2n^2 + 2ns)\,\tau \quad . \tag{10}$$

5 Comparison and Conclusions

The first version of the one-sided algorithm has a higher cost than the two-sided method, but the difference is smaller in the algorithms working by blocks than in the algorithms not working by blocks. The second one-sided algorithm has the lowest cost, but has worse precision. Communications are less costly in the one-sided algorithms because it is not necessary to broadcast the rotation matrices. Also in the communications the second one-sided algorithm is better because it works with one matrix and only half of the data must be transferred. Table 1 shows the execution time predicted on the Touchstone Delta for matrix size 1024 and a variable number of processors. The estimated values of the constants are ([2]) $k_1 = 0.137\mu s$, $k_3 = 0.048\mu s$, $\beta = 61\mu s$ and $\tau = 0.149\mu s$. We can see the behaviour of the one-sided algorithms is better when the number of processors increases. This is why it could be interesting to implement the algorithms here analized and to compare them experimentally. This is what we are doing now.

References

1. B. A. Chartres. Adaptation of the Jacobi Method for a Computer with Magnetic-tape Backing Store. *The Computer Journal*, 5:51–60, 1963.
2. D. Giménez, V. Hernández, R. van de Geijn and A. M. Vidal. A block Jacobi method on a mesh of processors. *Concurrency: Practice and Experience*, 9(5):391–411, May 1997.
3. G. H. Stewart. A Jacobi-like algorithm for computing the Schur decomposition of a nonhermitian matrix. *SIAM J. Sci. Stat. Comput.*, 4:853–864, 1985.
4. Ana J. Viamonte. Métodos de Jacobi para o Cálculo de Valores e Vectores próprios de Matrizes Simétricas. Tesis de Mestrado. Universidade do Minho. 1996.

This article was processed using the LaTeX macro package with LLNCS style

Efficient sparse data distribution for the Conjugate Gradient on distributed shared memory systems

D.E. Singh, F.F. Rivera, and J.C. Cabaleiro

Dept. Electronics and Computer Science. Univ. Santiago de Compostela
Campus Sur. 15706 Santiago de Compostela. Spain.
david@dec.usc.es, fran@dec.usc.es, caba@dec.usc.es

Abstract. *In this paper we set out to study the performance of parallelization of iterative method on shared memory multiprocessor using different data distributions. We start with the study of block and cyclic distributions, and then propose a mixed distribution which combines advantages of both.*

Keywords: Conjugate Gradient, Data Distribution, Distributed Shared Memory Systems, Sparse Systems.

1 Introduction

This work tackles the parallelisation of the non-stationary iterative Conjugate Gradient method [1,6], which is used to solve sparse linear equation systems. This type of operation frequently appears during the resolution of partial differential equations, and one of its characteristics is that the matrix of coefficients must be symmetric and positive–defined. The results obtained can be generalised to other iterative methods, due to the fact that all of them use the same kind of computations.

The system on which the parallelisation of the algorithm was implemented was the distributed shared memory multiprocessor Origin 2000 by Silicon Graphics, which consists of 8 MIPS R10000 processors using a hardware cache coherence protocol based on the directory [4].

2 Data distributions

We used the data–parallel programming paradigm which, as well as being easy to program, presents high complexity in the establishment of optimisations. The programming language used was fortran77. The parallelization is expressed by means of parallelization directives [5], which direct the compiler in the generation of calls to the low level libraries in the multiprocessors. The elements of a vector can be allocated in the memory of the system using two distributions: block and

cyclic. By means of a block distribution, the elements of a vector of size N are divided into P blocks of size $B = N/P$ (where P is the number of threads). In a cyclic distribution, the elements are divided into pieces of size L (in our case $L = 1$), and then they are distributed cyclically over the threads.

On the Origin 2000 two techniques can be used to carry out these distributions, regular and reshaped. In the regular scheme the elements to be distributed have to be pages of 16Kb. In the case in which data must be allocated in different memories are in the same page, the compiler will not be able to resolve the conflict, and will place the whole page in one of the memories. This causes a great number of conflicts of false sharing, especially for cyclic distributions, as at the level of cache line consecutive elements will belong to different threads. This is reflected in a strong increase in the number of operations of coherency, like invalidations or exclusive to shared transitions in cache lines.

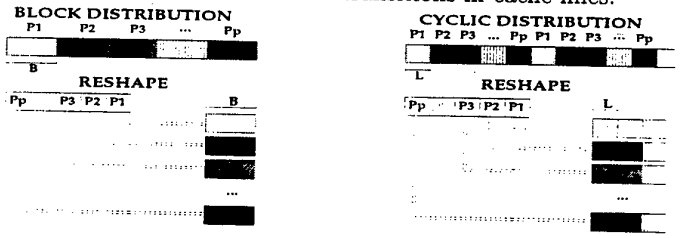


Fig. 1. Reshaped scheme over a vector.

In our case we use the reshaped scheme illustrated in figure 1, with which the compiler can reorganize the size of the blocks in the storage structure of the memory to obtain the desired distribution. This can be achieved by storing consecutively the array elements that corresponds to each local memory. Using the reshaped scheme, both the cyclic and block distributions obtain similar values in the number of coherency operations.

3 Computations

The parallelization of the algorithm is based on two types of operations which represents the greatest computation costs: sparse matrix–vector products and vectorial operations [3].

The sparse matrix–vector product is carried out by accessing the matrix by columns, which is the same as reading its rows, as the matrix is symmetric. In this way it is possible to parallelize the product so that each processor computes the value of the different elements of the resulting vector, thus eliminating possible conflicts in writes. The format for accessing the matrix is the Compressed Column Storage, by means of which the matrix is characterised by just three vectors [1].

The algorithm also uses some vectors to carry out various intermediate operations to compute, the residue, the successive approximation to the solution and the search direction. With these vectors two types of operations are carried

out: linear combinations of vectors and dot products. These operations have great influence over the efficiency of the parallel program. Moreover, the performance does not depend on the type of distribution, due to the use of the reshape technique.

4 Results

Initially two different distributions were evaluated: the block and the cyclic applied to all of the vectors, including those which characterise the matrix. Then, a new distribution was tested, which we will call hybrid block–cyclic in which the vectors that characterise the sparse matrix are distributed in blocks, and the rest of them cyclically.

Table 1. Matrices used as benchmark.

matrix	*bcsstk14*	*bcsstk17*	*zenios*	*random*
order	1806	10974	2873	10000
nnz	63454	428650	21842	110576

In our analysis we used sparse matrices of different sizes and patterns. All of these, come from the Harwell–Boeing collection [2], in addition to one matrix generated randomly. The characteristics of them are shown in table 1.

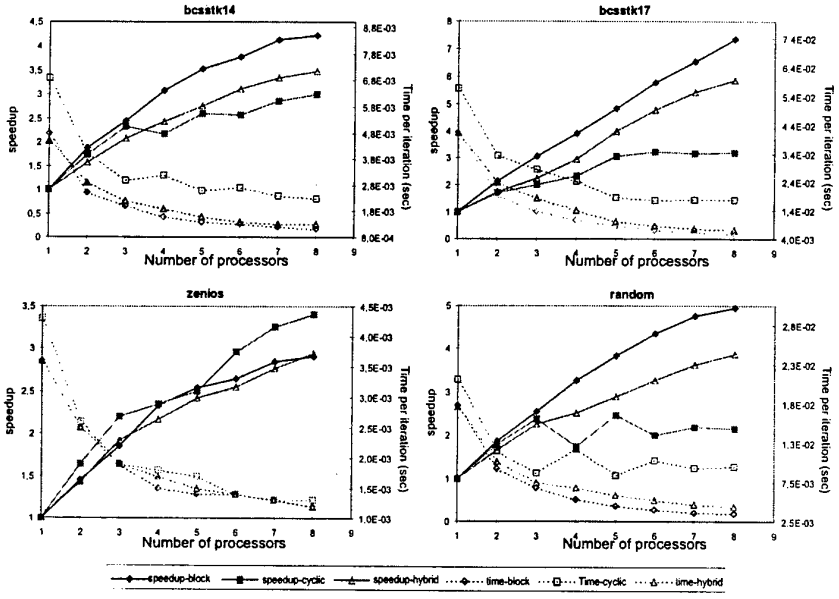
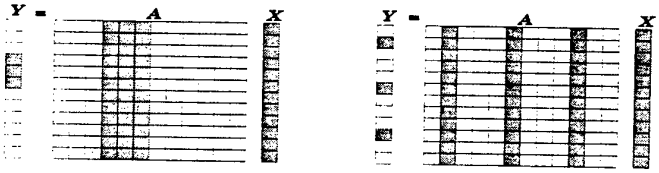


Fig. 2. *Speedup and run time per iteration.*

In figure 2 the speedups and run times for iteration are shown. Note that the best results have been obtained for the block distribution, whereas in the cyclic case there are irregularities in these values due to an increase in the number of cache misses. These irregularities are eliminated with the use of the hybrid distribution.

The performance of the parallelization of this algorithm will depend mainly on the management of the memory, an important factor being the volume of data accessed by each processor. The pattern of access to the data in the sparse matrix–vector product in each processor is shown in figure 3, and is determined by the distribution of vector Y. In other words, each processor will access those parts of the matrix which correspond to their elements of vector Y.



(a) BLOCK (b) CYCLIC

Fig. 3. Pattern of access in sparse matrix–vector product.

Note that, in the block distribution the access to the matrix is performed on adjacent columns. In this way, as the number of processors increases, each one must access a smaller number of pages and cache lines. In the case of a cyclic distribution of vector Y, it is necessary to access practically all the pages of the matrix. This is reflected in the large number of TLB misses. For matrix *bcsstk17* the number of pages it occupies is more than the number of TLB entries, so that in a single processor a great number of misses is generated as they have to access the whole matrix. When the number of processors increases, in the block case the number of TLB misses decreases, whilst in the case of the cyclic distribution it remains almost the same. However, the main problem with the latter distribution is that the consecutive elements of the matrix belong to different cache lines, so that the number of lines accessed by each processor is much larger than in the block case.

Figure 4 shows the number of cache misses. In a block distribution the lowest values in cache misses are obtained. With a cyclic distribution a marked increase in the number of cache misses in the case of four processors can be observed. The reason is that in this case all the lines read by each processor do nor fit in its cache, thereby producing a large number of operations of replacement. In the next iteration these replaced lines are demanded again, thus provoking capacity misses in the cache. By means of a hybrid distribution it is possible to solve this problem to a great extent, as consecutive elements in the matrix will be consecutive in memory (except if they are assigned to different processors) and therefore will probably belong to the same cache line. In this way it is possible to reduce the number of accesses to cache lines, and then the replacement problems have been eliminated. However, these values will always be larger than in the case of block distributions, given that it is necessary again to access a greater

memory space than in the block case, due to the cyclic distribution of the vector. The higher the number of non zero elements the greater will be this effect. Thus, in the case of the matrix *zenios*, as it has a small number of non zero elements, the results archived in the run time for iteration are similar for all distributions. The size of matrices *bcsstk17* and *random* is higher than that of the secondary cache and this produces a large number of capacity misses when the algorithm is executed in a single processor. This also produces superlineal speedups for the matrix *bcsstk17*.

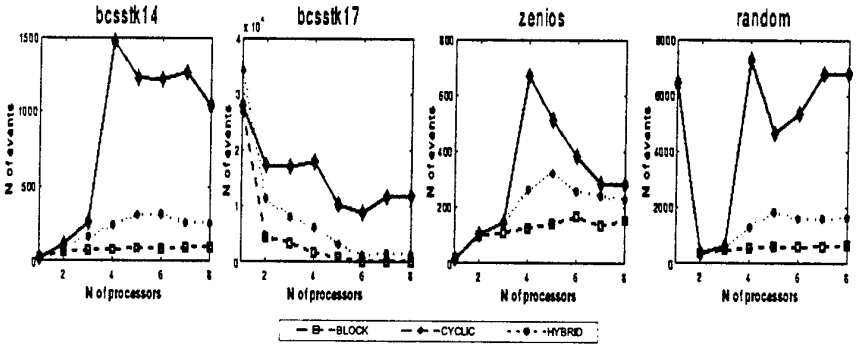


Fig. 4. Cache misses.

The main problem of this distribution arises when matrices with non uniform patterns, such as *zenios*, are used, as they cause a load unbalance between the processors which operate over the densest parts as against those that operate over the sparsest parts. This can be noted in figure 5, which represents the load unbalance given by $B = C_{max}/C_{med}$, where C_{max} is the number of floating point operations of the thread which has the greater work load, and C_{med} is the average value. High values of B limit the value of the speedup when a high number of processors is used. By means of the use of a cyclic distribution, the problem of load unbalance is then solved. Note that speedup for the *zenios* matrix is more scalable for the hybrid distribution.

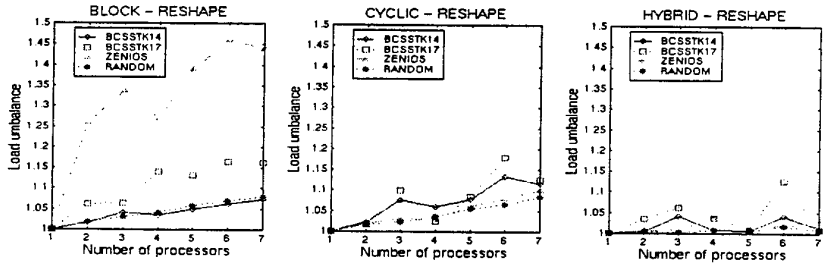


Fig. 5. Load unbalance.

5 Conclusions

The use of regular distributions is inefficient in sparse systems, given that in these cases the pattern of the matrices is not known at compile time. By using a hybrid distribution the advantages with regard to the load balancing of the cyclic distribution are maintained, and the execution times per iteration are similar to the block distribution. In this way, the results using matrices with regular patterns are similar to the block distribution, and faced with matrices with irregular patterns, the load unbalance is resolved.

Acknowledgements

This work was supported in part by the CICYT under grant TIC96-1125-C3-02 and Xunta de Galicia under grant XUGA20605B96. We would like to thank the University of Malaga for the use of their systems.

References

1. R. Barrett and M.Berry et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, 1994.
2. Iain S. Duff, Roger G. Grimes, and John G. Lewis. *Users' Guide for the Harwell-Boeing Sparse Matrix Collection.* Research and Technology Division, Boeing Computer Services, Mail Stop 7L-21, P.O. Box 24236, Seattle, WA 98124-0346, USA, release I edition, October 1992.
3. K.Dincer, K.A.Hawick, A.Choudhary, and G.C.Fox. High perfomance Fortran and possible extensions to support Conjugate Gradient algorithms. Technical report, Northeast Parallel Architecture Center, 111 College Place, Syracuse, NY 13244-4100, 1996.
4. James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Computer Architecture News*, volume 25, pages 241–251. SCA'24 Proceedings, May 1997. Special Issue.
5. Guy-Ren Perrin and Alain Darte. *The data parallel Programming model.* Springer, 1996.
6. Yousef Saad. *Iterative Methods for Sparse Lineal Systems.* PWS PUBLISHING COMPANY, 1996.

Synchronized Parallel Algorithms on RedBlack trees*

Xavier Messeguer and Borja Valles

Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Campus Nord–Mòdul C6
C/ Jordi Girona Salgado, 1–3
08034 Barcelona, Spain
Contact author: peypoch@lsi.upc.es

Abstract. We present an approach for designing synchronized parallel algorithms to update RedBlack trees. The resulting algorithms update k keys with k processors on trees of size n in time $O(\log n + \log k)$ which is very close to the optimal speedup of $O(\log n)$ (sequential time for one search or update). The algorithms are designed as a pipeline of waves of processors, which are created at the bottom of the tree and flow up to the root. The design is made following the E.W.Dijkstra approach by first choosing the invariant properties and then the rules to update the tree.

Keywords: Synchronized parallel algorithms, PRAM algorithms, RedBlack trees.

1 Introduction

The so called *Synchronized parallel algorithms* are those that manage data types in a synchronized manner (PRAM algorithms [Akl89]). They can be envisaged as many sequential algorithms running simultaneously and executing the same sentence at the same time. Therefore, it may happen that several processes read or write on the same memory location at the same time. Our goal is to avoid these concurrent accesses.

