Automatic Computation and Data Partitioning
on Scalable Shared Memory Multiprocessors

by

Sudarsan Tandri

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy.
Graduate Department of Computer Science.
in the University of Toronto

© Copyright by Sudarsan Tandri 1997
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

0-612-27739-9
Automatic Computation and Data Partitioning
on Scalable Shared Memory Multiprocessors

Sudarsan Tandri
Doctor of Philosophy. 1997
Department of Computer Science
University of Toronto

Abstract

Scalable shared memory multiprocessors are becoming increasingly popular platforms for high-performance scientific computing because they both scale to large numbers of processors and support the familiar shared memory abstraction. In order to improve application performance on these machines, it is essential to divide computation among processors and to place data carefully in the distributed shared memory. In particular, relying on only the native operating system page placement policies to manage data often results in poor performance of applications.

Computation and data partitioning is necessary for the management of computation and data on shared memory multiprocessors. The primary focus of this dissertation is the automatic derivation of computation and data partitions for regular scientific applications on scalable shared memory multiprocessors. Ideally, such partitions maximize parallelism and cache locality and minimize remote memory accesses, memory contention, synchronization overhead, and false sharing. In general, the problem of deriving optimal computation and data partitions is NP-hard. The complexity results from the combinatorics of possible computation and data partitions and the interdependence of the above optimization objectives. The degree to which these optimization objectives affect the performance of an application also varies, according to the characteristics of the application and the architecture of the scalable shared memory multiprocessor.

This dissertation presents a heuristic algorithm called the Computation and Data Partitioning (CDP) algorithm for deriving computation and data partitions on scalable shared memory multiprocessors. The CDP algorithm establishes affinity relationships between where data is located, based on array accesses in the program, and where computations are performed. These data-computation affinity relationships are used as a basis to determine the computation partitions for all the parallel loops and to determine static and/or
dynamic data partitions for all the arrays in the input program. The CDP algorithm takes into account shared memory effects in selecting the appropriate computation and data partitions. Experimental results from the implementation of the CDP algorithm in a prototype compiler demonstrate that the algorithm is computationally efficient and that it improves the performance of a suite of benchmark applications, compared to the native operating system page placement policies for data management.
Acknowledgements

First and foremost, I would like to thank my supervisor Tarek Abdelrahman for guiding me throughout my Ph.D. I deeply respect his ability to translate ideas into technical documents. My special thanks to Michael Stumm for channelizing my ideas onto a right track, and for his invaluable suggestions which has transformed the structure of this thesis. I thank Ken Sevcik, Songnian Zhou, and Charles Clarke for being my committee members and for providing valuable suggestions which improved the quality of this thesis. My thanks to Rudolf Eigenmann for being my external examiner, and for providing valuable comments for extending this research.

My greatest thanks to my friend, Dattatraya Kulkarni for his valuable suggestions, his “never give up” attitude, and for helping me endure the turbulent times during the culmination of this thesis. My thanks to Shankar and Ravindran, who were always there for me as good friends. I also thank Hui, Karim, and Danny for all the interesting discussions.

I thank the Hector team and the Hurricane team for providing the Hector multiprocessor. I am grateful to the Center for Parallel Computing at the University of Michigan for providing time on the Convex Exemplar multiprocessor. I am especially thankful to Andrew Caird of the CPC for his invaluable help with the system.

My heartfelt thanks to Vijaya, PSN, Papooz, and Jana for sharing my sorrows and jubilant times during the Ph.D. roller coaster which seemed to have many more troughs than peaks. I thank my brothers Shyam, Giri, Hari, and Balaji for encouraging me and for taking pride in all my endeavors. Finally and importantly, I wish to thank my wife, Radha, for her undaunted belief in my abilities and her patience.

My parents have been very supportive and it is to them that I dedicate this thesis.
To my parents
# Contents

1 Introduction ................................................. 1
   1.1 Computation and Data Partitioning ....................... 2
   1.2 Research Overview ...................................... 4
   1.3 Thesis Organization ..................................... 5

2 Background and Related Work ................................. 6
   2.1 The Problem ........................................... 6
   2.2 Notation for Computation and Data Partitions ............. 8
      2.2.1 Data Partitions .................................... 9
      2.2.2 Computation Partitions ............................. 11
   2.3 Related Work ........................................... 11
      2.3.1 Gupta and Banerjee ................................ 12
      2.3.2 Bixby, Kremer, and Kennedy ....................... 13
      2.3.3 Palermo and Banerjee .............................. 13
      2.3.4 Garcia et al. ..................................... 14
      2.3.5 Anderson and Lam ................................... 15
   2.4 Discussion ............................................ 16

3 New Computation and Data Partitions ....................... 17
   3.1 New Distribution Attributes .............................. 17
      3.1.1 Reverse Distribution Attributes ..................... 17
      3.1.2 Compound Distribution Attributes .................. 20
   3.2 New Data Partitions .................................... 21
      3.2.1 Many-to-One Mapping ............................... 22
4 Shared Memory Effects Influencing the Selection of Computation and Data Partitions

4.1 Performance Factors ............................................. 27
  4.1.1 Cache Affinity ............................................. 28
  4.1.2 Memory Contention ......................................... 28
  4.1.3 Synchronization Overheads ................................. 29
  4.1.4 False Sharing ............................................. 29

4.2 Experimental Evaluation ........................................ 30
  4.2.1 Cache-Conscious Computation and Data Partitions .......... 30

4.2.2 Contention- and Synchronization-Conscious Computation and Data Partitions ......................... 34

4.2.3 False-Sharing-Conscious Computation and Data Partitions .......... 39

4.3 Summary .................................................................. 40

5 The Computation and Data Partitioning Algorithm ......................... 42

5.1 The Input Program Model ........................................ 42

5.2 The Output of the CDP Algorithm ................................. 44

5.3 Outline of the CDP Algorithm .................................... 46

5.4 Processor Geometry Dimensionality ............................... 50

5.5 Data-Computation Affinity ........................................ 53
  5.5.1 The ALAG .................................................. 54

5.5.2 Deriving Distribution Attributes ............................. 56

5.5.3 Selecting Block Sizes ........................................ 63

5.5.4 Final Block Size Selection ................................... 65

5.5.5 Processor Geometry Dimension Assignment ................ 66

5.6 Dynamic Array Partitions ......................................... 68

5.7 Determining Final Computation Partitions ....................... 75

5.8 Processor Geometry Mapping ..................................... 77

5.9 Summary .................................................................. 78
4.1 Requirements for the \texttt{psinv} subroutine. ........................................ 33
4.2 Performance bottlenecks in various data and computation partitionings for ADI. .............................................................. 39
6.1 Parameters for performance estimation. ........................................... 86
6.2 Machine-dependent parameters for modelling multiprocessors. .............. 84
7.1 Application characteristics. ............................................................. 95
7.2 Multiprocessor characteristics. .......................................................... 97
7.3 Execution time of sequential application on 1 processor of the indicated machine in milli Seconds (Problem Size is shown in parenthesis). ............... 100
7.4 Data partitions derived by the CDP algorithm. ........................................ 101
7.5 Main machine characteristics that affect the choice of computation and data partitions on the chosen suite of benchmarks. ................................. 104
7.6 Impact of cache on data partition selection: performance of \texttt{psinv} application. 104
7.7 Measurements from data-computation affinity phase. ......................... 107
7.8 Measurements from dynamic array partition phase. ............................... 109
7.9 Measurements from processor geometry mapping phase. ....................... 109
7.10 Features of the algorithm applications exercise. .................................. 110
7.11 Bixby, Kremer, and Kennedy's approach. ........................................... 111
7.12 Anderson and Lam's approach. .......................................................... 113
7.13 Quality of solutions obtained by Anderson and Lam's algebraic solution on the Hector multiprocessor. .................................................. 113
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Scalable shared-memory multiprocessor architecture</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>An example of computation and data partitioning</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Choices for partitioning data and computation for an example program</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Examples of data partitions</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Examples of data partitions using reverse distribution attributes (P=4)</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Mapping functions for traditional distribution attributes</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Mapping functions for reverse distribution attributes</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Examples of data partitions using compound distribution attributes</td>
<td>20</td>
</tr>
<tr>
<td>3.5</td>
<td>Mapping functions for compound distribution attributes</td>
<td>21</td>
</tr>
<tr>
<td>3.6</td>
<td>Two array dimensions are mapped onto the same dimension of processor</td>
<td>22</td>
</tr>
<tr>
<td>3.7</td>
<td>Functions for many-to-one mapping</td>
<td>22</td>
</tr>
<tr>
<td>3.8</td>
<td>An array dimension mapped onto multiple processor geometry dimensions</td>
<td>24</td>
</tr>
<tr>
<td>3.9</td>
<td>Many-to-many mapping of array dimensions to processor geometry dimensions</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>The <code>psinv</code> subroutine</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Normalized execution time of <code>Multigrid</code> on KSR-1</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Observations from the PMON hardware monitor</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Outline of the ADI program</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Partitioning requirements of the different phases</td>
<td>35</td>
</tr>
<tr>
<td>4.6</td>
<td>(<code>Block{1}</code>) data partition with pipelined computation</td>
<td>36</td>
</tr>
<tr>
<td>4.7</td>
<td>(<code>Block{1}</code>) data partition with relaxed computation rule</td>
<td>37</td>
</tr>
<tr>
<td>4.8</td>
<td>(<code>Block{1}.Block{1}</code>) data partition with relaxed computation rule</td>
<td>38</td>
</tr>
</tbody>
</table>
4.9 ADI Performance on Hector (256x256). 39
4.10 Outline of the tred2 program. 40
4.11 Performance of tred2 on KSR-1. 41

