Computation and Data Partitioning on Scalable Shared Memory Multiprocessors

Sudarsan Tandri and Tarek S. Abdelrahman
Department of Electrical and Computer Engineering
The University of Toronto
Toronto, Ontario, Canada, M5S 1A4
e-mail: {tandri,tsa}@eecg.toronto.edu

Abstract

In this paper we identify the factors that affect the derivation of computation and data partitions on scalable shared memory multiprocessors (SSMMs). We show that these factors necessitate an SSMM-conscious approach. In addition to remote memory access, which is the sole factor on distributed memory multiprocessors, cache affinity, memory contention and false sharing are important factors that must be considered. Experimental evidence is presented to demonstrate the impact of these factors on performance using three applications on the KSR1 and the Hector multiprocessors.

1 Introduction

Scalable shared memory multiprocessors (SSMMs) are becoming increasingly popular and a viable alternative to distributed memory multiprocessors (DMMs). The Stanford DASH [20], FLASH [14], the KSR1 [24], Toronto's Hector [26], NUMAchine [1], and the Cray T3D [23] are some SSMMs currently in use or under development. Processors in a SSMM share a single coherent address space. However, shared memory is physically distributed to allow scalability, as shown in Figure 1. This distribution of shared memory results in non-uniform memory access latencies, depending on the distance between a processor and memory. Consequently, careful placement and management of data is essential for scaling performance.

We believe that data distribution[1] is a good paradigm for managing data in data-parallel applications on SSMMs [3, 21]. The division of array data allows a compiler to place data in the physical memory of the processor that uses it the most, and also allows the compiler to partition the computations of parallel loops. We have experimented with programmer specified data distributions on the Hector multiprocessor and have found them to be effective in improving performance. However, the task of selecting a good data distribution requires the programmer to understand both the parallel machine architecture and the data access patterns in the program. Porting programs to various machines and tuning them for performance becomes a tedious and laborious process. Consequently, it is desirable to derive data and computation partitions automatically using a compiler. The objective of this paper is to describe the factors that affect the derivation of computation and data partitions on SSMMs.

[1] In this paper we use the terms data distributions and data partitions interchangeably.

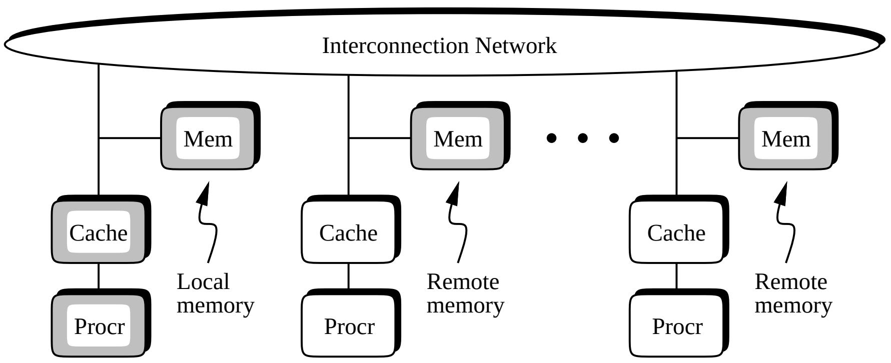

Figure 1: Scalable shared-memory multiprocessor architecture.

On DMMs, the main factor that affects the performance of an application is the cost of interprocessor communication. Consequently, scalable performance can be achieved by partitioning data and computations in a way that minimizes interprocessor communications. On SSMMs, processors communicate through shared memory, and the cost of interprocessor communications (i.e., remote memory access) is relatively inexpensive. We show that cache affinity, memory contention and false sharing are additional factors that must be considered in the selection of data distributions. Furthermore, the presence of a single shared address space allows flexibility in the selection of a computation partition. Specifically, we show that relaxing the commonly used owner-computes rule [15] has performance advantages. We present experimental results to support our conclusions using three applications on two SSMMs, the Hector and the KSR1 multiprocessors.