The first synchronized parallel algorithms on search trees were designed by W. Paul, U. Vishkin and H. Wagener for 2-3 trees in 1983 [PVW83]. They proved that the time needed to search or update k elements with k processors on a tree with n keys is $O(\log n + \log k)$ which is very close to the optimal speedup of $O(\log n)$.

They designed parallel algorithms to dynamically maintain a parallel dictionary working simultaneously with many keys. The algorithms first hang the keys from the leaves (*search phase*), and later rebalance the tree (*rebalancing phase*) using pipelines of processors. These pipelines can be envisaged intuitively

* This work has been partially supported by ESPRIT LTR Project no. 20244 — ALCOM-IT and DGICYT under grant PB95-0787 (project KOALA).

in terms of traveling plane waves. Assume, for instance, the basic insertion case in which every leaf incorporates at most one new key. Something like a *wave* of processors is generated at the bottom of the tree, namely a *plane wave*, because all leaves of a 2-3 tree have the same depth. This wave is sent up in further iterations until it disappears. Note that the wave goes to the root and at each iteration it strictly increases its height and decreases its depth. The *life-time* of each wave, i.e. the number of steps taken by a wave before it disappears, is an open problem, but some preliminary results [BYGM97] strongly suggest that it is logarithmic on k.

In the general insertion case, in which a packet of many new keys can hang from a single leaf, a pipeline of waves is generated to get something like harmonic traveling waves. Each new wave is created as follows: some iterations after the last wave has been created, the packets are split, the middle key of each one is attached as a new leaf and the remaining left and right subpackets are hung from the new leaf. This set of new leaves created by the middle keys constitute the new wave.

This rebalancing phase synchronizes the processors that belong to the same wave, and these processors locally manage the data and test the conditions to become inactive or to continue one step more. For this reason we say that processors are controlled by *Local Rules*. These are sequential algorithms composed by a small and fixed number of sentences that access a small number of neighbor nodes. The rebalancing phase can be written:

```
While there are active processors do
    For all waves do
        For all active processors of a wave do
            Select and apply rules
        endforall
    endforall
endwhile
```

These ideas were applied on B trees by L. Higham and E. Schenk [HS94], on Skip lists by J. Gabarró, C. Martínez and X. Messeguer [GMM96], and on AVL trees by J. Gabarró and X. Messeguer [GM96].

The RedBlack trees are an important basic data structure, namely a *balanced binary search tree*, which implements the *dictionary* abstract data type. The balancing criterion differentiates RedBlack trees from 2-3 trees, because it does not force the tree to be perfectly balanced: it is possible to deal with RedBlack trees whose leaves have significantly different depth. Therefore, it could be difficult to synchronize the processors of a wave because there is no obvious way to create plane waves.

We address in this paper the design of the synchronized insertion parallel algorithm on RedBlack trees with the same cost $O(\log n + \log k)$, and the *exclusive-read* and *exclusive-write* policy (EREW [Akl89]).

We omit the search phase of the update algorithms because it is well known (see previous references). We only design the rebalancing phase of this algorithm in which k keys are updated with k processors. The deletion algorithm can easily be designed using the same technique.

We prove the algorithm correctness following the approach developed by E.W.Dijkstra, in which the proofs are based on the preservation of some properties, called *invariants*, at each iteration, and the strict decreasement of a function, called *variant* function, at each iteration. This approach, very common in basic sequential algorithmic courses, has not been applied yet on parallel algorithms on balanced search trees.

The rest of paper is organized as follows. Section 2 recalls RedBlack trees. Section 3 addresses the synchronized insertion algorithm. Finally section 4 shows the local rules of the algorithm.

2 RedBlack trees.

Following [CLR90], each node n of a RedBlack tree stores a key, denoted **key**(n), and each internal node has three pointers **left**(n), **right**(n) and **parent**(n) pointing respectively to its sons and parent. A RedBlack tree satisfies the following properties:

P_1 : Every node is either red or black.
P_2 : Every leaf (NIL) is black.
P_3 : If a node is red then both its children are black. This is equivalent to, no path from the root to a leaf contains two consecutive red nodes.
P_4 : Every simple path from a node to a leaf contains the same number of black nodes.

The last condition P_4 allows the definition of the function called black-height in [CLR90]:

$$\textbf{blackh}(n) = \text{the number of black nodes on any path from, but not including, a node } n \text{ to a leaf.}$$

We recall the sequential insertion algorithm:

1. *Search phase.* The key to be inserted falls until it is attached to a new red node n at the bottom of the tree. As this new node n is red, the property P_4 is maintained.
2. *Rebalancing phase.* If the *parent* of n is black P_3 holds and the insertion is over. Otherwise, n and **parent**(n) are red and the bottom-up *rebalancing* phase really starts. By performing rotations and node recoloring, the redness of consecutive nodes disappears or rises up. Finally, if the root becomes red it is colored black. Figure 1 depicts the local rules applied in this phase.

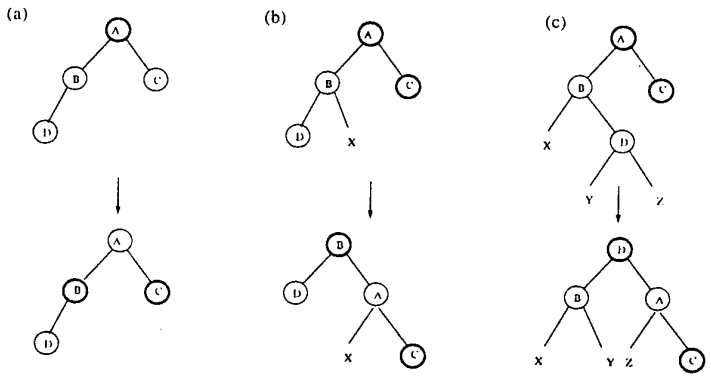


Fig. 1. The three basic local rules (under symmetry) of the sequential algorithm. The first rule (a) propagates up the redness and the following two rules, (b) and (c), rotate down the blackness.

3 Synchronized parallel insertion

Assume that the parallel search phase has ended and that the packets of keys hang from the leaves. We force each iteration of the rebalancing phase to hold the following invariants:

I_1: *Properties P_1, P_2 and P_4 of RedBlack trees.*

I_2: *Only those red nodes whose parent is also red have an active processor.* We identify the node with its processor, then we sometimes talk about "active nodes". Therefore, when there are no active nodes property P_3 holds, and by I_1 the tree is a RedBlack tree.

I_3: *All active processors of a wave have the same black-height,*

$$\forall p, q \in \text{wave} : \mathbf{blackh}(p) = \mathbf{blackh}(q).$$

This property allows us to define the black-height of a wave w:

$$\mathbf{blackh}(w) = \mathbf{blackh}(p) \text{ for any } p \text{ such that } p \in w.$$

I_4: *The black-height of the last created wave is at least two.* This property means that if the black-height of every wave gets increased by one unit at each iteration, then between two consecutive waves there is at least one black node. Therefore, if an active node has a grandparent gr, then gr is black.

The variant function involves the number of keys hanging at leaves, denoted $NKEYS$, and the sum of the depths of all existing waves, denoted $DEPTH$.

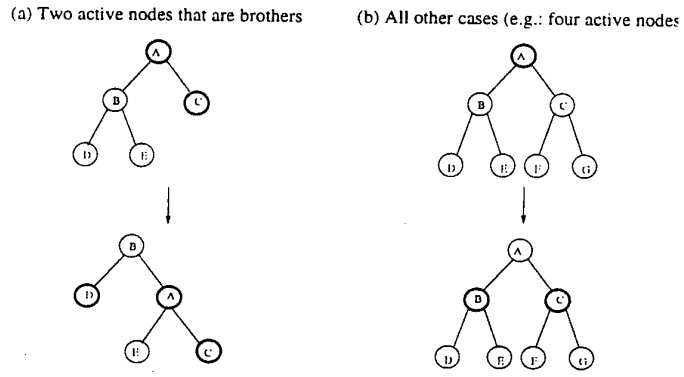


Fig. 2. The two basic new local rules (under symmetry) of the parallel algorithm

Namely, it is defined by the ordered pair $(NKEYS, DEPTH)$. It strictly decreases at each iteration because new keys are attached to the tree, and when there are no keys, we force waves to strictly increase their black-height.

Each iteration is composed of two separate actions: (i) the creation of a new wave and (ii) the moving up of all waves.

(i) A wave is created by selecting the middle key of each packet and attaching it into a new red node, so I_1 holds. Each new red node n is controlled by an active processor p_n. Then active processors test their parents color and become inactive if it is black, so I_2 holds. As all nodes of the last created wave satisfy $\mathbf{blackh}(n) = 1$ (black leaves hang from them), I_3 holds.

(ii) Active processors run local rules which will be showed in the following section. We design them so they satisfy the the previous invariant and so they increase the black-height of all waves. Finally, we again update the active nodes.

4 Local rules for insertion

Let us deal now with the rules we apply to make the waves go up. If there are active nodes without a grandparent, we simply turn the root black. For each active node n with a grandparent (that is black, by I_4) we consider the area defined by its grandparent gp, and the sons and grandsons of gp. In this area we can have active nodes other than n, but in any case they are all grandsons of gp and belong to the same wave, by I_4. Depending on the number of active nodes in the area we apply one rule or another.

If the grandparent of an area has only one active grandson we are in the same situation as the sequential case so we can try the same rules (see [CLR90]) and

check if they satisfy the invariants. If the grandparent has more of one active grandson we are in a specifically parallel case so we need new rules. For every area in this situation we need to select one representative of its active nodes so we can apply the rules with only one processor. Note that counting the active nodes in an area and selecting a representative may lead us to a concurrent read situation. We avoid that possibility by just properly sequentializing that process.

In the sequential case we have three rules (see Figure 1): in (a) we move the wave up just by recoloring. Note that the number of black nodes of each path does not change but the variant function decreases, because the black-height of the wave (whose only node is now the grandparent) is one unit higher than before. In (b) and (c) we need both rotations and recoloring. The number of black nodes of each path does not change and the active nodes become inactive.

In the parallel case we find two new situations: if we have two active nodes that are brothers (Figure 2(a)) we need one rotation and recoloring; otherwise (Figure 2(b)) recoloring is enough, because both parents are red. Again the wave moves up one level without changing the number of nodes of any path.

Summing everything up, in all cases the active nodes of a wave move up one level (their black-height increases one unit) or they become inactive, which means that the variant function actually decreases and I_3 holds. The last created wave has now black height two (I_4). We also guarantee that every path from every node to a leaf has the same number of black nodes, so we preserve I_1. Finally, as we keep updating the active nodes, we also satisfy I_2.

References

[Akl89] S. Akl. *The design and analysis of parallel algorithms.* Prentice-Hall, 1989.

[BYGM97] R.A. Baeza-Yates, J. Gabarró, and X. Messeguer. Fringe analysis for parallel macrosplit insertion algorithm in 2-3 trees. Technical Report LSI-97-38-R, Universitat Politècnica de Catalunya. Dep. de Llenguatges i Sistemes Informàtics, 1997.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw Hill, MIT, 1990.

[GM96] J. Gabarró and X. Messeguer. Massively parallel and distributed dictionaries on AVL trees and brother trees. In ISCA, editor, *Proc. of 9th International Conference on Parallel and Distributed Computing Systems*, pages 14–17, 1996. An extended version appeared as Technical Report LSI-96-27-R, LSI-UPC,1996.

[GMM96] J. Gabarró, C. Martínez, and X. Messeguer. A design of a parallel dictionary using skip lists. *Theoretical Computer Science*, (158):1–33, 1996.

[HS94] L. Higham and E. Schenks. Maintaining B-trees on an EREW PRAM. *J. of Parallel and Dist. Comp.*, 22:329–335, 1994.

[PVW83] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In J. Díaz, editor, *Proc. 10th International Colloquium on Automata, Programming and Languages, LNCS 154*, pages 597–609. Springer-Verlag, 1983. Also appeared as "Parallel computation on 2-3 trees" in RAIRO Informatique Théorique, pages 397-404, 1983.

Parallelization of GIS algorithms based on data partitioning

M. Luisa Córdoba Cabeza and Antonio Pérez Ambite

Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Universidad Politécnica de Madrid
mcordoba@fi.upm.es,
aperez@fi.upm.es

Abstract. In this paper we try to show that speeding up Geographical Information Systems (GIS) by their process in parallel architectures is possible. A spatial data partitioning and subdivision scheme is proposed, to process GIS data in a distributed memory parallel machine. We also provide solutions to classical problems in GIS systems and parallel processing, such as data boundary matching, and how to distribute and assign data among different processors to optimize both results quality and communication time. Finally, we show results obtained with different kind of hardware platforms: a net of computers organized in a cluster, and a massive parallel machine.

Key words: parallelization, data partitioning, Geographic Information Systems (GIS), massive parallel processors (mpp), multicomputers

1 Introduction

Geographic information is characterized by its distribution over terrain surface. This data organization makes their projection over an horizontal plane a good data model to be recorded and handled. We also must realize that most of process with these data is done considering parameters related with terrain surface [6].

A great deal of GIS algorithms (visualization, data interpolation, DTM generation from contour lines, intervisibility, shapes, planning, analysis, scheduling, retrievals, etc) do calculations on data representing a terrain characteristic, and therefore, easily structured as information data layers.

Both retrieval and data process of this information layers will be done on a delimited area of terrain surface, considering only spatially close data. This data neighbourhood property allows their process in parallel in a distributed system with not many communication requirements. Tasks may be distributed following data partitions of terrain surface, in such a way that partitions may be close to the proper subset of processors, though is not to others. A good data partitioning scheme among processors will allow parallel work with certain autonomy *distributed memory multiprocessor*).

2 Geographic Information Systems (GIS)

Geographic Information Systems handle spatial information with a particular behaviour. Geographic information includes cartographic or graphic elements, but also alphanumeric attributes. Main conceptual models in GIS are: vector and raster models. **Vector format** uses line as graphic primitive, while **raster format** uses point.

A vector representation of a geographical information uses points, lines, polylines and polygons as geometrical primitives. Attributes are linked to the geometry. In a raster representation, the information is projected into a grid, each grid-point defining the location and the attribute of the location.

Raster representation usually requires more memory, but on the other hand, yields a spatial distribution more homogeneous. This format has been more used than vector one, due to the fact that most algorithms to be applied to this kind of data are more efficient with this format.

Consequently, a classic problem in GIS is the huge requirements of memory to storage geographical data, with all their consequences: high access times, memory bandwidth saturation, concurrency problems, etc. These disadvantages would be considerably reduced with several processors working in parallel, following a distributed memory scheme [9].

3 GIS algorithms parallelization

The presented solution is based on spatial parallelism, partitioning data domain in square or rectangular partitions. Every rectangular partition is assigned to a virtual process node. This kind of data distribution is also named *domain decomposition* [2]. Work to be done is assigned to the the processor whose data are sited in, and that processor may communicate its neighbours as necessary.

An optimal data partition will optimize communications among different process virtual nodes. But these processors should communicate others when they require data sited on others. In general purpose applications, this communication overhead becomes a great and serious bottleneck.

3.1 Communication requirements

Communications requirements due to data partitioning in a distributed Geographical Information System are:

- Initial data partition distribution
- Data boundary partitions matching problem
- Connectivity and neighbourhood algorithms

Some geographic data analysis may be done in parallel on different data partitions with some autonomy. But connectivity and neighbourhood analysis evaluate characteristics over an area that may cover several adjacent partitions. Therefore, processors may require data from adjacent nodes.

Choosing the partition grain size is very important and not trivial. Partitioning grain must allow enough number of partitions to get the benefits of parallelism, but also partitions must be big enough to provide a minimum autonomy of work in case of connectivity or neighbourhood algorithms. Communications will be limited to the subgroup of adjacent nodes.

We propose a parallelization scheme that reduce this communication and solve the data boundary matching problem, trying to get the most benefit of parallelism. The proposed scheme minimizes communications even with a fine grain of parallelism.The solution is based in what we call the *search area.*

3.2 Search Area

The *search area* is a set of data surrounding every partition which is sent to a processor to help calculations near partition boundaries. Data and process of search area is really assigned to other processor, and are for read only to the partition which is around.

The search area is in fact an overlapped region replicated in several processing nodes. But only one processor should write on it. A synchronization and communication protocol is needed to guarantee data coherence and atomicy.

