# Automatic Partitioning of Data and Computations on Scalable Shared Memory Multiprocessors

Sudarsan Tandri
IBM Canada Ltd.
Toronto, Ontario, Canada, M3C 1V7
tandri@vnet.ibm.com

Tarek S. Abdelrahman
Dept. of Electrical and Computer Engineering
The University of Toronto
Toronto, Ontario, Canada, M5S 3G4
tsa@eecg.toronto.edu

Abstract—*This paper describes an algorithm for deriving data and computation partitions on scalable shared memory multiprocessors. The algorithm establishes affinity relationships between where computations are performed and where data is located based on array accesses in the program. The algorithm then uses these affinity relationships to determine both static and dynamic partitions for arrays and parallel loops. Experimental results from a prototype implementation of the algorithm demonstrate that it is computationally efficient and that it improves the parallel performance of standard benchmarks. The results also show the necessity of taking shared memory effects (memory contention, cache locality, false-sharing and synchronization) into account—partitions derived to minimize only interprocessor communications do not necessarily result in the best performance.*

## 1 Introduction

Scalable Shared Memory Multiprocessors (SSMMs) are becoming increasingly used as platforms for high-performance scientific computing because they both scale to large numbers of processors and support the familiar shared memory abstraction. Scalability is achieved by physically distributing shared memory among the processors. However, this fragmentation of memory results in *non-uniform* access latencies—the latency for accessing the local portion of shared memory is less than the latency for accessing non-local or remote portions. Consequently, careful placement of data in shared memory is essential for scaling performance. Examples of SSMMs include the Stanford Flash [10], the University of Toronto Hector [16] and NUMAchine [17], the KSR1 [13], the HP/Convex Exemplar [3], and the SGI Origin 2000 [7].

Automatic parallelization of scientific applications on shared memory multiprocessors has been mainly concerned with the detection of parallelism and the scheduling of parallel-loop iterations [2]. Hence, it is not surprising that on SSMMs issues related to data placement have been ignored by compilers and delegated to the operating system. Page placement policies, such as "first-hit" and "round-robin" place pages in the physically-distributed shared memory as these pages are initially accessed [11, 16]. Unfortunately, operating system policies are oblivious to application data access patterns and manage data at too coarse of a granularity. It is too often the case that such policies fail to enhance memory locality, cause contention and hot-spots, and lead to poor performance [11].

Data partitioning is an approach used by compilers for distributed memory multiprocessors, such as High-Performance Fortran (HPF) [9], to map array data onto separate address spaces. Although such partitioning of arrays is not necessary on SSMMs because of the presence of a single coherent address space, data partitioning can be used to enhance memory locality—the compiler can place an array partition in the physical memory of the processor that uses it the most. Furthermore, data partitioning can eliminate false sharing, reduce memory contention and enhance cache locality across loop nests [15]. However, the task of selecting good data partitions requires the programmer to understand both the target machine architecture and data access patterns in the program. Consequently, porting programs to different machines and tuning them for performance becomes a tedious and laborious process [8]. It is desirable to automatically derive data and computation partitions by the compiler.

In this paper, we describe and experimentally evaluate an algorithm (called CDP) for automatically partitioning data and computations on SSMMs. Automatic data and computation partitioning has been an active area of research in recent years [1, 5, 8]. However, this research targeted distributed memory multiprocessors and derived partitions that reduce interprocessor communications. The algorithm we describe in this paper takes into account shared memory effects, including memory contention, cache locality, false-sharing and synchronization, in deriving partitions. Experimental results on 3 multiprocessors show that the parallel performance of standard benchmarks using our derived data and computation partitions compares favorably to the parallel performance of the benchmarks using operating system policies. Parallel execution time is reduced on average by factors of 2.15 on the Hector multiprocessor, 1.91 on the Convex Exemplar, and 2.04 on the KSR1. The results also demonstrate the importance of taking shared memory effects into account, and the computational efficiency of our approach.

The remainder of this paper is organized as follows. Section 2 gives a brief background on data and computation partitioning. Section 3 describes the CDP algorithm. Section 4 presents experimental results. Section 5 reviews related work. Finally, Section 6 gives concluding remarks.

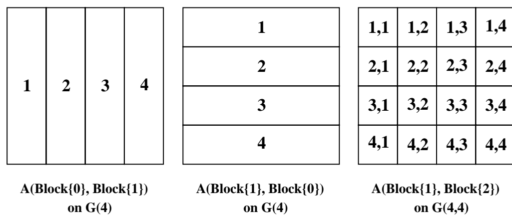


Figure 1: Examples of data partitions

# 2 Background

Data partitioning divides an array into regular partitions and assigns each partition to a processor. It is specified using a processor geometry, a distribution attribute for each array dimension and a mapping between the dimensions of the array and the dimensions of the processor geometry.

The processor geometry is a $n$-dimensional cartesian grid of virtual processors $G(P_1, P_2, \cdots, P_n)$, where $P_i$ is the number of processors in the $i^{th}$ dimension of $G$, and $P_1 \times P_2 \times \cdots \times P_n = P$ is the total number of processors.

An array dimension is partitioned or distributed by assigning it a distribution attribute. The `Block` attribute slices an array dimension into blocks of contiguous elements and assigns each block to one processor. The `Cyclic` attribute partitions an array dimension by distributing the array elements in this dimension to processors in a round-robin fashion. The `BlockCyclic` attribute specifies that $b$ contiguous array elements in a dimension are assigned to the same processor followed by the next $b$ elements to the next processor in a round-robin fashion; $b$ is the block size of the `BlockCyclic` distribution. The $*$ attribute is used to indicate that an array dimension is not partitioned.

The mapping between the distributed dimensions of an array and the dimensions of the processor geometry defines how the array is partitioned. A one-to-one mapping is used to assign a processor geometry dimension to each distributed dimension of an array [9]. In this paper, such mapping is denoted by appending the dimension of the processor geometry to the distribution attribute. For example, `Block`$\{$`1`$\}$ indicates that an array dimension is mapped onto dimension 1 of the processor geometry. A mapping of an array dimension to the non-existent dimension `0` of the processor geometry is also used to indicate that this dimension is not partitioned, and is hence equivalent to $*$. Thus, `Block`$\{$`0`$\}$ implies that a dimension is not partitioned, even though it is assigned a distribution attribute. Figure 1 shows some common data partitions.