The remainder of this paper is organized as follows. Section 2 presents an overview data distributions. Section 3 describes the factors that impact on the selection of computation and data partitions on SSMMs. Section 4 gives experimental evidence of the impact of cache affinity and false sharing on the choice of data partitions. Section 5 presents results to show that the flexibility in selecting the computation partitioning can be used to improve performance. Section 6 reviews related work. Finally, Section 7 presents concluding remarks and directions for future work.

2 Data Distributions

Data distribution [15, 16] is achieved by specifying a partitioning scheme for each array in the program and by specifying a processor geometry to which array partitions map. A processor geometry is an n-dimensional Cartesian grid of virtual processors $(V_0, V_1, \cdots, V_{n-1})$, where V_i is the number of processors in the i^{th} dimension of the grid, and $V_0 \times V_1 \times \cdots \times V_{n-1}$ = P, the total number of processors. A partitioning scheme assigns a *partitioning attribute* to each dimension the array. There are four partitioning attributes. The `Block` attribute divides the corresponding dimension of the array in approximately equal size blocks such that a processor owns a contiguous range of that dimension of the array. The `Cyclic` attribute divides the corresponding array dimension by distributing the array elements in this dimension to processors in a round-robin fashion. The `BlockCyclic` attribute first groups array elements in the corresponding dimension in contiguous blocks of a given size, and then assigns the blocks to processors in a round-robin fashion. The block size, called the *block-cyclic factor*, is supplied by the programmer. Finally, the `*` attribute is used to indicate that the corresponding dimension of the array is not distributed. The processor geometry on which the array is mapped determines the number of processors assigned to each distributed dimension of the array. For example, distributing a two dimensional array using the `(Block,Block)` attributes onto a two dimensional processor geometry of (2,4), distributes the array on to the 8 processors, assigning 2 processors to the first dimension and 4 processors to the second dimension.

3 Performance Factors

The main factor that affects the performance of a parallel application on a DMM is the relatively high cost of interprocessor communication. For example, the latency for a remote memory access on the CM5 multiprocessor is approximately 2560 processor cycles[2]. This necessitates the selection of computation and data partitions that minimize the cost of communication. In contrast, on SSMMs, processors communicate through shared memory and the cost of remote memory access is relatively small. For example, the cost of a remote read operation on the KSR1 is approximately 170 processor cycles [24]. Consequently, other factors come into play in the selection of computation and data partitions. In this section we elaborate on these factors and on how they affect performance, and consequently, affect the choice of data and computation partitions.

3.1 Cache Affinity

Caches are used in SSMMs to reduce effective memory access time and reduce contention in the interconnection network. Data is transferred between cache and memory in units of a *cache line*, typically a multiple of the processor word size. *Spatial reuse* occurs when other words on the same line are used by the processor before the line is flushed from the cache. Analogously, *temporal reuse* occurs when data on a cache line is used again before the line is evicted from the cache. The performance of an application depends to a large extent on the ability of the caches to exploit spatial and temporal reuse. In some cases, this may be difficult because of the limited capacity and associativity of caches. Data brought into a cache by a reference or a prefetch may be evicted before being used or reused, because of either a capacity or a conflict miss caused by a subsequent reference. Cache misses on SSMMs adversely affect performance, since evicted data must be retrieved from its home memory, which may be remote to the processor. Caches play less of an important role in DMMs because cache misses result exclusively in local memory accesses, which are inexpensive relative to interprocessor communications.

3.2 False Sharing

In SSMMs data on the same cache line may be shared by more than one processor, and the line may exit in more than one processor's cache at the same time. Hardware is used to maintain the consistency of the multiple copies of the line, typically using a write-invalidate protocol [24, 14]. *True sharing* occurs when two or more processors access the same data on a cache line, and it reflects necessary data communications in an application. On the other hand, *false sharing* occurs when two processors access different pieces of data on the same cache line. If processors write to the same cache line, the cache consistency hardware causes the cache line to be transferred back and forth between processors leading to a "ping-pong" effect [8]. False sharing causes extensive invalidation traffic and can considerably degrade performance. False sharing is non-existent on DMMs.

