Automatic Memory Management for Data Parallel Programs

Wellington Santos Martins

Instituto de Informática
Universidade Federal de Goiás, Goiânia, Brazil
E-mail: wsm@inf.ufg.br

Abstract: The exploitation of data locality has been crucial for achieving high performance in parallel computers. The techniques used for that range from those requiring programmer intervention, to those relying on a run-time system that automatically moves data. This paper presents a memory management system that, while moving data automatically, still requires the programmer to provide the size of a basic unit of access. Data locality is exploited by automatically moving blocks of global data to local memory as necessary. The system is developed for data parallel programs where the same computation is performed on disjoint groups of data. It is shown that the system can produce comparable results to those obtained by hand-coded versions. The effects of imposing a limit on the local memory size of the processors are also analyzed.

Keywords: Distributed systems and paralellism. Operating systems. Computer architecture.

1. Introduction

Efficient data management has been crucial for achieving high performance almost since the first computers were built. The differences in cost and access time of memory devices resulted in hierarchical systems where fast (more expensive) devices are at the top of the hierarchy, i.e. closer to the processor, while slow (cheaper) devices are located further away. High performance is achieved by carefully managing data so that currently used data is at the highest possible level. The effectiveness of this approach is due to a natural characteristic of programs known as locality of reference. Two types of locality are exploited: temporal locality which states that recently referenced items are likely to be referenced again in the near future, and spatial locality which refers to the tendency for items to be referenced in clusters. Given the effectiveness of exploiting data locality in sequential machines, it is not surprising that this technique has also been employed in parallel machines.

Besides temporal and spatial locality, two new types of locality become important in parallel computing. Parallel machines with no globally accessible data usually use direct interconnection networks forming a distributed memory hierarchy, i.e. the processors are at different distances from each other. To improve locality of reference in these machines, *network locality*, also known as communication locality, can be exploited. This means that remote accesses should be restricted to the neighbourhood of the processor so that long delays can be avoided. This approach can be quite effective in producing high performance programs, but it has the serious drawback of producing non-portable software.

On the other hand, parallel machines providing globally accessible data can improve locality of reference by moving global data close (cache or local memory) to the processor. In order for this two-level memory hierarchy to work efficiently, *processor locality* [AgGu88] should be exploited. Processor locality requires that references to a particular region of data come from the same processor. To keep high processor locality, unnecessary interference from other processors should be avoided. The reason why processor locality is desirable has to do with the memory coherency problem. With all processors being able to access global data, if replication of the data is allowed, the system must make sure that once one of these copies is modified the others are updated or invalidated. Unless there is high processor locality, the traffic generated by these operations can degrade the system's performance.

The techniques developed for exploiting data locality in parallel machines resemble those developed for sequential machines and in some cases represent an extension of them. These techniques either determine data mapping and data movement statically, i.e. before execution, or move data on demand during run time. In the first case the programmer is required to do the job himself or to provide information to guide the compiler in generating code to automate this process. In the second case, a run time system is used to move data to where it is needed.

This paper presents a memory management system that, while moving data automatically, still requires the programmer to provide the size of a basic unit of access. The next two sections give a brief overview of the techniques proposed so far to exploit data locality in parallel machines. The following section describes the target architecture used in the experiments conducted. The automatic memory management system proposed is then described in section 5. The final section then summarizes and discusses the results.

2. Data Locality Exploitation under Programmer Intervention

In the early days of computing the programmer was responsible for memory management. The programmer would determine, at each moment of time, how information should be distributed between main memory and backing storage. Whenever the program was expected to exceed the size of main memory, it had to be split into segments which could be overlaid and commands had to be inserted to make sure the correct overlay was in memory at any time. This approach was based on the assumptions that it was possible to predict both memory availability of the program and its behaviour (reference string) so that locality of reference could be exploited. These rather restrictive assumptions made the applicability of this approach limited to a small number of applications where static (preplanned) allocation was possible. Automatic folding systems were eventually proposed and showed to be of comparable performance and of wider applicability.