The partitioning of an array is *static* if it is fixed throughout the execution of the program. However, in some programs it is beneficial to have an array partitioned differently in different parts of the program. In these cases, the partitioning is referred to as *dynamic*. Dynamic partitioning requires the remapping the array elements, and hence, incurs run-time overhead.

Computation partitioning is specified in a similar manner. A distribution attribute specifies the partitioning of a parallel loop and a mapping to geometry dimensions specifies the assignment of loop iterations to processors.

# 3 The CDP Algorithm

## 3.1 Overview

The goal of the CDP algorithm is to derive partitions for arrays and loops such that overall execution time is minimized. *Affinity* is said to exist between a parallel loop and an array dimension because of a reference to the array, if the parallel loop iterator appears in the subscript expression of the array dimension. Non-local memory accesses can be avoided for such a reference by partitioning the parallel loop and the array dimension using the same distribution attribute *and* by mapping both the array dimension and the parallel loop to the same dimension of the processor geometry. In general, however, there can be multiple references to the same array in multiple loop nests. When conflicting selections of the distribution attribute and/or the mapping to processor geometry dimensions arise, a *partitioning conflict* is said to exist.

When there are no partitioning conflicts for an array, it is possible to derive a unique static data partition that minimizes non-local memory accesses to the array in all loop nests. However, when there are partitioning conflicts, there exists more than one possible partition for the array. The possible partitions must be evaluated to determine one(s) that result in minimal execution time.

Consider the example shown in Figure 2. It consists of two loop nests `L1` and `L2`, each having three loops. All loops in both loop nests are parallel. The subscript expressions in the first, second and third dimensions of all references to `B` in both loop nests are functions of the parallel loop iterators `i`, `j` and `k` respectively. Non-local memory access can be minimized if each array dimension and the corresponding parallel loop are partitioned and mapped similarly. Hence, there are no partitioning conflicts for this array. In contrast, the second dimension of array `A` is accessed in `L1` using the parallel loop iterator `j` and in `L2` using the parallel loop iterator `k`. If the partitionings of loops `j` in `L1` and `k` in `L2` are not the same, then a partitioning conflict exists for the second dimension of `A`. The second dimension of `A` may be partitioned similar to the `j` loop in `L1` or to the `k` loop in `L2`, with or without dynamic partitioning. In general, the number of possible partitions in a program with partitioning conflicts is exponential and examining all possible partitions is NP-hard [8]. The CDP algorithm uses the affinity relationships and a cost model to prune the space of possible partitions.

The CDP algorithm consists of 3 phases. In the first phase, affinity relationships between array dimensions and parallel loops are captured using a graph representation. Distribution attributes and a mapping to dimensions of the processor geometry are derived for the array dimensions and parallel loops using these affinity relationships. In the second phase of the algorithm, arrays with partitioning conflicts are re-examined to determine partitions that minimize overall execution time. It is in this phase that machine specific information, such as the cost of cache, local, and remote memory accesses, the penalty for contention, and the overhead of synchronization, are considered. In the last

```
L1: forall k = 2 to N-1
       forall j = k to N
          forall i = 1 to N
              A(i,j) = B(i,j,k-1)+ B(i,j,k)
                      +B(i,j,k+1)+A(i,j)
          end for
       end for
    end for

L2: forall k = 1 to N
       forall j = 2 to N-1
          forall i = 2 to N-1
                C(i,j,N-k+1) = A(i-1,k)
                        -A(i+1,k)+B(i,j,k)
          end for
       end for
    end for
```

Figure 2: The running example.

phase of the CDP algorithm, a mapping of the virtual processors of the processor geometry to physical processors is determined.

## 3.2 New Data Partitions

In this section we introduce a new set of distribution attributes. These attributes, when combined with non-one-to-one mappings of array dimensions to processor geometry dimensions, result in data partitions not previously feasible on distributed memory multiprocessors, but are possible to support on SSMMs because of the presence of a shared address space. The benefits of the new partitions will be seen in Section 3.3.

We define six new distribution attributes. The `RBlock`, `RCyclic` and `RBlockCyclic` are the "reverse" of the regular `Block`, `Cyclic` and `BlockCyclic` distribution attributes, respectively. The assignment of the array partitions to processors in the case of reverse attributes is done in a decreasing order starting with processor `P` instead of an increasing order starting with processor `1`, as in regular attributes [9]. In addition, we synthesize regular and reverse attributes to obtain the *compound* attributes: `BlockRBlock`, `CyclicRCyclic` and `BlockCyclicRBlockCyclic`. In these attributes, elements in an array dimension are divided into two halves. Elements in the first half are partitioned using regular attributes while elements in the second half are partitioned using corresponding reverse attributes.

The mapping between array dimensions and processor geometry dimensions is also relaxed to allow an array dimension to be associated with more than one dimension of the processor geometry and to similarly allow more than one array dimension to be associated with the same dimension of the processor geometry. For example, the $(\texttt{Block}\{1\}, \texttt{Block}\{1\})$ partitioning of the 2-dimensional array `A` shown in Figure 3 results in both array dimensions being partitioned and mapped onto a 1-

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 2 | 3 | 4 | 1 |
| 3 | 4 | 1 | 2 |
| 4 | 1 | 2 | 3 |

**A(Block{1}, Block{1})**
**on G(4)**

Figure 3: An example of new data partitions.

dimensional processor geometry. The array is sliced into blocks of contiguous elements along both its dimensions. The resultant blocks are then assigned to processors such that a processor is not assigned two blocks in the same row or column. That is, a block $(i, j)$ maps to processor $((i-1)+(j-1)) \bmod P+1$ in the linear array of $P$ processors. This partitioning of the array is called a *latin square* and is useful in alleviating contention [15]. This partitioning is computationally expensive on distributed memory multiprocessors because of the complexity of finding the owner of each element of the array.

## 3.3 Data-Computation Affinity

The purpose of this phase of the algorithm is to establish affinity between arrays and parallel loops in the program and to derive static partitions for arrays with no partitioning conflicts.