The spatial parallelization scheme includes *search area* management, creation, and updating, as schematically showed in the following steps (fig. 3):

1. Data assign and distribution among virtual processing nodes
2. Search area creation
3. Local process in parallel of the partitions considering each processing node its partition and its search area.
4. Search area updating, considering results already obtained in adjacent nodes.
5. Optional boundaries data correction at each partition, considering search area already updated

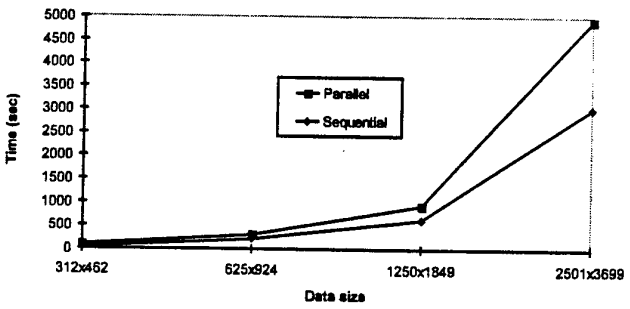


Fig. 1. Execution time in T3E with increasing data sizes.

4 Results

We have implemented our proposed parallelization scheme in different parallel hardware platforms because we wanted to propose a general scheme, independent of the hardware: a cluster with several computers, and a massive parallel processor.

Our main goal with this implementation is not only to demonstrate our parallelization scheme works properly, but also showing that contiguity and neighbourhood algorithms may also be parallelized without loosing information and therefore without significantly communication overhead. We have concretized our tests for spatial interpolation from contour lines, one of the most representative neighbourhood algorithm in GIS.

We have also studied influence in time and quality results of different parameters in partitioning scheme, such as: size and number of data partitions, number of real processor nodes, search area size, etc.

We have tested the following two hardware platforms: a cluster with 4 RS6000 (programming model PVM); and a massive parallel processor: T3E (Cray), with up to 32 processors (programming Model of shared variables (HPF)).

In both platforms we have analyzed both quality of results and the execution time.

Analyzing results quality, we studied the proper search area size and the partitioning grain (partitions size). Obviously, we obtained the same results quality at both hardware platforms.

We established that the search area size depends on data distribution. For spatial interpolation, we estimated that the search area size should obey the following expression:

$$1 - P^{sas} \geq 0.8$$

where *sas* is the search area size expressed in number of data rows, and P is the density of points with known latitude in input data.

This minimum search area size guarantees quite similar results quality near boundaries partitions than in sequential processing.

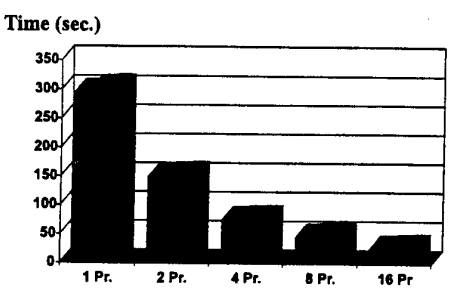
We also got that with this search area size, partitioning grain could be fine to get the benefits of parallelism. Therefore, the best partition size is determined by the number of available real processors.

About execution time, results were rather different in the two platforms. Tests in the cluster show that time processing heavily depends on the network load. This network may become soon a bottleneck, and speed up with 4 processors is not really spectacular for small files.

But our tests also show that speed up improves as data size grows up (fig. 1). This is important, as these systems (characterized by managing huge quantities of data) are involving more an more data.

Execution times with the mpp of Cray are quite better, with a high speed up, thanks to a higher number of available real processors, and a better and dedicated communication links. However, due to the fact that I/O was in this machine

sequential, total speed up is very influenced by sequential I/O. Considering just interpolation time, excluding I/O operations, the speed up for 16 processors were near 11, what is good (fig. 2 and 3).

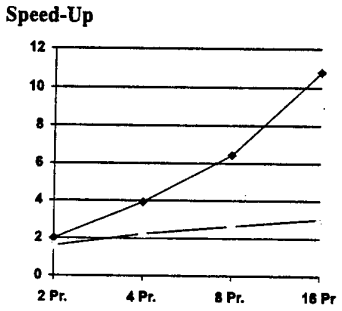


Interpolation time (sec) for eje9 (1090x1720)

sas=0 sas=15 sas=30

sas = search area size (n. rows)

	1 Pr.	2 Pr.	4 Pr.	8 Pr.	16 Pr.
sas = 0	294	149	75	46	26
sas = 15	301	151	76	47	27
sas = 30	302	152	77	47	28



Speed up for eje9 (1090x1720)

Spatial interpolation speed up

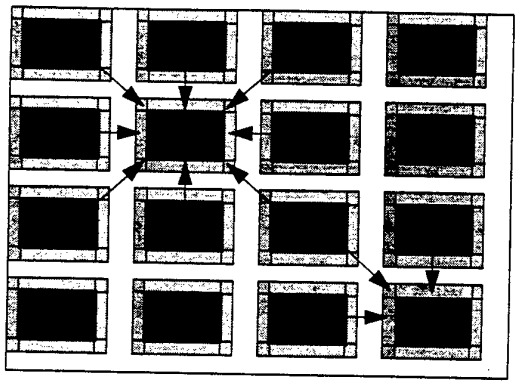
Total time speed up (+I/O)

	2 Pr.	4 Pr.	8 Pr.	16 Pr.
Interpolation speed up	1,99	3,92	6,43	10,79
Total speed up (+seq. I/O)	1,59	2,23	2,63	3,02

Fig. 2. Interpolation times and speed up for Cray T3E.

5 Conclusions

A general parallelization scheme based on data partitioning is presented. The proposed scheme minimize communication between process nodes, thanks to the *search area* concept introduced and therefore, response times are considerably reduced. The proposed scheme also presents solution to classical problems in GIS, such as data boundary matching in spatial subdivision, the influence of partitioning grain in quality and time of results, and data assignment to the different processor nodes. Execution times are significantly better in the massive parallel machine, where communications are not the bottleneck (in the cluster the network is a serious bottleneck). However, a parallel I/O file system is truly recommended when massive parallel processors are working.



Data input	183	185
First interpolation	158	153
Seach area updating	near 0	1
Second interpolation	2	9
Data output	236	237

Fig. 3. I/O, interpolation and search area updating times for Cray T3E.

References

1. Allan, R.J. *A Tutorial in Parallel Programming.* Parallel Algorithm Design and Implementation. Course at Physics Computing'92 Conference, Prague. Daresbury Laboratory, DL/SCI/P824T. August, 1992.
2. Allan, R.J. *An Introduction to Parallel Programming.* Daresbury Laboratory, Technical Memorandum DL/SCI/TM95T. May, 1993.
3. Burrough, P. A. *Principles of Geographical Information Systems for Land Resources Assessment.* Oxford University Press. Monographs on Soil and Resource Survey, 1986. ISBN 0-19-854592-4.
4. Carretero, J. *Un Sistema de Ficheros Paralelo con Coherencia de Cache para Multiprocesadores de Propósito General.* Tesis Doctoral. Facultad de Informática de Madrid. Univ. Politécnica de Madrid. DATSI. Mayo, 1995.
5. Córdoba, M. L.; Perez Ambite, A. *Modelización y Visualización del Terreno.* II Conferencia sobre Informática Gráfica. pp 1-8. Junio, 1990.
6. Córdoba, M. L. *Estudio y Paralelización de Algoritmos para Sistemas de Información Geográfica.* Tesis Doctoral. Facultad de Informática de Madrid. Univ. Politécnica de Madrid. DATSI. Febrero, 1996.
7. Cray Research, Inc. *PVM and HeNCE Programmer's Manual.* SR-2501 3.0. 1992.
8. Cray Research, Inc. *CRAY T3D Emulator User's Guide.* SR-2500 1.0.2. 1993.
9. Hwang, K.; *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill. 1993.
10. Shekhar, S.; Coyle, M.; Goyal B.; Liu D.; Sarkar, S. *Data Models in Geographic Information Systems.* Comm. of the ACM, vol. 40, nro 4, April 1997.
11. Ulmann, U. *Volume Reconstruction and Parallel Rendering Algorithms: A Comparative Analysis.* Ph D Dissertation. Univ. North Carolina at Chapel Hill. Department of Comp. Science. 1993.
12. Valiant, L. G. *A Bridging Model for Parallel Computation* Communications ACM, vol. 33, nro 8, pp 103-111. August 1990.

Emulating a superscalar processor to teach pipeline and superscalar concepts

Santiago Rodríguez de la Fuente
M. Isabel García Clemente
Rafael Méndez Cavanillas
José M. Pérez Villadeamigo

Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Universidad Politécnica de Madrid
srodri@fi.upm.es,
mgarcia@fi.upm.es,
rmendez@fi.upm.es

Abstract. Current processors use special techniques to improve performance such as pipeline and multiple instruction issue per cycle.
Using a real pipeline or superscalar computer to teach these concepts is actually impractical, because these computers are designed to be programmed in high-level languages.
Hence, we have implemented a superscalar processor emulator, where most of the processor parameters can be defined by the student. Its objective is to create a set of laboratory works allowing the student to observe the execution evolution of his assembly program through the different components of the computer, detecting the different kinds of hazards and their impact on performance. Then, the student can apply some software techniques to avoid them. Moreover, he can obtain statistics about caches.
Keywords: education, pipeline, superscalar, cache memory, emulator.

1 Introduction

This paper presents a superscalar processor (MC88110) emulator that we have implemented to teach classical and modern Computer Architecture concepts at the Facultad de Informática of the U.P.M.

The motivation which lead us to develop a new emulator, instead of using existing ones, is simple. We wanted an educational tool which could serve us to make different practical works in which we could increase the complexity of the concepts we want to cover. In a first stage, we want to use the emulator to teach assembly programming and later to teach cache behavior, and pipeline and superscalar computer concepts. Caches can be inhibited in beginner's laboratory works for avoiding memory hierarchy concepts.

Although some educational emulators which could serve for our practical works are available (spim, cl–spim, Dlx, DineroIII and SuperDlx), they are oriented to specific purposes. We were also looking for a tool running on conventional Unix stations and on personal computers with Linux.

Nowadays, the emulator is being used for laboratory works, to teach assembly programming using a RISC approach, cache behavior, pipeline and superscalar computers concepts. It is available for Solaris, Aix and Linux operating systems.

The emulator has an embedded debugger. It allows the user to control the program execution, and to observe the state of the different components of the computer at every clock cycle. The user can set breakpoints, execute the whole program or just a cycle, display and modify registers and memory contents, and display the instructions at the different pipeline stages and the history buffer contents.

The emulator has currently a textual interface, although an X-window based interface that will provide equivalent functionality is being finished.

2 Emulator description

The system emulates the functional units and behavior of the MC88110 processor. We chose the MC88110 because at the beginning of this project (1993) this processor had recently appeared and there was good documentation about it. It included the most interesting characteristics of superscalar processors like out-of-order completion of instructions, branch prediction, a mix of in-order and out-of-order issue, and used shelving for some instructions.

This superscalar processor can issue two instructions every clock cycle, a suitable throughput for our purposes. Instructions are issued in the order in which they appear in the program, but they can be finished out-of-order due to the different functional units latency. The processor also implements a partial out-of-order issue model for branch and store instructions, that can be issued even when its operands are not available.

Instructions are dispatched to ten different functional units that work in parallel, although the two graphics units have not been emulated.

The instruction pipeline is a conventional four stages RISC pipeline:

- *Fetch.* Two instructions are read together from the instruction cache.
- *Decode.* The instructions previously read are decoded and their source registers are read from the register file. The branch target address is computed to perform static branch prediction.
- *Execution.* If the operands and functional units are available, both instructions are dispatched and executed. At this stage branch instructions compute the branch condition while load and store instructions execute their memory accesses.
- *Write back.* The execution results are written into the register file.

Latency is defined to be one cycle for all except for the execution stage. In this case it depends on which functional unit is involved.

The evolution of instructions through pipeline stages can be displayed at every machine cycle, marking explicitly those executed due to a branch prediction.

Instructions dispatching can be stalled due to *structural*, *data* or *control* hazards. The sequencer dispatches instructions according to the order in which they

appear in the program, except for store and branch instructions. In these cases their functional units have two reservation stations, avoiding these instructions to produce stalls in the pipeline due to data dependencies.

In order to diminish the overhead produced by *structural* hazards most of the functional units, except divide, are pipelined, and a two writing ports register file has been implemented. Also, there are two caches, so implementing a Harvard architecture. Most of the cache parameters are also configurable: cache access time, whole and line sizes, organization policies and write policy.

Both the actual processor and the emulated one include the *scoreboarding* mechanism to track RAW and WAW *data* dependencies. Recent superscalar processors include hardware mechanisms to eliminate WAW dependencies by register renaming. The inclusion of this hardware mechanism makes tracking of program execution harder, which we do not consider appropriate due to the academic purpose of the emulator. We deal with register renaming statically, that is, at programming time.

Concerning *control* hazards, the emulated processor includes delayed branch instructions (one slot) as well as static branch prediction in the decode stage. This allows the student to use the branch instructions available in the instruction set to make their own predictions, comparing performance. The instructions fetched due to a branch prediction are tagged (conditionally executed). If the prediction was correct, the instructions that have been predicted are untagged and they are converted to normal instructions. If a missprediction has been detected, tagged instructions are aborted.

The emulator also implements the MC88110 *history buffer*, a FIFO queue storing the issued instructions in the program order and the previous value of the destination register, in order to restore the state previous to their execution when there is a missprediction.

When the first instruction of the history buffer completes its execution, the sequencer removes every instruction completed. If the instruction becoming the head of the history buffer is a branch whose prediction failed, all the tagged instructions are removed and the values of their destination registers are restored to those saved in the history buffer.

3 Program execution debugging and visualization

We have developed an Assembler which generates the binary files used by the emulator. This Assembler allows using a wide instruction subset of the actual MC88110, as well as some pseudoinstructions specified in IEEE-694 standard (`org`, `res` and `data`).

Figure 1 shows an assembly program fragment that performs the dot product of two vectors (V1 and V2). For instance, the instruction **`bb1.n 3, r3, loop`** branches if the third bit of `r3` (r4 not equal r0) is set. This instruction predicts that the branch will be be taken. The suffix `.n` means the following instruction will be executed before taking the branch (delayed branch).

```
        and r8, r0, r0        ;r8 contains the dot product
loop:   ld r5, r1, r0         ;r5 y r6 are loaded with an element
        ld r6, r2, r0         ;of both vectors
        sub r4,r4,1           ;The counter is decremented
        add r1, r1, 4         ;V1's pointer is incremented
        mulu.d r9, r5, r6     ;Multiply result is on r9 and r10
        add r2, r2, 4         ;V2's pointer is incremented
        cmp r3, r4, r0
        add.co r7, r7, r10    ;The result of mulu is accumulated
        bb1.n 3, r3, loop     ;if r4 <> 0 then branch to loop
        add.ci r8, r8, r9
        st.d r7, r11, r0
error:  stop                  ;End of emulation
```

Fig. 1. Assembly program to perform the dot product of two vectors

The embedded debugger allows the user to control program execution. Every time the program shows the prompt to the user, the emulator displays the processor internal state: register contents, status register and pipeline state. Figure 2 shows the information provided by the emulator: current instruction, program counter (PC), register file (only selected registers), processor status register, some cache statistics and the pipeline state. Also the contents of the history buffer at that instant can be visualized.

```
 PC=64          add      r01,r01,4          Tot. Inst: 13   Cycle : 31
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000074 h R02 = 0000009C h R03 = 00005998 h R04 = 0000000A h
 R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 00000000 h R11 = 0000006C h R12 = 00000000 h
  Instruction cache : 9 accesses, 3 misses, Hit ratio 66.6
  Data cache : 2 accesses, 1 misses, Hit ratio 50.0

FETCH:          68      mulu.d  r09,r05,r06
                64      add     r01,r01,4
DEC:
EXEC:           56      ld      r06,r02,r00
WBCK:           52      ld      r05,r01,r00
                60      sub     r04,r04,1

History buffer contents:

       52  ld  r05,r01,r00      Not executed    R05: 00000000
       56  ld  r06,r02,r00      Not executed    R06: 00000000
       60  sub r04,r04,1        Not executed    R04: 0000000A
```

Fig. 2. Emulator state after executing 30 machine cycles

```
FETCH: C        56      ld      r06,r02,r00
       C        52      ld      r05,r01,r00
DEC:   C        92      st      r07,r11,r00
       C        88      add.ci  r08,r08,r09
EXEC:            84      bb1.n   03,r03,-8
                 80      add.co  r07,r07,r10
WBCK:

History buffer contents:

        80  add.co      r07,r07,r10     Not executed    R07: 00000000
        84  bb1.n       03,r03,-8       Not executed
```

Fig. 3. Emulator state after executing 52 machine cycles

When the first loop iteration finishes (see Figure 3), the instruction **bb1** has been issued to the branch unit. Previously an effective branch has been predicted. So, instructions stored at addresses 52 and 56 are tagged as conditional. As the prediction was right, the branch unit will remove that tags at the end of this cycle. Furthermore, the tag of the instruction 88 will be removed because it is a delayed branch. On the other hand, the instruction 92 will be aborted. The final pipeline state is shown in figure 4.

```
FETCH:          64      add     r01,r01,4
                60      sub     r04,r04,1
DEC:            56      ld      r06,r02,r00
                52      ld      r05,r01,r00
EXEC:           88      add.ci  r08,r08,r09
WBCK:           80      add.co  r07,r07,r10
                84      bb1.n   03,r03,-8

History buffer contents

        80  add.co      r07,r07,r10     Not executed    R07: 00000000
        84  bb1.n       03,r03,-8       Not executed
        88  add.ci      r08,r08,r09     Not executed    R08: 00000000
```

Fig. 4. Emulator state after executing 53 machine cycles

4 Conclusions

This paper presents a superscalar processor emulator for educational purposes. Most of the processor parameters are fully configurable, so it may be used to teach cache behavior as well as pipeline and superscalar computer concepts.