The scenario described above resembles that of programming DADM (Disjoint Address-space, Distributed Memory) machines using the message passing paradigm. In these systems (e.g. TCGMSG [Har90], PVM [GeSu92], MPI [MPI93]) the programmer must distribute data across the processors and use explicit message passing to control data movement. Data movement is implemented by inserting *send* and *receive* commands into the program to make sure the required data is moved from one node to another. The need for these commands, though, is not due to limited size of local memory but because the processors (usually) need to share data to solve a problem in parallel. This approach requires the programmer to deal with a number of details: buffers must be set up, care must be taken to send data as early as possible and economically, the interleaving of messages must be deadlock-free etc. The performance that can be obtained is largely dependent on the data distribution selected because this determines where the computation takes place and what communication is necessary. The programmer has total control over data movement and is encouraged to exploit network locality to improve performance. Since the coding of data exchange (send/receive commands) is based on the data distribution chosen, experimentation with different data distributions usually requires a great deal of reprogramming. Given a good data distribution this approach can be quite effective in producing high performance programs but it imposes too much work on the programmer who is already faced with the challenge of devising a parallel algorithm for the problem in question.

Programming DADM machines using explicit message passing is tedious, time consuming and error prone. To free the programmer from this burden some researchers have proposed using languages based on a global name space, for example C* [RoSt87], Split-C [CDGK93], Fortran D [HKT92], and more recently High Performance Fortran [HPC93], an extension of Fortran 90 which has being proposed as a standard data parallel Fortran programming language. The idea is to facilitate the programmer's job by allowing him to write code using global data references, as on a shared memory machine. A sophisticated compiler is then used to automatically translate this code into a message passing program. However, since data distribution is crucial to performance, the programmer is required to annotate the program with directives specifying how the data should be mapped onto the distributed memory machine.

Compiler-managed memory, although much more convenient from the programmer's point of view than the use of explicit message passing, still forces the programmer to become aware of machine idiosyncrasies. In general, the best distribution scheme depends not only on program characteristics, but also on a number of machine-specific parameters, and on the kind of optimisations performed by the compiler. In addition the choice of a single data distribution for the entire program may result in poor performance. This is because although a data distribution may be well-suited for one phase of an algorithm, it may not be good, in terms of performance, for a subsequent phase. In these situations the programmer is required to include realign and redistribution directives between phases of the program.

3. Programmer Transparent Data Locality Exploitation

In the early computers, memory management was the programmer's responsibility. However, due to the increasing complexity of programs and new requirements demanded by multiprogramming and time-sharing systems, dynamic memory management techniques started being investigated. The idea was to have an adaptive system which would operate on-the-fly as the program runs, mapping the program into whatever memory size was available. This resulted in the development of automatic mapping units which later became the virtual memory systems that we know today. Following a similar line of development slave memories, today known as cache memories, were developed to improve the performance of the memory hierarchy between the CPU and the main memory. Both techniques, virtual memory and cache, move data automatically and take advantage of the locality of reference presented by programs. These two techniques were predicted to become part of most sequential machines and this has happened.

Similar ideas have been used in parallel machines. Parallel machines with a Single Address-space and Distributed Memory (SADM) have received a lot of attention recently. Having a physically distributed memory means that these machines are easy to scale. In addition the provision of a single address space frees the programmer from the burden of message passing programming. In order to provide a single address space in these distributed memory machines, each processor has to be able to directly access every other local memory. This can be done either by having processors with sufficient address space (address bits) to directly address the entire physical memory of the machine, or by concatenating node number and address within a node to achieve the same objective. However, due to the high communication costs of these machines, frequent remote accesses would degrade performance substantially. The solution adopted in practice is to move blocks of data in an attempt to exploit data locality and thus decrease the number of remote accesses. In addition, data is usually allowed to be replicated so that multiple read accesses can take place at the same time using local accesses. Provided the application presents a high processor locality, the system can be expected to perform well. Systems implementing these ideas are commonly known as Distributed Shared Memory (DSM) systems (e.g. IVY [LiSc89], KOAN [LaPr89], DDM [WaHa88], DASH [LLGG90]). They have their roots in the virtual memory and cache systems of sequential machines. A number of DSM's have been proposed, ranging from simple run-time library extensions to sophisticated hardware designs.