5.1 Examples of loop nests targeted by the CDP algorithm. 43
5.2 A sample output. 44
5.3 Running example #1. 47
5.4 Running example #2. 48
5.5 The CDP Algorithm. 49
5.6 Algorithm: select-processor-geometry-dimensionality (Phase 1a). 52
5.7 The parallelism matrix for the arrays in running example #1. 52
5.8 The parallelism matrix for the arrays in running example #2. 53
5.9 Algorithm: establish-data-computation-affinity (Phase 1b). 55
5.10 The ALAG for running example #1. 56
5.11 The ALAG for running example #2. 56
5.12 Example showing the effectiveness of Cyclic distribution attribute. 58
5.13 Conflict resolution graph. 60
5.14 Algorithm: propagate-resolve. 61
5.15 ALAG with the final distribution attributes for running example #1. 62
5.16 ALAG with the final distribution attributes for running example #2. 62
5.17 Algorithm: select-block-size. 65
5.18 Processor geometry assignment for running example #1. 67
5.19 Processor geometry assignment for running example #2. 68
5.20 Algorithm: assign-proc-dim. 69
5.21 The cost matrix for array A on Hector multiprocessor in running example #1. 72
5.22 The cost matrix for array A on Convex multiprocessor in running example #1. 72
5.23 Reaching matrix for array A in running example #1. 73
5.24 Algorithm: finalize-data-partitions (Phase 2a). 74
5.25 Subprogram DFS_Repartition. 75
5.26 Search space of possible partitions for array A in running example #1. 76
5.27 Algorithm: finalize-computation-partitions (Phase 2b). 76
5.28 Algorithm: processor-geometry-mapping (Phase 3). 77
CHAPTER 1

Introduction

Scalable shared memory multiprocessors (SSMMs) are becoming increasingly popular platforms for high-performance scientific computing both because they scale to large numbers of processors and because they support the familiar shared memory abstraction. A SSMM is typically built by connecting clusters of processors with a scalable interconnection network, as shown in Figure 1.1. The shared memory is physically distributed into memory modules, one in each cluster. Memory access latency within a cluster is usually uniform, but is non-uniform across clusters. The extent to which memory access latency is non-uniform depends on the interconnection network. Remote memory access latency can range anywhere from one to two orders of magnitude higher than local memory access latency. Examples of SSMMs include the Stanford Dash [1] and Flash [2], the University of Toronto Hector [3] and NUMAchine [4], the KSR1 [5], the HP/Convex Exemplar [6], and the SGI Origin 2000 [7].

A common approach to improving the performance of applications on SSMMs has been to parallelize the applications manually, which is complex and tedious. Hence, compilers that transform sequential programs into efficient parallel programs are required, to reduce programming effort and to promote portability. These compilers are termed parallelizing compilers, and have been the focus of research for many years. Parallelization technology has advanced considerably in the past decade. Polaris [8], SUIF [9], KAP [10], and IBM's XL Fortran Compiler [11] are some of the research and commercial parallelizing compilers that exist today.

The primary focus of parallelizing compilers has been the detection of parallelism in nested loops, because loops are the main source of parallelism. Unfortunately, the identification of parallelism is not sufficient to result in good parallel performance on SSMMs. It is
also essential to place data carefully in shared memory, so as to mitigate the effects of the non-uniformity of memory access latencies, to improve cache locality, to reduce contention, and to minimize false sharing. Compilers for SSMMs have largely ignored issues related to data placement and have left them to the operating system. Operating systems distribute and manage data using page placement policies such as "first-hit" and "round-robin" [3, 12], which locate pages in the physically distributed shared memory as the pages are initially accessed. Unfortunately, these policies often lead to poor performance, because they are oblivious to the data access patterns of an application and manage data at too coarse a granularity: they often fail to enhance memory locality and, in addition, may result in memory contention and hot-spots [12, 13, 14].

In this thesis, we show that compilers can be used to manage both computation and data in regular (i.e., dense-matrix) scientific applications on SSMMs using the framework of computation and data partitioning. More specifically, we propose and evaluate compiler techniques that automatically derive computation and data partitioning for such applications to enhance parallelism and locality, and thus improve performance.

1.1 Computation and Data Partitioning

Computation partitioning maps the iterations of nested loops onto processors of a multiprocessor. Similarly, data partitioning maps array data onto the memory modules of the multiprocessor. Figure 1.2 illustrates a possible computation and data partitioning for the
Following loop on a 4-processor SSMM:

\[
\text{forall } i = 1..100 \\
A(i) = i \\
\text{end for}
\]

The 100 iterations of the loop are divided among the four processors such that each processor executes 25 consecutive iterations. The 100 elements in array A are mapped onto the two memory modules such that each module contains 2 consecutive segments of 25 array elements, as shown in Figure 1.2.

It is crucial that the compiler choose appropriate computation and data partitions. The partitioning shown in Figure 1.2 has good memory locality, since each iteration of the loop accesses array elements that are mapped onto the memory module closest to the processor executing the iteration. In other words, the execution of the loop does not result in any remote memory accesses. Consider a scenario wherein the computation partitioning remains the same, but array elements 1..50 are mapped onto memory module 2, and array elements 51..100 are mapped onto memory module 1. With this computation and data partitioning, each iteration of the loop must access array elements remotely.
It is also important to choose a computation partitioning that balances the workload on the processors. For example, the computation partitioning shown in Figure 1.2 maps the iterations so that the workload on the processors is balanced. Consider a scenario wherein the data partitioning remains the same as shown, but the iterations are mapped (rather artificially) such that processor 1 executes iterations 1..10, processor 2 executes iterations 11..50, processor 3 executes iterations 51..60, and processor 4 executes the remaining 40 iterations. This partitioning of the iterations would result in poor performance, since the workload on the processors is unbalanced. In general, computation and data partitionings that are compatible with respect to data access patterns and workload distribution should be chosen in order to achieve good locality of data accesses and balance of workload.

In addition to improving memory locality and balancing the workload, the chosen computation and data partitions must also enhance cache locality, reduce contention, minimize synchronization overhead, and reduce false sharing.

Data partitioning is used on distributed memory multiprocessors to map array data onto the separate address spaces. The computation partitioning on these machines is implicit, because it is indirectly specified using a computation rule such as the owner-computes rule [15, 16]. In the past decade, many researchers have focused on compiler techniques that automatically derive data partitionings for distributed memory multiprocessors. This thesis extends their work in three ways: (i) we use data partitioning to enhance memory locality on SSMMs, (ii) we do not use a computation rule, but map computations explicitly onto processors, and (iii) we automatically derive both computation and data partitions that take into account shared memory effects. In contrast, data partitions on distributed memory multiprocessors are derived with the main objective of minimizing interprocessor communication.

1.2 Research Overview

The primary research focus of this thesis is to develop a compiler framework targeted to improve the performance of dense matrix scientific applications (i.e., applications in which the array subscript expressions are affine functions of the loop index variables, and the array and loop bounds are known) on SSMMs. An outcome of this thesis is a heuristic framework implemented in the Jasmine compiler [17] prototype, which automatically derives
computation and data partitions. The main contributions of this thesis are as follows.

1. We show that shared memory effects, such as cache affinity, memory contention, synchronization overhead, and false sharing, must be considered while deriving computation and data partitionings. Indeed, we show that the data partitions selected by existing techniques on distributed memory multiprocessors may not always be the best choice for a SSMM, although both architectures have a physically distributed memory.

2. We present a heuristic algorithm called the Computation and Data Partitioning (CDP) algorithm, which derives computation and data partitions for SSMMs. We show that the CDP algorithm is computationally efficient, while taking into account shared memory effects that affect the performance of applications. Experimental evaluation of the CDP algorithm on three SSMMs demonstrates the effectiveness of the algorithm in improving the performance of a suite of benchmark applications, compared to the native operating system for data management.

1.3 Thesis Organization

The definition of the computation and data partitioning problem and an overview of existing techniques to automatically derive data partitions on distributed memory multiprocessors are presented in Chapter 2. New computation and data partitions that we have introduced are presented in Chapter 3. These new partitions facilitate the establishment of data-computation affinity, which is the basis of the CDP algorithm. The performance factors that affect the selection of computation and data partitions on SSMMs are identified in Chapter 4. The CDP algorithm for heuristically deriving computation and data partitions for SSMMs is presented in Chapter 5. The cost model used in the CDP algorithm to statically evaluate the effectiveness of candidate computation and data partitions is presented in Chapter 6. The experimental evaluation of the CDP algorithm is presented in Chapter 7. Concluding remarks and directions for future research are presented in Chapter 8.
Chapter 2

Background and Related Work

In this chapter, we describe the background and related work with respect to the problem of automatically deriving computation and data partitions. This chapter is organized as follows. First, we define the problem of deriving optimal computation and data partitions, and illustrate the computational complexity involved with an example. Second, we describe the notation used to specify computation and data partitions. Finally, we review existing techniques for deriving computation and data partitions, principally on distributed memory multiprocessors.

2.1 The Problem

Data partitioning of an array divides the array into sub-arrays and maps these sub-arrays onto processors. Similarly, computation partitioning of a loop nest divides the iteration space of the loop and assigns the iterations to the processors.

Computation and data partitions must be derived so as to minimize the execution time of the application on a given multiprocessor. The problem of selecting optimal computation and data partitions (i.e., computation and data partitions that minimize interprocessor communication, minimize load imbalance, maximize cache locality, minimize contention, minimize synchronization overhead, and minimize false sharing) is computationally hard due to the combinatorics of possible computation and data partitions, given the dimensionality and the number of arrays, and the number of loops in each loop nest [18, 19]. The complexity of the problem can be illustrated using the program shown in Figure 2.1. It contains two loop

\[\text{footnote}{1}\]

1 Although sub-arrays are assigned to memory modules, in this thesis we use the phrase “assigning sub-arrays to a processor” to mean “assigning sub-arrays to the memory module which is close to the processor.” This phrasing implicitly associates a processor with the memory module close to the processor and simplifies the presentation.
nests accessing three 3-dimensional arrays, A, B, and C. The set of possible combinations of loops that can be partitioned in each loop nest is indicated to the right of each nest. The index of each loop that may be partitioned is shown to indicate that the corresponding loop is partitioned. For example, \( i \& j \) in the table indicates a computation partition that divides loops with iterators \( i \) and \( j \). The number of possible choices for computation partitioning is \( 2^d - 1 \), where \( d \) is the depth of the loop nest. Thus, there are seven choices for partitioning each of the two loop nests.

Possible data partitions for each array in the example are shown below the loop nests. The number of possible partitions for one array are \( 2^{d'} - 1 \), where \( d' \) is the dimensionality of the array. In this case, each array has seven possible partitioning choices.

The computation and data partitions must also be mapped onto processors. This is typically done by choosing a processor geometry, which is an \( n \)-dimensional Cartesian grid of processors, \( (p_1, p_2, \ldots, p_n) \). In this notation for processor geometry, \( p_i \) denotes the number of processors in the \( i^{th} \) dimension, and \( p_1 \times p_2 \times \cdots \times p_n = P \) is the total number of processors. The dimensionality of the processor geometry, \( n \), is usually limited by the
highest array dimensionality. In our example, the processors can be organized as a 1-dimensional, 2-dimensional, or 3-dimensional processor geometries. Clearly, there is only one possible 1-dimensional processor geometry for a given number of processors. With \( P = 16 \), there are 5 possible 2-dimensional processor geometries \((p_1, p_2)\): \((16, 1)\), \((8, 2)\), \((4, 4)\), \((2, 8)\) and \((1, 16)\). Similarly, 16 processors can be organized in 15 different 3-dimensional processor geometries.