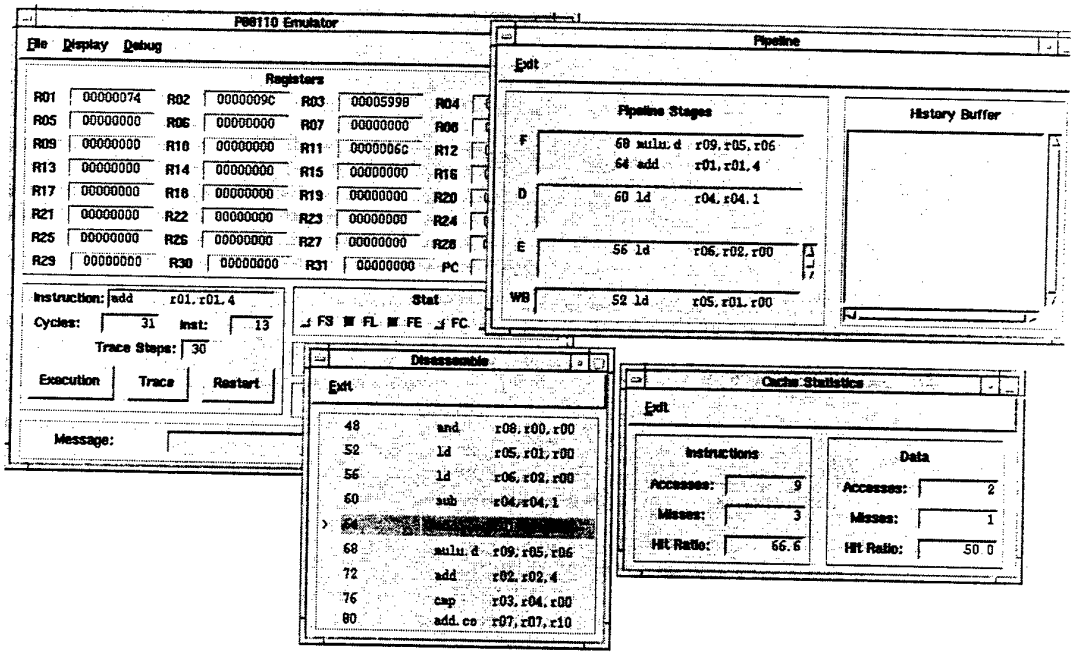


Fig. 5. em88110 emulator X–window interface

The emulator has currently a textual interface but we are implementing an X-window based one (Figure 5 shows the information it will provide).

Currently we are improving the emulator to allow selecting the number of instructions issued per cycle. The student will be able to choose whether one or two instruction will be issued, in order to emulate a conventional pipelined machine or a superscalar one.

References

1. Keith Diefendorff, Michael Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2):40–63, April 1992.
2. Moura, C. *SuperDLX. A generic superscalar Simulator.* ACAPS Technical Memo 64, McGill University Scool of Computer Science, 1993,
3. John L. Hennessy, David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Los Altos, USA, second edition, 1995.
4. David A. Patterson, John L. Hennessy. *Computer Organization and design. The hardware/software interface.* Morgan Kaufmann Publishers, Los Altos, USA, 1994.
5. *MC88110: Second Generation RISC Microprocessor. User's Manual.* Motorola Inc., 1991.
6. Mehdi R. Zargham. *Computer Architecture: Single and Parallel Systems.* Prentice-Hall International Editions, 1996.

A Parallel Genetic Algorithm for solving the partitioning problem in Multi-FPGA systems.

J. I. Hidalgo, M. Prieto, J. Lanchares and F. Tirado

Departamento de Arquitectura de Computadores y Automatica
Facultad de Ciencias Fisicas
Universidad Complutense
28040 Madrid, Spain
hidalgo@eucmax.sim.ucm.es, mpmatias@eucmos.sim.ucm.es,
{julandan,ptirado}@eucmax.sim.ucm.es

Abstract. In this paper we present the results we have obtained after applying a parallel genetic algorithm (PGA) to the Multi-FPGA partitioning problem. Solutions are based on Xilinx 3000 series FPGA's and satisfy some constraints allow the routing within the set of FPGA that constitutes the Multi-FPGFA system. To verify our studies we have used circuits from Partitioning Benchmark93 at the NCSU CAD Benchmarking Laboratory. The experimental results have been obtained using the CRAY T3E.

1.Introduction

Nowadays, FPGA systems are widely used because of their prototyping and correction capabilities. Every day, new FPGA´s are appearing in the market with higher density integration and tools with wider capabilities. However, these increasing capabilities do not support all the necessities of some designs, so it is necessary to distribute these designs among several FPGA´s. This is the major reason for Multi-FPGA systems [1]. The first step in the design flow is to partition the system. In other words, we have to decide how many FPGA's are needed to implement the system, their type and their distribution. We present a PGA to solve the partitioning problem of Multi-FPGA systems. These algorithms have been successfully used in other optimization problems [2]. If the partition process precedes the technology mapping, it is called functional partitioning, otherwise it is called structural partitioning [3].

In the case of structural partitioning of Multi-FPGA systems, this method allows us to obtain solutions with a great number of blocks. We can also use industrial tools, such as XACT [4], to accomplish the first stages of the design flow. Our partitioning algorithm then divides the results obtained after using XACT, on initial system specifications.

In the area of Multi-FPGA system partitioning there are a few tools which involve constraints, e.g., Kuznar´s research [5][6]. Its major drawback is that this method has been designed for heterogeneous systems but the implementation is undertaken on homogeneous systems.

This paper is organised as follows. In Section 2 we describe a parallel genetic algorithm. In Section 3 we show its application to the Multi-FPGA system-partitioning problem and the experimental results are presented in section 4. The paper ends with some conclusions and futures research.

2. Parallel Genetic Algorithms

Genetic algorithms [7] are optimization techniques which imitate the way that nature selects the best individuals (the best adaptation to the environment) to create descendants which are more highly adapted. The first step is to generate a random initial population, where each individual is represented by a character chain like a chromosome and with the greatest diversity, so that this population has the widest range of characteristics. Then, each individual is evaluated using a fitness function, which indicates the quality of each individual. Finally, the best-adapted individuals are selected to generate a new population, whose average will be nearer to the desired solution. This new population is created making use of three operators: reproduction, crossover and mutation.

One of the major aspects of GA is their ability to be parallelised. Indeed, because natural evolution deals with an entire population and not only with particular individuals, it is a remarkably highly parallel process [8].

It has been established that GA efficiency to find optimal solution is largely determined by the population size. With a larger population size, the genetic diversity increases, and so the algorithm is more likely to find a global optimum. A large population requires more memory to be stored, it has also been proved that it takes a longer time to converge. The use of today's new parallel computers not only provides more storage space but also allows the use of several processors to produce and evaluate more solutions in a shorter time.

We use a coarse grained parallel GA. The population is divided into a few subpopulations or demes, and each of these relatively large demes evolves separately on different processors. Exchange between subpopulations is possible via a migration operator. In the literature, this model is sometimes also referred as the island Model. Sometimes, we can also find the term 'distributed' GA, since they are usually implemented on distributed memory machines.

Technically there are three important features in the coarse grained PGA: the topology that defines connections between subpopulations, migration rate that controls how many individuals migrate, migration intervals that affect how often the migration occurs.

Many topologies can be defined to connect the demes. We present result using a simple stepping stone model and a master-slave model. In the former, the demes are distributed in a ring and migration is restricted to neighboring demes. In the latter there is a master population connected to all the slaves.

Choosing the right time for migration and which individuals should migrate appears to be more complicated and a lot of work is being done on this subject. Several authors propose that migrations should occur after a time long enough to allow the development of goods characteristics in each subpopulation[9]. However, it also appears that immigration is a trigger for evolutionary changes. In our algorithm

the migration occurs after each new generation, therefore the algorithm is more or less equivalent to a sequential GA with a larger population.

In our problem, migrants are selected from the best individuals in the population and they replace the worst in the receiving deme. The number of migrants may be selected at execution time. With this operator, our PGA has better convergence properties than the sequential version.

3. Genetic partitioning for Multi-FPGA systems

Figure 1 describes the design and implementation flow of a Multi-FPGA system. It starts from an initial specification (a netlist or a HDL description), that is used as XACT input. It returns the number of CLB´s and IOB´s. Then, it is necessary to determine the optimum distribution of the CLB´s on the different available FPGA´s. An optimum distribution has a minimal cost and guarantee the internal routability of each FPGA. For this purpose we use the PGA described in section 2.

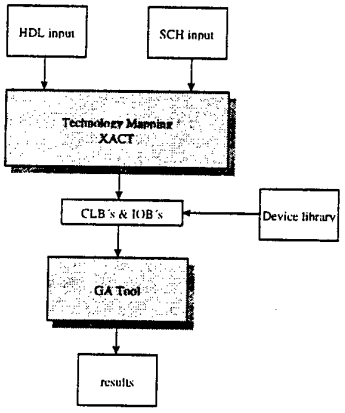
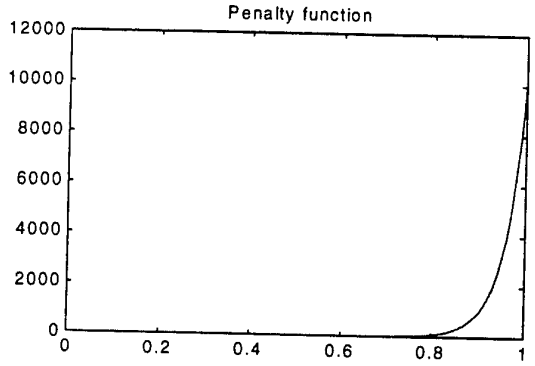


Fig. 1. Design and implementation flow of a Multi-FPGA system

The input to our algorithm must include the number of necessary CLB´s to implement the circuit. In order to evaluate the different solutions, it is also necessary to have a FPGA library. It must include the number of CLB´s and the cost of each FPGA. In our case it has been used the corresponding data to the three simplest devices of the series 3000 of Xilinx; XC3020, XC3030 and XC3042. After the optimisation, the algorithm returns the number of circuits of each type, the distribution of the CLB´s and the percentage of utilisation of the FPGA´s.

Our problem has been coded as follows: each individual represents a distribution of CLB´s in the set of FPGA´s. We have supposed we have three different types of Xilinx 3000 series FPGA´s and we can use as many as necessary [10]. Each individual is a chromosome with so many genes as the number of CLB´s in the original circuit. Each CLB is represented by a gene, which has a different value depending on which kind of FPGA it uses.

We solve the partition and placement problem simultaneously with the routing problem. A FPGA is routable whenever the percentage (*pc*) of busy CLB´s does not exceed 0.8. This is one of the constraints that our fitness function satisfies as figure 2 shows. Moreover, it minimizes the final cost (*cost*) of the circuit, according to 3000 series specifications and the number of holes (free CLB's). The term *penalty* that appears in the fitness function acts when the system is not routable.



$$F = \frac{1}{\sum_{i=0}^{2} [(hole) \cdot k_i + Penalty_i + Cost_i]}$$

$$Penalty_i = K_1 * e^{K_2 * pc}$$

Fig. 2. Cost and penalty functions used in the genetic algorithm.

The values of GA parameters are the followings: the crossover probability (P_c) is equal to 0.8, the mutation probability (P_m) is equal to 0.015, the population size is set to 60 individuals. The constants K_i, K_1 and K_2 have been adjusted experimentally to satisfy the constraints.

4. Experimental Results

The circuits that we have used for testing our algorithm have been obtained from the Partitioning Benchmarks 93 suite. The characteristics of these circuits after using the XACT tool are shown in table 1.

Table 2 compares our results with those obtained by Kuznar. The comparison is made in terms of cost and occupation of CLB´s.

Table 3 compares the sequential algorithms to the parallel versions (using 8 processors in the Cray T3E). The results show that the second approach has better convergence properties due to its non-overlapping replacement characteristic.

Circuit	Num CLB´s	num IOB´s
C3540	283	72
C5315	377	301
C7552	833	64
C6288	489	313
S5378	381	86
S9234	454	43
S13207	915	154
S15850	842	101

Table 1. Characteristic of the test circuits

Circuit	AG cost	AG Pc	Kuznar cost	Kuznar Pc
C3540	5.20	0.76	4.99	0.77
C5315	6.56	0.79	7.76	0.52
C7552	14.6	0.79	13.66	0.83
C6288	9.40	0.70	7.88	0.85
S5378	7.56	0.71	6.19	0.94
S9234	9.40	0.66	7.98	0.85
S13207	15.96	0.79	16.81	0.81
S15850	14.12	0.83	14.97	0.80

Table 2. Comparison between the Kuznar and the GA algortihms

Circuit	Sequential	Ring PAG	Master-Slave PAG
C3540	19.78 (500)	3.883	3.973
C5315	- (>2000)	10.298	10.548
C7552	85.31 (725)	11.465	-
C6288	61.78 (900)	6.682	8.525
S5378	38.75(725)	6.504	15.975
S9234	57.34(900)	18.574	18.995
S13207	116.627(900)	15.734	28.917
S15850	- (>2000)	25.980	26.546

Table 3. Comparison between the sequential and the parallel GA´s (time in seconds)

5. Conclusions

The main conclusions of our research can be summarised as follow: (1) The PGA improves Kuznar results. Although the cost is not always improved, the routability of the system is almost assured in all the cases. We always obtain a cost improvement or a routing improvement. (2) The logic blocks distribution that gives us the PGA assures the internal routability of the system in 88% of the cases. The cost in dollars of the resulting circuit has been reduced also in 45% in the experiments compared to the Kuznar results. (3) The sequential version of the GA needs more than 2000 generations to obtain an acceptable solution, but the 8 processors (in the worst case) Ring PGA only needs 225 generations and the Master-Slave PGA 300 generations. This result is due to simultaneous search and the implicit non-overlapping replacement of the PGA. (4) Finally, it is interesting to note that the Ring PGA gives better results than the Master-Slave, due to premature convergence effects.

Acknowledgments

This work has been supported by the Spanish research grants TIC 96-1071, TIC IN96-0510, UCM PR-181 / 96-6776, and the Human Mobility Network CHRX-CT94-0459. We would like to thank Ciemat for providing access to the parallel computer that have been used in this research.

References

[1] Hauck, S. "Multi-FPGA systems". Ph. D. Thesis. University of Washington.1994

[2] Hidalgo J. I., Lanchares J.. "Functional Partitioning for Hardware-Software Codesign using Genetic Algorithms" EuroMicro97, Budapest. IEEE Press, 1997.

[3] Wolf W. "Hardware-Software Co-Design of embedded Systems". Proc. Of the IEEE. Vol 82, n°7, July 1994.

[4] Xilinx ,"XACT User guide". 1994.

[5] Kuznar R., Brglez F., Kozminslei K. "Cost Minimization of Partitions into Multiple Devices" 30th ACM/IEEE DAC pp315-320. 1993.