### 3.3.1 Processor Geometry Dimensionality

The initial step of the CDP algorithm is to determine the dimensionality of the processor geometry. The processor geometry is determined by the number of array dimensions that contain parallel loop iterators, since it is those dimensions that potentially get distributed. It is desired to select a small geometry dimensionality while maintaining parallelism to reduce array fragmentation in shared memory.

A *parallelism matrix* $M = [m_{ij}]$ is built for an array. It has a row for each loop nest in the program and a column for each dimension of the array. An entry $m_{ij}$ is set to 1 if the subscript expression in dimension `j` of a reference to the array in loop nest $L_i$ contains one or more parallel loop iterators; otherwise, $m_{ij}$ is set to 0. The `OR_Mask` is defined as the result of `OR`ing the matrix elements along the columns, i.e. along the dimensions of the array. The `AND_Mask` is the result of `AND`ing the elements along the columns. The *visible parallelism* is defined as the sum of elements in a row of the parallelism matrix. Hence, the `OR_Mask` indicates which dimensions of the array are accessed by a parallel loop iterator in one or more loop nests. Similarly, the `AND_Mask` indicates which dimensions of the array are accessed by a parallel loop iterator in *every* loop nest in the program. Finally, the visible parallelism indicates the maximum number of array dimensions that can be distributed in a loop nest.

In order to determine the smallest number of array dimensions that can be distributed, each array is considered individually. If the `OR_Mask` of the array is 0, then none

|  | $A_1$ | $A_2$ | *Visible Parallelism* |
|---|---|---|---|
| L1 | 1 | 1 | 2 |
| L2 | 1 | 1 | 2 |
| OR_Mask | 1 | 1 |  |
| AND_Mask | 1 | 1 |  |

Figure 4: The parallelism matrix for array A of the running example.

of the dimensions of the array will be distributed. If the AND_Mask is not 0, then the number of non-zero elements in the mask is used as the dimensionality of the processor geometry for the array. However, it is possible for the OR_Mask to be non-zero, and for the AND_Mask to be 0, indicating that a dimension of the array is accessed by a parallel loop iterator in some, but not all, loop nests in the program. In this case, the dimensionality of the processor geometry for the array in a loop nest is determined by the visible parallelism in the loop nest; it is the one determined most often (i.e., statistical mode) for the array across the loop nests.

The dimensionality of the processor geometry for the entire program is chosen as the statistical mode of the dimensionalities for the arrays in the program.

The dimensions of each array that can potentially be distributed are determined using the AND_Mask and OR_Mask. If the number of 1's in the AND_Mask of an array is greater than or equal to dimensionality of the processor geometry, then the array dimensions indicated by 1 in the AND_Mask are chosen, and the array can potentially be partitioned only along these dimensions. However, if the number of 1's in the AND_Mask is less than the dimensionality of processor geometry, then all array dimensions indicated by 1 in the OR_Mask of the parallelism matrix are chosen since any of these dimensions can be distributed. Arrays in which both the AND_Mask and the OR_Mask are 0 are not distributed and are replicated to avoid contention.

The parallelism matrix for array A in the running example (Figure 2) is shown in Figure 4, where $A_1$ and $A_2$ correspond to the first and the second dimensions of A respectively. A 2-dimensional processor geometry is chosen for A based on the AND_Mask. A 3-dimensional processor geometry is chosen for B and C. Hence, a 3-dimensional geometry is chosen for the program. Both dimensions $A_1$ and $A_2$ are distributed based on the AND_Mask. All the three dimensions of B and C are also distributed based on their respective AND_Masks.

#### 3.3.2 The ALAG

A graph, called the *Array-Loop Affinity Graph (ALAG)*, is constructed to capture affinity relationships between arrays and computations. The ALAG is an undirected bipartite graph $(V, E)$, where each vertex or node $v \epsilon V$ corresponds to either a parallel loop or an array dimension, and each edge $e \epsilon E$ connects a loop node to an array dimension

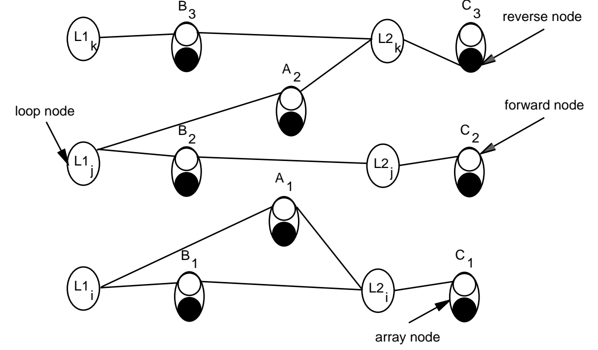


Figure 5: The ALAG for the running example.

node. There is a loop node for each parallel loop in the program. Similarly, there is an array dimension node for each array dimension that can potentially be distributed.

An array dimension node has two subnodes, a *forward* subnode and a *reverse* subnode. There is an edge between a loop node $l_i$ and the forward subnode of an array if there is a reference to the array in which a subscript expression has a positive coefficient of the iterator i. Similarly, there is an edge between a loop node $l_i$ and the reverse subnode of an array if there is a reference to the array in which a subscript expression has a negative coefficient of the iterator i. Two references to an array, one with positive and the other with negative coefficients of a parallel loop iterator result in two edges from the loop node to the array dimension node; one to the forward subnode and one to the reverse subnode.

Figure 5 shows the ALAG for the running example. The nodes labeled $L1_k$, $L1_j$, $L1_i$, $L2_k$, $L2_j$, and $L2_i$ correspond to the parallel loops in the program. Nodes labeled $A_1$, $A_2$, $B_1$, $B_2$, $B_3$, $C_1$, $C_2$, and $C_3$ correspond to the distributed dimensions of arrays A, B and C respectively. An edge connects the forward node of $A_1$ because of the references to A in L1. In contrast, an edge connects the reverse node of $C_3$ to loop node $L2_k$ because of the negative coefficient of j in the reference $C(i, j, N - k + 1)$ in loop L2.