4. Target Architecture

The experimental platform used in this work is a simulator [Nash93] of a distributed memory machine which supports uniform global access by the use of data randomisation. This Randomised Shared Memory (RSM) provides a single address space and automatically maps data to processors by the use of hash functions, i.e. data is spread over the processors so that the probability of overloading any local memory is diminished. The shared address space distinguishes two forms of data: global data which is randomly distributed amongst all processors and local data which is mapped to a single processor memory.

The simulator includes a detailed performance model which costs operations based on measured performance figures for the T9000 transputer processor and simulations of the C104 packet router. Local operations modelled by the simulator include arithmetic calculation, context switching, message handling and local process management. Messages entering the network are assumed to be split up to guarantee that no one message ties up a switch for long periods of time. Work on validating the simulator has been carried out at Leeds with a close match being found between theoretical predictions and experimental results [NDD95].

The simulator is written in C and provides a rich programming interface to execute algorithms written in C. The programming interface consists of a set of library procedures which support process management, shared data access and process synchronisation. Only a small subset of these library calls were used in this research, including: procedures for process management (fork, join, my_node and my_index), read and write procedures for data access, and a procedure to barrier synchronise processes. Parallel algorithms, implemented using the programming interface, are executed directly on the simulator. This way the sequence of operations generated by the program drives the simulator (execution-driven discrete-event simulation).

5. The Memory Management System Proposed

The RSM systems provide by the target architecture facilitates the design and analysis of algorithms since the programmer does not have to worry about data mapping, and the access patterns to memory generated by the processors can be assumed random. However, it destroys data locality because data is kept local to processors that do not necessarily need it. Thus most memory accesses tend to be non local. If the costs involved in non-local access are high, RSM becomes prohibitive. One way to overcome this problem is to have part of the local memory reserved for local data which is not mapped via the hashing algorithm. This way, substantial improvements in performance can be obtained by requiring the programmer to manage data carefully by using local memory where cost effective. This approach, however, forces the programmer to be concerned with memory management in addition to algorithm design and implementation. The extra work involved in static memory management can pay off for some algorithms with a predictable communication structure; knowledge of the algorithm's data reference patterns allows for good data allocation which reduces communication costs. However, in other cases, where the algorithm's communication requirements are less predictable, for example where communication patterns are input data dependent, exploitation of data locality is much harder and may require dynamic memory management.

To alleviate this problem, the following automatic memory management system is proposed. The programmer defines global data together with the size of a basic unit of access. A run-time system uses this information to copy a block of data once one of its elements is required. The blocks copied to local memory are kept locally until bulk-synchronisation (barrier) takes place when all modified blocks are written back to global memory. One problem with this approach is that all replicas of a block have to be updated (or invalidated) each time its contents change (write operation). However, this is not a problem for data parallel algorithms which operate on disjoint groups of data. The run-time system acts as a cache-like system by prefetching data which is expected to be used later on. Each time a memory access is requested, the corresponding block number is calculated and the memory access is performed either locally or remotely (globally). Data is kept coherent when the processors are synchronised.

To test the system proposed, a parallel version of the matrix multiplication algorithm, and a parallel sort using a balanced merge [FrMa88] were used. For the matrix multiplication algorithm (C=AxB), each processor calculates a ($n/p^{1/2}$ x $n/p^{1/2}$) sub matrix of C by reading $n/p^{1/2}$ rows of A and $n/p^{1/2}$ columns of B. In the parallel sort algorithm, each of the p processors starts by sorting n/p elements and then proceeds in a log(p)-phase merge. The number of elements read in each phase varies from n/p in the first phase to n elements in the final phase, doubling each time. This is because partition's boundaries are calculated using a binary search over an increasing search space.