The number of possible partitions for an array can be computed using the number of dimensions of the array, \( d \), and the number of processors. Assuming that the number of processors is a power of some prime number \( p \), i.e., \( P = p^k \), the number of possible partitions, \( n \), for a \( d \)-dimensional array is \( \binom{k+d-1}{d-1} \) [20]. Note that this formula can also be used to obtain the number of partitions for loops in a loop nest with a depth of \( d \).

The number of choices for partitioning arrays and iteration spaces increases exponentially as the number of arrays and the number of loop nests in the program increases. The search space that must be considered when there are \( a \) arrays accessed in each of \( l \) loop nests in the program is \( (n^a \times n)^l = n^{(a+1)\times l} = \binom{k+d-1}{d-1}^{(a+1)\times l} \). In our example, with 16 processors, one must consider over 2.5 billion \((15^{4\times2})\) possible computation and data partitions to derive an optimal solution by an exhaustive search. The number of choices grows to nearly 400 billion \((28^{4\times2})\) when we have 64 processors.\(^2\)

The problem of selecting optimal computation and data partitions is further complicated by the replication of arrays, the frequency of execution of loop nests, and the presence of selection statements, such as if-then-else, that alter the execution frequency of loop nests. Hence, techniques for selecting computation and data partitions are heuristic, because of the exponential complexity of the problem. These techniques reduce the complexity by making simplifying assumptions and/or by employing efficient search techniques.

### 2.2 Notation for Computation and Data Partitions

Computation and data partitions are specified using textual notations for convenience. Languages such as HPF include such a notation as a part of language specification [16]. In this section, we describe these textual notations. The computation and data partitioning

\(^2\)Note that we have only considered the choice of loops and array dimensions for partitioning and not the manner in which they are divided. Considering the manner in which the loops and array dimensions are divided further increases the number of choices for computation and data partitioning.
is specified as a mapping from a grid of iterations and a grid of data elements onto the Cartesian grid of processors.

### 2.2.1 Data Partitions

The notation for the data partitioning of an array has two components for each array dimension: a distribution attribute, and a processor dimension assignment. A distribution attribute is assigned to each array dimension, and it specifies how an array dimension should be divided. There are three distribution attributes typically used on distributed memory multiprocessors: Block, Cyclic, and BlockCyclic. In this dissertation, we refer to these three attributes as the traditional distribution attributes. 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 to processors in a round-robin fashion. The BlockCyclic attribute specifies that \( b \) contiguous array elements in a dimension must be mapped onto the same processor, followed by the next \( b \) elements to the next processor, in a round-robin fashion; \( b \) is called the block size. Thus, the BlockCyclic distribution attribute with a block size of 1 and the Cyclic distribution attribute are equivalent. A special distribution attribute, denoted by \(*\), is used to indicate that an array dimension is not partitioned at all.

The processor dimension assignment maps array dimensions onto the dimensions of the processor geometry. This assignment is specified by using a one-to-one mapping between the dimensions of arrays assigned distribution attributes and the dimensions of the processor geometry. Thus, in conjunction with the distribution attribute, the processor dimension assignment determines the number of subdivisions into which an array dimension must be divided.

In this dissertation, we denote the processor dimension assignment by appending a dimension of the processor geometry to the distribution attribute. For example, \( A(\text{Block}\{1\}) \) indicates that the single dimension of array \( A \) is mapped onto dimension 1 of the processor geometry, and the array dimension is divided into \( p_1 \) blocks, where \( p_1 \) is the number of processors in the first dimension of the processor geometry. It is important to note that the distribution attribute not only specifies the manner in which the array dimension should be divided but also imposes an implicit ordering for mapping the sub-arrays: the first sub-array is mapped onto the first processor, the second sub-array is mapped onto the second
When an array dimension is mapped onto a non-existent dimension of the processor geometry, i.e., dimension 0, then the notation indicates that the array dimension is not partitioned. Hence, such a mapping is equivalent to the * attribute. For example, $A(\text{Block}\{0\})$ is equivalent to $A(*)$.

Figure 2.2 illustrates some common data partitions of a 2-dimensional array. Figure 2.2(a) illustrates the case in which a 2-dimensional array, $A$, is partitioned onto a 1-dimensional processor geometry containing 4 processors. Both dimensions of $A$ have the Block distribution attribute. However, the first dimension is associated with dimension 0 of the processor geometry, and the second dimension is associated with dimension 1 of the processor geometry. Hence, the first dimension is not partitioned, and the second dimension is divided into 4 blocks of columns. Figure 2.2(b) illustrates the case where array $A$ is divided into 4 blocks of rows and mapped onto a 1-dimensional processor geometry. A partitioning of array $A$ onto a 2-dimensional processor geometry of 16 processors, with 4 processors along each dimension, is shown in Figure 2.2(c). The 2-dimensional array is partitioned along both the dimensions into 4 blocks for a total of 16 blocks. Each of the 16 processors is assigned one block of the array.

An important locality optimization used in practice is to replicate arrays and sub-arrays on processors. Replication of arrays and sub-arrays can be specified using the above notation for data partitions. For example, the $A(\text{Block}\{1\})$ partitioning of an array $A$ onto a 2-dimensional processor geometry will result in the array being divided and mapped onto the first dimension, and the array being replicated along the second dimension of the processor geometry. When the number of array dimensions with a mapping is less than the dimensionality of the processor geometry, then an array is replicated along the dimensions of the processor geometry that are not associated with any array dimension.
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 repartitioning of arrays and hence incurs run-time overhead.

2.2.2 Computation Partitions

Computation partitions can be explicitly specified using a notation similar to the notation for data partitions. A distribution attribute associated with a loop specifies the division of loop iterations into subsets of iterations, and a processor dimension assignment specifies how these subsets are assigned to processors.

On distributed memory multiprocessors, computation partitioning is implicit and is specified indirectly, using a computation rule such as the owner-computes rule [15, 16]. The processor that has a data partition assigned to its memory is said to own the data in the partition. With the owner-computes rule, the processor that owns a data item is mapped with all the computations that update/write the data item. Using a computation rule simplifies the compiler's task of generating the messages required on a distributed memory multiprocessor [15]. In contrast, we explicitly use a computation partition, whether the partition adheres to the owner-computes rule or not.

It is interesting to note that static loop scheduling on shared memory multiprocessors can be viewed as a restricted form of computation partitioning. Loop scheduling only partitions parallel loops, while computation partitioning can partition sequential loops as well: computation partitioning of a sequential loop corresponds to a wavefront or pipelined scheduling of loop iterations [19]. In addition, loop scheduling is restricted to a linear array of processors to map parallel loop iterations, while computation partitioning maps the iterations of a loop nest (consisting of many loops) onto an n-dimensional processor geometry.

2.3 Related Work

In this section, we summarize heuristic algorithms developed in the past decade for deriving data partitions on distributed memory multiprocessors. These algorithms require
compile-time performance estimation in order to evaluate the effectiveness of candidate data partitions. Hence, compile-time performance estimation has also been an active area of research [41, 31].

2.3.1 Gupta and Banerjee

Gupta and Banerjee develop an algorithm to determine static data partitions on distributed memory multiprocessors [21, 22]. Their technique is developed as part of the Parafrase-2 restructuring compiler [23]; the compiler accepts a Fortran-77 program and generates a SPMD program with explicit communications.

The algorithm analyzes array references in each statement in the program to determine the array elements that must be allocated together. For the purpose of representing array references in the program, Gupta and Banerjee extend Li and Chen's Component Affinity Graph (CAG) [24, 25]. The CAG is a weighted undirected graph, in which each dimension of every array in the program is a node. There is an edge between two nodes if the subscript expressions of corresponding dimensions of the arrays are affine functions of same loop iterator. Gupta and Banerjee assign each edge a weight, which corresponds to the communication penalty incurred if the nodes connected by the edge are not aligned, i.e., are not ultimately mapped to the same dimension of the processor geometry. These weights are estimated based on the analysis of communication in the program with an arbitrary alignment other than the one suggested by the edge.

Gupta and Banerjee use a heuristic algorithm to partition the CAG into disjoint sets of nodes, such that the sum of weights of edges between nodes in different sets is minimized, thus minimizing the cost of interprocessor communication. Either a Cyclic or Block distribution attribute is chosen to partition array dimensions, based on the relative merit of each attribute.

A 2-dimensional processor geometry is used to partition the arrays to minimize the search space of possible processor geometries. The number of processors onto which the arrays are mapped is selected using an exhaustive search of possible processor geometries.

This was the first algorithm to present a complete solution for deriving static data partitions for a program. As Bixby et al. [26] and Anderson and Lam [19] have shown, however, selecting only static data partitions may not be sufficient, and sometimes it is necessary to repartition the arrays to achieve good performance.
2.3.2 Bixby, Kremer, and Kennedy

Bixby et al. formulate the problem of deriving data partitions on distributed memory multiprocessors as a 0-1 integer programming problem [20, 26, 27]. Their basic approach is to partition a program into phases, each consisting of one or more loop nests. They determine partitions for arrays in each phase and allow the arrays to be repartitioned only between phases.

The alignment constraints in each phase are represented by a separate CAG. As in Gupta and Banerjee [21]'s approach, the edges of the CAG are assigned a weight that indicates the communication penalty for not aligning the nodes connected by the edge. The CAG is partitioned into disjoint sets, using 0-1 integer programming, to minimize the sum of weights of edges that connect nodes in different sets. The result is the alignment of arrays with respect to each other in each phase. This alignment is captured by using a template. Candidate data partitions to be evaluated are obtained by partitioning each of the dimensions of the template, using Block and Cyclic distribution attributes.

The dimensionality of the processor geometry is chosen according to the dimensionality of the template. The candidate partitions, along with possible processor geometries, are used to derive the global static/dynamic data partitions for the program, also by the use of a similar 0-1 integer programming formulation.

Although the 0-1 integer programming formulation suggests that an optimal solution is obtained for the problem, Bixby et al. reduce the search space through their problem formulation. For example, they do not consider the replication of arrays, and their prototype implementation assumes a single dimensional processor geometry. As a result, only a subset of the data partitions are evaluated. Nonetheless, the computational complexity of their approach is very high [20].