3.3 Memory Contention

Memory contention occurs when many processors access data in a single memory module at the same time. Since the communication protocol in SSMMs is receiver-initiated, and transfers data in units of relatively small cache lines, a large number of requests to the same memory can overflow memory buffers and cause excessive delays in memory response time [13]. Contention has been considered less of a performance bottleneck on DMMs

[2]Calculated based on the elapsed time for a send-reply message of 128 bytes [19].

because a sender-initiated communication protocol is employed, and because programmers typically communicate data in large infrequent messages. Applications on DMMs also use collective communications [15] that further reduce contention.

3.4 Overhead of Parallelism

In DMMs, synchronization is achieved through data communication. However, on SSMMs, synchronization is explicit and is independent of data communication. The resulting overhead can become a performance bottleneck [27], and must be minimized. The performance of an application is also affected by the overhead involved in parallelizing loops, manifested in the form of computation partitioning tests [25]. These tests can be eliminated in some cases by compiler analysis, but when not possible, can degrade performance. This overhead though also present in the case of DMMs, is not considered significant because of the predominantly high cost of remote memory access.

4 Impact on Data Distribution

In this section we use two applications, `Multigrid` and `Tred2`, to illustrate the impact of cache affinity and false sharing on the choice of a data distribution. The KSR1 system is used because of its large cache size, and because of the presence of monitoring hardware that enables the measurement of the number of non-local memory accesses and the number of caches misses for a processor.

The KSR1 is a Cache only Memory Architecture (COMA) configured as a hierarchy of slotted rings with processing cells on the leaf-level rings. The local portion of shared memory associated with a processor is organized as a cache. There is no home location for data, rather, data may exist in more than one local memory. The hardware maintains the consistency of possible multiple copies of the data.

The KSR1 implicitly implements the owner-computes rule, since data written by a processor must exclusively reside in the processor's local portion of the shared memory. Hardware automatically migrates data to the processor that requests the data in units of *subpages*. Hence, the computation partitioning of a loop dictates the residence of a data item and hence the distribution of the arrays in the loop. Data which is read by the processors may exist in multiple local memories, and read requests to this data from different processors may be satisfied from different portions of the shared memory.

4.1 Cache-Conscious Data Distribution

The `Multigrid` application from the NAS suite of benchmarks illustrates how data distributions must be cache-conscious. `Multigrid` is a three dimensional solver calculating the potential field on a cubical grid. We focus on the subroutine `psinv` which uses two 3-dimensional arrays U and R. The subroutine mainly performs the following computation inside a triply nested loop: $U(i,\ j,\ k) +=\ \alpha(\ R(f(i),\ g(j),\ h(k)))$, where $f(i) = i - 1$, i or $i + 1$, as are the functions g and h. The loop nest is fully parallel. The application has nearest neighbor communications along all three dimensions, which is typical of many scientific applications.

In this application, we choose not to parallelize the innermost loop to avoid cache line false sharing and cache interference; successive iterations of this loop access successive elements on the same cache line. Hence we use a two dimensional grid for the processor geometry. Since the application has nearest neighbor communications, `Block` distribution performs the best. The restriction of the innermost loop to be sequential requires the arrays to be distributed