Two versions of each algorithm were implemented: one using manual (explicit) copying of data blocks and the other relying on the system proposed. To illustrate this point, figure 1 shows a code fragment of the two versions for the parallel sort, one doing the copying manually and the other using the automatic memory system. To be able to measure the impact of the system on performance, routines implementing the automatic memory system were incorporated in the simulator so that completion times of the algorithms, taking into account the extra management overheads, could be obtained. Figure 2 shows the performance achieved for direct global memory access (RSM) and that obtained with the programmer intervention (RSM+Copying). It is clear the advantage of the latter approach. However, it is also noted that in this case little gain is obtained with the addition of processors, since the execution time function approaches its asymptotic value. The performance obtained with the automatic system proposed, shown in figure 3, is as good as that, and in some cases even outperforms it.

```
/* version doing copying manually */                  /* version using automatic memory system */

array = gmalloc ( array_size );                       array = gmalloc ( array_size, block_size );
        .                                                                 .
        .                                                                 .

local_array = lmalloc ( seg_size );
read ( array, offset, seg_size, aux_array);
switch ( );  /* guarantee aux_array is accessible */
write ( local_array, 0, seg_size, aux_array );

sort ( local_array, upper, lower );                    sort ( array, upper, lower );

read ( local_array, 0, seg_size, aux_array );
write ( array, offset, seg_size, aux_array );
free ( local_array);
sync ( );                                              sync ( );
```

Figure 1 - Code fragment of the programs

Figure 3a shows the performance dependence on block size when multiplying a 32x32 matrix as well as the performance obtained with manual copying. As expected the best performance of the implementation using the proposed memory system is achieved when the block size is equal to $n^2/p^{1/2}$, the total number of elements required from each matrix for the calculations. As the number of processors increase the performance of the cache system approaches that obtained by manual copying. This is because increasing the number of processors, and keeping the matrix size constant, reduces the number of blocks to be dealt with by the run-time system and the consequent overhead associated with their management. It can also be observed that the performance variation is small for different block sizes showing that the system responds reasonably well even when the block size chosen is not optimal.

The performance of the balanced sorting algorithm for different block sizes is showed in figure 3b. For fixed 1k data elements the performance obtained with manual copying and with automatic copying for block sizes of 8, 16 and 32 is shown. The difference between the two implementations is more accentuated when only a few processors are utilised. However, in contrast to the results for matrix multiplication, the performance obtained with automatic copying exceeds that obtained with manual copying when more than 32 processors are employed and the block size is 8 or 16. This is due to the fact that a binary search is required before each merge phase. The copy of whole blocks increases the chances of avoiding global access, especially when the search space is small compared with the block size which is the case when several processors are used.

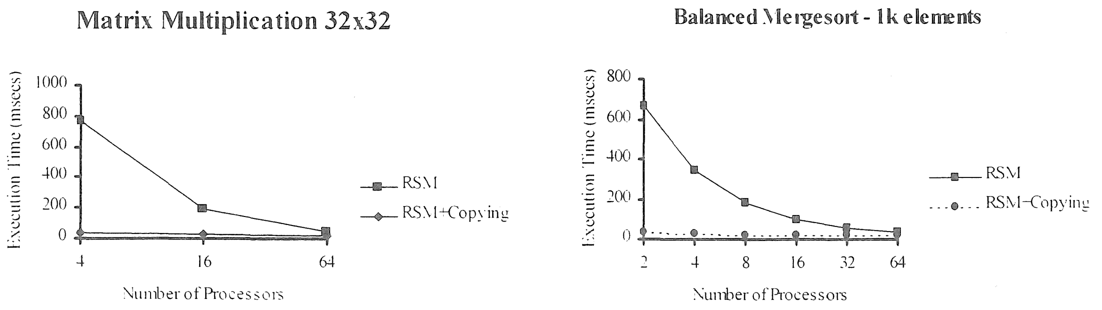

Figure 2 - Performance achieved with RSM and Copying - (a) Matrix Multiplication (b) Balanced Mergesort