2.3.3 Palermo and Banerjee

Palermo and Banerjee extend Gupta and Banerjee's algorithm in order to derive dynamic data partitions on distributed memory multiprocessors [28]. Similar to the approach of Bixby et al., they decompose the program into phases and assume that data repartitioning is beneficial only between phases. They derive the phases recursively. Initially, the entire program is viewed as a single phase, and a static data partition is determined for the pro-
gram using Gupta and Banerjee’s algorithm. They use a weighted directed graph, called the 
communication graph, to represent data flow between statements. The nodes of the graph 
correspond to the statements, and edges correspond to the data flow between statements. 
The incoming edges of each node are assigned weights that correspond to the cost of comm-
unication using the derived static data partitioning. The graph is partitioned into two, 
using a maximal-cut algorithm. The program is divided into two phases that correspond 
to the partitions if and only if the sum of the execution times of the phases with their best 
static partitions is less than the execution time of the original program with the original 
static partitions. This process of dividing the program into phases is recursively applied to 
the derived phases until no more benefit can be obtained.

The cost of array redistribution is then considered. For this purpose, Palermo and 
Banerjee analyze the hierarchy of phases obtained during the recursive decomposition of 
the program. They use an acyclic graph, called the phase transition graph, in which the 

nodes represent phases and the edges represent data flow between phases. The nodes 
are assigned the cost of execution of the corresponding phases with their best static data 
partitions, while the edges are assigned a weight that is the cost of repartitioning between 
the phases. The shortest path in the phase transition graph yields the data partitions for 
the program.

Although the formulation of the problem is elegant, the algorithm relies on static par-
titions in each given phase, obtained using Gupta and Banerjee’s algorithm, which obtains 
the communication costs for partition selection using arbitrary alignments.

2.3.4 Garcia et al.

Garcia et al. propose a method to derive data partitions for distributed memory multipro-
cessors using a 0-1 integer programming formulation similar to that of Bixby et al. [29]. The 
authors use a variant of the CAG called the Communication Parallelism Graph (CPG). As 
in a CAG, nodes in a CPG correspond to the dimensions of arrays in the program. However, 
the CPG has edges corresponding to all possible alignments between nodes. In contrast 
to a CAG, which uses communication penalties as edge weights, the edges in CPG have 
a weight that represents the benefit of aligning two dimensions. In addition, the nodes in 
the CPG that correspond to array dimensions having a parallel loop index in their sub-
script expressions are connected by a hyperedge. The weight of a hyperedge represents the
execution time that is saved when the loop is parallelized.

Similar to the partitioning of CAGs, the CPG of the program is partitioned into disjoint sets. However, the partitions of the CPG are such that the sum of the weights of edges between nodes in different sets is maximized, and the sum of the weights of hyperedges that connect nodes in different sets is minimized. Integer programming is used to partition the CPG.

One limitation of this approach is that the processor geometry is limited to a linear array of processors. In addition, the algorithm derives only static data partitions, in order to limit computational effort.

2.3.5 Anderson and Lam

Anderson and Lam present an algebraic framework for determining data and computation partitions for both distributed memory multiprocessors and SSMMs [19, 30]. The input consists of a program with a sequence of loop nests in which parallelism has been moved to outer nesting levels using unimodular transformations.

The algorithm has three main steps. In the first step, the algorithm determines communication-free partitions for each of the loop nests in isolation. For this purpose, Anderson and Lam formulate an algebraic representation of the relationship between computation and data, and derive static communication-free data and computation partitions. This formulation ensures that computation is implicitly performed where the data resides. They then use an iterative algorithm to derive static data partitions, considering all the loop nests in the program. In doing so, they trade off extra degrees of parallelism that may be available in loops to avoid communication.

When it is not possible to determine communication-free static data partitions, they derive pipelined computation partitions in the second step of their algorithm. This step is performed to evaluate the benefit of partitioning sequential loops, while maintaining static data partitions.

The third step of the algorithm is used to derive dynamic data partitions, either when communication-free partitioning cannot be obtained or when a loop nest cannot be pipelined.\(^3\) This step eliminates the largest amounts of communication, by obtaining static partitions for the most frequently executed loop nests. If the static partition chosen for the frequently

\(^3\)Typically, a loop nest cannot be pipelined when the dependence distances are unknown.
executed loop nest does not result in communication-free partitions for the less frequently executed loop nests, then a new data partition is chosen for the less frequently executed loop nests. Thus, by using this "greedy" approach, they place communication in the least executed segments of the program.

It should be noted that, by default, the algorithm uses the Block distribution attribute. The Cyclic distribution attribute is used when the workload is not balanced, while the BlockCyclic distribution attribute is used when a loop nest with unbalanced workload is pipelined. Data transformations are finally used to allow the processors to access contiguous data regions.

This technique uses the owner-computes rule to partition computations. i.e., it requires that data be allocated where the computation is performed. Consequently, when a communication-free data partition cannot be determined, data will be repartitioned, even if the cost of accessing remote data is less than the cost of repartitioning. Hence, the algorithm considers significantly fewer data partitioning candidates than are possible and may result in sub-optimal data partitions for applications on SSMMs. The algorithm does not determine a processor geometry and block sizes for BlockCyclic distribution attributes. In addition, this technique relies on accurate profiling information to indicate the execution frequencies for loop nests.

2.4 Discussion

All of the existing techniques, except Anderson and Lam's algorithm, derive data partitions for distributed memory multiprocessors. Although Anderson and Lam's algorithm is intended for SSMMs as well, it is strongly influenced by techniques for distributed memory multiprocessors. in that the algorithm uses the owner-computes rule. Therefore, the major focus of existing techniques has been the minimization of communication between processors.

Our research focuses on taking into account shared memory effects on SSMMs while deriving computation and data partitions. In Chapter 4, we identify the shared memory effects that must be considered in the derivation of computation and data partitions on SSMMs. In the next chapter, we introduce new computation and data partitions that are especially useful on SSMMs.

4We will see an example of this in fft3d, in Chapter 7.
This chapter describes the new computation and data partitions that we have introduced. This chapter is organized as follows. First, we describe six new distribution attributes that can be used to partition data and computation. Subsequently, we describe how new data partitions are derived by mapping each distributed dimension of an array to the same dimension or many dimensions of the processor geometry. Finally, we conclude this chapter by describing why the usefulness of these new computation and data partitions may be limited to scalable shared memory multiprocessors.

3.1 New Distribution Attributes

In addition to the traditional distribution attributes\(^1\), we introduce six new distribution attributes: RBlock, RCyclic, RBlockCyclic, BlockRBlock, CyclicRCyclic, and BlockCyclicRBlockCyclic. Of these distribution attributes, the first three are reverse distribution attributes and the last three are compound distribution attributes.

3.1.1 Reverse Distribution Attributes

The reverse distribution attributes are RBlock, RCyclic, and RBlockCyclic. These distribution attributes are the reverse of the Block, Cyclic, and BlockCyclic distribution attributes respectively. The assignment of different segments of an array dimension (or iterations of a loop) to the processors is made in a decreasing order, starting with processor \(P\) and ending with 1; hence the name "reverse." This is unlike the traditional distribution attributes, where assignment to processors occurs in an increasing order, starting with processor 1 and ending with \(P\). Figure 3.1 shows examples of partitioning a 1-dimensional array.

\(^1\)These are the Block, Cyclic, and BlockCyclic distribution attributes, described in Chapter 2.
using the reverse distribution attributes. For example, consider the partition \( A(\text{RBlock}\{1\}) \) mapped onto a one dimensional processor geometry consisting of 4 processors. The first block of the array is mapped onto processor 4 and the last block of the array is mapped onto processor 1. In contrast, had the array been partitioned using the \text{Block} distribution attribute, instead of the \text{RBlock} distribution attribute, the first block of the array will be mapped onto processor 1 and the last block will be mapped onto the last processor, i.e., processor 4.

It is interesting to note that the “reverse” distribution attributes can be viewed as renumbering the processors in reverse. However, it is not sufficient to implement reverse distribution attributes by processor renumbering since this can affect the mapping of other arrays and loops.

The reverse distribution attributes can be obtained in HPF by using an alignment directive along with a template. For example, the data partition \( A(\text{RBlock}\{1\}) \) can be obtained in HPF by using the following four statements, assuming that the dimension of the array is \( N \).

```
CHPF$ PROCESSORS P(4)
CHPF$ TEMPLATE TMPL(N)
CHPF$ DISTRIBUTE TMPL(BLOCK) ONTO P
CHPF$ ALIGN A(I) WITH TMPL(N-I)
```

In this example, the alignment statement associates the last element of the array with the first element of the template and vice versa. Hence, by partitioning the template, using only the traditional attribute \text{Block}, the \text{RBlock} distribution attribute is obtained. The reverse distribution attributes obtained using HPF can be viewed as \textit{derived} attribute types. However, we set out the reverse distribution attributes as \textit{fundamental} attribute types required for partitioning computation and data. This distinction is important, because it allows us to define the compound distribution attribute, as will be described in the next section.
Before we discuss the functions used to map data using the reverse distribution attributes, it is important to introduce the functions that map data using the traditional distribution attributes. This is because the functions that map data using the reverse distribution attributes are defined based on the functions of the traditional distribution attributes. The generic functions that map array elements onto processors using the traditional distribution attributes, which are Block, Cyclic, and BlockCyclic, are given in Figure 3.2. Given an array A with lower and upper bounds $LB(A_i)$ and $UB(A_i)$ in dimension $i$, and a processor grid with $P_k$ processors in dimension $k$, the functions map an index $l$ of $A$ in dimension $i$ onto a processor in dimension $k$ of the processor grid. $Partition\_Size$ for the Block distribution attribute specifies the size of each block along an array dimension. Recall that the block size, $b(A_i)$, for the BlockCyclic distribution attribute, on the other hand, is the number of contiguous elements that must be co-located as specified by the distribution attribute.