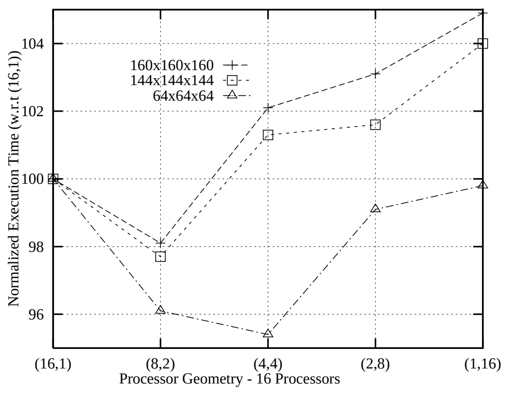

Figure 2: Normalized Execution time of `Multigrid`.

with `(*,Block,Block)` since the arrays are assumed to be stored using column major ordering. With 16 processors, it is possible to choose one of the (16,1), (8,2), (4,4), (2,8) and (1,16) processor geometries. The choice of the processor geometry affects the number of processors that execute each parallel loop. For example, a processor geometry of (8,2), implies 8 processors assigned to the inner parallel loop and 2 processors assigned to the outer parallel loop.

Figure 2 shows the execution time of the application for various processor geometries with the `(*,Block,Block)` distribution for the arrays on the KSR1 with 16 processors, normalized with respect to the (16,1) processor geometry. For a small data size (64x64x64), execution time is minimized by a distribution with equal number of processors in each dimension, i.e., (4,4). This is the same distribution scheme suggested in the Syracuse High Performance Fortran applications suite[3] for DMMs. However, when the data size is large, the processor geometry (4,4) no longer performs the best. The execution time is minimized with a processor geometry of (8,2).

The impact of processor geometry on performance is due to cache affinity, as can be deduced from Figures 3 and 4. Figure 3 shows the measured number of cache lines accessed from remote memory modules, normalized with respect to the processor geometry (16,1). The number of remote memory accesses is minimal when the processor geometry is (4,4) for all data sizes. Figure 4 shows the average measured number of cache misses from a processor cache, again normalized with respect to the processor geometry (16,1). When the data size is small (64x64x64), the data used by a processor fits into the 256k processor cache and the misses from the cache in this case reflect remote memory accesses that occur in the parallel program. Hence, the predominant factor affecting performance is interprocessor communication, and the best performance is attained using the (4,4) geometry.

However, when the arrays are relatively large (144x144x144), the cache capacity is no longer sufficient to hold data from successive iterations of the outer parallel loop, and the number of cache misses increases. When the number of processors assigned to the outer loop increases, the number of misses from the cache also increases. The (4,4) processor geometry minimizes the amount of remote memory access, but the (16,1) processor geometry minimizes the amount of cache misses. The distribution with (8,2) processor geometry strikes a balance between the cost of remote memory access and the cost of cache misses, resulting in best overall performance, in spite of higher interprocessor communication cost.

4.2 *False Sharing Conscious Data Distribution*

The programs `Tred2` (which is part of Eispack), `mdg`, and `trfd` (which are both part of the Perfect Club Benchmark Suite) exhibit parallelism which results in considerable false sharing.

[3]http://www.npac.syr.edu/hpfa/ .

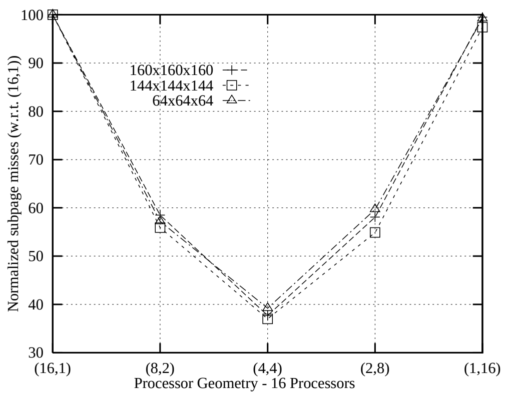

Figure 3. Remote Memory Access.