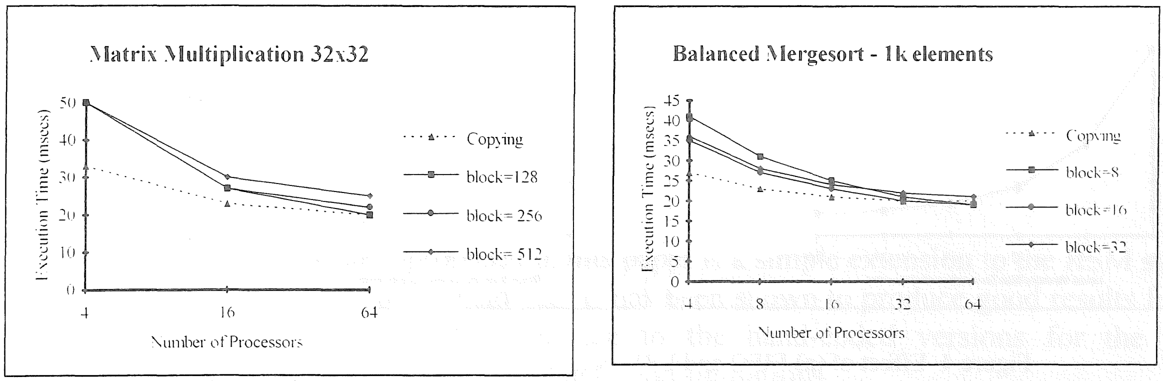

Figure 3 - Dependence of performance on block size - (a) Matrix Multiplication and (b) Balanced Mergesort

Imposing a Limit on the Local memory Capacity

The experiments described so far assumed that the processors had local memories with infinite capacity. Whenever a data block was needed it was copied to local memory and kept there until bulk synchronisation had taken place. What follows is a discussion of the implications of using a more realistic assumption, that of having local memories with a fixed size.

One immediate consequence of imposing a limit on the local memory capacity is the necessity of establishing a block replacement policy. Because the local memory has a fixed size, and consequently can store only a finite number of blocks, the block replacement policy has to decide which block should be discarded when a new block is fetched and the local memory is full. Two popular replacement policies are: FIFO (First In First Out), which replaces the block which has been in memory for the longest time, and LRU (Least Recently Used) which replaces the block which has been least referenced in the past. The experiments described next made use of these commonly used replacement policies.

Table 1 Parallel balanced mergesort results using FIFO and LRU policies

Memory Size (in number. of elements)	*Hit Ratio: FIFO (LRU) 1st phase*	*Hit Ratio: FIFO (LRU) 2nd phase*	*Hit Ratio: FIFO (LRU) 3rd phase*	*Execution Time (msecs) FIFO(LRU)*
8	0.53 (0.52)	0.33 (0.33)	0.32 (0.32)	347 (352)
16	0.81 (0.80)	0.66 (0.66)	0.65 (0.65)	167 (172)
32	0.93 (0.92)	0.83 (0.83)	0.82 (0.82)	87 (92)
64	0.97 (0.91)	0.91 (0.91)	0.90 (0.90)	58 (63)
128	0.99 (0.95)	0.95 (0.95)	0.93 (0.94)	45 (49)
256	1.00 (1.00)	0.98 (0.98)	0.96 (0.97)	41 (45)

