Automatic Data and Computation Partitioning on Scalable Shared Memory Multiprocessors

Sudarsan Tandri and Tarek S. Abdelrahman
Department of Electrical and Computer Engineering
The University of Toronto
Toronto, Canada, M5S 3G4
e-mail: {tandri,tsa}@eecg.toronto.edu

Abstract

This paper describes an algorithm for deriving data and computation partitions to improve locality on scalable shared memory multiprocessors. The algorithm first determines a computation partitioning by establishing affinity between where the computations are performed and where the data is located. It then determines a data partitioning for each array separately. It takes into account performance issues that are specific to shared memory multiprocessors, like contention, synchronization and false sharing using machine-specific cost models. The algorithm is implemented in a prototype compiler. Experimental results on three multiprocessor systems demonstrate the impact of shared memory issues and the effectiveness of the partitions derived using the algorithm.

1 Introduction

Scalable Shared Memory Multiprocessors (SSMMs) are becoming increasingly viable platforms for high-performance scientific computing; 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 shared memory results in *non-uniform* memory access latencies—the latency for accessing the local portion of shared memory is less than the latency for accessing non-local or remote portions. Hence, careful placement of data in shared memory is essential for scaling performance. Examples of SSMMs include the Stanford Dash [1] and Flash [2], the University of Toronto Hector [3] and NUMAchine [4], the KSR1 [5], and the Convex Exemplar [6].

Automatic parallelization of scientific applications on bus-based shared memory multiprocessors has been mainly concerned with the detection of parallelism and the scheduling of parallel-loop iterations [7, 8]. Hence, it is not surprising that on SSMMs issues related to data placement have been ignored by compilers and delegated to the operating system as part of page management. Policies, such as "first-hit" and "round-robin" place pages in the physically-distributed shared memory as these pages are initially accessed [3, 4, 9].

However, it is too often the case that such policies fail to enhance memory locality and cause contention and hot-spots, leading to poor performance [9]. Operating system policies are oblivious to application data access patterns and manage data at too coarse of a granularity.

Data partitioning can be a good paradigm for compiler-based data management in scientific applications on SSMMs. 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 accordingly partition the computations of parallel loops. We have experimented with programmer specified data partitions on the Hector and the Convex multiprocessors and have found them to be effective in improving performance [10, 11]. However, the task of selecting a good data partition 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 automatically derive data and computation partitions by the compiler.

In this paper, we describe an algorithm for automatically deriving data and computation partitions on SSMMs. The automatic derivation of data and computation partitions has been an active area of research in recent years [12, 13, 14, 15, 16]. This research targets distributed memory multiprocessors and derives partitions that minimize remote memory accesses (i.e. interprocessor communication). We show that an algorithm that derives data and computation partitions on SSMMs must take into consideration shared memory effects, including memory contention, false-sharing and synchronization. These effects if ignored, can cause more performance degradation than loss of memory locality. Furthermore, the presence hardware support for a shared coherent address space allows a different view of the role of data partitions. Whereas on distributed memory multiprocessors data partitions are used to map data onto the separate address spaces and to maintain data coherence, on SSMMs data partitions need only be used to introduce affinity between the location of computation and the location of data.

The remainder of this paper is organized as follows: Section 2 presents an overview of data and computation partitions used by our algorithm. Section 3 describes the algorithm for deriving data and computation and partitions. Section 4 presents an experimental evaluation of the algorithm using applications on 3 multiprocessor platforms. Section 5 reviews related work. Finally, Section 6 presents concluding remarks.

2 Data and Computation Partitions

In this section, the notion of data and computation partitioning is formally defined. A partitioning of an array tessellates the array into regular partitions that are mapped onto available processors. Such a partitioning is specified using a processor geometry, a set of distribution attributes for the dimensions of the array, 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 processors $(P_1, P_2, \cdots, P_n)$, where P_i is the number of processors in the i^{th} dimension of the grid, and $P_1 \times P_2 \times \cdots \times P_n = P$, the total number of processors used to execute the program.

An array dimension is partitioned or distributed by assigning a distribution attribute to that dimension. There are 9 possible distribution attributes. The `Block` attribute specifies that the array data should be distributed across processors by slicing the array dimension into blocks of contiguous elements. The `Cyclic` attribute maps the array dimension by distributing the array elements in this dimension to processors in a round-robin fashion. The `BlockCyclic` attribute specifies that n successive array elements must mapped onto the same processor followed by the next set of elements to the next processor in round-robin fashion, where n is the block size for the `BlockCyclic` distribution. `Block`, `Cyclic` and `BlockCyclic` are the traditional distribution attributes that have been used to partition data in other compilers, like HPF [17, 18, 19].

In addition, we define six new distribution attributes. `RBlock`, `RCyclic` and `RBlockCyclic` are the reverse of the `Block`, `Cyclic` and `BlockCyclic` distribution attributes[1]. The assignment of the array tessellations to the processors is done in a decreasing order starting with processor `P` in the case of reverse distribution attributes, unlike the traditional distribution attributes where the assignment to the processors occurs in an increasing order starting with processor `1`. It is also possible to synthesize the traditional and the reverse distribution directives. An array dimension may be partitioned using `BlockRBlock`, `CyclicRCyclic` or `BlockCyclicRBlockCyclic` distribution attributes. In these cases, the dimension is divided into two halves with the first half of the array dimension partitioned using the `Block`, `Cyclic` or `BlockCyclic` distribution attribute while the second half is partitioned using `RBlock`, `RCyclic` or `RBlockCyclic` distribution attribute

[1] `RBlock`, `RCyclic` and `RBlockCyclic` can be obtained in HPF by using an alignment attribute with the `Block`, `Cyclic` and `BlockCyclic` templates.

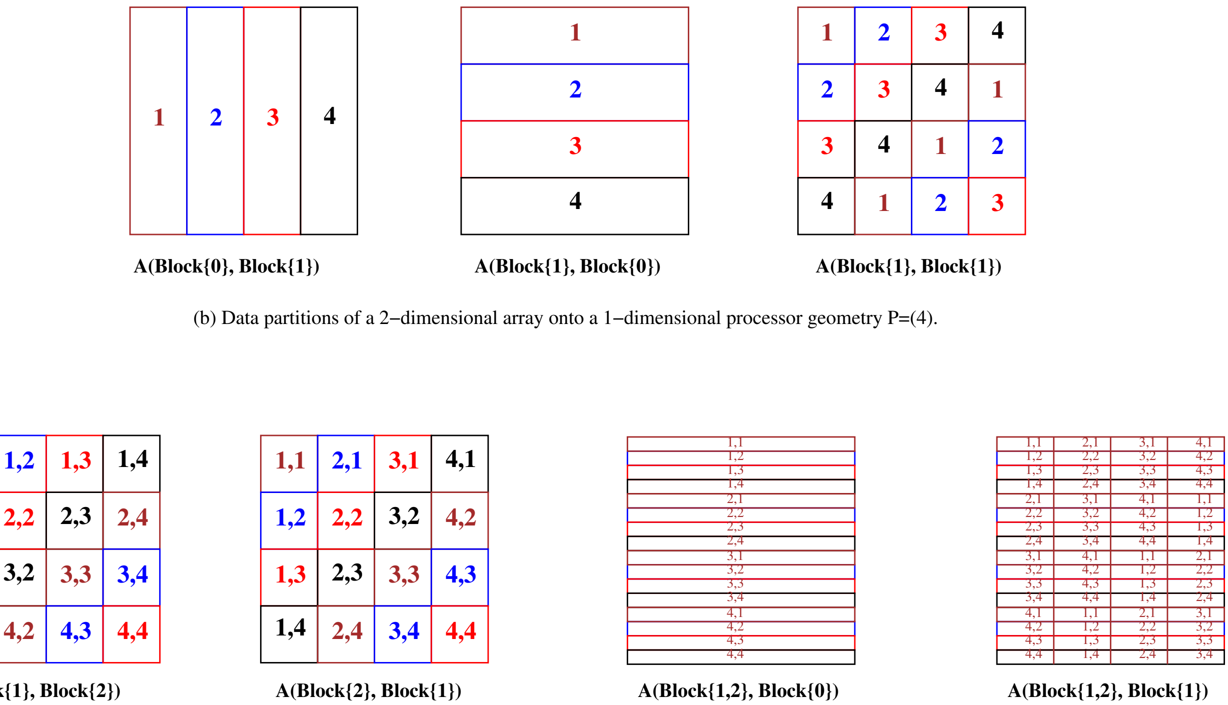
respectively. Finally, in addition to the nine distribution attributes, the $*$ attribute is used to indicate that the corresponding dimension of the array is not partitioned.