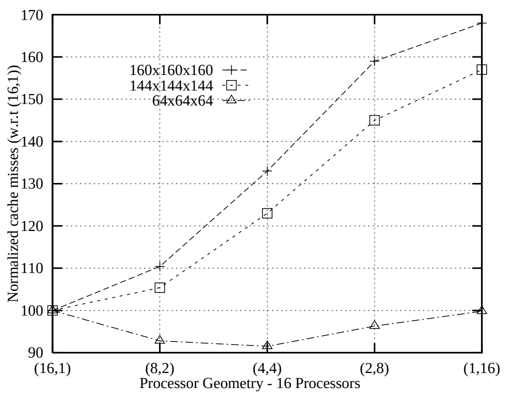

Figure 4. Cache Misses.

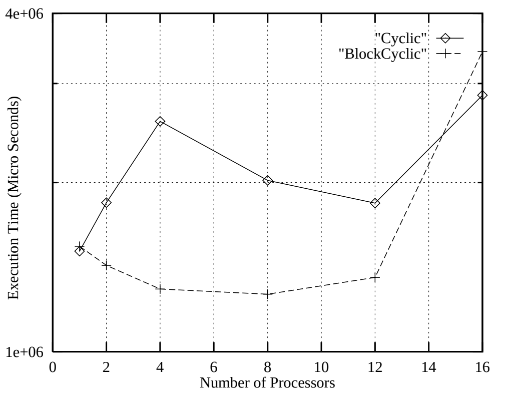

Figure 5. Effect of False Sharing.

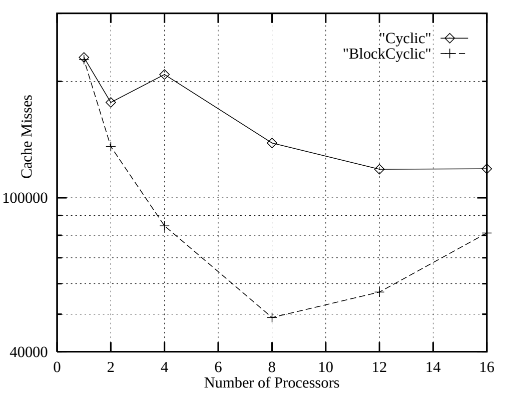

Figure 6. Cache Misses.

These programs have triangular iteration spaces which necessitate cyclical distribution for load balancing. The choice of this distribution combined with the storage order of the arrays cause more than one processor to share the same cache line, leading to false sharing. The impact of this false sharing is shown in Figure 5 for the `Tred2` application on the KSR1 multiprocessor. The figure shows the execution time of the application for `Cyclic` and `BlockCyclic` distributions using 1 to 16 processors. The use of the `Cyclic` distribution results in a large number of cache misses, as can be seen in Figure 6. The resulting overhead causes execution time to increase as the number of processors increases. The arrays are distributed using a `BlockCyclic` distribution, where the size of the block is equal to the size of the cache line, which effectively eliminates false sharing. When the number of processors is small, the load is relatively well-balanced, and the elimination of false sharing improves performance. However, as the number of processors increases, the load becomes increasingly imbalanced, and the negative impact of this load imbalance begins to outweigh the benefits of eliminating false sharing. A compiler for SSMMs must consider this tradeoff between load imbalance and false sharing when determining data distributions.

5 Impact on Computation Partitioning

The owner-computes rule has been the computation partitioner of choice for compiling HPF-type languages on DMMs [16]. The owner-computes rule maps a statement such that the the computation is executed on the processor on which the data element that is written is local. All the data elements that are required to compute the result (which may be remote) are communicated to the processor. A strict rule such as owner-computes is not necessary on a SSMM because message passing code is not generated at compile time [3]. In some

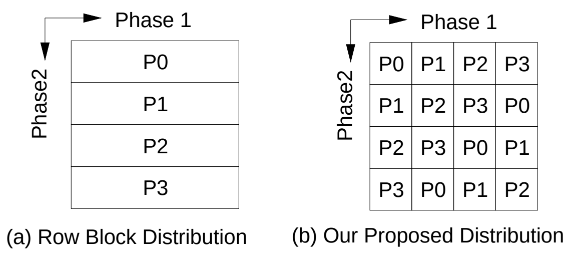

Figure 7: Data Distribution used to alleviate Memory Contention.