#### 3.3.3 Initial Distribution Attributes

Loop nodes are assigned initial distribution attributes to balance the load of the corresponding parallel loop. Load imbalance occurs when the computations inside a parallel loop increase or decrease with the iteration number of the loop, or with the iteration number of a loop that encloses the parallel loop. Traditionally, the Cyclic distribution attribute is used to balance the load. The CyclicRCyclic distribution attribute better balances workloads, but introduces additional overhead [14]. Hence, we opt to use CyclicRCyclic only for the outermost parallel loop, and use Cyclic for inner parallel loops. Loop nodes are assigned a $*$ attribute when their corresponding parallel loops are load balanced.

Array dimension nodes are assigned distribution attributes to enhance access locality for corresponding array

references. An array that is referenced only once in a loop nest for which each subscript expression is a function of a single loop iterator does not pose any requirements with respect to the selection of a distribution attribute. Hence, array nodes introduced because of only such references are assigned the $*$ distribution attribute.

However, if an array is referenced in an iteration of a parallel loop more than once, all elements accessed by the iteration must be co-located to enhance locality. For example, if $\mathtt{X}(\mathtt{a} * \mathtt{i})$ and $\mathtt{X}(\mathtt{a} * \mathtt{i} + \mathtt{c})$ are two references to an array X in an iteration of a parallel loop i, then both data elements accessed must be assigned to the same processor. This is done by assigning the corresponding array dimension nodes the Block attribute.

When an array subscript expression is a function of multiple loop iterators, the relative nesting of parallel and sequential loops determines the distribution attribute of the array dimension node. Consider an array X that is referenced as $\mathtt{X}(\mathtt{a} * \mathtt{i} + \mathtt{b} * \mathtt{j})$ in a loop nest where i (outer) and j (inner) are loop iterators. If the j loop is parallel and the i loop is sequential, small segments of the array X are accessed by the iteration space of the j loop and the outer sequential loop i controls a sweep through the entire array. Hence, the array dimension node is assigned a BlockCyclic distribution attribute, with the block size equal to b, the coefficient of the parallel loop iterator in the array subscript expression. In contrast, when the i loop is parallel and j is sequential, large chunks of the array are accessed by each processor. In this case, the array dimension node is assigned a Block distribution attribute. If both i and j loops are parallel, the array dimension node is also assigned a Block distribution attribute to favor outer loop parallelism.

Initial distribution attributes are assigned to the forward and reverse subnodes of an array dimension separately. Since an array can be referenced in multiple loop nests, conflicting initial attributes of an array dimension node can result. In such cases, the BlockCyclic attribute is used.

In the running example, loop node $\mathtt{L1}_\mathtt{k}$ is assigned CyclicRCyclic attribute to balance the load. In contrast, loop node $\mathtt{L1}_\mathtt{j}$ is assigned Cyclic attribute to balance the load. All other loop nodes are assigned a $*$ distribution attribute because load balancing is not an issue. The forward subnode of array dimension nodes $\mathtt{B}_3$ and $\mathtt{A}_1$ are assigned a Block distribution attribute. All other array dimension nodes are assigned a $*$ distribution attribute.

### 3.3.4 Final Distribution Attributes

The initial distribution attributes do not ensure that parallel loops and array dimensions whose subscript expressions are functions of the parallel iterators are partitioned similarly. Hence, in this step of the algorithm, connected nodes in the ALAG are made to have the same distribution attribute. We refer to such a condition as *consensus*.

When the distribution attributes of two connected nodes in the ALAG are not the same, a *conflict* is said to exist. This conflict is resolved using the conflict resolution graph shown in Figure 6. The resolution of two distribution attributes is the least common ancestor in the conflict resolution graph. For example, the attributes Cyclic and Block resolve to BlockCyclic. The BlockCyclic attribute allows both locality of access to be maintained through the selection of an appropriate block size, and allows load to be balanced by distributing the blocks Cyclicly. A node in the conflict resolution graph can be its own least common ancestor. A distribution attribute in conflict with the $*$ attribute resolves to the attribute itself.

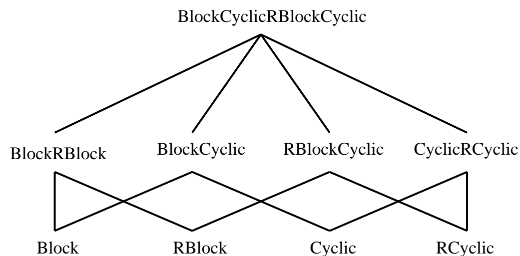


Figure 6: Conflict resolution graph.

For each array dimension node, the distribution attributes of the forward and reverse subnodes are first made consistent by having the distribution attribute of the forward subnode be the reverse of the reverse subnode. An iterative process is then used to reach consensus. In each iteration, the distribution attribute of a forward node is compared to that of each node directly connected to it. A conflict is resolved as described above. Since all distribution attributes have one common ancestor, the process is guaranteed to terminate.

In the running example, the distribution attribute of $\mathtt{B}_3$ is forced to be BlockCyclicRBlockCyclic(BCRBC) resolving its distribution attribute Block and the CyclicRCyclic distribution attribute of loop node $\mathtt{L1}_\mathtt{k}$ connected to it. When consensus is reached, nodes $\mathtt{L1}_\mathtt{k}$, $\mathtt{L1}_\mathtt{j}$, $\mathtt{L2}_\mathtt{k}$, $\mathtt{L2}_\mathtt{j}$, $\mathtt{A}_2$, $\mathtt{B}_3$, $\mathtt{B}_2$, $\mathtt{C}_3$ and $\mathtt{C}_2$, are partitioned using the BCRBC distribution attribute. Nodes $\mathtt{L1}_\mathtt{i}$, $\mathtt{L2}_\mathtt{i}$, $\mathtt{A}_1$, $\mathtt{B}_1$ and $\mathtt{C}_1$ are partitioned using the Block distribution attribute.

### 3.3.5 Processor Geometry Dimension Assignment

Parallel loops and distributed array dimensions must be assigned processor geometry dimensions to complete the partitioning of computations and data. The goal of this assignment is to ensure that elements of arrays required by an iteration of a parallel loop are assigned to same processor that executes the iteration. In other words, it is desired to have connected nodes in the ALAG map onto the same processor geometry dimension.