The mapping between the distributed dimensions of an array and the dimensions of the processor geometry defines how the array is tessellated. Each distributed dimension of an array can be mapped onto one or more of the dimensions of the processor geometry. This mapping is denoted by appending the distribution attribute associated with an array dimension with the appropriate dimensions of the processor geometry. For example, $\texttt{Block}\{1,2\}$ indicates that the array dimension is mapped onto dimensions 1 and 2 of the processor geometry. A mapping of an array dimension to the non-existent dimension `0` of the processor geometry is used to indicate that this dimension is not partitioned. Hence, $\texttt{Block}\{0\}$ implies that the associated dimension is not partitioned, even though it has been assigned a distribution attribute.

An array dimension may be associated with more than one dimension of the processor geometry. Similarly, more than one array dimension may be associated with the same dimension of the processor geometry. This can lead to interesting and useful array tessellations, some of which have not been used on distributed memory multiprocessors. Figure 1 shows some examples. The array partitioning shown in Figure 1(b) illustrates a tessellation that cannot be defined in HPF. The $(\texttt{Block}\{1\}, \texttt{Block}\{1\})$ partitioning of the array `A` results in both the array dimensions being tessellated and mapped onto a 1-dimensional processor geometry. The array is sliced into blocks of contiguous elements along both its dimensions because of the `Block` distribution attribute. The resultant blocks in the tessellated geometry are assigned to processors such that same processor is not assigned two blocks in the same column or row of the tessellated geometry. That is, a block $(\texttt{i}, \texttt{j})$ of the tessellated array maps to processor $((\texttt{i}-1)+(\texttt{j}-1)) \texttt{ mod P} + 1$ in the linear array of `P` processors. This partitioning of the array is called a *latin square* and is useful in alleviating contention.

When the partitioning of all the arrays is fixed for the entire program, the partitioning is called *static*. In some applications, it might be beneficial to have different array partitions across the loop nests in the program. In this case, the partitioning is referred to as *dynamic*. Dynamic partitioning requires *repartitioning* i.e. remapping the array from the original partition to the new transformed partition, which incurs the overhead of copying the data from the original data partitions to the new partition.

Computation partitioning is also specified in a manner similar to data partitioning. A distribution



(a) Data partitions of a 1−dimensional array onto a 1−dimensional processor geometry P=(4).

(b) Data partitions of a 2−dimensional array onto a 1−dimensional processor geometry P=(4).

(c) Data partitions of a 2−dimensional array onto a 2−dimensional processor geometry P=(4, 4).

Figure 1: Examples of data partitions

attribute and a mapping of a loop to the dimensions of the processor geometry define the computation partitioning of the loop. The distribution attributes specifies the partitioning of the loop and the processor dimension mapping specifies the assignment of the loop iterations to processors.

3 The Algorithm

The input to the algorithm is a program that consists of a sequence of loop nests, each containing both parallel and sequential loops. Loop nest that contains only sequential loops are not considered, and each loop nest is assumed to contribute equally to the execution time of the program. The output of the algorithm is: (1) a processor geometry; (2) a distribution attribute for each dimension of every array in the program; (3) a distribution attribute for each parallel loop in the program; (4) a mapping between distributed dimensions of arrays and dimensions of the processor geometry; and (5) a mapping between parallel loops and the dimensions of the processor geometry. The algorithm consists of 6 major steps:

1. Determining the dimensionality of the processor geometry.
2. Selecting array dimensions and parallel loops that will be partitioned.
3. Assigning a distribution attribute to each partitioned array dimension and to each parallel loop.
4. Mapping distributed array dimensions and parallel loops to the dimensions of the processor geometry.
5. Determining final data partitions and repartitioning points in the program.
6. Determining the number of processors in each dimension of the processor geometry.

The objective of the first 4 steps of the algorithm is to establish an affinity between the data and computation in the program. These steps rely solely on the structure of the program and on the data access patterns in loops, and are hence, machine-independent. The last 2 steps on the other hand use machine specific information such as cache, local and remote memory access costs, cache size, cost of synchronization, penalty for contention and the number of processors. Hence, they are machine dependent. Analytical and/or experimental models are used to determine the costs needed for these two steps.

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

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

Figure 2: Running example

The following sections describe the algorithm in detail. The program fragment depicted in Figure 2 will be used to illustrate the steps of the algorithm. The program consists of two loop nests `L1` and `L2`. The program accesses two 2-dimensional arrays `A` and `C` and a 1-dimensional array `C`. In loop nest `L1`, the inner i loop is parallel, while in the loop nest `L2`, the outer j loop is parallel. The array `A` is accessed along the rows by the parallel loop in the loop nest `L1`, and is accessed along the columns in the parallel loop in the loop nest `L2`.

3.1 Determining Data-Computation Affinity

This part of the algorithm derives affinity between the array data and the parallel loops in the program and it encompasses the first 4 steps of the algorithm. When this part terminates, every parallel loop and each dimension of the arrays accessed in the program will have: a distribution attribute, a mapping to processor dimension(s), and in the case of `BlockCyclic` attributes, a block size.

3.1.1 Determining Processor Geometry Dimensionality

The selection of the processor geometry dimensionality is driven by two conflicting goals. The first is the desire to maximize the number of parallel loops that indeed execute in parallel. This is necessary to exploit the large number of processors in SSMMs. The second is to minimize the number of dimensions of an array that get partitioned. This is necessary to reduce the fragmentation of arrays in shared memory.

A *reference matrix* [20] is constructed for every array access in the program. This matrix summarizes an array access by extracting the coefficients corresponding to the loop iterators (i.e., loop variables) in the array

$$\begin{array}{l c} & \begin{array}{cc} \mathtt{A}_1 & \mathtt{A}_2 \end{array} \\ \begin{array}{l} \text{i} \\ \text{j} \\ \text{Constant} \end{array} & \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \end{array}$$

Figure 3: Reference matrix for access A(i, j+1)

$$\begin{array}{l c c} & \begin{array}{cc} \mathtt{A}_1 & \mathtt{A}_2 \end{array} & VisibleParallelism \\ \begin{array}{l} \mathtt{L1} \\ \mathtt{L2} \end{array} & \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \begin{array}{c} 1 \\ 1 \end{array} \\ & & \\ \begin{array}{l} \mathtt{AND\_Mask} \\ \mathtt{OR\_Mask} \end{array} & \begin{array}{cc} 0 & 0 \\ 1 & 1 \end{array} & \end{array}$$

Figure 4: Parallelism Matrix

subscript expressions along every dimension. For example, the reference matrix for the access $\mathtt{A(i, j+1)}$ in loop L1 of the running example is shown in Figure 3; $\mathtt{A}_1$ and $\mathtt{A}_2$ correspond to the first and the second dimensions of `A` respectively. A *parallelism matrix* of an array identifies the dimensions of the array in which subscript expressions contain a parallel loop iterator. The reference matrices of the accesses to an array in the program are used to construct the parallelism matrix of the array. The parallelism matrix for an array is a matrix with rows equal to the number of loop nests, and columns equal the number of dimensions of the array. An entry corresponding to a dimension of an array and a loop nest is marked 1 if and only if that dimension has a non-zero entry in any of the reference matrices of the array for a parallel loop iterator in the loop nest. Hence, in the running example, the parallelism matrix for array `A` contains a 1 entry along $\mathtt{A}_1$ dimension in the loop nest `L1`. The parallelism matrix for array `A` is shown in Figure 4.