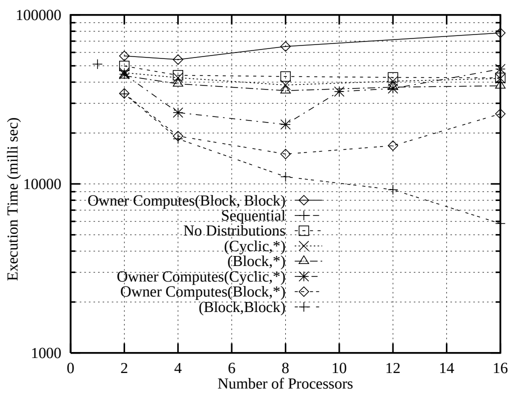

Figure 8: ADI Performance (256x256).

cases adhering to owner-computes rule can incur severe synchronization or ownership test overhead which exceeds the cost of accessing remote memory. We use the Altering Direction Integration (`ADI`) to illustrate that the shared address space provides flexibility in the choice of computation partitions, reducing contention and synchronization overhead, and resulting in significant performance improvements.

We use the Hector, a Non-Uniform Memory Access multiprocessor, as an experimental platform. Hector consists of 4 sets of processor-memory pairs connected by a bus to form a station; 4 stations are connected by a local ring to form a cluster; 4 local rings are connected by a global ring. We use a system with one cluster. Each processor-memory pair consists of a Motorola MC88100 CPU, a 16 KB instruction cache, a 16 KB data cache and 4 MB of the globally addressable memory. The hardware provides no support for cache coherence. The coherence of data is maintained by software at cache line granularity [10]. Data distributions are implemented using the array allocation techniques described in [21, 3].

5.1 Contention and Synchronization Conscious Distribution

The `ADI` program has two phases with parallelism along orthogonal dimensions in each phase. It operates on three 2-dimensional arrays A, B and X. A single iteration of an outer sequentially iterated loop consists of a forward and a backward sweep phase along the rows of three arrays, followed by another forward and backward sweep phase along the columns of the arrays [18]. This application is typical of other programs such as `2D-FFT` and `Erlebacher` that have parallelism in orthogonal directions in different phases of the program.

The best data distribution scheme for `ADI` remains an issue of debate [18, 4]. The two proposed schemes partition arrays along a single dimension, either in blocks or cyclically. These distributions, in conjunction with the owner-computes rule result in a wavefront type computation, leading to heavy synchronization overhead in one of the phases of the program. Figure 7(a) shows a `Block` distribution of the rows of the arrays. With such a distribution, during the first phase of the program all the processors access data that is local and require no communication. During the second phase, however, the parallelism is orthogonal to the

Table 1: Performance Bottlenecks for various data and computation partitioning for ADI.

Data Distribution	Compute Rule	Performance Bottleneck
None	Relaxed	Memory Contention
(*, Block)	Owner-Computes	High Synchronization
(*, Block)	Relaxed	Memory Contention
(*, Cyclic)	Owner-Computes	High Synchronization
(*, Cyclic)	Relaxed	Memory Contention
(Block, Block)	Owner-Computes	Ownership tests
(Block, Block)	Relaxed	High Remote Memory Access

direction of distribution. Strict adherence to the owner-computes rule implies ordering of the computations by processors on the corresponding chunk of the columns they own. Thus, processor i has to wait for processor $i-1$ to finish the computation on its chunk of the column before proceeding. A larger number of synchronizations are required to maintain the ordering involved in the wavefront computation.