Processor geometry dimensions are first assigned to loops and distributed array dimensions in loop nests in which the number of parallel loops is equal to the dimensionality of the processor geometry. It is in these loop nests that all parallel loops must indeed execute in parallel. The first such loop nest in the program is considered first. Parallel loops are assigned geometry dimensions from outermost

to innermost, assigning the highest geometry dimension to the outermost loop and the lowest dimension to the innermost loop. The assignment of dimensions to loops is then *propagated* to array dimensions using the ALAG. The geometry dimension assigned to a loop node is also assigned to all array nodes connected to the loop node. An array dimension node may be assigned more than one dimension of the processor geometry based on the number of loop nodes connected to it.

The next loop nest in which the number of parallel loops is equal to the geometry dimensionality is considered. Parallel loops whose corresponding loop nodes in the ALAG are connected to array dimension nodes that have already been assigned a geometry dimension are assigned the same geometry dimension. Parallel loops that are not assigned geometry dimensions in this way are assigned a dimension as described for the first loop nest from outermost to innermost, but only geometry dimensions that have not been assigned to a loop node in the current loop nest are considered. The dimension assignment to each loop node is then propagated to all the array dimension nodes connected to it, and the process is repeated for the remaining loop nests in which the number of parallel loops is equal to the geometry dimensionality.

In loop nests in which the number of parallel loops is not equal to the geometry dimensionality, either some parallel loops will not be assigned a geometry dimension and hence will execute sequentially, or some of the geometry dimensions will go unused. Such loop nests are also considered one at a time. First, geometry dimensions are assigned to parallel loops by propagating geometry dimensions from array nodes to loop nodes and by assigning available geometry dimensions to parallel loops that remain without a geometry dimension outermost loop to innermost loop in a loop nest. If the number of parallel loops in a loop nest is greater than the geometry dimensionality, then some loops will not be assigned a geometry dimension. However, if the number of parallel loops is less than the geometry dimensionality, some of the geometry dimensions will not be utilized.

Figure 7 shows the processor geometry assignment for the running example. Loop nodes $\mathtt{L1}_k$, $\mathtt{L1}_j$ and $\mathtt{L1}_i$ are assigned geometry dimensions 3, 2 and 1 respectively. These assignments are propagated to array dimension nodes $\mathtt{A}_2$, $\mathtt{B}_1$, $\mathtt{B}_2$, and $\mathtt{B}_3$. Loop nodes $\mathtt{L2}_k$, $\mathtt{L2}_j$ and $\mathtt{L2}_i$ are assigned dimensions 3, 2 and 1 respectively based on the assignments of $\mathtt{B}_3$, $\mathtt{B}_2$ and $\mathtt{B}_1$. Consequently, the array dimension nodes $\mathtt{C}_3$, $\mathtt{C}_2$ and $\mathtt{C}_1$ also are assigned geometry dimensions 3, 2 and 1 respectively. Because $\mathtt{A}_2$ is connected to both $\mathtt{L1}_j$ and $\mathtt{L2}_k$, it gets the geometry assignment of both these loop nodes 2, 3. For arrays B and C no partitioning conflicts exist and static partitions are determined. However there is a partitioning conflict for A and all possible data partitions must be evaluated to determine if dynamic partitioning is necessary.

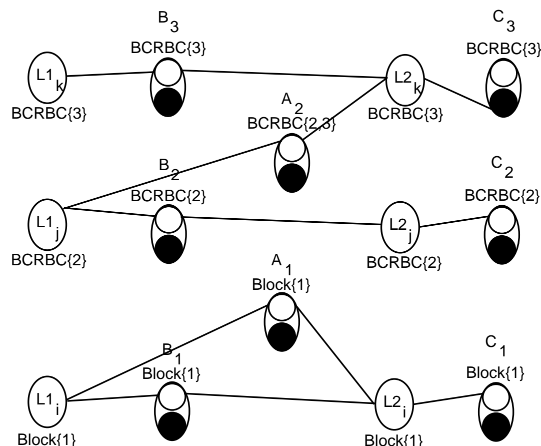


Figure 7: Processor geometry assignment for running example.

## 3.4 Dynamic Array Partitions

The partitions derived so far by the algorithm establish affinity between data accessed and computations through static array partitions. For arrays for which a one-to-one mapping between the distributed dimensions of the arrays and the dimensions of the processor geometry, static partitions make accesses to the arrays local across all loop nests in the program. However, for arrays for which a one-to-one mapping is not used, remote memory accesses result in some or all of the loop nests. The geometry dimension assignment was made for such arrays to satisfy multiple locality requirements in the different loop nests. Since a parallel loop is partitioned along only a single dimension of the processor geometry, partitioning an array dimension along multiple dimensions of the geometry results in non-local accesses. Similarly, partitioning multiple dimensions of the array along a single dimension of the processor geometry also results in non-local accesses. Hence, the partitioning of these arrays in the loop nests in which they are referenced is re-evaluated to determine if dynamic partitioning may result in better program performance.

The assignment of geometry dimensions to array dimension is explored in selecting appropriate data partitions. The permutations of the geometry dimensions (including dimension 0 or no partitioning) for each dimension of an array is the set of possible data partitions for the array in the program. In the running example, the 2-dimensional array is partitioned on a 3-dimensional processor geometry using $\mathtt{A}(\mathtt{Block}\{1\}, \mathtt{BCRBC}\{2,3\})$, and its partitioning should be re-evaluated. The set of possible data partitions are: $\mathtt{A}(\mathtt{Block}\{0\}, \mathtt{BCRBC}\{0\})$ which replicates the array completely; $\mathtt{A}(\mathtt{Block}\{1\}, \mathtt{BCRBC}\{0\})$, $\mathtt{A}(\mathtt{Block}\{0\}, \mathtt{BCRBC}\{2\})$, $\mathtt{A}(\mathtt{Block}\{0\}, \mathtt{BCRBC}\{3\})$, and $\mathtt{A}(\mathtt{Block}\{0\}, \mathtt{BCRBC}\{2,3\})$ which partially replicate the array; $\mathtt{A}(\mathtt{Block}\{1\}, \mathtt{BCRBC}\{2\})$ which results in local accesses to the array in the first loop nest, but not the second; $\mathtt{A}(\mathtt{Block}\{1\}, \mathtt{BCRBC}\{3\})$ which results in local accesses to the array in the second loop nest, but not

the first; and A(Block{1}, BCRBC{2, 3}) which results in non-local accesses in both loops.