The `AND_Mask` is a result of `AND`ing the elements in the columns of the parallelism matrix, i.e. along the dimensions of the array. It indicates whether a particular dimension of the array has a subscript expression with a parallel loop iterator in every loop nest in the program. This dimension can be partitioned across the whole program. The number of non-zero elements in the `AND_Mask` indicates the number of such dimensions, and is the preferred dimensionality of the processor geometry for the array. However, it is not always the case that a dimension has an expression with a parallel loop index in every loop nest, and the `AND_Mask` is 0. In this case, the processor geometry selection is based on the number of parallel loops in each loop nest and *visible parallelism*. Visible parallelism is the number of dimensions of an array that can be partitioned in a loop nest. It is the sum of elements in a row of the parallelism matrix. On one hand, the number of distributable dimensions of the array should not exceed the visible parallelism. On the other hand, the number of distributed dimensions of an array should not exceed the number of parallel loops that enclose the array access— this could lead to idle processors. Consequently, the minimum of the number of parallel loops that enclose the array access and the visible parallelism is chosen as the dimensionality of the processor geometry for the array in that loop nest. The dimensionality of the processor geometry chosen most often (i.e. statistical mode) across all loop nests is chosen for the array. The dimensionality of the processor geometry most often chosen by the arrays is selected for the whole program.

The `OR_Mask` is the result of `OR`ing the elements along the columns. Each bit in the mask indicates the presence of at least one loop nest with a parallel loop iterator associated with the corresponding dimension of the array. If both the `AND_Mask` and the `OR_Mask` are 0, the array does not have any parallel loop iterators associated with the subscript expressions. Hence, the array does not contribute in the processor dimension selection.

For array `A`, in the running example, the `AND_Mask` is 0 and the `OR_Mask` is non zero. Hence, the dimensionality of the processor geometry is based on the visible parallelism. The visible parallelism is 1 across both the loop nests. Thus a 1-dimensional processor geometry is chosen. A 1-dimensional processor geometry is also chosen by arrays `B` and `C`. Consequently, a 1-dimensional processor geometry is chosen for the whole program.

The first goal in the choice of processor dimensionality is satisfied by using the `AND_Mask` in the selection; each 1 bit in the `AND_Mask` indicates the minimum number of array dimensions that can be partitioned without sacrificing parallelism across the whole program. The second goal in the choice of processor dimensionality is satisfied by using the visible parallelism; it indicates the maximum number of dimensions that can be partitioned retaining the parallelism in each loop nest.

3.1.2 The NUMA-CAG

A variant of the Component Alignment Graph [21], called the NUMA-CAG, is defined to establish the affinity between parallel loop iterations and the array partitions. The NUMA-CAG is an undirected 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 node. There is a loop node for each parallel loop in the program. Similarly, there is an array dimension node for each distributed dimension of each array in the program. In order to determine which dimensions of an array can be distributed, the **AND_Mask** and **OR_Mask** of the array are examined. If the number of 1's in the **AND_Mask** is greater than or equal to dimensionality of the processor geometry, then the array dimensions indicated by 1 in the **AND_Mask** are chosen as array dimension nodes. In this case, the array is sure to be partitioned only along the dimensions corresponding to 1's in the **AND_Mask**, dimensions of the array that are not partitioned are not included in the NUMA-CAG. 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 as the array dimension nodes. In this case, all the dimensions of the array that can be partitioned are added to the NUMA-CAG. Arrays in which both the **AND_Mask** and the **OR_Mask** are 0 are not selected in building the NUMA-CAG. These arrays are replicated to avoid contention.

An array dimension node has two sub-nodes, a *forward* sub-node and a *reverse* sub-node. There is an edge between a loop node and the forward sub-node if the coefficient in a reference matrix along the array dimension corresponding to the loop iterator is non-zero and positive. Similarly, there is an edge between a loop node and the reverse sub-node if the coefficient in a reference matrix along the array dimension corresponding to the loop iterator is negative. Two accesses of an array, one with positive and the other with negative coefficient corresponding to a parallel loop iterator in the reference matrices will result in two edges from the loop node to the array dimension node; one of the edges will be to the forward sub-node and the other will be to the reverse sub-node.

Figure 5 shows the NUMA-CAG for the running example. $\mathtt{L1}_\mathtt{i}$ and $\mathtt{L2}_\mathtt{j}$ (corresponding to the loop iterators i and j respectively) are the parallel loops in the program and form the loop nodes. Since the **AND_Mask** of array `A` is 0, both dimensions $\mathtt{A}_1$ and $\mathtt{A}_2$, are chosen as nodes in the NUMA-CAG based on the

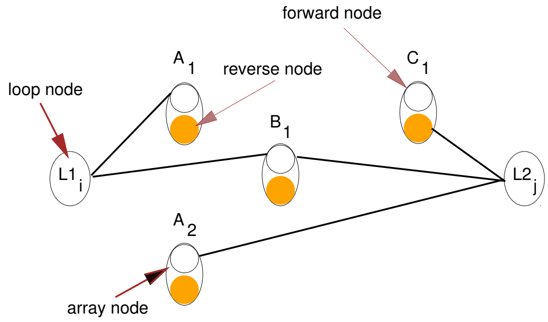


Figure 5: NUMA-CAG for the running example.

OR_Mask. Nodes $\mathtt{B}_1$ and $\mathtt{C}_1$ are chosen as nodes based on the AND_Mask of arrays B and C respectively. The first dimension of array A contains 1 in the reference matrix for the access in loop L1 corresponding to the parallel loop iterator i. Hence an edge connects the forward node of $\mathtt{A}_1$ with the loop node $\mathtt{L1}_\mathtt{i}$. Similarly, the second dimension of A contains 1 in the reference matrix for the access in loop L2 corresponding to the parallel loop iterator j. An edge connects the forward node of $\mathtt{A}_2$ with the loop node $\mathtt{L2}_\mathtt{j}$. The first dimension of C contains -1 in the reference matrix corresponding to the loop iterator j. Therefore, the loop node $\mathtt{L2}_\mathtt{j}$ is connected to the reverse node of $\mathtt{C}_1$.

3.1.3 Assigning Initial Distribution Attributes

Loop nodes are assigned initial distribution attributes based on the load balancing requirements of the corresponding parallel loop. Array dimension nodes, on the other hand, are assigned distribution attributes based on the locality requirements of the array accesses.