The functions that map the elements of an array onto processors using the reverse distribution attributes are shown in Figure 3.3. Given an array A with lower and upper bounds $LB(A_i)$ and $UB(A_i)$ respectively, in dimension $i$, and a processor geometry with $P_k$ processors in dimension $k$, the functions map an array element indexed by $l$ in dimension $i$ of $A$ onto a processor in dimension $k$ of the processor geometry. It should be noted that the division of arrays into smaller segments for both the traditional and reverse distribution attributes is the same. It is the mapping of the divided segments of a dimension of an array.

\[ Block(A_i, P_k, l) = \left\lfloor \frac{(l - LB(A_i))}{Partition\_Size(A_i, P_k)} \right\rfloor + 1 \]
where $Partition\_Size(A_i, P_k) = \left\lceil \frac{UB(A_i) - LB(A_i)}{P_k} \right\rceil$

\[ Cyclic(A_i, P_k, l) = \left\{ (l - LB(A_i)) \mod P_k \right\} + 1 \]
\[ BlockCyclic(A_i, P_k, l) = \left\{ \left\lfloor \frac{(l - LB(A_i))}{b(A_i)} \right\rfloor \mod P_k \right\} + 1 \]

Figure 3.2: Mapping functions for traditional distribution attributes.

\[ RBlock(A_i, P_k, l) = P_k - Block(A_i, P_k, l) + 1 \]
\[ RCyclic(A_i, P_k, l) = P_k - Cyclic(A_i, P_k, l) + 1 \]
\[ RBlockCyclic(A_i, P_k, l) = P_k - BlockCyclic(A_i, P_k, l) + 1 \]

Figure 3.3: Mapping functions for reverse distribution attributes.
onto the processors that distinguishes the reverse attributes from the traditional attributes. The functions that map the iterations of a parallel loop are similar to the array mapping functions.

### 3.1.2 Compound Distribution Attributes

The *compound distribution attributes* are a synthesis of both the traditional and reverse distribution attributes. BlockRBlock, CyclicRCyclic, and BlockCyclicRBlockCyclic are the compound distribution attributes. When a dimension of an array (or iterations of a loop) is partitioned using a compound distribution attribute, the array dimension (or iterations of the loop) is divided into two halves. The first half of the array dimension (or iterations of the loop) is partitioned using the appropriate traditional distribution attribute. The second half of the array dimension (or iterations of the loop) is partitioned using the appropriate reverse distribution attribute. Figure 3.4 shows examples of partitioning a 1-dimensional array using the compound distribution attributes. Consider the partition A(BlockRBloc{1}) mapped onto a one dimensional processor geometry consisting of 4 processors. The 1-dimensional array being partitioned is first divided into two halves. In order to map the first half segment using the Block distribution attribute, the half segment is further split into smaller pieces. The first piece of the first half segment of the array will be mapped onto processor number 1. The last piece of the first half segment of the array will be mapped onto processor number 4. However, to map the second segment, the RBlock distribution attribute is used. The first piece of the second half segment will be mapped onto processor number 4. The last piece of the second half segment will be mapped onto processor number 1.

The functions that map an array A using the compound distribution attributes are shown in Figure 3.5. These functions assume that the size of an array dimension with a compound distribution attribute is divisible by $2 \times P_k$. When this is not the case, an equal number of data elements cannot be assigned to each processor along the array dimension. The array
\[
\text{BlockRBlock}(A_i, P_k, l) = \begin{cases} 
\text{Block}(A_i, P_k, l) & \text{if } l < (UB(A_i) - LB(A_i))/2 \\
\text{RBlock}(A_i, P_k, l) & \text{otherwise}
\end{cases}
\]

\[
\text{CyclicRCyclic}(A_i, P_k, l) = \begin{cases} 
\text{Cyclic}(A_i, P_k, l) & \text{if } l < (UB(A_i) - LB(A_i))/2 \\
\text{RCyclic}(A_i, P_k, l) & \text{otherwise}
\end{cases}
\]

\[
\text{BlockCyclicRBlockCyclic}(A_i, P_k, l) = \begin{cases} 
\text{BlockCyclic}(A_i, P_k, l) & \text{if } l < (UB(A_i) - LB(A_i))/2 \\
\text{RBlockCyclic}(A_i, P_k, l) & \text{otherwise}
\end{cases}
\]

Figure 3.5: Mapping functions for compound distribution attributes.

dimension is divided into two segments such that the subsequent divisions in the first half segment are all of equal sizes. Each processor may not get equal number of data elements in the second half segment, which is partitioned using the reverse distribution attribute.

### 3.2 New Data Partitions

Traditionally, each dimension of an array with a distribution attribute has been implicitly associated with a unique dimension of the processor geometry [16, 21, 27, 30]. We introduce a number of interesting and useful data partitions by relaxing the strict one-to-one relationship between the distributed dimension of an array and the dimension of the processor geometry. Three new data partitions can be derived by using the mapping between the dimensions of an array and the dimensions of the processor geometry. They are \textit{many-to-one}, i.e., mapping many array dimensions onto one dimension of the processor geometry; \textit{one-to-many}, i.e., mapping one array dimension onto more than one dimension of the processor geometry; and \textit{many-to-many}, i.e., mapping many array dimensions onto many dimensions of the processor geometry.

The three new mappings are not used to partition computation. The relaxed mapping is not generally used between parallel loops and processor geometry dimensions because such a relaxation \textit{i}) adds to the loop overhead introduced to map iterations onto the dimensions of the processor geometry,\textsuperscript{2} \textit{ii}) increases the search space of possible computation partitions, and \textit{iii}) has not proved to be beneficial, in our experience.

\textsuperscript{2}We do not have to explicitly identify the home location of each array element because of the shared memory. However, we do not have such an option with respect to the iteration spaces. We have to partition computation among the processors.
3.2.1 Many-to-One Mapping

The many-to-one mapping is a result of mapping multiple array dimensions onto a single dimension of the processor geometry. Figure 3.6 shows an example in which two dimensions of an array \( A \) are mapped onto a one dimensional processor geometry with 4 processors, using a \( \text{Block}\{1\}, \text{Block}\{1\} \) partitioning. The array is sliced into blocks of contiguous elements along both its dimensions because of the \text{Block} distribution attribute. The resultant sub-array blocks 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 \((i, j)\) of the tessellated array maps to processor \([(i - 1) + (j - 1)]\mod P + 1\) in the linear array of \( P \) processors. This partitioning of the array is called a \textit{latin square}, and it represents one possible mapping of multiple dimensions of the array onto the same dimension of the processor geometry.

Figure 3.7 shows the function that determines which processor an array element is mapped to when many array dimensions are mapped onto the same dimension of the processor geometry. This function determines the processor in the \( k \)th dimension of the processor geometry \( P \) onto which an array reference \( A(l_1, l_2, ..., l_n) \) is mapped. The processor onto which an array element is mapped is determined in two steps. First, the processor numbers onto which the array element is mapped are determined along each of the dimensions of the array.
array, using the distribution attribute functions. Second, the processor numbers of the array dimensions that map onto the $k$th dimension of the processor geometry are added together to determine the processor in that dimension. Since the processor in the $k$th dimension of the processor geometry must remain within the limits of $1 \ldots P_k$, to be a valid processor, a modulus operation is performed to obtain the processor onto which the array element is mapped.

### 3.2.2 One-to-Many Mapping

The one-to-many mapping is a result of mapping a single dimension of an array onto multiple dimensions of the processor geometry. Figure 3.8 shows an example of a 2-dimensional array $A$ that is partitioned using (Block{1,2},*). In this example, the first dimension of the array is mapped onto both dimensions of a two-dimensional processor geometry, with 4 processors in each of its dimensions. Only the first dimension of the array has a Block distribution attribute. Hence, the array must be sliced into blocks of contiguous elements along the first dimension. The number of blocks into which the dimension must be divided is dictated by the number of processors associated with this dimension. Since both dimensions of the processor geometry partition the array, the number of segments into which the array must be divided is the product of the number of processors along each of the dimensions of the processor geometry. Hence, the array is divided into 16 segments, with one segment for each of the 16 processors ($4 \times 4$). Conceptually, the partitioning of the first dimension of $A$ onto the two dimensions of the processor geometry can be viewed as a two-step process. The first step is to divide the array into 4 segments and map these segments onto the second dimension of the processor geometry. The second step is to further divide and map the segments onto the first dimension of the processor geometry.

The function that maps an element of an array in which one of the array dimensions has a one-to-many mapping is made up of three steps. First, the multiple processor geometry dimensions partitioning an array dimension are linearized\(^2\) into a single dimension. Second, the processor number in the linearized geometry onto which the array element is mapped is determined, using the distribution attribute functions. Finally, the processor number along each of the multiple processor geometry dimensions is determined, based on the processor number in the linearized geometry and the linearization function.

\(^2\)The Fortran column major order is followed.
For example, consider a one dimensional array \( A(1:100) \) that is mapped onto a three dimensional processor geometry \((2.5.5)\), using a \((\text{Block}[1.2.3])\) mapping. Let us determine the processor onto which the array element \( A(77) \) maps. The three dimensional processor geometry consists of a total of 50 processors. In a linear processor geometry of 50 processors, we can determine that the array element \( A(77) \) maps to processor number 39, using the mapping functions described for the distribution attributes. Based on the linearization function, in this example we find that the processor number 39 is \((1.5.4)\) in the three dimensional processor geometry.

### 3.2.3 Many-to-Many Mapping

The combination of many-to-one and one-to-many partitioning results in many-to-many mapping. Figure 3.9 shows an example in which a two dimensional array \( A \) is mapped onto a two dimensional processor geometry with 4 processors in each dimension, using \((\text{Block}[1.2].\text{Block}[1])\). This partitioning of the array results in both the dimensions of the array being divided and mapped onto the processor geometry dimension 1, and the first dimension is further divided and partitioned along the processor geometry dimension 2. The combination of \((\text{Block}[1].\text{Block}[1])\) and \((\text{Block}[1.2].*)\) partitions discussed in the previous sub-sections is the partition \((\text{Block}[1.2].\text{Block}[1])\).

The functions for the many-to-many mapping are obtained by applying the many-to-one and one-to-many functions. The order in which these functions are applied is determined by the mapping of the array onto the higher dimensions of the processor geometry. First, the mapping onto the highest dimension of the processor geometry is determined.
subsequently, the processor numbers along the lower dimensions of the processor geometry are determined. In this example, a one-to-many function is first applied to obtain the processor numbers along the second dimension of the processor geometry. The many-to-one mapping is then applied on the subdivided array to determine the processor number along the first dimension of the processor geometry. However, the many-to-one and one-to-many functions are not sufficient when many dimensions of an array are mapped onto many of the same dimensions of the processor geometry. Consider, for example, the data partition \((\text{Block}\{1.2\}, \text{Block}\{1\})\). In such cases, linearization, along with the one-to-many and/or many-to-one functions, is applied to determine the processors onto which the array elements are mapped.