In order to select an appropriate data partition for any array, the time to access the array in each loop nest must be estimated for each of the above partitions. When the partitions differ across loop nests, the array must be repartitioned between loop nests, or loop nests must be executed in a pipelined fashion. Machine-specific information, such as cache, local, and remote memory accesses latencies, cost of synchronization, penalty for contention, and cost of redistribution are used to estimate the time for array accesses. Cost estimation heuristic used are described in [14], and are beyond the scope of this paper.

The set of possible data partitions is searched to determine the data partitions that minimize the overall execution time. A depth-first search with pruning is used to limit the size of the search space and make the search practical.

### 3.5 Processor Geometry Mapping

The final step of the CDP algorithm is to determine the number of processors in each dimension of the processor geometry, and to map its virtual processors to physical processors.

Factors of the number of processors $P$ are used to generate possible processor geometries. For example, when $P = 8$, possible 2-dimensional processor geometries are (1, 8), (2, 4), (4, 2) and (8, 1). The costs of cache, local and remote memory accesses for loop nests are determined given the data and computation partitions for each geometry and the one with the minimal cost is selected. The heuristics used to calculate the costs of local and remote memory accesses and the number of cache misses are beyond the scope of this paper and are discussed in [14].

The physical processors are viewed as a linear array and virtual processor $(p_1, p_2, \cdots, p_n)$ is assigned to the physical processor numbered $\sum_{i=1}^{n} p_i \prod_{j=1}^{i-1} P_j$. Thus, for a 2-dimensional processor geometry, this mapping implies a column-major order assignment of virtual processors to physical processors. This mapping allows inner loops to execute on adjacent physical processors, which typically improves performance because on SSMMs remote memory access latency to close-by processors is lower.

## 4 Experimental Results

We implemented our algorithm in a prototype compiler called *Jasmine*, which is being developed at the University of Toronto. The compiler has four major phases: parallelism detection, cache locality enhancement, memory locality enhancement and code generation. We use the Polaris compiler [2] from the University of Illinois for parallelism detection. The algorithm described in this paper constitutes the memory locality enhancement phase.

The compiler was used to automatically determine data and computation partitions for applications and to generate parallel code. The resulting parallel code was executed on 3 multiprocessor platforms to measure performance; Hector [16], an HP/Convex Exemplar SPP1000 [3] and a KSR1 [13].

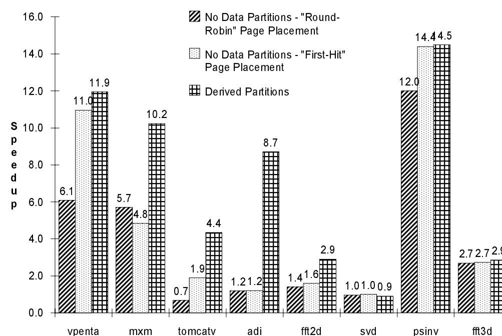

Figure 8: Performance on a 16-processor Hector.

We present three sets of results in this section. The first set shows that the overall performance of the applications is improved by placing array data in memory using the partitions derived by our compiler, as opposed to relying on operating system policies. The second set shows that partitions derived by our compiler taking into account shared memory effects in addition to interprocessor communications lead to better performance than partitions derived taking only interprocessor communications into account. The last set of results shows the computational efficiency of our framework and that it compares favorably to that of other approaches.

### 4.1 Overall Performance

We applied our compiler prototype to the 8 benchmark applications summarized in Table 1. The data partitions derived by the compiler for the main arrays in each application on each of the 3 multiprocessor platforms are shown in Table 2. The processor geometry is a linear array unless otherwise indicated.

The speedup (with respect to sequential execution) of the parallel applications with and without the use of our data and computation partitions on the Hector multiprocessor is shown in Figure 8. The performance of the applications without data partitions refers to their performance using supported operating system page placement policies and partitioning the outermost parallel loop in each loop nest. In the case of Hector, two policies are supported, “first-hit” and “round robin” [16]. The performance of the applications with data partitions is better on average by a factor of 2.15 compared to their performance with page placement policies, which validates our approach to data placement.

The speedup of the applications on the HP/Convex Exemplar with and without data partitions shown in Figure 9. The Exemplar only supports the “round-robin” page placement policy [3]. Similar to Hector, the performance of the applications on the Exemplar is better on average by a factor of 1.91 using our algorithm compared to relying on the page placement policy.

The speedup of the applications on the KSR1 is shown in Figure 10 (fft2d and fft3d were not executed on the

Table 1: Application characteristics.

| Application | Lines of Code | No. of Proc- -edures | No. of parallel loop nests | No. of Arrays | Array dimensi- -onality | Brief Summary |
|---|---|---|---|---|---|---|
| vpenta | 126 | 2 | 7 | 9 | 2/3 | inverts three pentadiagonal matrices |
| mxm | 50 | 2 | 2 | 3 | 2 | matrix multiplication |
| tomcatv | 195 | 1 | 9 | 7 | 2 | mesh generation program |
| ADI | 35 | 1 | 5 | 3 | 2 | stencil computation |
| fft2d | 250 | 2 | 11 | 4 | 1/2 | 2-D fast fourier butterfly computation |
| svd | 319 | 1 | 16 | 5 | 1/2 | decomposes a 2-D array |
| psinv | 20 | 1 | 1 | 2 | 3 | computes 3-D potential field |
| fft3d | 70 | 2 | 3 | 2 | 3/1 | 3-D fast fourier butterfly computation |

Table 2: Derived data partitions.