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. In such cases, proper partitioning of the iterations of the parallel loop is required to balance the load. The Cyclic and CyclicRCyclic partitioning attributes can be used to balance the load among the various processors. A loop node is assigned the CyclicRCyclic distribution attribute when the iteration space of the corresponding parallel loop is *triangular*, i.e. when the computations inside the loop vary with the index of the loop. When the iteration space of a parallel loop is non-triangular, but varies with an outer loop index, the loop node is assigned the Cyclic distribution attribute. Parallel loops that do not give rise to load imbalance, are assigned a $*$ attribute because they can be partitioned using any distribution attribute.

Elements of one array accessed in an iteration of a parallel loop must be collocated to enhance locality of access. For example, if $\mathtt{X(a*i)}$ and $\mathtt{X(a*i+c)}$ are accessed in an iteration of a parallel loop i, both the data elements must be assigned to the same processor such that a contiguous chunk of the array is allocated to a processor. When the subscript expression in a dimension of an array is a function of a single iterator, the `Block` distribution attribute meets the goal. However, when an array subscript expression is a function of multiple loop iterators, the distribution attribute of the array dimension node depends on the nesting level of the loops. Consider an array `X` that is accessed as $\mathtt{X(a*i+b*j)}$ in a loop nest where i (outer) and j (inner) are loop iterators. If the j loop is parallel and i is sequential, small segments of the array `X` are accessed by the iteration space of j and the outer sequential loop i controls a sweep along 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 favour outer loop parallelism.

The `Block` and `BlockCyclic` distribution attributes are assigned both to the forward and reverse sub-nodes of an array dimension node by examining the forward access and the reverse access of the array dimension separately. Arrays can be accessed in multiple loops, and can present conflicting requirements for the distribution attribute of a array dimension node. When a conflict occurs between `Block` and `BlockCyclic` attributes, the `BlockCyclic` attribute is used. The array dimension nodes without a `Block` or `BlockCyclic` distribution attribute are assigned a $*$ attribute.

In the running example, the parallel loop i in loop `L1` has a triangular iteration space. To balance the load, the loop node $\mathtt{L1_i}$ is assigned `CyclicRCyclic` distribution attribute. Load balancing is not an issue in the parallel loop j in loop `L2`. Hence, the loop node $\mathtt{L2_j}$ is assigned $*$ distribution attribute. The forward sub-node of array dimension node $\mathtt{B_1}$ is assigned a `Block` distribution attribute. Array dimension nodes $\mathtt{A_1}$, $\mathtt{A_2}$ and $\mathtt{C_1}$ are assigned $*$ distribution attribute.

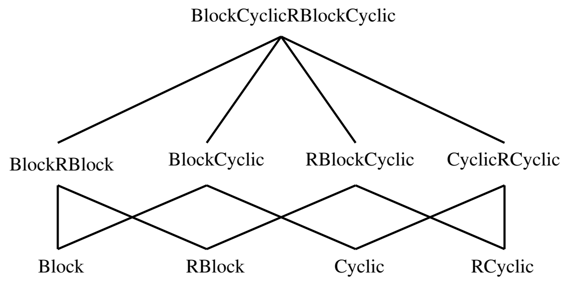


Figure 6: Conflict resolution graph.

3.1.4 Deriving Final Distribution Attributes

The initial distribution attributes do not ensure that parallel loops and array dimensions whose subscript expressions are a function of the parallel loop iterators are partitioned similarly. Hence, in this step of the algorithm the nodes in the NUMA-CAG until all nodes that are connected have the same distribution attribute. We refer to this as *consensus*.

When the distribution attributes of the connected nodes are not the same, a *conflict* is said to exist. Such a 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. A node in the conflict resolution graph itself can be its own least common ancestor. For example, the distribution attributes `Cyclic` and `Block` resolve to `BlockCyclic`, the distribution attributes `Cyclic` and `CyclicRCyclic` resolve to `CyclicRCyclic`, and the distribution attributes `Cyclic` and `BlockRBlock` resolve to `BlockCyclicRBlockCyclic`. A distribution attribute with the $*$ attribute resolves to the distribution attribute itself.

The distribution attribute of the forward node is the distribution attribute of the array dimension. Each array dimension node ensures that the distribution attribute of the reverse node is the complement of the distribution attribute of the forward node. The complement of the traditional attribute is the corresponding reverse distribution attribute, while the complement of the synthesized attribute is the same attribute. The forward and reverse node of the array were treated as separate nodes in the initial distribution directive assignment. Since the forward and reverse nodes can have different distribution directives, these directives can conflict and may not be the complement of each other. In this case, a resolution is made based on the conflict resolution graph. The array dimension node consisting of the forward node and the reverse node are then considered as one unit. The process of resolving conflicts is followed by all the array dimension nodes

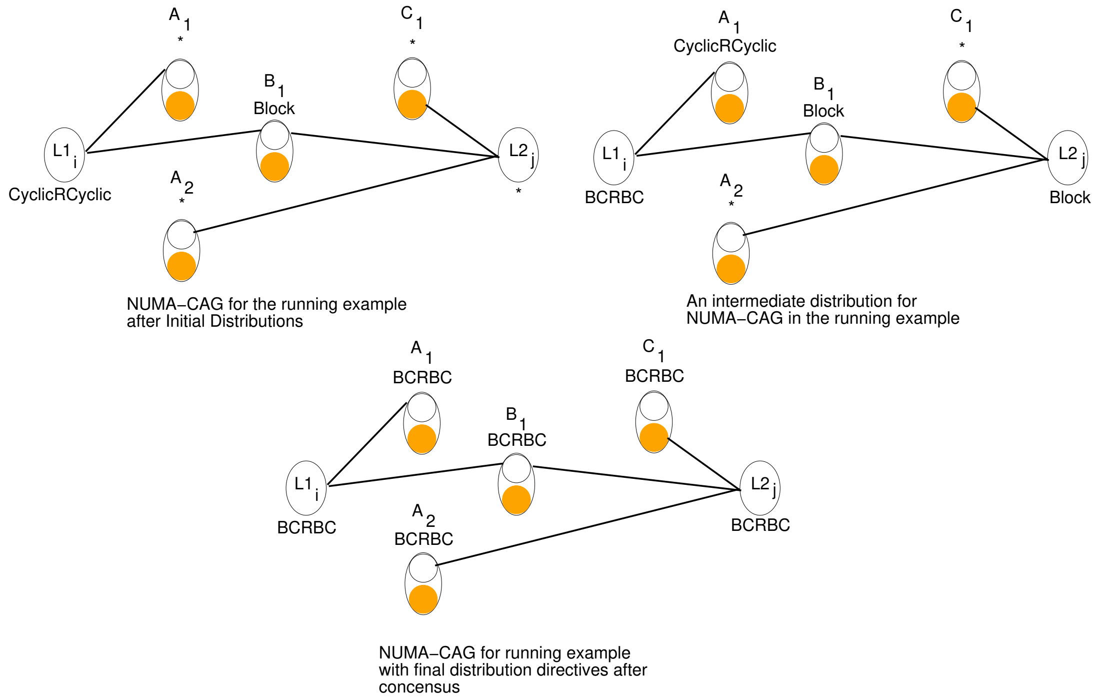


Figure 7: Deriving distribution attributes for the running example

and the loop nodes until all the nodes that are connected have the same distribution attribute.