3.3 Summary

In this chapter we have described six new distribution attributes. These distribution attributes extend the set of possible computation and data partitions that have been discussed in the literature. In addition, by allowing any mapping between dimensions of an array and the dimensions of the processor geometry, we have derived new data partitions that are unique and interesting.

The one-to-one mapping of array dimensions onto processor geometry dimensions allows for simple functions that determine the processor onto which the data are mapped. However, if array dimensions can be mapped onto any dimension of the processor geometry, the functions that determine the processor onto which data are mapped become complicated.
On distributed memory multiprocessors, the owner-computes rule is used to partition computation and to determine the owner of each of the array elements. The statements in the program are guarded to ensure that only the owner of an array element can update it. With one-to-one mappings, the guards in most cases are simple and can be eliminated by using compile time analysis and by rewriting the loop bounds. In contrast, the one-to-many, many-to-one, and many-to-many mappings will generate a program in which the elimination of such guards is a non-trivial task, because simple loops cannot be written. On SSMMs, we can obtain guard-free code, because coherence is maintained by hardware. We hypothesize that in the absence of guards, these mappings can be beneficial in improving program performance on SSMMs.
This chapter presents experimental evidence to show that shared memory effects influence the selection of computation and data partitions on SSMMs. We show that the computation and data partitions derived for SSMMs may be different from those derived for distributed memory multiprocessors. This chapter is organized as follows. First, we identify the performance factors that affect the derivation of computation and data partitions on SSMMs. Subsequently, we present experimental results to show that factors such as cache affinity, memory contention, synchronization overheads, and false sharing are important, and that they must be considered, in addition to remote memory access, in the derivation of computation and data partitions. Moreover, we also show how a data partition using the many-to-one mapping described in the previous chapter can help to minimize memory contention and synchronization overheads. Finally, we conclude this chapter by reviewing the problem of deriving computation partitions in the presence of SSMM specific performance factors.

4.1 Performance Factors

The primary factor that affects the performance of a parallel application on a distributed memory multiprocessor is the relatively high cost of interprocessor communication. However, on SSMMs, additional performance factors, such as cache affinity, memory contention, synchronization overheads, and false sharing, can have a significant impact on the choice of appropriate computation and data partitions. In this section, we describe these factors and explain why they were not considered significant for distributed memory multiprocessors.
4.1.1 Cache Affinity

Caches are used in SSMMs to reduce memory access time and reduce contention in the interconnection network. Cache affinity results when array elements that are reused across the program are retained in the cache. Memory access times are reduced because of two types of data reuse: spatial reuse and temporal reuse. Data is transferred between cache and memory in units of cache line. Typically, a cache line contains multiple array elements. Although only a single data element is accessed by a processor, a cache line is transferred from the memory to the cache. Spatial reuse occurs when all the array elements in a cache line are used by a processor before the line is flushed from the cache. Temporal reuse occurs when an array element in a cache line is reused before the cache line is evicted from the cache.

The performance of an application depends to a large extent on the ability of the cache to exploit spatial and temporal reuse. In some cases, exploiting reuse may be difficult because of the limited capacity and associativity of caches. Data brought into a cache by a reference or a prefetch may be evicted before being used or reused, because of either a capacity or a conflict miss caused by a subsequent reference. Cache misses on SSMMs adversely affect performance, since evicted data must be retrieved from their home memory, which may be remote to the processor.

Although caches are important in improving the performance of applications on distributed memory multiprocessors, remote data is not directly fetched to a processor's cache. First, the remote data is fetched, using the communication routines, into the local memory of the processor. Subsequently, data is cached from the local memory when a particular data item is accessed by the processor. Hence, caches play a less important role in distributed memory multiprocessors than in SSMMs, because cache misses result exclusively in local memory accesses, which are inexpensive in comparison to interprocessor communications.

4.1.2 Memory Contention

Memory contention occurs when many processors access data in a single memory module at the same time. Memory contention can degrade the performance of an application. Since the communication protocol in SSMMs is receiver-initiated, and transfers data in units of relatively small cache lines, a large number of requests to the same memory can overflow
memory buffers and cause excessive delays in memory response time [32].

Memory contention has been considered less of a performance bottleneck on distributed memory multiprocessors because of sender-initiated communication (i.e., each processor is responsible for sending data that it owns). The order in which the data is to be sent is determined at compile time. In addition, in programs written for distributed memory multiprocessors, data is communicated using large messages and infrequently. Applications on distributed memory multiprocessors also use collective communications [33] to further reduce memory contention.

4.1.3 Synchronization Overheads

Synchronization is introduced in programs to maintain program correctness. On SSMMs, synchronization is explicit, and is generally in the form of barriers at the start and the end of each parallel loop. Barriers are expensive, because the cost of barrier synchronization typically increases linearly with the number of processors. The speedup of parallel applications on SSMMs can be significantly limited due to barrier synchronization overhead [34]. The overhead due to barrier synchronization on SSMMs can become a performance bottleneck [35], and must be minimized. However, in distributed memory multiprocessors, synchronization is implicit, and is achieved through data communication.

4.1.4 False Sharing

In SSMMs, data on the same cache line may be shared by more than one processor, and the line may exist in more than one processor's cache at the same time. True sharing occurs when two or more processors access the same data on a cache line, and it reflects necessary data communications in an application. On the other hand, false sharing occurs when two processors access different pieces of data on the same cache line. On a hardware cache-coherent multiprocessor, if at least one of the processors continuously writes while the other processor continuously reads or writes to the same cache line, the cache consistency hardware causes the cache line to be transferred back and forth between processors, leading to a "ping-pong" effect [36]. False sharing causes extensive invalidation traffic and can considerably degrade performance. False sharing does not occur on distributed memory multiprocessors, because the arrays are partitioned and mapped into separate address spaces.
4.2 Experimental Evaluation

The degradation caused by such factors as the amount of remote data retained in the cache, memory contention, synchronization overheads, and false sharing can be mitigated by changing the computation and data partitions. First, we present an experiment using the Multigrid application, in which the selection of an appropriate processor geometry ensures that cache affinity and program performance are improved. Subsequently, we describe an experiment using the ADI application, in which memory contention and synchronization overheads are reduced by selecting a data partition with many-to-one mapping. Finally, we present an experiment using the tred2 application, in which false sharing is minimized by selecting appropriate data and computation partitions.

4.2.1 Cache-Conscious Computation and Data Partitions

The KSR-1 multiprocessor is used to present how cache affinity plays a key role in the selection of computation and data partitions. The KSR-1 multiprocessor has a medium-sized cache and hardware monitoring. The presence of this hardware enables us to measure the number of non-local memory accesses and the number of cache misses that occur for a processor. In addition, on this machine, we focus only on determining computation partitions. The data partitions are dictated by the computation partitions, as data migrates to the cache memory of a processor when it is accessed, because of the COMA architecture. The architecture of the KSR-1 multiprocessor is described in Chapter 7 on page 98.

The Multigrid application from the NAS suite of benchmarks illustrates why computation and data partitions must be cache-conscious. Using this application, we also illustrate that problem size is critical in determining computation and data partitions. Multigrid is a three dimensional solver that calculates the potential field on a cubical grid. We focus on the subroutine psinv, which uses two three-dimensional arrays U and R, as shown in Figure 4.1. This application has data access patterns typical of many scientific applications.

The psinv subroutine contains a loop nest with three parallel loops. We choose not to parallelize the innermost loop to avoid cache line false sharing and cache interference, because successive iterations of this loop access successive elements on the same cache line. Hence, to partition the two outer parallel loops onto the processors, we choose a two dimensional processor geometry. Since this application does not have load imbalances, and
\begin{verbatim}
1 REAL U(N, N, N), R(N, N, N), C(0:3)
2 forall i = 2 to N-1
3    forall j = 2 to N-1
4        forall k = 2 to N-1
5            U(k, j, i) = U(k, j, i) +
6                C(0)*R(k, j, i) +
7                   C(1)*( R(k-1, j, i) + R(k+1, j, i) + R(k, j-1, i) + R(k, j+1, i) +
8                       R(k, j+1, i) + R(k, j-1, i-1) + R(k, j+1, i-1) +
9                       R(k+1, j+1, i) + R(k-1, j-1, i) + R(k-1, j-1, i-1) +
10                      R(k-1, j, i-1) + R(k-1, j, i+1) + R(k, j+1, i) +
11                         R(k, j+1, i) + R(k, j, i-1) + R(k, j, i+1) + R(k+1, j, i-1) +
12                            R(k+1, j, i+1) + R(k+1, j+1, i) +
13                              R(k-1, j-1, i-1) + R(k+1, j-1, i-1) + R(k-1, j, i-1) +
14                                 R(k, j, i-1) + R(k, j, i+1) + R(k+1, j-1, i-1) +
15                                     R(k, j-1, i) + R(k+1, j+1, i-1) +
16                                          R(k+1, j, i+1))
17    end for
18 end for
\end{verbatim}

Figure 4.1: The psinv subroutine.

has near neighbour references such as \( i - 1, i \), and \( i + 1 \). The \textbf{Block} distribution attribute is typically chosen [22] to partition the parallel loops. We map the outermost parallel loop onto dimension 2 of the processor geometry and the inner parallel loop onto processor dimension 1. Thus, based on computation partitioning, the data partitioning of the arrays are \((*,\text{Block}{1},\text{Block}{2})\).

We have to choose the number of processors to be assigned to each of the two dimensions of the processor geometry. With 16 processors, it is possible to choose the \((16,1),(8,2),(4,4),(2,8)\), or \((1,16)\) processor geometry. The choice of processor geometry affects the number of processors that execute each parallel loop in the loop nest. For example, a processor geometry of \((8,2)\) implies that there are 8 processors assigned along the first dimension and 2 processors along the second dimension of the processor geometry.