| Application | Hector (16 Processors) | Convex (30 Processors) | KSR1 (16 Processors) |
|---|---|---|---|
| vpenta | F(∗, Block{1}, ∗) | F(∗, Block{1}, ∗) | F(∗, Block{1}, ∗) |
| mxm | A(∗, ∗) | A(∗, ∗) | A(∗, ∗) |
| | C(∗, Block{1}) | C(∗, Block{1}) | C(∗, Block{1}) |
| tomcatv | RX(Block{1}, ∗) | RX(Block{1}, ∗) | RX(Block{1}, ∗) |
| ADI | X(Block{1}, Block{1}) | X(∗, Block{1}) | X(∗, Block{1}) |
| | | /Pipelining | /X(Block{1}, ∗) |
| fft2d | X(∗, Block{1}) | X(∗, Block{1}) | X(∗, Block{1}) |
| | /X(Block{1}, ∗) | /X(Block{1}, ∗) | /X(Block{1}, ∗) |
| svd | U(BCRBC{1}, BCRBC{1}) | U(BCRBC{1}, BCRBC{1}) | U(BCRBC{0}, BCRBC{1}) |
| psinv | U(∗, Block{1}, Block{2}) | U(∗, Block{1}, Block{2}) | U(∗, Block{1}, Block{2}) |
| proc. geom. | (16, 1) | (6, 5) | (8, 2) |
| fft3d | Z(∗, Block{1}, Block{2}) | Z(∗, Block{1}, Block{2}) | Z(∗, Block{1}, Block{2}) |
| proc. geom. | (1, 16) | (1, 30) | (1, 16) |

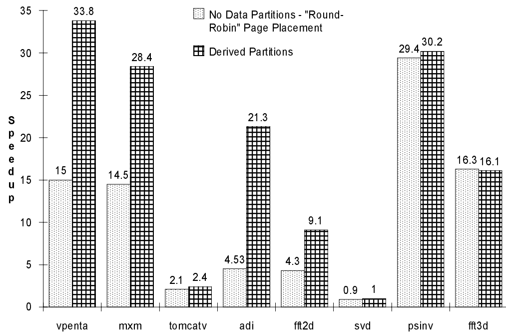

Figure 9: Performance on a 30-processor HP/Convex.

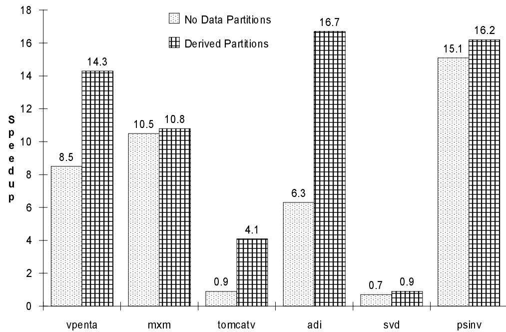

Figure 10: Performance on a 16-processor KSR1.

KSR1 multiprocessor due to native compiler limitations).

Since the KSR1 is a COMA multiprocessor, the performance of the applications without data partitions refers to their performance with the underlying hardware which copies and replicates data on demand. Again, the performance of the applications with our derived partitions is better on average by a factor of 2.04.

## 4.2 Impact of Shared Memory Effects

In this section we show the impact of shared memory effects. That is, we show that data and computation partitions derived by our compiler taking into account cache locality, false sharing, contention and synchronization in addition to interprocessor communications outperform partitions derived for distributed memory multiprocessors taking only interprocessor communications into account. We use one application due to space limitations, and refer the reader to [14] for more results.

The psinv kernel is used to show the impact of cache locality. The kernel performs a nearest-neighbor computation using two 3-dimensional arrays inside a triply nested loop nest. All three loops in the nest are parallel. With 16 processors, both arrays would be partitioned on distributed memory multiprocessors using (∗, Block{1}, Block{2}) and a (4,4) 2-dimensional processor geometry in order to minimize interprocessor communications. Using our algorithm, the arrays are also partitioned using (∗, Block{1}, Block{2}), but using a (8,2) processor geometry with the outermost parallel loop

mapped to the 2nd dimension of the geometry and the next inner loop mapped to the 1st dimension. The innermost loop is executed sequentially to avoid false sharing.

The execution time of psinv on the KSR1 is shown in Figure 11 for possible processor geometries and for three array sizes. The curves are normalized with respect to the (16,1) geometry. When the size of the arrays is small (64x64x64), execution time is minimized by the (4,4) geometry. However, when the size of the arrays is larger execution time is minimized using the (8,2) geometry.

The impact of processor geometry on performance is due to cache locality, as can be deduced from Figures 12 and 13. Figure 12 shows the average measured number of cache lines accessed from remote memory modules (i.e., interprocessor communications), normalized with respect to the (16,1) geometry. The number of remote memory accesses is minimal when the processor geometry is (4,4) for all data sizes, which validates its choice for distributed memory multiprocessors. Figure 13 shows the average measured number of cache misses from a processor cache, again normalized with respect to the (16,1) geometry. When the data size is small (64x64x64), the data used by a processor mostly fits into the 256-Kbyte processor cache and the misses from the cache in this case reflect remote memory accesses. Hence, the best performance is attained using the (4,4) geometry, which minimizes remote accesses.

However, when the arrays are larger, the cache capacity is no longer sufficient to hold all array data across successive iterations of the outer parallel loop, and the number of cache misses increases. When the number of processors assigned to the inner parallel loop increases (1st geometry dimension), less inner loop iterations are assigned to a processor, and the size of data accessed those iterations decreases. The size of data accessed across iterations of the outer parallel loop on a processor decreases, and hence, the number of misses from the cache decreases. The (4,4) geometry minimizes the amount of remote memory access, but the (16,1) geometry minimizes the amount of cache misses. The CDP algorithm considers both factors and strikes a balance with the (8,2) geometry to result in best overall performance.

### 4.3 Computational Efficiency

Table 3 gives the time taken by the prototype compiler to derive data and computation partitions for each application on a Sun SPARCstation 10. It also shows the number of array and loop nodes in the ALAG, the number of array and loop nodes re-evaluated for dynamic partitioning, the total size of possible partitions and the number of partitions actually examined, and the number of processor geometry choices considered. In half of the benchmarks static data partitions are derived directly from the ALAG. The pruned search is successful in limiting the number of partitions examined when partitioning conflicts exist.

The computational efficiency of our algorithm is favorable compared to that of other approaches (see Section 5). For example, in cases where partitions are obtained directly from the ALAG our algorithm is more efficient than solving 0–1 integer programming problems of sizes 2595 × 338, 170 × 59, 1940 × 354, and 15 × 1 for vpenta, mxm, tomcatv and psinv respectively, using Bixby's approach.