The synchronization overhead can be eliminated by relaxing the owner-computes rule in the second phase and allowing the processor to write the results to remote memory modules. This eliminates synchronization overhead at the expense of increased remote memory accesses. However, the use of this relaxed compute rule with the `(*,Block)` distribution results in heavy contention. Each processor is responsible for computing a column, and hence, each processor accesses every memory module in sequence. Thus, a given memory module is accessed by every processor at the same time, leading to contention.

The data distribution scheme depicted in Figure 7(b)[4] eliminates contention and results in the best possible performance with the relaxed compute rule. With this distribution, processors access data from remote memory modules in both phases of the program. In both phases, processors start working on the columns assigned to them by accessing data that is in different memory modules thus avoiding contention. There is no wavefront type parallelism, and hence no overhead involved due to synchronization.

The use of owner-computes rule with the distribution of Figure 7(b) will not result in good performance. Either ownership tests must be introduced in the body of the loops to enforce the owner-computes rule, or the loops must be rewritten with additional strip-mined controlling loops to schedule the computations on sub-blocks of the array. The former leads to overhead and the latter introduces synchronization similar to the wavefront computation.

The result of executing the `ADI` application on the Hector multiprocessor for a data size of 256x256 with various data distributions and compute rules is shown in Figure 8. The `(Block,Block)` data distribution that relaxes the owner-computes rule outperforms all data distribution schemes that adhere to the rule. The figure also indicates that the overhead due to the ownership tests, when using the owner-computes rule with a `(Block,Block)` distribution, degrades performance. It is also clear that the use of data distribution improves performance over the use of operating system policies to manage data (the no distributions curve). The performance bottlenecks of various distributions for `ADI` are summarized in Table 1.

[4]This is equivalent to !HPF$ PROCESSORS PROCS(N) with !HPF$ DISTRIBUTE B(BLOCK, BLOCK), X(BLOCK, BLOCK) ON PROCS in `HPF`. In the current `HPF` specification, this distribution is not valid; the rank of each distributee must equal the rank of the named processor grid [16]. Distributions in which this is not the case introduce additional complexity on DMMs [17]. In contrast, SSMMs provide the flexibility to implement these distributions.

6 Related Work

Several researchers have focused on the problem of deriving data distributions automatically for DMMs. Li and Chen [22], Gupta and Banerjee [12], Zima et al. [9] and Garcia et al. [11] follow the approach of finding the alignment constraints between different dimensions of the arrays and derive a data distribution that minimizes interprocessor communication. To avoid a heuristic approach, Bixby et al. [7] formulate a 0-1 integer programming problem for deriving data distributions. Their approach relies on the assumption that a good data distribution for the entire program can be found by merging easier-to-determine data distributions of smaller segments of the program. They minimize the interprocessor communication using the "performance estimator" developed by Balasundaram et al. [6]. Anderson [5] presents an algebraic framework for determining data and computation partitions by minimizing communication across processors. Data transformations are then applied so that the processors access contiguous data regions to reduce false sharing. This technique is oblivious to SSMM specific issues such as contention and cache affinity.

7 Concluding Remarks

Although large SSMMs are built based on an architecture with distributed memory, the shared memory paradigm introduces performance issues that are different from those encountered in DMMs. The high cost of interprocessor communication in distributed memory multiprocessors makes the minimization of communication the predominant issue in selecting data distributions and in partitioning computations. On SSMMs, a methodology for selecting data distributions must also consider cache affinity, memory contention and false sharing in addition to the cost of interprocessor communication. Furthermore, the single shared address space present in SSMMs provides flexibility in the selection of computation partitions. This should be exploited in applications in which owner-computes results in poor performance. The *Jasmine* compiler project [2] is investigating the issues discussed in this paper through the development of a framework for automatically deriving data distributions on SSMMs.

References

[1] T.S. Abdelrahman et al. An overview of the NUMAchine multiprocessor project. In *Proc. of the Canadian Supercomputing Conf.*, pages 283–295, 1994.