[6] Kuznar R., Brglez F., Zafc B."Multi Way Netlist Partitioning into Heterogeneous FPGA´s and Minimization of total Device Cost and Interconnect", 31th ACM/IEEE DAC 1994. PP 238-243.

[7] Michalewicz, Z., "Genetic Algorithms+ Data Structures= Evolution Programs" Springer-Verlag. 1994.

[8] Xavier Hüe "Genetic Algorithms for Optimisation Background and Applications". Edinburgh Parallel Computing Centre. Version 1.0 February 1997. Available from: http://www.epcc.ed.ac.uk/epcc-tec/documents/

[9] Liening, J. "A Parallel Genetic Algorithm for Performance-Driven VLSI Routing". IEEE Trans on evolutionary Computation VOL.1 NO.1 April 1997.

[10] Xilinx, "Xilinx Data Book". 1994.

$Haskell_{\#}$: A Functional Language with Explicit Parallelism

R.M.F.Lima & R.D.Lins

Departamento de Informática, UFPE, Recife, PE, Brazil
e-mail: rmfl, rdl@di.ufpe.br

Abstract In his 1978 Turing Lecture[1], John Backus draw the attention of the computer science community to functional languages. One of the claims he made was that pure functional languages offer a greater potential for parallelism than other programming paradigms, because their property of referential transparency means less interdependence between parts of a program. Meeting this promise has been a challenge. This paper, presents $Haskell_{\#}$ a parallel functional language with explicit parallelism based on MPI (Message Passing Interface) for communication between functional blocks of code.

1 Introduction

Functional languages are a nicer syntax to the λ-Calculus, a function theory widely used to provide semantics to programming languages of all paradigms[2]. The Church-Rosser theorems state that normal forms of λ-expressions are unique modulo variable renaming and that reductions of the leftmost-outermost reducible expression at each point of the reduction sequence leads to normal form, if it exist[2]. Thus, the reduction can be done even in parallel. This suitability of functional languages for parallel processing have led various researchers to propose different parallel implementation of functional languages.

The history of parallel functional programming complies two phases. The first period, the 1980s and before, corresponds to the time in which parallelism was sought as a way to make functional languages run as fast as imperative ones. The second period is the time in which "real" parallel processing can find an alternative in functional programming.

The first attempt to exploit parallelism of functional programs targeted either the evaluation of actual parameters before replacing them by the formal parameters or was done at combinator argument level. These strategies lead to a very fine grained parallelism. As a result, it was believed that novel architectures were necessary to achieve high performance with functional languages, and this led to a spate of designs for special-purpose machines, such as ALICE (Transputer-based), ICL Flagship, EDS/Goldrush and GRIP.

Unfortunately, these experiments proved that building special purpose hardware is costly and, too slow a process to meet the advantages of commercial general purpose machines.

Nowadays, sequential implementations of functional languages present performance figures in the same order of magnitude as important imperative languages, such as C or FORTRAN. In this new context, parallel processing might find a partner in parallel programming and some parallel implementation of functional programming languages have been made publically available[10].

These implementations, have some common features:

- In order to provide a higher workload, they create speculative tasks dynamically. However, in our opinion, dynamic management of speculative tasks represents an expensive overhead.
- Explicit parallel implementations employ a higher-level parallel environment (e.g. PVM), which actually manages the load-balance depending on system workload. Users give only some indicatives to potential parallel tasks.
- Parallel graph reduction proceeds on a shared program/data graph, so a primary function on the run-time system of these parallel functional languages is to manage the virtual shared memory in which the graph resides[6].

In contrast with other parallel implementations of functional languages, we suggest here a novel approach which makes explicit static task allocation, through *Message Passing Interface* (MPI)[5]. MPI provides a low-level parallel programming model and will be used to manage the creation, distribution and communication between tasks. Furthermore, each task will possess its own local sequential run-time system.

Finding the suitable-grained sized of tasks is certainly not easy. Some approaches have been proposed to deal with the granularity problem. Some systems rely on the programmer's ability for indicating expressions which are worth evaluating in parallel. These systems provide explicit constructs/annotations into the language. Other make use of some heuristics to estimate the execution costs of some expressions and then automatically annotate the program. Other yet use execution-profiling information for providing empirical measurement of the time spend in each function and then decide which expressions must be annotated.

1.1 Implicit Parallelism

The strategy to exploit implicit parallelism of functional programs takes different form in strict and non-strict languages:

- In a *strict* language it is easier to partition a program, but it yields a larger number of very fine grained tasks;
- In a *non-strict* language, parallelism is obtained from needed expressions, which are detected by strictness analysis - however, neededness is undecidable in general, and strictness analysis has failed to deal effectively with realistic parallel programs.

Despite efforts of several research groups around the world, trying exploit implicit parallelism in functional languages, results are still very shy. In our opinion this is a direct consequence of communication overhead brought by very fine granularity tasks generated for this strategy.

1.2 Explicit Parallelism

In explicitly parallel languages - such as Occam[4] - it is up to users setting parallel tasks. Results of implicit parallel implementations of functional languages as well as the belief that the bottom line of any parallel system is raw performance, and a program's performance can only be improved if it can be understood[10], led a number of researchers to exploit explicit parallelism.

Improving a sequential program by partitioning it in parallel tasks is not a simple work and requires a complete knowledge of the program as well as the architecture it will execute. Annotation for parallelism are usual. Hope+ on Flagship employs strictness annotation to control the precise degree of evaluation.

2 Haskell

Haskell is a general purpose, pure functional programming language which incorporates higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, monads, and a rich set of primitive datatypes, including arrays, arbitrary and fixed precision integers, and floating-point numbers[3, 9]. Haskell has now become *de facto* standard for the non-strict functional language.

Among the implementations of Haskell compilers Concurrent Haskell[8] and GUM[10] seems to be very promising.

Concurrent Haskell is a concurrent extension to lazy functional Haskell, which provide a more expressive substrate to build sophisticated I/O-performing programs, notably ones that support graphical user interfaces for which the usefulness of concurrency is well established. The goal of the designers of Concurrent Haskell is to attain implicit, semantically transparent parallelism, but the version available now uses explicit parallelism.

GUM is a portable, massage-based parallel implementation of Haskell. Portability is facilitated by using PVM communications harness that is available on many multi-processors. GUM is available for both shared-memory distributed-memory (network workstations) architecture. Initial performance figures demonstrate speedups relative to sequential compiler technology.

3 MPI

Message Passing is a paradigm widely used on loosely coupled parallel machines. Although there are many variations, the basic concept of processes communi-

cating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm.

The main advantages of using a message-passing standard are: efficiency, portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and/or abstraction are built upon lower level message passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard, such as that proposed in [5], provides vendor with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability. MPI also:

- provides an application programming interface;
- allows efficient communication: avoid memory-to-memory copying and allows overlap of computation and communication and offload to communication co-processor, where available;
- allows for implementations that can be used in heterogeneous environments;
- allows convenient C and Fortran 77 bindings for the interface;
- assumes a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem;
- defines an interface that is not too different from current practice, such as PVM, NX, Express, p4, etc., and provides extensions that allow greater flexibility;
- defines interfaces implemented on many vendors' platforms.

The parallel programming model supported by our implementation is message passing: a set of tasks, each executing in its own address space, communicating via calls to the Message-Passing Library. Such a parallel programming model offers a multitude of alternatives: some functions supported by microcode on the adapter and some by software on the computing processor; some functions executed in user space and some by kernel; trade-offs between more extensive use of buffering and data copying and more eager use of interrupts; "push" versus "pull" protocols; flow control; etc.

4 Haskell#

$Haskell_{\#}$ is a new language composed by parallel constructors (a subset of MPI with a more suitable syntax) and functional programs (Haskell programs). An IBM SP2 System with 9 (nine) processor nodes was chosen as testbed.

$Haskell_{\#}$ has some important differences from other implementations:

- an explicit static task allocation is adopted;

- MPI is used to manage a coarse task program distribution;
- each task is actually a functional program, with a local run-time system completely independent of the manager task module.

4.1 Parallel Module

Now, we will describe the main ideas on using MPI to implement $Haskell_{\#}$.

Program Structure Communication functions of MPI will be used to express the parallelism following the same mechanism present in Occam[4] programming language. According to the parallel constructors inserted by the user in a given $Haskell_{\#}$ program, MPI spawns the required number of processes to the available processors. Thus, $Haskell_{\#}$ enables an application to be described as a collection of processes, where each process executes concurrently, and communicates with other processes through channels. Each process in such an application describes the behaviour of a particular aspect of the implementation, and each channel describes the connection between each of the processes.

Communication Library MPI supports two classes of message passing functions: point-to-point calls, which send a message from one task to another task, and collective communication calls, which establish a communication pattern within a group of tasks.

MPI point-to-point communication includes blocking and non-blocking send and receive functions. Use of non-blocking sends and non-blocking receives are both safe (in terms of deadlock avoidance) and efficient. Some extra programming effort is required, since the programmer must determine the status of the communication before reusing the buffer (the memory location in the user's program that holds the message data before transmission or after receipt)

Blocking routines protect naive programmers from accidentally altering message buffer contents. The trade-off can be increased communication cost. Deadlock can occur in cases where a large message volume is being sent. The situations most appropriate for blocking routines are those in which there is little work that can be done between initiation of the communication and use (or reuse) of the buffer.

In this first approach, $Haskell_{\#}$ uses MPI point-to-point call functions. Furthermore, in order to provide safety and higher performance, we adopt the MPI non-blocking communication library.

4.2 Run-Time System

The Recife Haskell Compiler (RHC) run-time system was adopted as evaluation environment of the value expressions executing in a SP2 processor node. Each process represents an individual sequential Haskell program, evaluated by $\mu\Gamma$CMC, an abstract machine for efficient implementation of lazy functional languages. $\mu\Gamma$CMC transfers the control of the execution flow to C, as much as

possible, to take advantage of the extremely low costs of procedure calls in modern RISC architectures. This yielded a substantial improvement in performance.

Almost all implementations of parallel graph reduction proceeds on a shared program/data graph[10], thus a primary function of the run-time system of these parallel functional languages is to manage the virtual shared memory in which graphs resides. However, in contrast with previous implementations, $Haskell_{\#}$ do not proceed parallel graph reduction on a shared program/data graph. Here, individual task (process) has its own local stacks and heap. As a result, $Haskell_{\#}$ performs garbage collection locally.

4.3 Conclusions

In this paper, we presented the fundamental ideas behind $Haskell_{\#}$ and drew comparisons with its supposedely competitors. $Haskell_{\#}$ is a simple explicit parallel functional language where the MPI-based communication combinators "glue" together large chunks of pure Haskell code, allowing a hierarchical programming discipline that rescues the ability of reasoning about parallel functional programs, feature lost by our competitors by including the parallel combinators in the language themselves.

Reference [7], presents further details of $Haskell_{\#}$ language such as its semantic model of parallelism as well as performance figures for benchmarks running on a 9-node IBM-SP2 platform.

References

[1] J. Backus. *Can Programming be Liberated from the von Neumann style ? A Functional Graph Style and its Algebra of Programming.* CACM 21(8):613-641,1978.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and The Foundations of Mathematics, vol. 103.

[3] P. Hudak, S. Peyton Jones & P. Wadler. *Report on the Programming Language Haskell, version 1.2.* ACM SIGPLAN Notices, 27(5), 1992.

[4] *Inmos Limited, Occam 2 Reference Manual.* 1988.

[5] *MPI: A Message-Passing Interface Standard.* Message Passing Interface Forum, University of Tennessee, 1995.

[6] R. Jones & R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[7] R.M.F.Lima & R. D. Lins. *The Parallel Functional Language $Haskell_{\#}$.* To appear.

[8] S. L. Peyton Jones, A. Gordon & S. Finne. *Concurrent Haskell.* 23rd ACM SPPL, pp. 295-308, 1996.

[9] S. Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 1996.

[10] P.W. Trinder. et al. S. L. Peyton Jones. *GUM: a portable parallel implementation of Haskell.* D. Comp. Science, Glasgow University, 1995.

Parallel and Distributed Algorithm in State Estimation of Power System Energy

José Beleza Carvalho [1], F. Maciel Barbosa [2]

[1] Instituto Superior de Engenharia do Porto
R. de S. Tomé, 4200 Porto, Portugal

[2] Faculdade de Engenharia da Universidade do Porto
Rua dos Bragas, 4099 PORTO Codex, Portugal

Abstract. The State Estimation is nowadays considered the fundamental element of modern electrical power networks control centers. In this paper we develop a theoretically robust and computationally efficient state estimator algorithm, to solve the WLS problem by using parallel processing. The computational aspects of the parallel processing, was analysed and tested using the IEEE 30, 57 and 118 bus systems. Computational experiments are compared with standard WLS methods, in the integral and distributed version. An evaluation of the degree of natural decoupling in the state estimation problem is also performed. The results indicate that a distributed processing for state estimation, is the better way to adopt the parallel computing in power systems energy.

1. Introduction

The implementation of robust methods for power system state estimation, which maintain performance suitable to the large models encountered in modern control centres is a topic that has received significant attention. The estimator processes real-time redundant telemeter and pseudo measurements to provide a complete, coherent and reliable system database, which can describe the electrical state of the network [1]-[2]. These measurements, which include voltage magnitudes, real and reactive line flows and nodal power injections, are measured from the network at a certain moment, thus getting an estimate for the respective state vector (vector of voltages modules and phases on different buses) [3]. The higher frequency in state estimation execution requires the development of faster state estimation algorithms. The larger size of the supervised networks will increase the demand on the numerical stability of the algorithms. At same time, conventional centralised state estimation methods have reached a development stage in which important improvements in either speed or numerical robustness are not likely to occur. These facts, together with the technical developments in fast data communication network technology, opens up the possibility of parallel and distributed implementations of the state estimation algorithms [4]-[5]. The nature geographically distributed of power system applications, can benefit from this form of decentralised computer architecture, in which several remote processors perform local state estimation in network areas and the results are send to a central computer that refines the calculation. The power system under consideration may be partitioned into k areas, and each area is supervised by a local control center. The measurement data in each area will be collected in each individual local control center that has at least one computer system for data acquisition, data processing, and computation [9]. The computer systems of adjacent areas are connected by fast data communication links, and these decentralised computer systems form a computer network.

2. WLS State Estimation Problem

Mathematically, the information model used in power system state estimation is represented by the equation:

$$z = h(x) + e \tag{1}$$

Where z is a (m*1) measurement vector, x is a (n*1) true state vector, $h(.)$ is a (m*1) vector of non-linear functions, e is a (m*1) measurements error vector, m is the number of measurements, and n is the number of state variables. The static-state estimation problem of a N bus power system, is a weight-least-squares (WLS) optimisation problem:

$$\min J(x) = \sum_{i=1}^{m} w_i (z_i - h_i(x))^2 = [z - h(x)]^T W [z - h(x)] \tag{2}$$

Weight w_i represent the weight associated with measurement z_i. Weights are chosen as proportional to the accuracy of the measurements: the higher the accuracy of a measurement the bigger its weight. The solution of this optimisation problem gives the estimated state $\hat{x}$, which must satisfy the following optimality condition:

$$\frac{\partial J(x)}{\partial x} = 0 \Rightarrow H^T(\hat{x}) W [z - h(\hat{x})] = 0 \tag{3}$$

Where

$$H(x) = \frac{\partial h(x)}{\partial x}$$

is the Jacobean matrix of the measurement function h(x). The solution of the non-linear equation (3) may be obtained by an iterative method in which a linear equation of following type is solved at each iteration to compute the correction,

$$x^{t+1} = x^t + \Delta x^t$$

$$[G(x^t)]\Delta x^t = H^T(x^t) W [z - h(x^t)] \tag{4}$$

where $G(x)$ is called the gain matrix and is usually chosen as

$$G(x) = H^T(x) W H(x)$$

Eq.(4) is called the normal equation of the WLS problem. As in loadflow calculations, it has been found that state estimation algorithms based on decoupled versions behave adequately for the usual power networks [2]. Therefore, the decoupled model that has been mostly adopted is:

$$z_p = h_p(\theta, v) + e_p \tag{5}$$

$$z_q = h_q(\theta, v) + e_q \tag{6}$$

where θ (n_θ*1) and v (n_v*1) are the vectors of true voltage magnitudes and phase angles, p and q indicating partitions of vectors and matrices corresponding to active and reactive measurements, respectively;

$$n_\theta = N\text{-}1 \quad ; \quad n_v = N,$$

N is the number of network nodes. This naturally decoupled characteristic, make this method suitable for parallel processing implementation, with a great reducing of the required computation time.