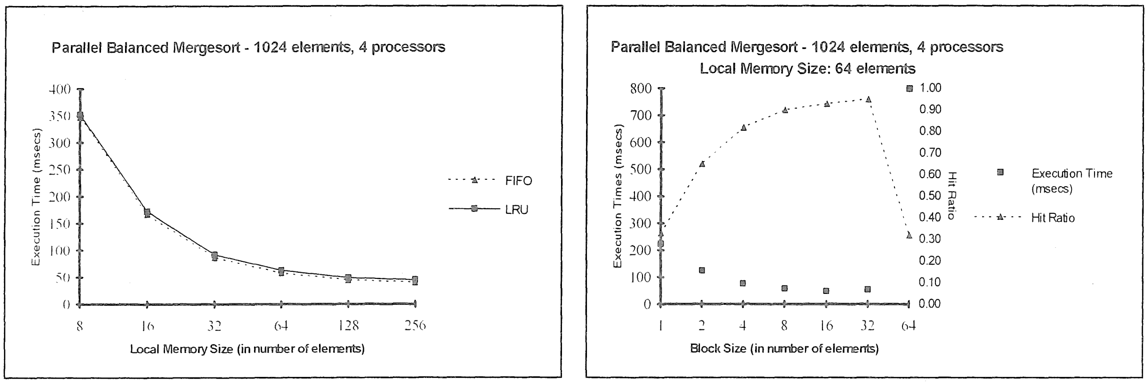

Figure 4 Effect of (a) FIFO and LRU policies and (b) block size on performance

The parallel balanced mergesort was used to collect the hit ratio (ratio of references not requiring remote accesses to the total number of references made) and execution time for different local memory sizes. The number of elements to be sorted was set to 1024 and 4 processors were utilised to do the sorting. This implied in each processor being assigned 256 (*n/p*) elements. With four processors available, three phases are required to perform the sorting. In the first phase. each processor sorts its 256 elements, then in the second and third phase they are merged to produce the final result. In each phase each processor deals with (1/*p*)th of the total number of processors, in this case, 256 elements. The memory sizes used varied from 8 to 256, doubling at each time. In order to guarantee that the processors had to manage the same number of blocks (8 in this case), independent on the memory size, the block sizes were chosen to be equal to 1, 2, 4, 8, 16, 32 and 64 respectively. The hit ratios are given for each phase of the algorithm since at the end of each phase a bulk synchronisation takes place, the modified data is copied back to global memory and the local memory is reset.

Table 1 shows the results obtained. The hit ratios obtained with FIFO and LRU policies are practically the same. This is because the memory accesses generated by the program present low temporal locality, i.e. in the program under study recently referenced items tend not be referenced again. In this situation, FIFO and LRU produce similar hit ratios since the least referenced block becomes the block which has been in memory for the longest time. As expected, the hit ratio increases with the memory size, and when this size approaches 256 elements (the slice each processor works with), the hit ratio gets close to one. The hit ratios for the second and third phase are not equal to one, when the memory size is 256, because in these phases the processors need to do a binary search and end up accessing more than their slice of 256 elements. Although producing similar hit ratios, FIFO and LRU policies result in different performance. As shown in figure 4a, FIFO performance is slightly better than that of LRU. The reason for this is the extra overhead involved in keeping a list of the least referenced blocks updated.

To analyse the effects of block size on hit ratio and performance the memory size was fixed in 64 elements while the block size was increased from 1 to 64, doubling at each time. Figure 4b shows the execution times and hit ratios (for the last merge phase) obtained when sorting 1024 elements using 4 processors. As can be noted, the hit ratio increases, reaches a maximum and starts to fall. This is due to the spatial locality of the program, i.e. once a particular item is referenced a nearby item is often referenced in the near future. When the block size is small, most of the items fetched are referenced in the near future. However, if the block size gets too big the mean utility of the elements being fetched drops and the hit ratio starts to fall. In contrast, the performance has an inverse behaviour since the hit ratio is inversely proportional to the number of remote accesses.

6. Conclusions

The memory management system proposed in this paper is a simple extension to the RSM system already existent on the architecture considered. It has been shown to produce good results for the algorithms studied, of comparable performance to the hand-coded versions for the same algorithms. However, the system has its drawbacks and limitations.