Figure 4.2 shows the execution time of the application for various processor geometries, with the \((*,\text{Block}{1},\text{Block}{2})\) partition for the arrays, on the KSR-1 with 16 processors, normalized with respect to \((16,1)\) processor geometry. For a small data size \((64x64x64)\), execution time is minimized by a partition with an equal number of processors in each dimension, i.e. \((4,4)\). However, when the data size is large, the processor geometry \((4,4)\) no longer performs the best. The execution time is minimized with a processor geometry of \((8,2)\).
The impact of processor geometry on performance is due to cache affinity, as can be deduced from Figures 4.3(a) and 4.3(b). Figure 4.3(a) shows the normalized result of the number of cache lines accessed from remote memory modules for the various processor geometries, with respect to (16.1) processor geometry. Figure 4.3(b) shows the normalized number of cache misses, again with respect to (16.1) processor geometry. In Figure 4.3(a), the number of remote memory accesses is minimal when the processor geometry is (4.4) for all data sizes. When the data size is small (64x64x64), the data used by a processor fits into the 256k processor cache, and the misses from the cache, in this case, reflect remote memory accesses that occur in the parallel program. Hence, the predominant factor affecting performance is interprocessor communication, and the best performance is attained using the (4.4) geometry. This confirms the appropriateness of this partition for distributed memory multiprocessors.

However, when the arrays are relatively large (144x144x144), the cache capacity is no longer sufficient to hold data from successive iterations of the outer parallel loop, and the number of cache misses increases. When the number of processors assigned to the outer loop increases, the number of misses from the cache also increases. The (4.4) processor geometry minimizes the number of remote memory accesses, but the (16.1) processor geometry minimizes the number of cache misses. The partition with (8.2) processor geometry strikes a balance between the cost of remote memory access and the increased cache affinity, resulting in the best overall performance, in spite of a relatively high number of remote memory accesses.

Similar conclusions that indicate that cache affinity plays a key role can also be derived analytically for the program by determining the amount of data accessed by an iteration.
of the outer parallel loop, as shown in Table 4.1. As the number of processors assigned to the inner parallel loop increases, the amount of data accessed by one iteration of the outer parallel loop decreases. When the data size is small (64x64x64), the data used by a processor is less than 128K for each of the processor geometries. Hence, the best performance is attained by minimizing interprocessor communication, which occurs at the (4,4) geometry. However, when the data size is large (160x160x160), the data used by a processor is greater than the cache size for some processor geometries. Hence, the best performance for the application is obtained only when there is a balance between the number of remote memory accesses and the amount of the data retained in the cache across iterations of the outer parallel loop.
REAL x(N, N), a(N, N), b(N, N)

for iter = 1 to MaxIter

// Phase 1: ADI forward & backward sweeps along rows
for j = 2 to N
    forall i = 1 to N
        b(i, j) = f{a(i, j), b(i, j-1), x(i, j)}
    end for
end for

// Phase 2: ADI forward & backward sweeps along columns
forall j = 1 to N
    for i = 2 to N
        b(i, j) = h{a(i, j), b(i-1, j), x(i, j)}
    end for
end for

Figure 4.4: Outline of the ADI program.

4.2.2 Contention- and Synchronization-Conscious Computation and Data Partitions

The Hector multiprocessor is used to illustrate how contention and synchronization can have adverse effects on the performance of an application without the interference of cache effects: cache coherence is maintained by the software. The architecture of the Hector multiprocessor is described in Chapter 7 on page 97.

The Altering Direction Integration (ADI) program is used to illustrate the choice of data and computation partitions to minimize contention and synchronization overheads. In addition, we use this experiment to show that, on SSMMs, computation does not necessarily have to be partitioned according to data, and vice versa.

The outline of the ADI program is shown in Figure 4.4. This program accesses three 2-dimensional arrays A, B and X. A single iteration of an outer sequentially iterated loop consists of a forward and a backward sweep phase along the rows of three arrays, followed by another forward and backward sweep phase along the columns of the arrays [37]. In this application, parallelism is along the columns in the first phase, and along the rows in the second phase. This application is typical of other programs, such as 2D-FFT, 3D-FFT, turbo-3d.
data partition
computation partition

Figure 4.5: Partitioning requirements of the different phases.

and Erlebacher, that have parallelism in orthogonal directions in different phases of the program.

We partition the parallel loops using the Block distribution attribute, because the ADI application does not have any load imbalances.

Page placement policies fail to enhance memory locality for the ADI application. Pages can be placed in a first-hit or round-robin manner among the processors. When the first-hit policy is used, most of the data accessed by a processor will be placed in the local memory of the processor in the first phase of the program. A page is shared only when data accessed by two iterations assigned to different processors are on the same page. Since Block distribution attributes are used to partition computation, the sharing of data only occurs in a small number of pages in the first phase. However, performance degrades in the second phase of the program, because all the processors contend to read data from the memory module of one processor in phase-lock with one another. The round-robin policy of page placement does not offer relief. Processors continue to access data in a phase-lock on each page, except now the pages are in different memory modules. This policy also increases memory access latency, because data is not resident in the local memory during the first phase. Hence, irrespective of the page placement policy, the performance of the program degrades because of contention.

The data partitioning requirements for each phase of the program are shown in Fig-
Each phase has a different requirement for partitioning the array. There are three possible alternatives for partitioning the data in the program. The first choice is to partition the data that satisfies one of the loops and incur the additional penalties for not conforming to the requirement of the loop with the conflicting requirement. The second choice is to partition data in the loops such that the data partition is optimal for each of the loops. This method requires data to be repartitioned (redistributed), i.e., re-mapped between the execution of the different loops. Depending on the multiprocessor, the cost of repartitioning can be very high, offsetting the benefit of the optimal partitioning of the arrays. The third choice is to select a partition that is suboptimal for both the loops, but which minimizes the overall program execution time. The best data partition for ADI remains an issue of debate [9, 37]. Amarasinghe et al. [9] follow the first choice for partitioning the data, while Kremer [37] follows the second approach and concludes that repartitioning is beneficial.

First, we will analyze the $(*, \text{Block}\{1\})$ data partition in conjunction with the owner-computes rule. The $(*, \text{Block}\{1\})$ data partition conforms to the requirement of the first phase of the program. Following the owner-computes rule, the second phase of the program results in a pipelined computation partitioning, as shown in Figure 4.6. During the first phase of the program, all the processors access local data and require no remote accesses. However, during the second phase, the parallelism is orthogonal to the direction of data partition. Strict adherence to the owner-computes rule requires that the computations
be performed by processors on the chunk of columns that they own. Thus, processor $i$ has to wait for processor $i - 1$ to finish the computation on its chunk of the column before proceeding. A large number of synchronizations is required to maintain the ordering involved in the pipelined computation.

The synchronization overhead can be eliminated by relaxing the owner-computes rule in the second phase and allowing the processor to write the results to remote memory modules. Figure 4.7 illustrates the data and computation assigned to each processor in both phases of computation. This partitioning of computation eliminates synchronization overhead at the expense of an increased number of remote memory accesses. The performance of the program does not improve, because the relaxed computation rule, along with the $(*, \text{Block}{1})$, partition results in heavy memory contention. Each processor is responsible for computing a chunk of columns: hence, each processor accesses every memory module in sequence. Thus, each memory module is accessed by every processor at the same time, leading to the memory contention.

The data partition $(\text{Block}{1}, \text{Block}{1})$, along with the relaxed computation rule, is another alternative. Figure 4.8 illustrates the data accessed by the processors in each of the two phases. With this partition, processors access data from remote memory modules in both phases of the program. In both phases, processors start working on the columns assigned to them by accessing data in different memory modules, thus avoiding memory
The use of the owner-computes rule with this data partition will not result in good performance. Ownership tests must be introduced in the body of the loops, leading to increased overhead.

Figure 4.9 shows the results of executing the ADI application on the Hector multiprocessor for a data size of 256x256, with various data partitions and compute rules. The (Block\{1\}, Block\{1\}) data partition that relaxes the owner-computes rule outperforms all other data partitions that adhere to the rule. Repartitioning of the array across the two phases, i.e., adhering to the best data partitions required in the individual phases, performs almost as well as the (Block\{1\}, Block\{1\}) data partition with relaxed computation rule. The figure also indicates that ownership tests degrade performance. It is also clear that the use of data partitioning improves performance over the use of operating system policies to manage data (the no distributions curve). The performance bottlenecks of various
Table 4.2: Performance bottlenecks in various data and computation partitionings for ADI.

<table>
<thead>
<tr>
<th>Data Partition</th>
<th>Compute Rule</th>
<th>Performance Bottleneck</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Relaxed</td>
<td>Memory Contention</td>
</tr>
<tr>
<td>(*. Block{1})</td>
<td>Owner-Computes</td>
<td>Synchronization</td>
</tr>
<tr>
<td>(*. Block{1})</td>
<td>Relaxed</td>
<td>Memory Contention</td>
</tr>
<tr>
<td>(<em>. Block{1})/ (Block{1})</em></td>
<td>Owner-Computes</td>
<td>Repartitioning</td>
</tr>
<tr>
<td>(Block{1}). Block{1})</td>
<td>Owner-Computes</td>
<td>Ownership tests</td>
</tr>
<tr>
<td>(Block{1}. Block{1})</td>
<td>Relaxed</td>
<td>High Remote Memory Access</td>
</tr>
</tbody>
</table>

partitions for ADI are summarized in Table 4.2.

4.2.3 False-Sharing-Conscious Computation and Data Partitions

The KSR-1 multiprocessor is used to illustrate the effect of false sharing. The KSR-1 multiprocessor is chosen because of its long cache lines.

The trrd2 program (which is part of eispack) is used to illustrate the choice of data and computation partitions to minimize false sharing. This program contains inner loop parallelism and has considerable load imbalances. The parallel loop iterators are part of the subscript expressions along the cache line dimension of the arrays in the program. The outline of the application is shown in Figure 4.10. This application is similar to mdg and trfd, which are part of the perfect club suite of benchmarks.

Typically, to balance the load, the parallel loops are partitioned using Cyclic computation partitioning [22]. This partitioning of computation causes more than one processor to share the same cache line, leading to false sharing. The impact of this false sharing is shown in Figure 4.11(a). The figure shows the execution time of the application using Cyclic and
1 REAL Z(N, N), D(N)
2 for i = 2 to N
3 . . .
4   for j = 1 to N-i
5     forall k = j to N-i
6        . . . Z(k, j) . . .
7    end for
8 end for
9 . . .
10 forall j = 1 to i-1
11     . . . D(j) . . .
12 end for
13 . . .
14 end for

Figure 4.10: Outline of the tred2 program.

BlockCyclic partitioning on 1 to 16 processors. The use of the Cyclic partition results in a large number of cache misses, as can be seen in Figure 4.11(b). The resulting overhead due to cache misses causes the execution time to increase as the number of processors increases. Each data element contained in the cache line will be written to by a different processor. When the loop and, therefore, the arrays are partitioned in a BlockCyclic manner, making the size of the block equal to the size of the cache line, false sharing is effectively eliminated. When the number of processors is small, the load is relatively well-balanced, and the elimination of false sharing improves performance. However, as the number of processors increases, the load becomes increasingly imbalanced, and the negative impact of this load imbalance begins to outweigh the benefits of eliminating false sharing.