[2] T.S. Abdelrahman, N. Manjikian, and S. Tandri. The Jasmine Compiler. In preparation.

[3] T.S. Abdelrahman and T.N. Wong. Distributed array data management on NUMA multiprocessors. In *Proc. of SHPCC*, pages 551–559, 1994.

[4] S.P. Amarasinghe, J.M. Anderson, M.S. Lam, and A.W. Lim. An overview of a compiler for scalable parallel machines. In *Languages and Compilers for Parallel Computing*, pages 253–272. Springer-Verlag LNCS-768, 1993.

[5] J.M. Anderson. Demonstration of automatic data and computation decomposition techniques. In *Proc. of the Workshop on Automatic Data Layout and Performance Prediction*, 1995.

[6] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proc. of PPOPP*, pages 213–223, 1991.

[7] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, 1994.

[8] W.J. Bolosky and M.L. Scott. False sharing and its effect on shared memory multiprocessors. In *Proc. of 4th Symp. on Experiences with Distributed and Multiprocessor Systems*, pages 57–71, 1993.

[9] B.M. Chapman, T. Fahringer, and H. Zima. Automatic support for data distribution on distributed memory multiprocessor systems. In *Languages and Compilers for Parallel Computing*, pages 184–199. Springer-Verlag LNCS-768, 1993.

[10] B. Gamsa. Region-oriented main memory management in shared-memory NUMA multiprocessors. Master's thesis, Department of Computer Science, University of Toronto, Toronto, CANADA, 1992.

[11] J. Garcia, E. Ayguade, and J. Labarta. A novel approach towards automatic data distribution. In *Proc. of the Workshop on Automatic Data Layout and Performance Prediction*, 1995.

[12] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):179–193, 1992.

[13] K. Harzallah and K.C. Sevcik. Hot spot analysis in large scale shared memory multiprocessors. In *Proc. of Supercomputing'93*, pages 895–905. ACM, 1993.

[14] M. Heinrich et al. The Stanford FLASH Multiprocessor. In *Proc. of the 21st Int'l Symp. on Computer Architecture*, pages 302–313, 1994.

[15] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proc. of Supercomputing'91*, pages 86–100, Albuquerque, NM, 1991.

[16] HPF. High Performance Fortran Language Specification (High Performance Fortran Forum). Technical report CRPC-TR92225, Rice University, 1994.

[17] C. Koelbel. HPF constraints. Personal Communications, 1995.

[18] U. Kremer. Automatic data layout for distributed-memory multiprocessors. Technical report CRPC-TR93229-S, Center for Research on Parallel Computation, 1993.

[19] T.T. Kwan, B.K. Totty, and D.A. Reed. Communication and computation performance of the CM5. In *Proc. of Supercomputing'93*, pages 192–201. ACM, 1993.

[20] D. Lenoski et al. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[21] H. Li and K.C. Sevcik. Numacros: Data parallel programming on NUMA multiprocessors. In *Proc. of 4th Symp. on Experiences with Distributed and Multiprocessor Systems*, pages 247–263, 1993.

[22] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *Journal of Parallel and Distributed Computing*, 2(3):361–376, 1991.

[23] Cray Research. The Cray Research Massively Parallel Processor System - Cray T3D. Technical report 80922, Munchen, Germany, 1993.

[24] Kendall Square Research. *KSR1 Principles of Operation*. Waltham, MA, 1991.

[25] J. Torres, E. Ayguade, J. Labarta, and M. Valero. Align and distribute-based linear loop transformations. In *Languages and Compilers for Parallel Computing*, pages 321–339. Springer-Verlag LNCS-768, 1993.

[26] Z. Vranesic, M. Stumm, R. White, and D. Lewis. The Hector Multiprocessor. *IEEE Computer*, 24(1):72–80, 1991.

[27] R.W. Wisniewski, L.I. Kontothanassis, and M.L. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors. In *Proc. of PPOPP*, 1995.