The system is only applicable to data parallel programs where the same computation is performed on disjoint groups of data. The programmer is responsible for inserting barrier synchronisation points in the program so as to keep memory coherent. In addition, the programmer is still indirectly involved with data management since he/she has to provide the basic unit of access (block size) for each global object declared. The choice of a good block size is not always trivial. For example, a chosen block size may be good for a(some) superstep(s) but not for others. Also, the system deals only with one dimensional arrays. It could be extended to deal with multi dimensional arrays by requiring the programmer to define the 'shape' of the basic unit of access.

The exploitation of data locality requires the programmer to know something about the behaviour of the algorithm and to be able to predict the best pattern of memory management. This is undesirable but unavoidable unless an architecture can be built which has high bandwidth so that the performance of local and non-local memory is comparable, that is, there is a uniform memory architecture. Until such time as these systems exist it will be necessary for the programmer to optimise the performance of an algorithm by performing memory management either directly or indirectly.

7. Acknowledgements

The author would like to thank Jonathan Nash for the simulator used in this work and Roy Dowsing for his comments. This work has been supported by the *Instituto de Informática* (UFG, Goiânia, Brazil) and CNPq (Brazil).

References

[AgGu88] A. Agarwal and A. Gupta, "Memory-reference characteristics of multiprocessor applications under MACH". In Proc. 1988 ACM Sigmetrics Conf. Measurement Modeling Comput. Syst., Vol. 16. No. 1, pp. 215-225, May 1988.

[CDGK93] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. *Proceedings of Supercomputing 93*, Ch. 95, pp. 262-273.

[FrMa88] Francis, R. S. and Mathieson, I. D. "A Benchmark Parallel Sort for Shared Memory Multiprocessors", *IEEE Transactions on Computers*, Vol. 37, No. 12, pp. 1619-1626, December 1988.

[GeSu92] G. A. Geist and V. S. Sunderam., "Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, pages 293-311, June 1992.

[Har90] J. R. J. Harrison, "Portable tools and applications for parallel computers. In *International Journal of Quantum Chemistry*, vol. 40, pages 847-863, February 1990.

[HKT92] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling Fortran D for MIMD distributed-memory machines", *Communications of the ACM*, vol. 35, pp. 66-80, Aug. 1992.

[HPF93] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, Jan. 1993.

[LaPr89] Z. Lahjomri and T. Priol, "KOAN: A Shared Virtual memory for the iPSC/2 Hypecube". *Technical Report 597, IRISA*, France, July 1989.

[LiSc89] K. Li and R. Schaefer. "A hypercube shared virtual memory system". *Proceedings of the 1989 International Conference on Parallel Processing*, 1:125-131, 1989.

[LLGG90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-based Cache Coherence Protocol for the DASH Multiprocessor". *Proceedings of the 17h Annual International Symposium on Computer Architecture*, pages 148-159, 1990.

[MPI93] Message Passing Interface Forum, "MPI: A message passing interface". In *Proc. Supercomputing '93*, pages 878-883. IEEE Computer Society, 1993.

[Nash93] Nash, J. M. "A study of the XPRAM Model for Parallel Computing", *PhD Thesis, University of Leeds*, 1993.

[NDD95] J. M. Nash, M. E. Dyer and P. M. Dew, "Designing Practical Parallel Algorithms for Scalable Message Passing Machines". *Proceedings of the 1995 World Transputer Congress*, pages 529-541. September 1995.

[RoSt87] J. Rose and G. Steele Jr., C*: An extended C language for Data Parallel Programming. *Proceedings of the Second International Conference on Supercomputing, Vol. 2*, pages 2-16, May 1987.

[Smit87] A. J. Smith, "Line (Block) Size Choice for CPU Cache Memories". *IEEE Transactions Computers*, Vol. 36, No. 9, pp. 1063-1075, 1987.

[WaHa88] D. H. D. Warren and S. Haridi, "The Data Diffusion Machine - a scalable virtual memory multiprocessor". *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 943-952, Tokyo, Japan, Dec. 1988.