Deriving data distribution attributes for the running example is shown in Figure 7. The node $\mathtt{L1_i}$ in NUMA-CAG for the program has `CyclicRCyclic` as its initial distribution attribute, whereas $\mathtt{B_1}$ has `Block` as its distribution attribute. All the other nodes, $\mathtt{L2_j}$, $\mathtt{A_1}$, $\mathtt{A_2}$ and $\mathtt{C_1}$ are assigned the $*$ distribution attribute. In an intermediate state shown, $\mathtt{A_1}$ takes on `CyclicRCyclic`, the distribution attribute of $\mathtt{L1_i}$. $\mathtt{L1_i}$ is forced to be `BlockCyclicRBlockCyclic`(`BCRBC`) resolving its distribution attribute `CyclicRcyclic` and the attribute of node $\mathtt{B_1}$ connected to it. $\mathtt{L2_j}$ takes the `Block` attribute of the node $\mathtt{B_1}$. When consensus is reached, all the nodes in the NUMA-CAG are partitioned using the `BlockCyclicRBlockCyclic` distribution attribute.

3.1.5 Processor Dimension Assignment

The nodes of NUMA-CAG must be assigned dimension(s) of the processor geometry to partition the computations and the arrays. The objective is to enhance locality by having the distribution attribute and the processor dimension of a loop node match that of an array dimension node connected to it. To simplify the mapping of the parallel loops to processors, the loop nodes are only assigned a single processor dimension. However, multiple dimensions of the processor geometry may be assigned to array dimension nodes to satisfy multiple locality requirements across the loops.

The algorithm assigns processor dimensions to the nodes by examining all the loop nodes in the NUMA-CAG. Loop nodes are sorted according to the nesting level of the parallel loop in each loop nest of the program to favour outer loop parallelism over inner loop parallelism. Each parallel loop in a loop nest is then assigned a unique processor dimension. The loop nests are examined one at a time in their lexical order in the program. A processor dimension is said to be *available* if it has not yet been assigned to any loop node in the loop nest. When none of the connected array dimension nodes have a processor dimension assigned to them, the first processor dimension that is available is assigned to the loop node. All the array dimension nodes that are connected to the loop node are then assigned the same processor dimension. If a connected array dimension node has a processor dimension, then the loop node is assigned the same processor dimension as the array dimension node. Since an array dimension node can have multiple processor dimension assignments, and also a loop node can be connected to multiple array dimension nodes, the loop node is assigned a processor dimension that is common to all the array dimension nodes. However, a common dimension may not always exist. In the case of conflict, the processor dimension that occurs the most in the array dimension nodes connected to the loop node is chosen to satisfy the locality requirements of most of the arrays. A loop node is assigned a non-existent processor dimension `0` when there are no available processor dimensions; however, this dimension is not assigned to the array dimension nodes connected to the loop node.

During processor dimension assignment, an array dimension node adds processor dimensions of connected loop nodes to the ones it already has been assigned. Hence, an array dimension may be partitioned across several processor dimensions. In addition, the same processor dimension can also be assigned to different array dimension nodes of an array. At the end of processor dimension assignment for all the loop nests in the program, all the array dimension nodes without a processor dimension are assigned the non-existent processor dimension `0`.

The parallel loop corresponding to the loop node that is mapped onto non-existent processor dimension is executed sequentially. Nodes with $*$ distribution attribute along with a valid processor dimension are assigned the `Block` distribution attribute and are executed in parallel.

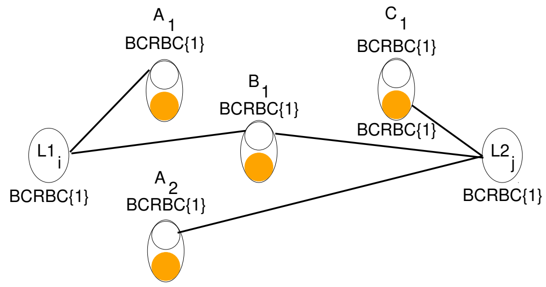


Figure 8: Processor dimension assignment for running example

Figure 8 shows the processor dimension assignments for the running example. In the example, the processor geometry is a linear grid. The processor dimension available for assignment is $\{1\}$. In the NUMA-CAG, loop node $\mathtt{L1_i}$ is assigned processor dimension 1. Consequently, the array dimension nodes $\mathtt{A_1}$ and $\mathtt{B_1}$ connected to $\mathtt{L1_i}$ are also assigned the processor dimension 1. In the second loop nest `L2`, the loop node $\mathtt{L2_j}$ is assigned the processor dimension 1 since the array dimension node $\mathtt{B_1}$ connected to it is assigned the dimension. Array dimension nodes $\mathtt{A_2}$ and $\mathtt{C_1}$ that are connected to $\mathtt{L2_j}$ are also assigned the processor dimension 1. Both the dimensions of the array `A` have a distribution attribute and a processor dimension associated with it satisfying the requirements of both loop nests in the program.

The computation partition in the program has been determined based on the parallelism available. Using the NUMA-CAG, all the dimensions of the array accessed in the body of the parallel loop with the loop index are also partitioned using the same distribution attribute as the data. Hence, the relationship between computation and data has been established and captured by the distribution attribute, block size and processor dimension attributes.

3.2 Determining Array Partitions

The derived partitionings establish locality relationship between data accessed and computation through static partitioning of arrays. By considering the requirements of the individual loop nests in the program, the performance of the program may be improved using dynamic partitioning of the arrays. Only the arrays that do not have a one-to-one correspondence between the array dimensions and processor dimensions, and the loop nests in which they are accessed need be examined. The processor dimension assignment was made for these arrays to satisfy multiple locality requirements in the different loop nests. Partitioning one array dimension along multiple dimensions of the processor geometry will result in non-local accesses for the array when the dimension is accessed using a parallel loop iterator because the parallel loop is partitioned along a single dimension of the processor geometry. Similarly, partitioning multiple dimensions of the array along a single dimension of the processor geometry also results in non-local accesses in the loop nests. Hence, a static partitioning of the array may not be the best. Arrays in which the one-to-one correspondence exists between the processor dimension and the array dimensions, locality is present across all the loop nests in the program. Hence, these arrays do not have to be examined.

The assignment of the processor dimension to the array dimension is explored in identifying the data partitions that are suitable for the loop nests. The permutations of the processor dimensions (including processor dimension `0`) along each dimension of the array is the possible set of data partitions for the array in the program. This set of possible data partitions forms the search space for each array. In the running example, the 2-dimensional array `A` is partitioned on a linear processor geometry using `A(BCRBC{1}, BCRBC{1})`, the set of all possible data partitions are `A(BCRBC{0}, BCRBC{0})`, `A(BCRBC{0}, BCRBC{1})`, `A(BCRBC{1}, BCRBC{0})` and `A(BCRBC{1}, BCRBC{1})`. `A(BCRBC{0}, BCRBC{0})` corresponds to the replication of the array. `A(BCRBC{1}, BCRBC{0})` is the data partition that satisfies the first loop nest in the program, while `A(BCRBC{1}, BCRBC{0})` is the data partition that satisfies the second loop nest in the program. `A(BCRBC{1}, BCRBC{1})` is the compromise data partition.

The cost of accessing the array in each loop nest is determined for each data partition in the search space. The cost estimation is dependent on the array ranges that are accessed from the cache, local or remote memory. The cost estimation heuristics used in determining the cache, local and remote access of an array is described in Section 3.4. If a data partition of an array in a loop nest results in contended accesses (e.g. when the subscript expression of a non-distributed dimension has a parallel loop iterator), the cost of executing the loop nest is scaled by *contention factor* to account for the contended accesses. Similarly, if the loops in a loop nest must be pipelined (i.e. wavefront), synchronization overhead incurred in pipelining the loop is added. In addition, repartitioning cost must be included when the data partition changes between the loop nests.

The set of possible data partitions is exhaustively considered to determine the data partitions that minimize the overall execution time and determine repartitioning points. The problem is clearly NP-Complete [14]. However, by only considering the arrays that lack one-to-one mapping between array dimensions and processor dimensions, the size of the search space is limited in most applications, allowing an exhausting search to be practical.