3. Parallel and Distributed State Estimation Problem

If we decompose the power network into "K" areas, connected through boundary buses which belongs simultaneously to both adjacent areas, the state estimation problem introduced in (5) and (6) can be presented as

$$z_p^k = h_p^k(\theta^k, v^k) + e_p^k, \qquad k=1,.....K \tag{7}$$

$$z_q^k = h_q^k(\theta^k, v^k) + e_q^k, \qquad k=1,.....K \tag{8}$$

where z_p and z_q are vectors of active and reactive measurements in area k; θ^k and v^k are vectors of voltage phase angles and magnitudes in area k, including the ones corresponding to the boundary buses. The number of boundary buses may be kept to a minimum and there are no injection measurements in the overlapping area buses. This is not a limitation, because actual injection measurement buses in overlapping areas, can be replaced by fictitious buses with no injection measurements connected to the actual buses, now placed outside the overlapping area, by zero impedance lines [10]. Then, the problem of distributed state estimation is to use the computer network associated with the measurement data collected in each local control center to solve the following weighted least square (WLS) problem in a distributed way:

$$\min \sum_{k=1}^{K} \left[z_p^k - h_p^k(.)\right]^T \left[R_p^k\right]^{-1} \left[z_p^k - h_p^k(.)\right] = 0$$

$$\min \sum_{k=1}^{K} \left[z_q^k - h_q^k(.)\right]^T \left[R_q^k\right]^{-1} \left[z_q^k - h_q^k(.)\right] = 0 \tag{9}$$

The iterative solution of above problem, for k =1.....K, is:

$$\theta^k(i+1) = \theta^k(i) + \left[G_p^k\right]^{-1} \left[H_p^k\right]^T \left[R_p^k\right]^{-1} \left[z_p^k - h_p^k(\theta_k(i), v_k(i))\right] \tag{10}$$

$$v^k(i+1) = v^k(i) + \left[G_q^k\right]^{-1} \left[H_q^k\right]^T \left[R_q^k\right]^{-1} \left[z_q^k - h_q^k(\theta_k(i), v_k(i))\right] \tag{11}$$

Where

$$G_p^k = \left[H_p^k\right]^T \left[R_p^k\right]^{-1} H_p^k \qquad G_q^k = \left[H_q^k\right]^T \left[R_q^k\right]^{-1} H_q^k$$

$$H_p^k = \frac{\partial h_p^k(\theta^k, v^k)}{\partial \theta^k} \qquad H_q^k = \frac{\partial h_q^k(\theta^k, v^k)}{\partial v^k}$$

are the Jacobean matrix, calculated for the initial conditions and kept constant in the iterative process. In the boundary buses, the elements (θ, v) obtained in (10) and (11) must be affected with a weight medium of the values calculated in the neighbouring areas k and j [8], and take the form,

$$\overset{b}{\theta}{}^k(i+1) = \theta^k(i+1) + \Delta\theta^k(i+1) \tag{12}$$

$$\overset{b}{v}{}^k(i+1) = v^k(i+1) + \Delta v^k(i+1) \tag{13}$$

Where

$$\Delta\theta_r^k(1+1) = \frac{g_{rr}^{\wedge}}{g_{rr}^k + g_{rr}^j}\left[\theta_r^k(i+1) - \theta_r^j(i+1)\right] \quad (14)$$

$$\Delta v_r^k(1+1) = \frac{g_{rr}^{\wedge}}{g_{rr}^k + g_{rr}^j}\left[v_r^k(i+1) - v_r^j(i+1)\right] \quad (15)$$

$g_{rr}^{\wedge}$ and g_{rr}^j are diagonal elements corresponding to boundary bus r of the inverse gain matrices of the neighbouring area k and j, respectively.

4. Analysis of Computation Experiments

The Parallel and Distributed State Estimation methodology analysed in this paper was tested and simulated with a PVM 3.1 (Parallel Virtual Machine) software, with program coded in Fortran 77 and running in a DEC Alpha machine with a Ultrix operating system. The distributed computer system, connected in a network, used in practice for parallel or distributed areas processing, was simulated with recurrence to PVM performances [6], that enables one to distribute tasks on various processors, to control message-passing between tasks, to synchronise tasks, etc. The convergence, accuracy and numerical efficiency of the proposed simulation study are presented in the following sections.

4.1 Parallel Processing in the Integral Version

The algorithm implemented for this integral study version is represented in figure 1. The nature decoupled of equation (10) and (11) make the algorithm suitable for parallel implementation. The algorithm presented in flowchart, calculates the θ and v, update at every iteration in a synchronous way. The IEEE 30, 57 and 118 bus standard networks were used to perform this test. Two levels of global redundancy were specified for each measurement system: normal and low level. Table 1 shows the data for each test case. In this table J is the sum of squared errors in the estimates of measured variables.

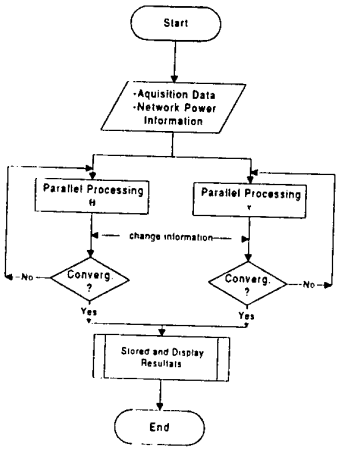


Fig. 1. Parallel Processing. Integral version.

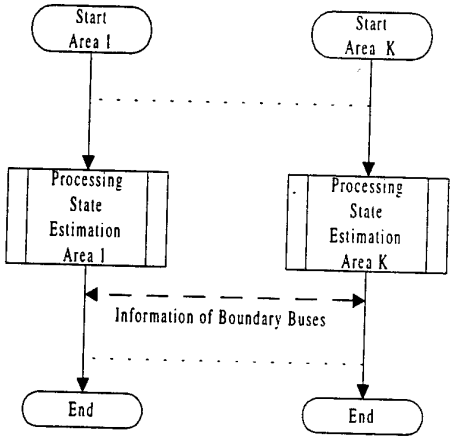


Fig. 2. Parallel Processing. Distributed version.

All test simulations converge in 2 iterations for standard WLS method and 8 iterations for standard decoupled estimator (MDE) and Parallel Processing. The convergence is obtained at 0.001 pu and 0.001 rad. for module and phase of voltage. From we can see that figure 3, the Parallel Processing in integral version is not so accurate like the MDE method. In a

synchronous process, due to idle times, the algorithm has to wait until the state vector is updated before it starts a new iteration. If we run the above algorithm in an asynchronous way, the precision of state estimation vector will be drastically deteriorated.

Table 1: Test Case Data

Test Case	N° of Bus	Redundancy	WLS		MDE		P.Process.	
			t(s)	J	t(s)	J	t(s)	J
A1	30	1.4	0.39	30.5	0.24	32.7	0.57	32.7
A2	30	2.4	1.00	95.0	0.34	97.0	0.69	97.0
B1	57	1.7	5.40	91.5	1.60	100	1.60	100
B2	57	2.3	9.00	172	2.70	185	2.70	185
C1	118	2.3	115	419	30.0	425	34	425
C2	118	3.2	220	802	70.0	806	74	806

4.2 Parallel Processing in the Distributed Version

Synchronous computation become too expensive when the processors are geographically distributed [7]. So, asynchronous concurrent processing is an attractive alternative. We analyzed this fact, dividing the test cases presented in table 1 in some areas and processing the equations (10) and (11) for each area, like shown in flowchart of figure 2. For the boundary buses, in the end of the asynchronous iterative process, we applied the restriction (12) and (13). The convergence obtained for 0.001 pu and 0.001 rad, the processing time, accuracy and numerical efficiency are shown in table 2 for WLS version, and table 3 for MDE version. The results presented demonstrate that in parallel distributed state estimation, we can get an elevated reduction of processing time, for essentially the same number of iterations, compared with integral methods showed in table 1. The accuracy of results, generally, is better for cases with more redundancy of measurements and for WLS state estimation version. The improvement in processing time for MDE method, compensate the small depreciate of results, compared with WLS version. In figure 3 we can see the performance of Parallel Processing in the Distributed Version (PPD), applicated to test case C2 (118 buses), comparing the processing time for standard WLS and MDE state estimation methods and the Parallel Processing in the Integral (PPI) and Distributed version. comparing the processing time for standard WLS and MDE state estimation methods and the Parallel Processing in the Integral (PPI) and Distributed version.

Table 2: Parallel Processing of Distributed Areas.
Estimation accuracy for WLS version.

Test Case	N° of Iter	Time (s)	Average error in phase angles (rad*1000)	Average error in voltage magnitud (pu*1000)	J
A1	5 ; 8	0.11	1.88 - 10.9	1.31 - 1.79	14.5 – 19.1
A2	5 ; 7	0.13	1.49 - 6.53	0.99 - 1.01	39.0 - 57.6
B1	6 ; 8	0.37	1.08 – 2.1	1.45 – 2.07	38.3 – 49.1
B2	5 ; 8	0.54	1.02 – 1.9	1.0 – 0.78	81 - 95
C1	5; 9; 11; 6	2.00	0.56-2.37-0.30-1.55	0.81-0.97-0.82-0.96	63-75-114-146
C2	5; 6; 11; 5	4.00	0.41-1.85-0.18-0.31	0.95-0.96-0.94-1.08	167-161-228-335

Table 3: Parallel Processing of Distributed Areas.
Estimation accuracy for MDE version.

Test Case	N° of Iter	Time (s)	Average error in phase angles (rad*1000)	Average error in voltage magnitud (pu*1000)	J
A1	2 ; 4	0.17	1.92 - 8.27	1.26 - 1.98	12.8 - 17.7
A2	2 ; 4	0.29	1.33 - 5.13	0.99 - 1.26	35.8 - 57.8
B1	2 ; 3	1.15	0.92 – 1.90	1.30 – 2.00	35.5 – 50.1
B2	2 ; 3	1.70	0.88 – 1.70	0.95 – 1.06	71 - 86
C1	2; 2; 2; 4	8.00	0.59-1.69-0.35-1.02	0.81-0.95-0.81-1.29	51-69-111-174
C2	2; 2; 2; 3	15.00	0.36-0.84-0.18-0.3	1.02-0.95-0.96-0.3	152-143-226-290

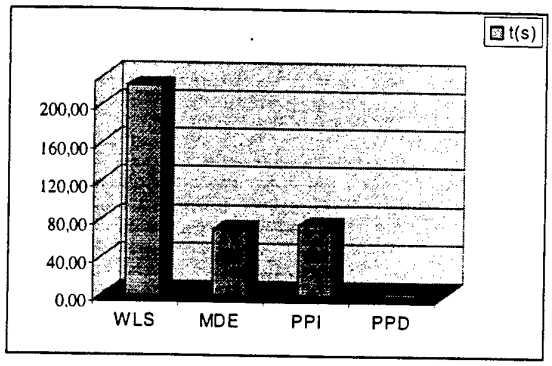


Fig. 3. Parallel Processing Improvement.

5. Conclusions

In this paper some methodologies for parallel state estimation were introduced and tested, based in conventional algorithms, like standard WLS version and standard decoupled MDE version. The results of computational experiments show that for integral processing of state estimation, the parallelism of algorithms does not bring any improvement, compared with the conventional decoupled MDE algorithm. A distributed computing is the better way to adopt the parallel computing in power systems energy. This fact was simulated tearing the IEEE standard test cases in some areas. The PVM software tool, enables the simulation of distribute tasks on various processors. The idle times of processors, synchronous computations become too expensive when the processors are geographically distributed, so we tested the asynchronous processing. For boundary buses, we apply the restrictions indicated in (12) and (13). The computational results show that with this distributed methods we get a very high improvement in manner of time processing, compared with integral standard version. The only drawback is the discrepancy in values of boundary bus state variables estimated using different sets of measurements, but in cases with higher redundancy levels, the values of the discrepancies are acceptable and the effect on computational efficiency is minimal.

6. References

[1] Do Couto Filho, M.B.; Leite da Silva, A.M. e Falcão, D.M., - *Bibliography On Power System State Estimation (1968-1989).* IEEE Trans. on PWRS, Vol.5, nº 3, August 1990.

[2] A. Bose, K.E. Clements – *Real Time Modeling of Power Networks.* Proceedings of the IEEE,Vol. 75, nº12, December 1987.

[3] Masiello, R.D. e Schweppe, F.C. - *A Tracking Static State Estimator*, IEEE Trans. on PWRS, Vol. PAS-90, March / April 1971.

[4] D.P. Bertsekas and J.N. Tsitsikilis, *Parallel and Distributed Computation*, Prentice Hall, 1989.

[5] Felix F. Wu and al, *Parallel Processing in Power Systems Computation.* IEEE Trans. On PWRS, Vol.7, August 1992.

[6] Jack Dongarra, Adam Berguelin. – *PVM: Parallel Virtual Machine.* Vaidy Sunderam, 1994.

[7] V.C. Ramesh, - *On Distributed Computing For On-Line Power System Applications.* Electric Power & Energy Systems, Vol. 18, 1996.

[8] Falcão, Djalma M. - *Parallel And Distributed Processing Applications in Power Systems Simulation and Control.* COPE, Universidade Federal do Rio de Janeiro, July 1995.

[9] Falcão, D.M.; Felix F. Wu e Liam Murphy, - *Parallel And Distributed State Estimation.* IEEE PES Summer Meeting, San Francisco, July 1994.

[10] Monticelli, A. e Garcia, A. - *Modeling Zero Impedance Branches in Power System State Estimation,* IEEE Trans. on PWRS, Vol. 6, November,1991.

Parallel Block Two-Stage Preconditioners for the Conjugate Gradient Method

M. Jesús Castel, Violeta Migallón, and José Penadés

Departamento de Ciencia de la Computación e Inteligencia Artificial,
Universidad de Alicante, E-03071 Alicante, Spain
{chus, violeta, jpenades}@dtic.ua.es

Abstract. Linear systems of the form $Ax = b$, where the matrix A is symmetric and positive definite, often arise from the discretization of elliptic partial differential equations. A very successful method for solving these linear systems is the preconditioned conjugate gradient method. In this paper we study parallel preconditioners for the conjugate gradient method based on the block two-stage iterative methods. Sufficient conditions for the validity of these preconditioners are given. Computational results of these preconditioned conjugate gradient methods on two parallel computing systems are presented.

1 Introduction

We study the parallel solution of a linear system

$$Ax = b, \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ is a symmetric and positive definite matrix (i.e., $A = A^T$ and $x^T Ax > 0$, for all real $x \neq 0$) and x and b are n–vectors.

Preconditioned conjugate gradient methods (PCG) can be used for the solution of (1). Descriptions of these methods can be found e.g., in Concus, Golub and O'Leary [3] or Ortega [9]. The idea of the PCG method consists of applying the conjugate gradient method (see [5]) to a better conditioned linear system $\hat{A}\hat{x} = \hat{b}$, where $\hat{A} = SAS^T$, $\hat{x} = S^{-T}x$, and $\hat{b} = Sb$. The matrix $\mathcal{M} = (S^T S)^{-1}$ is called the preconditioner or preconditioning matrix. The PCG method may be applied without computing $\hat{A}$, but solving the auxiliary system

$$\mathcal{M}s = r, \tag{2}$$

at each conjugate gradient iteration, where $r = b - Ax$ is the residual at the corresponding iteration.

One of the general preconditioning techniques is the use of the truncated series preconditioning. These preconditioners consist of considering a splitting of the matrix A as

$$A = P - Q, \tag{3}$$

and performing m steps of the iterative procedure defined by the splitting (3), toward the solution of $As = r$, choosing $s^{(0)} = 0$. It is well known that the solution of the auxiliary system (2) is effected by $s = (I + R + R^2 + \cdots + R^{m-1})P^{-1}r$, where $R = P^{-1}Q$, and the preconditioning matrix is $\mathcal{M}_m = P(I + R + R^2 + \cdots + R^{m-1})^{-1}$, cf. [1].

It is in these terms that in Section 2, we construct the preconditioner based on the two-stage methods and we study its validity. Moreover, in Section 3 we evaluate the performance of the resulting PCG algorithms on two different parallel distributed memory multiprocessors.