Table 3: Measurements from the Jasmine compiler.

| Application | No. of Array/Loop nodes | No. of Arrays/Loops re-eval. | Possible/Examined Part. | No. of Processor Geometry Choices | Total Compile Time (mS) |
|---|---|---|---|---|---|
| vpenta | 9/9 | –/– | –/– | 1 | 31 |
| mxm | 3/4 | –/– | –/– | 1 | 5 |
| tomcatv | 7/11 | –/– | –/– | 1 | 14 |
| ADI | 6/5 | 3/5 | $3*4^5$/116 | 1 | 280 |
| fft2d | 4/11 | 1/9 | $4^9$/1591 | 1 | 5184 |
| svd | 5/16 | 1/13 | $4^{13}$/240 | 1 | 1993 |
| psinv | 4/3 | –/– | –/– | 5 | 197 |
| fft3d | 3/6 | 1/3 | $16^3$/16 | 5 | 697 |

# 5 Related Work

Automatic approaches for deriving data and computation partitions have focused primarily on distributed memory multiprocessors. These approaches first derive data partitions that minimize some metric of performance, then use the *owner-computes* rule [6] to determine corresponding computation partitions. Since communications is the main factor that affects performance on distributed memory multiprocessors, the cost of interprocessor communications is used as the metric.

Li and Chen [12] and Gupta and Banerjee [5], represent the alignment constraints between array dimensions in a program in the form of a component alignment graph, then apply graph-partitioning heuristics to derive only static array partitions which minimize interprocessor communications.

Bixby et al. [8] formulate a 0–1 integer programming problem to represent all possible data partitions and their associated costs. Their approach relies on the assumption that good partitions for the program can be obtained by dividing the program into phases or segments and analyzing data partitions of each segment alone. Data partitions are allowed to change across segments. The cost of a data partition is determined using a static performance estimator. Garcia et al. [4] also use 0–1 integer programming to derive static data partitions for arrays, but use profiling to more accurately estimate the costs of partitions.

Anderson, Amarasinghe and Lam [1] present an algebraic framework for both distributed memory multiprocessors and SSMMs. They determine data and computation partitions that minimize interprocessor communications. Profiling is used to determine the loops that are most commonly executed. A static data partition is then derived for these loops, hence reducing the cost of array repartitioning. Transformations are then applied to reduce false sharing

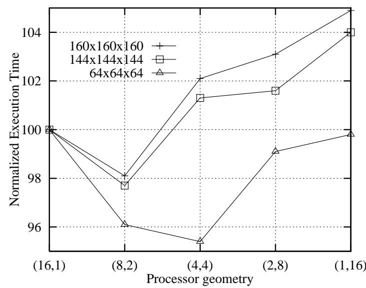

Figure 11: Execution time of `psinv`.

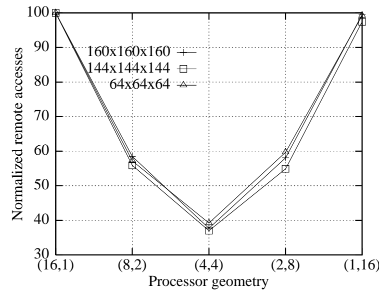

Figure 12: Remote accesses in `psinv`.

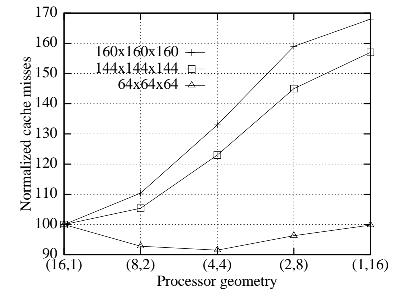

Figure 13: Cache misses in `psinv`.

and synchronization. The effect of memory contention is ignored.

In contrast, the algorithm presented in this paper targets SSMMs and takes all shared memory affects into consideration. The partitions are obtained without the run-time profiling. Our algorithm derives dynamic partitions using an efficient pruned search.

# 6 Conclusion

Data partitioning is proposed as a mechanism for data placement on SSMMs. Determining good partitions can be difficult for programmers. In addition, cache locality, contention, synchronization and false sharing must be considered in the selection of partitions on SSMMs. This paper described an algorithm to automatically derive data and computation partitions for a program taking into account such factors. Experimental results show that the use of the data and computation partitions derived by our algorithm improve the performance of the standard benchmark applications over the use of operating system policies for page placement. Experimental results also demonstrate the importance of taking shared memory effects into consideration; data and computation partitions that are derived to only minimize interprocessor communications do not necessarily lead to the best performance.

# References


[1] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. of PPoPP*, 1995.

[2] W. Blume et al. Polaris: Improving the effectiveness of parallelizing compilers. In *Languages and Compilers for Parallel Computing*, pages 141–154, 1994.

[3] Convex Computer Corporation. *Convex Exemplar System Overview*. Richardson, TX, USA, 1994.

[4] 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.

[5] M. Gupta and P. Banerjee. Automatic Data Partitioning on Distributed Memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):179–193, 1992.

[6] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D . *CACM*, 35(8):66–79, 1992.

[7] Silicon Graphics Inc. *The SGI Origin 20000*. Mountain View, CA, 1996.

[8] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proc. of Supercomputing*, 1995.

[9] C. Koelbel et al. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[10] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proc. of ISCA*, pages 302–313, 1994.

[11] R. LaRowe Jr., J. Wilkes, and C. Ellis. Exploiting Operating System Support for Dynamic Page Placement on a NUMA Shared Memory Multiprocessor. In *Proc. of PPoPP*, pages 122–132, 1991.

[12] J. Li and M. Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *Journal of Parallel and Distributed Computing*, 2(3):361–376, 1991.

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

[14] S. Tandri. *Automatic Data and Computation Partitioning on Scalable Shared Memory Multiprocessors (in preperation)*. PhD thesis, Department of Computer Science, University of Toronto, 1997.

[15] S. Tandri and T. Abdelrahman. Computation and data partitioning on scalable shared memory multiprocessors. In *Proc. of PDPTA*, 1995.

[16] Z. Vranesic et al. The Hector Multiprocessor. *IEEE Computer*, 24(1):72–79, 1991.

[17] Z. Vranesic et al. The NUMAchine Multiprocessor. Technical Report CSRI-324, Computer Systems Research Institute, University of Toronto, 1995.