3.3 Processor Geometry Selection

The number of processors along each dimension of the processor geometry must be chosen when the dimension of the processor geometry is greater than 1. The selection of an appropriate processor geometry is driven by two objectives. The first is to minimize remote memory accesses. The second is to reduce the number of misses from the processors cache. These objectives can generate conflicting processor geometries for some applications.

The number of processors (`P`), the application is to be executed on, must equal the product of the number of processors along each dimension of the processor geometry. Hence, the factors of `P` are utilized in generating the candidate processor geometries. For example, when P=8, the candidate processor geometries are (1, 8), (2, 4), (4, 2) and (8, 1). The candidate processor geometries constitute the search space to be analyzed to determine the number of processors along each dimension of the processor geometry.

The costs of local and remote memory accesses for the program are determined by analyzing the data and computation partitions for every loop in the program, for each processor geometry. The number of cache misses are estimated by analyzing the amount of data required by each loop in the loop nest. The cost estimation heuristics are discussed in section 3.4. The geometry with the minimal cost estimate is chosen to be the processor geometry for executing the program.

3.4 Cost Models

Cost models are required to estimate: the array access cost and the cache misses in a loop nest. The array access cost comprises of the cost of accessing cache, local and remote memory, the cost of contention and the cost of synchronization.

The access ranges of arrays along a dimension are determined based on the array subscript expressions and the loop bounds. Multiple ranges along a dimension are merged together. Parts of this array range is accessed from the cache, local and remote memory.

The range of the array accessed from the cache is dependent on all the parent loop nests from which this loop nest can be reached. An array range is said to be *live* in a loop nest if it remains in the cache after the execution the loop nest. Hence, only if an array range is live in all the parent loop nests, the access will be from the cache. If the amount of cache required by the parent loop nest is greater than the cache size of the machine, the array range does not remain in the cache. Otherwise, the intersection of all the array ranges accessed by the parent loop nodes is the minimal region of the array that could be accessed from the cache.

For each dimension of the array, local access can be determined from the data partition of the array that is local and the range of the array accessed. Since part of the local access can be from the cache, local access is adjusted based on the array range in the cache. Similarly, for each dimension of the array, remote data access ranges can be determined. This analysis is similar to the regular section analysis used in distributed memory multiprocessors [19, 22]. The cost for accessing the array is the sum of accessing the data from the cache, local and remote memory modules using the cache, local and remote memory latencies.

Modeling contention is a difficult task and depends on many parameters like the size of memory buffers, network latency, routing protocols etc [23]. A simple factor is used to model contention. Contention factor is determined by measuring the performance "training set" loop nests on the multiprocessor. These programs are used measure the performance of a loop nest with and without contended local and accesses. Local contention factor is the ratio of the performance of the loop nest with contended local access and with uncontended local access. Similarly remote contention factor is determined. The average of the local and remote contention factors is used as the contention factor. This factor is used to scale the cost of array access when the array is contended for in a loop nest.

An analytical model is available for some multiprocessors to determine the cost of synchronization. For example, the cost of a barrier synchronization on the Convex SPP1000 [6] is approximately $(8 + P)$ μsec, where P is the number of synchronizing processors. In the absence of analytical models, the synchronization cost is determined by using "training set" loop nests.

Cache misses have to be determined only if the data required in a loop nest exceed the cache capacity.

When the data requirement exceeds the cache capacity in loop nests, data will not be retained across the loop nests and hence, only local and remote accesses are required in determining the cost of array accesses in a loop nest. However, if the cache is larger than the data required in the loop nests, cache capacity does not play a role in determining the processor geometry. The number of processors along each dimension of the processor geometry is chosen to minimize remote memory access.

The behaviour of the cache is hard to predict at compile time. Hence, based on the data requirements in the loop, to account for the cache misses, the cost is scaled by the *cache requirement factor*. In a parallel loop, as the number of processors increase the amount of data accessed decreases. Given a number of processors to partition a loop nest, the amount of data accessed in a loop nest is a constant. However, data required by each of the inner loops in the loop nest varies with the number of processors assigned to each parallel loop. The amount of data required for an iteration of the outer loops for each loop in a loop nest is determined. If the data accessed in the loop is less than the cache size, potentially the data will remain in the cache. Data in an iteration of a loop will be re-accessed (depending on the data access pattern) in the following iteration of the outer loop if the cache size is smaller than the data accessed. The number of retries that will occur is estimated as the ratio of the data accessed and the size of the cache. The product of the number of retries of all the loops in the program gives the cache requirement factor. The cost of memory access and cache requirement factor gives an approximate estimate of the cost of executing the loop nest.

4 Experimental Results

In this section we experimentally evaluate our data and computation partitioning algorithm. In particular, we give experimental results demonstrating the impact of the shared memory effects on derived data and computation partition. We also give experimental results showing the effectiveness of the algorithm in managing data compared to operating system policies.

We implemented the algorithm in a prototype compiler called *Jasmine*, which is developed at the University of Toronto. It consists of four major phases: parallelism detection, cache locality enhancement, memory locality enhancement and code generation. We use the Polaris compiler [8] from the University of Illinois as well as the KAP preprocessor from Kuck and Associates for the parallelism detection phase. The algorithm described in this paper constitutes the memory locality enhancement phase.

The compiler is used to automatically determine data and computation partitions for a number of applications. The resulting parallel code is executed on three multiprocessor platforms to measure performance; Hector [3], a Convex SPP1000 [6] and a KSR1 [5]. We present two sets of results. The first set shows the impact of shared memory on the type of data and computation partitions derived by the compiler. The second set gives the improvements in overall performance of applications from the use of data and computation partitioning to manage data as opposed to operating system policies.

4.1 Impact of Shared Memory

We use two benchmark applications to show the impact of shared memory in the choice of data partitions on a SSMM: the `ADI` stencil computation and `psinv` subroutine in `Multigrid` from the NAS suite of benchmarks.

The `Multigrid` application illustrates how data partitions must be cache-conscious. The application shows the importance of the selection of an appropriate processor geometry that increases cache affinity and minimizes remote memory accesses. `Multigrid` is a three dimensional solver calculating the potential field on a cubical grid. The subroutine `psinv` 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, using two 3-dimensional arrays `U` and `R`. The loop nest is fully parallel. To avoid false sharing the inner most loop is not parallelized. The presence of two parallel loops in the loop nest requires a two-dimensional processor geometry. The arrays are partitioned as $(*, \texttt{Block}\{1\}, \texttt{Block}\{2\})$ and the outermost parallel loop is associated with the 2nd dimension of the processor geometry and the inner parallel loop is associated with the 1st dimension of the processor geometry. With 16 processors, the compiler can choose one of the (16,1), (8,2), (4,4), (2,8) and (1,16) processor geometries.

Figure 9 shows the execution time of the application for various processor geometries on the KSR1 with 16 processors, normalized with respect to the (16,1) processor geometry. Equal number of processors in each dimension of the processor geometry (4,4) minimizes the remote memory accesses. The data from the successive iterations of the loop must be retained in the cache to decrease the number of cache misses[2].

[2]Misses due to cache capacity. We are not reducing misses due to the associativity of the cache.

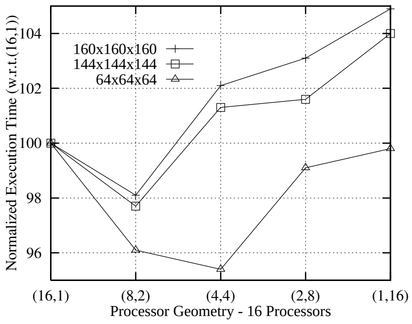


Figure 9: Normalized execution time of `Multigrid`.