2 Parallel block two-stage preconditioners

Let us consider the splitting (3), where P is a block diagonal matrix, denoted by

$$P = \text{Diag}(P_1, \ldots, P_p), \tag{4}$$

and P_j, $1 \leq j \leq p$, are square nonsingular matrices of order n_j, $\sum_{j=1}^{p} n_j = n$. Note that performing m steps of the iterative procedure defined by the above splitting to approximate the solution of $As = r$, corresponds to perform m steps of a Block-Jacobi type method. Thus, at each step l, $l = 1, 2, \ldots, m$, of a Block-Jacobi type method, p independent linear systems of the form

$$P_j s_j^{(l)} = (Qs^{(l-1)} + r)_j, \quad 1 \leq j \leq p, \tag{5}$$

with $s_j^{(0)} = 0$, need to be solved; therefore each linear system (5) can be solved by a different processor. However, when the order of the diagonal blocks P_j, $1 \leq j \leq p$, is large, it is natural to approximate their solutions by using an iterative method, and thus we are in the presence of a two-stage iterative method; see e.g., [4], [6], [7], [8]. In a formal way, let us consider the splittings

$$P_j = B_j - C_j, \quad 1 \leq j \leq p, \tag{6}$$

and at each lth step perform, for each j, $1 \leq j \leq p$, $q(j)$ iterations of the iterative procedure defined by the splittings (6) in order to approximate the solution of (5). That is, to solve the auxiliary system (2) of the PCG method, we use m steps of the iteration

$$s^{(l)} = Ts^{(l-1)} + W^{-1}r, \quad l = 1, 2, \ldots, m,$$

choosing $s^{(0)} = 0$, where $T = H + (I - H)P^{-1}Q$, $W = P(I - H)^{-1}$, with P defined in (4) and $H = \text{Diag}((B_1^{-1}C_1)^{q(1)}, \ldots, (B_p^{-1}C_p)^{q(p)})$; see e.g., [7]. Then, the updated vector from m steps is given by $s^{(m)} = (I + T + T^2 + \cdots + T^{m-1})W^{-1}r$. Therefore, the preconditioner related to the block two-stage methods is given by

$$\mathcal{M}_m = W(I + T + T^2 + \cdots + T^{m-1})^{-1}. \tag{7}$$

In the rest of this section we check the validity of this preconditioner. We give sufficient conditions on the splittings to assure that $\mathcal{M}_m$ is symmetric and positive definite. Given a square real matrix A, the splitting $A = P - Q$ is P-regular if and only if $P^T + Q$ is positive definite.

Theorem 1. *Let A be a symmetric positive definite matrix. Let $A = P - Q$ be a splitting of A, where $P = \mathrm{Diag}(P_1, \ldots, P_p)$ is the block diagonal matrix defined in (4). Suppose that P is symmetric and Q is positive semidefinite. Let $P_j = B_j - C_j$, $1 \leq j \leq p$, be P-regular splittings such that B_j is symmetric. Then the preconditioning matrix $\mathcal{M}_m$ defined by (7) is symmetric.*

Proof. The matrix $W^{-1} = (I - H)P^{-1}$ can be written as

$$W^{-1} = \mathrm{Diag}((I - (B_1^{-1}C_1)^{q(1)})P_1^{-1}, \ldots, (I - (B_p^{-1}C_p)^{q(p)})P_p^{-1})$$
$$= \mathrm{Diag}\left(\sum_{i=0}^{q(1)-1} (B_1^{-1}C_1)^i B_1^{-1}, \ldots, \sum_{i=0}^{q(p)-1} (B_p^{-1}C_p)^i B_p^{-1}\right). \qquad (8)$$

Since P_j and B_j, $1 \leq j \leq p$, are symmetric, C_j is also symmetric. Then, it is easy to see that W^{-1} is symmetric. On the other hand, the matrix T can be written as $T = I - W^{-1}A$. Then, from (7) it obtains $\mathcal{M}_m^{-1} = (I + T + T^2 + \cdots + T^{m-1})W^{-1} = \sum_{i=0}^{m-1} (I - W^{-1}A)^i W^{-1}$. Thus, the matrix $\mathcal{M}_m^{-1}$ is a linear combination of terms of the form $(W^{-1}A)^i W^{-1}$, $i = 0, 1, \ldots, m-1$, which are symmetric. Then, the proof is completed.

Theorem 2. *Let A be a symmetric positive definite matrix. Let $A = P - Q$ be a splitting of A, where $P = \mathrm{Diag}(P_1, \ldots, P_p)$ is the block diagonal matrix defined in (4). Suppose that P is symmetric and Q is positive semidefinite. Let $P_j = B_j - C_j$, $1 \leq j \leq p$, be P-regular splittings such that B_j is symmetric. Then the preconditioning matrix $\mathcal{M}_m$ defined by (7) is positive definite.*

Proof. Since $P_j = B_j - C_j$, $1 \leq j \leq p$, are P-regular splittings, from Corollary 3.6 of [2] it follows that the block diagonal matrix $W = P(I - H)^{-1}$ is positive definite. On the other hand, from (7) we can write

$$\mathcal{M}_m^{-1}W = (I + T + T^2 + \cdots + T^{m-1}), \qquad (9)$$

with $T = I - W^{-1}A$. From Theorem 3.5 of [2] it follows that $\rho(T) < 1$, and reasoning in a similar way as in the proof of Theorem 3.4.2 of [9] it is obtained that the eigenvalues of $\mathcal{M}_m^{-1}W$ are positive. Then from Theorem A.2.7 of [9] the proof is completed.

3 Numerical experiments

In the experiments the problem to be solved comes from the discretization of the Laplace's equation, $\nabla^2 u = u_{ss} + u_{tt} = 0$, satisfying Dirichlet boundary conditions

on the unit square $\Omega = [0,1] \times [0,1]$. The discretization of the domain Ω, using five point finite differences, with $J \times J$ points equally spaced by h, yields a linear system $Ax = b$, where A is block tridiagonal, $A = \text{tridiag}[-I, C, -I]$, where I and C are $J \times J$ matrices, I is the identity, and $C = \text{tridiag}[-1, 4, -1]$. Note that A has $J \times J$ blocks of size $J \times J$. Clearly, A is a symmetric positive definite matrix.

Let $A = \bar{P} - \bar{Q}$ be the Block-Jacobi splitting of A, i.e., $\bar{P} = \text{Diag}(A_{11}, \ldots, A_{pp})$. Let us consider square diagonal nonnegative matrices D_j, of size n_j, $1 \leq j \leq p$, such that $\bar{Q} + \text{Diag}(D_1, \ldots, D_p)$ is positive semidefinite. Then, it is easy to see that the splitting $A = P - Q$, where

$$P = \text{Diag}(P_1, \ldots, P_p), \quad P_j = A_{jj} + D_j, \quad Q = \bar{Q} + \text{Diag}(D_1, \ldots, D_p), \quad (10)$$

satisfies the assumptions of Theorems 1 and 2.

Therefore, in order to ensure the hypotheses of the above theorems we considered in our examples a block splitting as in (10), where $A_{jj} = \text{tridiag}[-I, C, -I]$, $1 \leq j \leq p$, and $D = \text{Diag}(\sum_{j=1, j\neq 1}^{n} |\bar{q}_{1j}|, \ldots, \sum_{j=1, j\neq n}^{n} |\bar{q}_{nj}|)$, with $\bar{Q} = [\bar{q}_{ij}]_{1 \leq i,j \leq n}$.

In these experiments reported here, we use as inner iterative procedure the Jacobi method.

The parallel experiments have been run on two different parallel computer systems. The first platform is an IBM RS/6000 SP with 8 nodes. The second platform is an ethernet network of five 120 MHz Pentiums. The peak performance of this network is 100 Mbytes per second.

We experimented with different matrix sizes. The matrices were partitioned according to the number of available processors . The conclusions were similar for all tested matrices. Here we discuss the results for two matrices of size 1024 and 4096 which correspond to grid sizes of 32 and 64, respectively.

The initial vector used was $x^{(0)} = (0, 0, \ldots, 0)^T$ and the right hand side was $b = (1, 1, \ldots, 1)^T$. The stopping criterion used was $r^T \cdot r < 10^{-5}$, where r is the residual at the corresponding iteration. All times are reported in seconds. In the results we use the notation $2^1 6^1$ to represent that $q(j) = 2$, $j = 1$, and $q(j) = 6$, $j = 2$. Similar notation is used for other block two-stage PCG methods.

Tables 1 and 2 show the behavior of some PCG methods for the above Laplace matrices. We compare these methods with the well-known m-step Block-Jacobi PCG method that has potentially excellent parallel properties. In this case, the subdomain problems are solved by using the Choleski complete factorization (see e.g., [9]). One can observe that the use of two-stage preconditioners gives better results than the use of the Block-Jacobi preconditioner. The conclusions are similar on both multiprocessors. However, the computing platform has obviously an influence in the performance of a parallel implementation. So, the efficiency decreases notoriously when the number of processors increases. This fact is due to the inadequate use of the processors when the number of processors increases for a fixed matrix, because the cost of the operations performed in parallel can be smaller than the cost of communications. For example, in the last block partitioning of Table 2 using four processors for the cluster of Pentiums it obtains

REAL times between 3.04 and 7.21 seconds, however the CPU times are between 0.68 and 1.54 seconds. Here the network is very slow compared to the network in the other computing platform.

On the other hand we observed that generally the optimal number of steps m is two for any size of the diagonal blocks. However, it seems that the choice of the number of inner iterations ($q(j)$) is dependent of the size of the diagonal blocks. So, an optimal sequence of inner iterations is that a little greater than one producing a priori a load balance based on the block size assigned to each processor.

We have observed, in some cases, that when the number of steps is odd, then the number of iterations increases with respect to the previous even number of steps. This fact is due to the condition number of the matrix $\hat{A} = SAS^T$ that is similar to the matrix $\mathcal{M}_m^{-1}A$. Then, $\text{cond}(\hat{A}) = \frac{1-\lambda_{min}(T^m)}{1-\lambda_{max}(T^m)}$, where $\lambda_{min}(T^m)$ and $\lambda_{max}(T^m)$ are respectively the minimum and maximum eigenvalues of T^m. Therefore, if T has negative eigenvalues and m is odd, the numerator of $\text{cond}(\hat{A})$ is greater than one. However, if m is even, the numerator is always less than one. Thus, we must expect a better decreasing of $\text{cond}(\hat{A})$ for even values of m.

Table 1. Parallel implementation of the PCG method on the solution of Laplace problems. Size of matrix A: 1024.

# Proc.		Block two-stage PCG				Block-Jacobi PCG		
n_j	m	$q(j)$	It.	Time cluster	Time sp2	It.	Time cluster	Time sp2
2	1	1^2	49	0.95	0.090			
512	1	2^2	27	0.51	0.050			
512	1	3^2	31	0.61	0.062			
	1	4^2	21	0.44	0.047	8	1.28	0.71
	2	1^2	25	0.57	0.056			
	2	5^2	14	0.43	0.051			
	2	6^2	12	0.41	0.050	7	1.13	0.65
2	1	1^2	59	1.13	0.125			
768	1	$2^1 6^1$	30	0.58	0.062	11	2.64	1.73
256	2	2^2	23	0.58	0.066			
	2	$3^1 6^1$	19	0.54	0.066	6	2.42	1.71
3	1	1^3	50	1.08	0.128			
352	1	4^3	20	0.49	0.051	11	1.17	0.24
352	2	1^3	26	0.66	0.084			
320	2	5^3	13	0.39	0.053	8	0.93	0.17
	3	1^3	28	0.80	0.135			
	3	2^3	16	0.50	0.069			
	3	3^3	17	0.56	0.075			
	3	4^3	12	0.44	0.057	7	0.97	0.18

Table 2. Parallel implementation of the PCG method on the solution of Laplace problems. Size of matrix A: 4096.

# Proc.		Block two-stage PCG				Block-Jacobi PCG		
n_j	m	$q(j)$	It.	Time cluster	Time sp2	It.	Time cluster	Time sp2
3	1	1^3	102	5.93	0.56			
1344	1	4^3	47	3.18	0.31	19	16.47	9.01
1344	2	1^3	52	4.12	0.36			
1408	2	4^3	32	3.02	0.32	11	16.40	8.06
4	1	1^4	101	7.21	0.63			
1024	1	4^4	38	3.04	0.27	21	9.58	4.01
1024	2	1^4	51	4.95	0.41			
1024	2	4^4	27	3.12	0.27	14	9.72	4.02
1024	3	4^4	22	3.36	0.29	13	10.28	4.08

References

1. Adams, L.: M-step preconditioned conjugate gradient methods. SIAM Journal on Scientific and Statistical Computing, Vol. 6 (1985) 452–462
2. Castel, M. J., Migallón, V., Penadés, J.: Parallel two-stage iterative methods for hermitian positive definite matrices. Technical Report 96–03, Departamento de Tecnología Informática y Computación, Universidad de Alicante, Spain, (1996)
3. Concus, P., Golub, G.H., O'Leary, D.P.: A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations. In: Buch, J. R., Rose, D. J. (eds.): Sparse Matrix Computations. Academic Press, (1976) 309–332
4. Frommer, A., Szyld, D. B.: H-splittings and two-stage iterative methods. Numerische Mathematik, Vol. 63 (1992) 345–356
5. Hestenes, M. R., Steifel, S. R.: Methods of conjugate gradient for solving linear systems. J. of Res. Nat. Bureau Standards, Vol. 49 (1952) 409–436
6. Lanzkron, P. J., Rose, D. J., Szyld, D. B.: Convergence of nested iterative methods for linear systems. Numerische Mathematik, Vol. 58 (1991) 685–702
7. Migallón, V., Penadés, J.: Convergence of two-stage iterative methods for hermitian positive definite matrices. Applied Mathematics Letters, Vol. 10(3) (1997) 79–83
8. Nichols, N. K.: On the convergence of two-stage iterative processes for solving linear equations. SIAM Journal on Numerical Analysis, Vol. 10 (1973) 460–469
9. Ortega, J. M.: Introduction to Parallel and Vector Solution of Linear Systems. Plenum Press, New York (1988)

Parallelization of a Direct Method for Systems of Linear Equations

M. F. Costa[1] and R. M. Ralha[2]

[1] Departamento de Matemática, Universidade do Minho,
Campus de Azurém, 4800 Guimarães, Portugal
mfc@math.uminho.pt
[2] Departamento de Matemática, Universidade do Minho,
Campus de Gualtar, 4710 Braga, Portugal

Abstract. In this paper we study a sequential version of the Gaussian elimination method in which several pivots are used in each reduction step. We carry out an error analysis and establish an upper bound for the error in the solution. In all our tests (in which we have used random matrices as well as matrices of special types) the numerical results produced by an implementation of the algorithm are as good as those produced by the classical method. From the point of view of sequential processing, the new method is as efficient as the classical method and we believe that it has advantages for parallel processing since it allows better load balancing and computation/communication overlap. We develop a parallel implementation of the new method in a distributed memory system with a ring topology and give a performance analysis of the parallel algorithm based on the study of the load balancing and the cost of communication between processors. We present preliminary results of some computational experiences with the parallel algorithm.

1 Introduction

Much work has been published in the last years on the parallel solution of large systems of linear equations. A considerable number of publications treat the parallelization of the old method of Gauss with partial pivoting [3][5][6][7][9][11][12] [14][15]. The main problem of any implementation of this method in a multiprocessor machine resides in the need to incorporate partial pivoting to guarantee the numerical stability of the method. This happens because, at each step, the search for the pivotal row forces the synchronization of the activity of several processors and part of the time is spent on communication and waiting. To minimize these problems, we propose a modification of the method of Gaussian elimination which consists in the use of several pivots in each reduction step; we first study a sequential version of the modified method and then proceed with its parallelization. Our proposal is significantly different from another variant of the method know as "pairwise pivoting" which has been introduced by Wilkinson [1] and more recently used by others in the context of parallel processing [3][5]. As it is also the case with pairwise pivoting [2][4], one possible drawback of our

pivoting strategy is that the theoretical upper bound for the error in the solution is larger than in the classical method; nevertheless, in our numerical experiments the errors produced by both methods were found to be comparable.

2 Gaussian elimination with several pivots in each step

Given a system $Ax = b$ with $A \in R^{n\times n}$ non singular, consider the matrix $(A|b)$ divided into nB blocks of R contiguous rows. In the process of reducing A to triangular form, we consider the kth reduction step ($k = 1, 2, ..., n-1$) as a sequence of two phases. The first phase occurs at an internal level within each block and the second phase involves the various blocks.

$$\left(\begin{array}{ccccc}
a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\
\vdots & \vdots & \cdots & \vdots & \vdots \\
a_{R,1} & a_{R,2} & \cdots & a_{R,n} & b_R \\
\hline
a_{R+1,1} & a_{R+1,2} & \cdots & a_{R+1,n} & b_{R+1} \\
\vdots & \vdots & \cdots & \vdots & \vdots \\
a_{2R,1} & a_{2R,2} & \cdots & a_{2R,n} & b_{2R} \\
\hline
\vdots & \vdots & \vdots & \vdots & \vdots \\
\hline
a_{(n-R)+1,1} & a_{(n-R)+1,2} & \cdots & a_{(n-R)+1,n} & b_{(n-R)+1} \\
\vdots & \vdots & \cdots & \vdots & \vdots \\
a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n
\end{array}\right)$$

Description of the first step of reduction: for $L = 1, 2, ..., nB$ select a pivotal row, called local pivotal row, let us say row p_L where:

$$|a_{p_L,1}| = \max_{(L-1)R+1 \le i \le LR} |a_{i,1}|$$

Next, if $a_{p_L,1} \neq 0$ each row i ($i = (L-1)R+1, ..., LR$, $i \neq p_L$) is replaced by its sum with row p_L multiplied by $m_{i,1} = -a_{i,1}/a_{p_L,1}$.

Once these elementary operations are concluded in each block, one still needs to annihilate nB-1 elements in the first column. To do this, a global pivotal row is selected among the nB local pivotal rows, which is row p, where:

$$|a_{p,1}| = \max_{1 \le L \le nB} |a_{p_L,1}|$$

Assuming that $a_{p,1} \neq 0$, we finalize the first step of reduction by replacing the remaining local pivotal rows with its sum with the global pivotal row multiplied by $m_{p_L,1} = -a_{p_L,1}/a_{p,1}$ ($L = 1, ..., nB$ and $p_L \neq p$), where one interchanges rows p and 1 if $p \neq 1$, so that in the end the matrix of the system is in triangular

form. In the remaining $n-2$ reduction steps one proceeds in an analogous way. Note that initially the number of local pivotal rows equals the number nB of blocks but such number will decrease along the process of elimination, as the number of blocks involved in the reduction to triangular form decreases.

3 Matrix formulation of the method

A matrix formulation of the method with several pivots in each reduction step can be described in terms of products with non unitary elementary matrices (Gauss transformations)[18]. Denoting the matrices involved in the local and global stages of the kth step respectively by $M_{k,L}$ and M_k, we have:

$$M_{k,L} = I - \tau^{(k_L)} e_{k_L}^T$$

$$M_k = I - \tau^{(k)} e_k^T$$

where:

- k_L is the index of the local pivotal row in block L
- $\tau^{(k_L)}$ represents the vector of multipliers of used in the local stage, in block L ($m_{i,k} = {}^{(k-1)}a_{i,k} / {}^{(k-1)}a_{k_L,k}, \quad i = k_L+1, ..., LR$)
- $e_{k_L}^T$ is the k_Lth column of the identity matrix
- k is the index of the global pivotal row
- $\tau^{(k)}$ represents the vector of multipliers used in the global phase

We will also denote the elementary permutation matrices by $P_{k,L}$ and P_k when referring to permutation of rows in the local phase (i.e., interchange of two local rows in block L) and in the global phase (i.e., permutation of rows from two distinct blocks), respectively. Therefore, at the end of step $n-1$ we have a triangular matrix U given by

$$\overbrace{M_{n-1}P_{n-1}}^{\text{step } (n\text{-}1)} \cdots \overbrace{M_{(n-R)+1}P_{(n-R)+1}}^{\text{step } (n\text{-}R)+1} \overbrace{M_{(n-R)}P_{(n-R)}M_{n-R,nB}P_{n-R,nB}}^{\text{step } (n\text{-}R)} \cdots$$
$$\cdots \underbrace{M_R P_R M_{R,nB} P_{R,nB} \cdots M_{R,2} P_{R,2}}_{\text{step } R} \cdots \underbrace{M_1 P_1 M_{1,nB} P_{1,nB} \cdots M_{1,1} P_{1,1}}_{\text{step } 1} A = U.$$

In terms of factorization, we have $\mathbf{A = LU}$ where

$$\mathbf{L} = P_{1,1} M_{1,1}^{-1} \cdots P_{1,nB} M_{1,nB}^{-1} P_1 M_1^{-1} \cdots\cdots P_{n-1} M_{n-1}^{-1}$$

is not necessarily a lower triangular matrix. However, if L is required for practical purposes, it can be readily obtained as a product of simple matrices, according to the previous expression.

Example (n=6, nB=2):

$$\underbrace{\left(\begin{array}{cccccc} 1 & -1 & -1 & -1 & -1 & -1 \\ -1 & 2 & 0 & 0 & 0 & 0 \\ -1 & 0 & 3 & 1 & 1 & 1 \\ \hline -1 & 0 & 1 & 4 & 2 & 2 \\ -1 & 0 & 1 & 2 & 5 & 3 \\ -1 & 0 & 1 & 2 & 3 & 6 \end{array}\right)}_{A} = \underbrace{\left(\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & -1 & -\frac{1}{2} & -\frac{1}{4} & 1 \\ -1 & -1 & -1 & \frac{1}{2} & -\frac{1}{4} & 1 \\ -1 & -1 & -1 & \frac{1}{2} & \frac{3}{4} & 1 \end{array}\right)}_{L} \underbrace{\left(\begin{array}{cccccc} 1 & -1 & -1 & -1 & -1 & -1 \\ 0 & 1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 1 & -1 & -1 & -1 \\ 0 & 0 & 0 & -2 & 3 & 1 \\ 0 & 0 & 0 & 0 & -2 & 3 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4} \end{array}\right)}_{U}$$

4 Error analysis

A detailed error analysis for the new method is given in [18] where it is shown that the calculated solution $\hat{x}$ satisfies the system:

$$(A + E)\hat{x} = b$$

with

$$||E||_\infty \leq 15n^4\rho||A||_\infty + O(u^2)$$

In the Gauss elimination method with partial pivoting one has [10]:

$$||E||_\infty \leq 8n^3\rho||A||_\infty + O(u^2)$$

Therefore, the limit for the rounding errors in the new method is more pessimist because of the factor $15n^4$. At this point, one should bear in mind that the factor n^3 is usually ignored in the discussion of the stability of Gaussian elimination. As stated in [10], p.65: "...usually, the bound itself is weaker than it might have been because of the necessity of restringing the mass of detail to a reasonable level and because of limitations imposed by expressing the errors in terms of matrix norms". It is usually considered that the numerical stability of the method depends on the size of a growth factor ρ. We adopted the definition

$$\rho = \frac{\max\limits_{i,j,k}|a_{i,j}^{(k)}|}{\max\limits_{i,j,k}|a_{i,j}|}$$

given in [16]. Although, in theory, ρ can be as large as 2^{n-1}, in practice such growth is extremely improbable and ρ is generally of the order 10. Indeed, in the computational experiences carried out with both methods, we found ρ to be always of such order of magnitude (see table 1). Based on this, we claim that the numerical properties of the new method are comparable to those of the classical algorithm.

5 Computational experiences

We implemented our algorithms (both sequential and parallel) on a transputer based machine. In the computational tests we found out that the new method and the classical method with partial pivoting produce solutions with the same precision, independently of the type of matrix used. This can be appreciated in table 1 for random matrices of different sizes. In all cases we have used a vector b corresponding to the exact solution $x_i = 1$ $(i = 1, ..., n)$, so that we can indicate the absolute error $||\hat{x}\text{-}x||_\infty$. Also, the execution times are essentially the same for both methods, although in the case of our method we are using several concurrent processes (in this set of experiments we set $R = 10$, i.e., we decomposed each matrix in $n/10$ blocks of 10 rows each); the transputer hardware handles efficiently the execution of concurrent processes and the overhead due to this is very small, as it can be better understood for the matrix of size $n = 100$, since in this case a single processor is running 10 concurrent processes.

n	method	$\|\|A\hat{x}$ -b$\|\|_\infty$	$\|\|\hat{x}$-x$\|\|_\infty$	ρ	run time (sec.)
10	Ours	1.78E-15	1.64E-14	1.49	0.00614
10	Classic	1.78E-15	1.64E-14	1.49	0.00589
20	Ours	5.33E-15	6.88E-15	1.77	0.0346
20	Classic	3.55E-15	1.24E-14	2.68	0.0331
50	Ours	2.13E-14	2.13E-13	6.84	0.398
50	Classic	1.78E-14	7.37E-14	3.86	0.384
100	Ours	8.53E-14	2.17E-13	9.81	2.84
100	Classic	3.55E-14	2.99E-13	7.18	2.76

Table 1: results obtained with the two methods on a single processor.

To make more clear that the numerical precision of the solution does not vary significantly with the number nB of blocks used, and that the execution time increases only slightly we tested our method with a certain matrix of size $n = 100$, using successively 1, 4, 5, 10 and 20 blocks. The results are listed in table 2.

nB	$\|\|A\hat{x} - b\|\|_\infty$	$\|\|\hat{x} - x\|\|_\infty$	run time (sec.)
1	$6.39E-14$	$6.57E-13$	2.77
2	$9.95E-14$	$1.40E-13$	2.77
4	$6.39E-14$	$1.16E-13$	2.78
5	$5.68E-14$	$1.26E-13$	2.78
10	$8.53E-14$	$2.17E-13$	2.84
20	$7.82E-14$	$1.66E-13$	2.89

Table 2: varying the number nB of blocks for a matrix of size $n = 100$.

6 The parallel algorithm

In the development of the parallel application we used a ring topology. Parallelizing the algorithm consists in assigning a block of R contiguous rows of the

matrix $(A|b)$ to each process of the ring. In this way, a local reduction is carried out concurrently in each process of the ring. Furthermore, the task of finding (and broadcasting to the still active processes) the global pivotal row can proceed concurrently with the local computation. After this, the processes finish "simultaneously" the reduction step.

6.1 Load balancing

The load balancing of the parallel algorithm is not predictable since it is not possible, in general, to know in advance which process is the owner of the global pivotal row in each one of the $n-1$ steps. Because of this, we studied two extreme cases: the best case occurs when the pivotal row belongs cyclically to each process (and all processes will be active almost till the end of the reduction to triangular form), the worst case occurs when the first R global pivotal rows belong to a particular process (this process will be idle in the remaining $n-R$ steps), the next R belong to another process, and so on. In this respect it is interesting to note that for matrices generated randomly the load balancing is always near to the ideal situation (see [18] p.69-72).

6.2 Efficiency and speedup

A theoretical study of the *speedup* $S{:=}T(1)/T(P)$ and *efficiency* $E := S/P$ of the parallel algorithm, was carried out for the extreme cases described before; we obtained the following expressions:

best case:

$$S \simeq 1 \Big/ \left[\frac{4n^3+3n^2(P+2)-n\left(P^2-6P+12\right)}{(4n^3+9n^2-7n)P} + 12\theta\,(P-1)\left(\frac{P(n-1)+2n}{(4n^3+9n^2-7n)P}\right)\alpha_d + \right.$$

$$\left. +6\theta\,(P-1)\left(\frac{n^2+4n(P+1)-4P}{(4n^3+9n^2-7n)P}\right)\beta_d \right]$$

worst case:

$$S \simeq 1 \Big/ \left[\frac{2n^3(3P^2-1)+3n^2(4P^2-P)-7nP^2}{(4n^3+9n^2-7n)P^3} + 12\theta\,(P-1)\left(\frac{P(n-1)+2n}{(4n^3+9n^2-7n)P}\right)\alpha_d + \right.$$

$$\left. +6\theta\,(P-1)\left(\frac{n^2+4n\,(P+1)-4P}{(4n^3+9n^2-7n)P}\right)\beta_d \right]$$

where:

- θ represents the number of flops per second
- α_d is the start-up time
- β_d is the time required to send a floating-point number through a physical link.

In any case, the efficiency and speedup increases when n grows and P remains fixed and decreases when P is grows and n is kept constant. Using the values $\theta = 10^6$, $\alpha_d = 2.6\mu s$, $\beta_d = 4.5\mu s$ given in [13] for the T800 transputer and considering $P = 4$ and $n = 100, 200, 300, 400, 600$, one obtains from the previous expressions the estimated values given in table 3.

	$S = \frac{T(1)}{T(4)}$	$E = \frac{S}{4}(\%)$
n	best case	worst case
100	3.14(78,5%)	2.33(58,3%)
200	3.56(88,9%)	2.53(63,3%)
300	3.70(92,6%)	2.60(64,9%)
400	3.78(94,4%)	2.63(65,7%)
600	3.85(96,3%)	2.66(66,5%)

Table 3: estimated values for the speedup and efficiency with 4 processors.

In computational experiences applied to problems of dimension $n = 100$ and using 4 processors we obtained the values for the speedup and efficiency given in table 4.

Matrix	T_{seq}(sec.)	T_{par}(sec.)	$S{=}T_{seq}/T_{par}$	$E{=}S/4$
Moler	2.39	1.39	1.72	43,0%
Frank	2.44	1.40	1.74	43,6%
Border	3.48	1.72	2.02	50,6%
Dingdong	2.61	1.46	1.79	44,7%
Random	2.78	1.24	2.24	56,0%

Table 4: execution times, speedup and efficiency of the parallel algorithm (with 4 processors).

The best results were obtained with random matrices, as expected, since the load balancing of the parallel algorithm was found to be good for such matrices, as mentioned before.

References

1. L. Fox, E. Goodwin, J. Wilkinson: Modern Computing Methods. (First edition, Philosophical Library, New-York) (1961).
2. J. Stern: A fast Gaussian elimination scheme and automated roundoff error analysis for SIME machines. (Dept. of Computer Science, University of Illinois) (1979).
3. W. Gentleman, H. T. Kung: Matrix triangularization by systolic arrays, in Proc. SPIE 298, Real Time Signal Processing. (San Diego, CA) (1981).
4. D. Sorensen: Analysis of Pairwise Pivoting in Gaussian Elimination. (IEE Trans. Comput. C-34) (1985) pp. 274-278.
5. A. Sameh: On some parallel algorithms on a ring of processors. (Comp. Phys. Comm.) (1985) pp. 159-166.
6. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker: Solving Problems on Concurrent Processors. (Prentice-Hall) (1988).
7. B. Tourancheau, M. Cosnard and G. Villard: Gaussian elimination on message passing architecture. (Supercomputing Lecture Notes in Computer Science) (1988) pp. 611-628.
8. Jagdish J. Modi: Parallel Algorithms and Matrix Computation. (Oxford Universty Press) (1988).

9. D. P. Bertsekas and John N. Tsitsiklis: Parallel and Distributed Computation. (Prentice-Hall) (1989).
10. G. H. Golub and Charles F. Van Loan: Matrix Computations. (The Johns Hopkins University Press) (1989).
11. A. Benaini, Y. Robert: Spacetime-minimal systolic arrays for gaussian elimination and algebraic path problem. (Parallel Computing) (1990) 15:211-225.
12. K. A. Gallivan, Michael T. Heath, and James M. Ortega: Parallel Algorithms for Matrix Computations. (Society for Industrial and Applied Mathematics) (1990).
13. R. Ralha: Parallel Computation of Eigenvalues and Eignvectors using Occam and transputers. (PhD thesis, University of Southampton) (1990).
14. F. F. Rivera, R. Doallo, J. D. Bruguera and E. L. Zapata: Gaussian elimination with pivoting on hypercubes. (Parallel Computing) (1990) 14:51-60.
15. T. L. Freeman and C. Phillips: Parallel Numerical Algorithms. (Prentice Hall International) (1992).
16. N. J. Higham: Accuracy and Stability of Numerical Algorithms. (Society for Industrial and Applied Mathematics) (1996).
17. L. V. Foster: The growth factor and efficiency of Gaussian Elimination with rook pivoting. (Journal of Computational and Applied Mathematics) (1997) pp.177-194.
18. M. F. Costa: Paralelização de um método directo para sistemas de equações lineares. (MSc. thesis, Universidade do Minho) (1997).