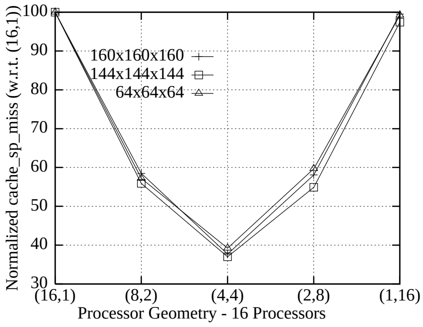


Figure 10: Remote memory access.

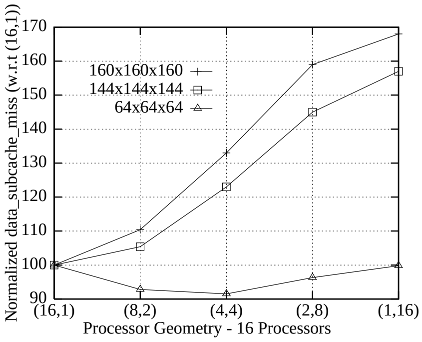


Figure 11: Cache Misses.

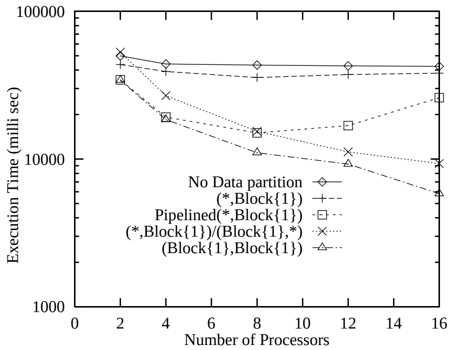


Figure 12: ADI performance (256x256).

Processor geometry (16,1) minimizes the amount of cache misses. When the data size is small (64x64x64), the data used by a processor fits into the 256k processor cache and hence, the processor geometry that minimizes remote memory access performs the best. However, as the data size increases (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 increase. The processor geometry (8,2) strikes a balance between the cost of remote memory access and the increased cache affinity, resulting in best overall performance, in spite of higher remote memory access cost. The cache statistics were collected *pmon* hardware monitor on the KSR1. Figure 10 shows the measured number of cache lines accessed from remote memory modules, normalized with respect to the processor geometry (16,1). Figure 11 shows the average measured number of cache misses from a processor cache, again normalized with respect to the processor geometry (16,1).

The Altering Direction Integration (`ADI`) is used to illustrate the choice of data partition that reduces contention and synchronization overhead. A single iteration of an outer sequentially iterated loop consists of a forward and a backward sweep phase along the columns of the three 2-dimensional arrays `A`, `B` and `X`, followed by another forward and backward sweep phase along the rows of the arrays [24].

The best data partition for `ADI` remains an issue of debate [24, 25]. Amarasinghe et al [25] partition the array as $(*, \texttt{Block}\{1\})$ satisfying the partition along the first phase and use pipelining to increase data locality in the second phase of the program. Kennedy et al [24] propose to partition the array using $(*, \texttt{Block}\{1\})$ in the first phase followed by repartitioning the array for the second phase using $(\texttt{Block}\{1\}, *)$. The former approach incurs synchronization cost while the latter incurs the cost due to data repartitioning.

The result of executing the `ADI` application on the Hector multiprocessor for a data size of 256x256 with various data partitions is shown in Figure 12. The compromise data partition $(\texttt{Block}\{1\}, \texttt{Block}\{1\})$ outperforms the other data partitioning strategies. Despite the increase in remote memory costs it does not incur the synchronization, repartitioning nor contention costs. The application performance using $(*, \texttt{Block}\{1\})$ data partition is poor due to contention in the second phase. The figure also indicates that the use of data partition improves the performance over the use of operating system policies to manage data (the no data partitions curve).

4.2 Overall Performance

We applied our compiler prototype to six application benchmarks that are representative of scientific applications to show the overall performance with the derived data partitions. They are: the `ADI` stencil computation, **`fft2d`**, **`mxm`**, **`Vpenta`** kernels from nasa7 in the SPEC92 floating-point benchmark suite, **`tomcatv`** also from the SPEC92 floating-point benchmark suite and **`fft3d`** from the ARCO suite of benchmarks. Table 1 summarizes the applications. The data partitions derived by the compiler for the major arrays in the applications on the multiprocessors is shown in Table 2. For all the applications, the processor geometry chosen was a linear array of processors unless otherwise indicated.

The performance of the applications on the Hector multiprocessor without data partitions, and the performance with the computation and data partitions derived using our algorithm is shown in Figure 13[3]. The performance of the applications without data partitions delegates data management to the operating system, which uses a "first-hit" page placement policy. The performance numbers reported without data

[3] `fft3d` was not executed on Hector because of system limitations.

Table 1: Application Characteristics

Name	Lines of Code	Loops/Parallel	Brief summary
ADI	35	8/4	Sweep along columns followed by sweep along the rows
fft2d	250	31/15	fast fourier butterfly computation on a 2-D array
mxm	50	10/4	matrix multiplication
vpenta	126	13/8	Inverts three pentadiagonal matrices
tomcatv	195	20/11	Mesh generation program
fft3d	70	9/6	FFT on a 3-D array

Table 2: Derived Data Partitions

Name	Hector	Convex	KSR
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},*)`
mxm	`A(*,*)`	`A(*,*)`	`A(*,*)`
	`C(*,Block{1})`	`C(*,Block{1})`	`C(*,Block{1})`
vpenta	`F(*,Block{1},*)`	`F(*,Block{1},*)`	`F(*,Block{1},*)`
tomcatv	`RX(Block{1},*)`	`RX(Block{1},*)`	`RX(Block{1},*)`
fft3d	`Z(*,Block{1},Block{2})`	`Z(*,Block{1},Block{2})`	`Z(*,Block{1},Block{2})`
	processor geometry (1,16)	(1, 30)	(1,16)

partitions are the better of the page placement policies for each application.

The performance of the application with data partitions outperforms the operating system data management. In the `ADI` application the compromise data partition eliminates contention that occurs in the operating system data management. Replication of a matrix was the key to improving the performance of the `mxm` application. The memory locality and contiguous allocation of data using data partitions in each processors memory module improves the performance of `vpenta`. Repartitioning of the arrays improves the performance of `fft2d` by eliminating contention. The performance of `tomcatv` is improved by exploiting the data reuse across the loop nests in the program. The overall performance of `tomcatv` is poor because of the overhead due to synchronization between the loop nests.

The performance of the applications on the Convex Exemplar multiprocessor without data partitions and outer loop parallelism, and the performance with the computation and data partitions derived using our algorithm is shown in Figure 14. The data partitions derived by the algorithm outperforms the operating system page placement policies on the Convex Exemplar multiprocessor as well. The only page placement policy on Convex is "round-robin" policy. Hence, unlike Hector, the improvement in performance is signif-

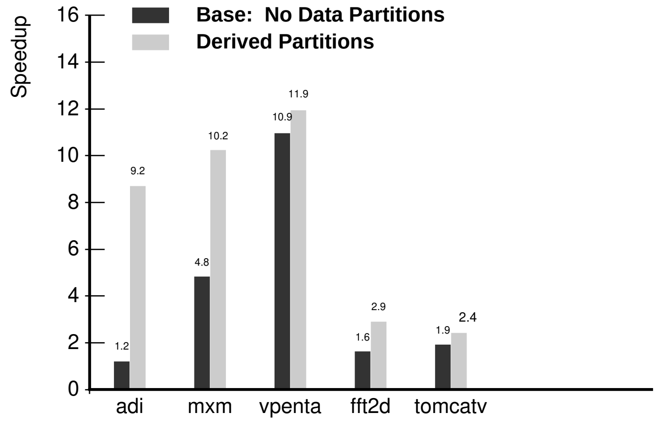


Figure 13: Performance on a 16-processor Hector.

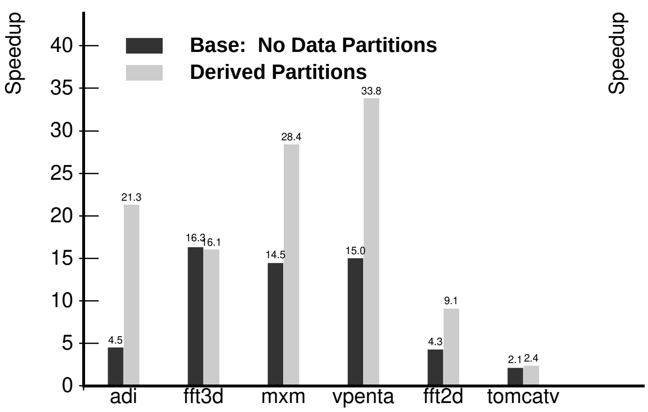


Figure 14: Performance on 30-processor Convex.

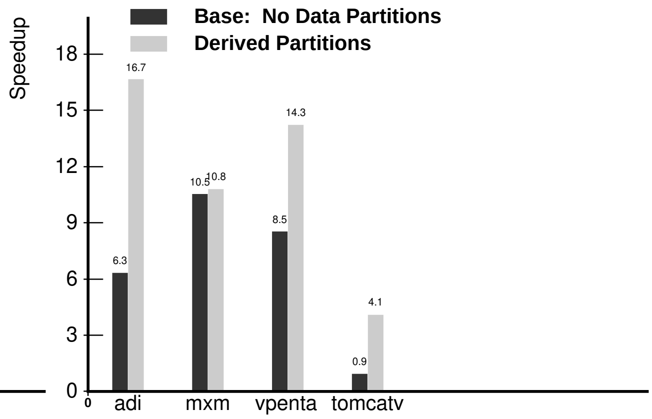


Figure 15: Performance on 16-processor KSR1.

icant using data partitions. The performance of all applications except `ADI` improved primarily because of the same data partitions derived for the Hector multiprocessor. The performance of ADI improved because pipelining was utilized to execute the loops in the second phase of the application. `Vpenta` exhibits superlinear speedup because of improved cache utilization with data partitions. `Fft3d` is the application where the performance of the application without data partitions is slightly better than the partition derived by the algorithm. The application is compute intensive and the remote or local memory access cost incurred in the program is small in comparison.

The performance of the applications on the KSR1 multiprocessor with outer loop parallelism and the computation partitions derived by the algorithm are shown in Figure 15[4]. The performance of `ADI` results in superlinear speedup because of repartitioning. Conclusions similar to the other two systems can be drawn.

[4] `fft2d` and `fft3d` were not executed on the KSR1 multiprocessor because of system limitations.

5 Related Work

Automatic approaches for deriving data and computation partitions on distributed memory multiprocessors have focussed on minimizing interprocessor communications. These approaches first derive a data partitioning then use the owner-computes rule [19] to determine a corresponding computation partitioning.

Li and Chen [26], Gupta and Banerjee [13, 27], and Zima et al. [15] follow the approach of finding the alignment constraints between different dimensions of the arrays and then derive a data partition that minimizes interprocessor communication. They derive only static data partition for the arrays. Bixby et al. [14, 24] formulate a 0-1 integer programming problem for deriving data partitions. Their approach relies on the assumption that a good data partition for the entire program can be found by analyzing the data partitions of smaller program segments. They minimize interprocessor communication using the "performance estimator" developed by Balasundaram et al. [28]. The program segments can have different data partitions. Data repartitioning is introduced in the program if it is beneficial. Garcia et al. [16] propose a technique to relate parallelism and data alignment. A static data partition is derived for the program using 0-1 integer programming.

Anderson and Lam [12] present an algebraic framework for both distributed memory multiprocessors and SSMMs, and determine data and computation partitions by minimizing communication across processors. Profiling is used to determine the loops that are most commonly executed. A static data partition is derived for the segments of the program that are most commonly executed. Data repartitioning is avoided in the commonly executed segments of the program. Data transformations are then applied so that the processors access contiguous data regions to reduce false sharing on SSMMs [29]. In their framework, the relationship between data and computation is strict in that the computation is performed where the data is allocated.

The approach presented in this paper is different in that it first derives suboptimal computation and data partitions by analyzing the data accesses in the program. Once the computation partition is determined, it improves the data partitions appropriate for the arrays in the program by re-evaluating the suboptimal partitions. The data partitions for each individual array is determined separately and can vary in the program.

6 Conclusion

Data partitioning is important on SSMMs because of the inherent non-uniformity of memory access. The shared address space of these machines requires an approach that considers contention, synchronization and cache affinity in deriving data and computation partitions. This paper has described an algorithm to automatically derive data and computation partitions for a program. The algorithm consists of two main parts. The first part derives a computation partitioning by establishing affinity between computation and data. The second part derives an appropriate data partitioning for each array separately using machine-specific costs. Experimental results indicate that the data and computation partitions derived by our algorithm improve the performance of the applications over operating system policies for data management. Experimental results also demonstrate the importance of taking shared memory effects into consideration; data and computation partitions that are derived to only minimize interprocessor communication are not necessarily the best data partitions.

Acknowledgments

We are grateful to the Center for Parallel Computing at the University of Michigan for providing time on the Convex Exemplar multiprocessor. We are especially thankful to Andrew J. Caird of the CPC for his invaluable help with the system.

References

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

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

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

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

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

[6] Convex Computer Corporation. *Convex Exemplar system overview*. Richardson, TX, USA, 1994.

[7] C. Polychronopolis et al. The structure of Parafrase-2: an advanced parallelizing compiler for C and Fortran. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 423–453. Pitman, London, 1990.

[8] W. Blume et al. Polaris: Improving the effectiveness of parallelizing compilers. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 141–154. Springer-Verlag, 1994.

[9] 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 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog.*, pages 122–132, 1991.

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

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

[12] J. Anderson and M. Lam. Global Optimization for Parallelism and Locality on Scalable Parallel Machines. In *Proc. of the SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 112–125, 1993.

[13] P. Banerjee et al. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, 1995.

[14] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. Technical report, Dept. Computer Science, Rice University, 1993.

[15] T. Fahringer, R. Blasko, and H. Zima. Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Proc. of Int'l Conf. on Supercomputing*, pages 347–356, 1992.

[16] 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.

[17] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. Technical report, NASA Langley Research Center, Hampton, October 1990.

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

[19] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed Machines. In *Proc. of Int'l Conf. on Supercomputing*, pages 1–14, 1992.

[20] U. Banerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic Publishers, 1993.

[21] J. Li and M. Chen. Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays. In *Proc. Frontiers of Massively Parallel Computation*, pages 424–433, 1990.

[22] V. Balasundaram. A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

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

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

[25] S. Amarasinghe et al. An overview of a compiler for scalable parallel machines. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 253–272. Springer-Verlag, 1993.

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

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

[28] V. Balasundaram et al. A static performance estimator to guide data partitioning decisions. In *Proc. of 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog.*, pages 213–223, 1991.

[29] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. of 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog.*, 1995.