4.3 Summary

Although large SSMMs are built based on an architecture with distributed memory, the shared memory paradigm introduces performance issues that are different from those encountered in distributed memory multiprocessors. The high cost of interprocessor communication in distributed memory multiprocessors makes the minimization of communication the predominant issue in selecting data partitions and in partitioning computation. In this chapter, we have shown that, on SSMMs
Figure 4.11: Performance of tred2 on KSR-1.

- Cache affinity, memory contention, synchronization, and false sharing play an important role in the derivation of computation and data partitions on.

- We need not strictly adhere to the owner-computes rule. We can use the shared address space and the hardware/software coherence mechanisms to implement a relaxed computation rule.

We use the observation that we do not have to adhere strictly to the owner-computes rule to derive the CDP algorithm for finding computation and data partitions that improve the performance of applications on SSMMs.
In Chapter 4, we discussed why shared memory effects must be considered in deriving computation and data partitions. This chapter describes the CDP algorithm, which derives appropriate computation and data partitions for an application on SSMMs. The chapter is organized as follows. First, we describe the input program model. Second, we describe the output of the CDP algorithm. Third, we present an outline of the different steps involved in obtaining the required output. These steps are elaborated in the subsequent sections.

5.1 The Input Program Model

The input to the CDP algorithm is a program consisting of a sequence of loop nests, in which parallel loops are identified. The loop nests targeted by the CDP algorithm are identified as follows. In each loop nest in the program, the nesting level\(^1\) \(N_s\) of the outermost parallel loop in the nest is identified. The smallest value of \(N_s\) (i.e., the value for the outermost parallel loop) for all nests in the program is used to delineate the target loop nests for the algorithm. This is illustrated in Figure 5.1. The loop nests targeted by the CDP algorithm are highlighted for each example. Figure 5.1(a) illustrates a case in which all the loop levels in each of the loop nests are considered. Figure 5.1(b) illustrates a case in which only the innermost parallel loops are considered. Figure 5.1(c) illustrates a case in which an outer sequential loop enclosing a parallel loop is ignored. Figure 5.1(d) also illustrates a similar case. Hence, by delineating target loop nests using the smallest \(N_s\), the CDP algorithm ensures that all parallel loops in the program are considered. Although we do not directly include the sequential loops that enclose the target loop nests in the selection of computation

\(^1\)The nesting level of a loop reflects the number of loops that encloses it [38]. Hence, the nesting level of the outermost loop is 0, and the nesting level of a loop enclosed by \(n\) outer loops is \(n\). We denote the nesting level of a loop \(m\) by \(N_m\).
Figure 5.1: Examples of loop nests targeted by the CDP algorithm.
and data partitions, they are used in the cost estimation of the CDP algorithm to increase the cost of execution of the target loop nest by the number of iterations of the sequential loop.

5.2 The Output of the CDP Algorithm

The output of the CDP algorithm is a computation partition for the loop nests and data partitions for the arrays in the input program. The derived computation and data partitions improve the performance of the input program. The computation and data partitions are represented as directives, which can be used by either a compiler or a run-time system to execute the program in parallel.

A sample program with output directives that specify computation and data partitions is shown in Figure 5.2, where the directives are preceded by “CSCDP.”
The output directives specify (i) processor geometry, (ii) data partition, and (iii) computation partition. The first directive in the program shown in Figure 5.2 specifies a two dimensional processor geometry, where each dimension of the processor geometry contains 4 processors. Thus, a total of 16 processors is used to partition both computation and data.

The output directives for data partitions are specified using a distribution attribute for each dimension of the array and a mapping of the array dimensions onto the dimensions of the processor geometry. In the program of Figure 5.2, the first dimension of array A is partitioned using a distribution attribute. The second and third dimensions of the array are partitioned using the Block distribution attribute. The second dimension of the array is mapped onto the processor geometry dimension 1. The third dimension is mapped onto the processor geometry dimensions 1 and 2.

Figure 5.2 does not show dynamic partitions that the CDP algorithm is capable of deriving. When an array is dynamically partitioned, the algorithm outputs more than one directive for the array. These directives are output at the start of a loop nest where the data partition of an array changes.

The computation partitioning directive for a loop is placed immediately before the loop in the program. The directives for computation partitioning specify the distribution attribute and an assignment of a processor geometry dimension. In the sample program, the two parallel loops in loop nest $L_1$ have a computation partitioning directive. Both the parallel loops are partitioned using the Block distribution attribute. The outermost parallel loop is mapped onto dimension 2 of the processor geometry, whereas the inner parallel loop is mapped onto dimension 1 of the processor geometry.

Figure 5.2 does not show the output directives for pipelined execution of loops that the CDP algorithm is capable of deriving. A computation partition directive is used for pipelined execution, and it is placed immediately before the sequential loop to be pipelined. The distribution attribute is used to select the granularity of the pipelined computation partitioning.

\[2^\text{When BlockCyclic distribution attributes are chosen, the block sizes follow the distribution attribute specification for each dimension of the array.}\]
5.3 Outline of the CDP Algorithm

The goal of the CDP algorithm is to derive good partitions for arrays and loops. This is accomplished by using the notion of *affinity* between parallel computation and data. 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 an array reference by using the same distribution attribute to partition the parallel loop and the array dimension, and by mapping both the array dimension and the parallel loop to the same dimension of the processor geometry. Since multiple references can occur to the same array in one or multiple loop nests, a conflict in the selection of the distribution attribute and/or the mapping to processor geometry dimensions can arise, and a *partitioning conflict* is said to exist. There are three types of partitioning conflicts that can exist for an array in a program: *sequential conflicts*, *distribution conflicts*, and *parallelism conflicts*.

A sequential conflict occurs when the subscript expression in a partitioned dimension of an array is not a function of a parallel loop iterator (i.e., the subscript expression is a function of only sequential loop iterator(s)). This type of conflict occurs, for example, in a program when the subscript expression in an array dimension is a function of a parallel loop iterator in one loop nest, and the subscript expression in the same array dimension in another loop nest is a function of only a sequential loop iterator. If the array dimension is partitioned, then remote memory accesses will result in the second loop nest. Similarly, if the array dimension is not partitioned, remote memory accesses will result in the first loop nest.

A distribution conflict occurs when an array is accessed in more than one loop nest and there is a mismatch in the distribution attribute and/or the processor geometry dimension assignment for an array dimension. For example, consider an array A that is accessed in two loop nests. Let the subscript expression in a dimension of A be the function of a parallel loop iterator that is assigned a processor dimension {1} in the first loop nest, and let the subscript expression in the same dimension of A be a function of a parallel loop iterator that is assigned a processor dimension {2} in the second loop nest. The array dimension must be mapped onto the processor dimension {1} to minimize remote memory accesses in the first loop nest, and must be mapped onto processor dimension {2} to minimize remote memory
L1: for $j = 1$ to $N-1$
  forall $i = j$ to $N-1$
    $A(i,j) = (B(i) + B(i+1)) \times A(i,j+1)$
  end for
end for

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

Figure 5.3: Running example #1.

accesses in the second loop nest. Mapping the array dimension onto both dimensions of the processor geometry will also result in remote memory accesses in both loop nests.

A parallelism conflict occurs when the subscript expressions in more than one dimension of the array are functions of a single parallel loop iterator in a loop nest. For example, this type of conflict can occur when the subscript expressions in both dimensions of a 2-dimensional array $A$ are functions of a single parallel loop iterator $i$, say $A(i,i)$. In this case, only elements of $A$ along the diagonal are accessed. When both dimensions of $A$ are partitioned and mapped onto a single-dimensional processor geometry, then many remote memory accesses occur due to poor memory locality (Chapter 3). Parallelism conflicts seldom occur in programs.

Sequential and distribution conflicts are illustrated by the examples in Figure 5.3 and Figure 5.4. Figure 5.3 shows an example program in which a sequential conflict occurs for array $A$. The program consists of two loop nests L1 and L2, each having two loops. The program accesses two 2-dimensional arrays $A$ and $C$ and a 1-dimensional array $B$. In loop nest L1, the inner $i$-loop is parallel, while in the loop nest L2, the outer $j$-loop is parallel. The first dimension of array $A$ is accessed using the parallel loop iterator $i$ in the loop nest L1, and is accessed along the second dimension using the parallel loop iterator $j$ in the loop nest L2. Non-local memory accesses can be minimized for array $A$ in loop nest L1, if the first dimension is partitioned similar to the parallel loop $i$. However, non-local memory accesses can be minimized for array $A$ in loop nest L2, if the second dimension is partitioned similar to the parallel loop $j$. Sequential conflict exists for array $A$ in loop nest L1 because the subscript expression in the second dimension of $A$ is a function of only a
L1: forall l = 2 to N-1
    forall k = 1 to N
        forall j = 1 to N
            forall i = 1 to N
                A(i, j, k) = B(i, j, k, l-1) + B(i, j, k, l+1) + A(i, j, k)
        end for
    end for
end for

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

Figure 5.4: Running example #2.

sequential loop iterator j. Similarly, sequential conflict exists for array A in loop nest L2 because the subscript expression in the first dimension of A is a function of only a sequential loop iterator i. In order to resolve this conflict, array A can be partitioned such that only the first dimension is partitioned similar to the i loop in L1. Only the second dimension is partitioned similar to the j loop in L2, or A is repartitioned between the two loop nests.

Figure 5.4 shows an example program in which a distribution conflict occurs for array A. The program consists of two loop nests L1 and L2, each having four parallel loops. The third dimension of array A is accessed in L1 using the parallel loop iterator k and in L2 using the parallel loop iterator l. There is a distribution conflict for the third dimension of array A, when the processor geometry dimension assignments of k-loop in L1 and l-loop in L2 are not the same. In contrast to array A, the subscript expressions in the first, second, third, and fourth dimensions of all references to B in both loop nests are functions of the parallel loop iterators i, j, k, and l respectively. Therefore, there is affinity between the dimensions of B and corresponding loop iterators. There are no partitioning conflicts for array B. Hence, non-local memory accesses can be minimized if each array dimension and the corresponding parallel loop are partitioned and mapped similarly.

When there are no partitioning conflicts for an array, it is possible to derive a unique
Algorithm: cdp-algorithm
begin
  // Phase 1
  1a) select-processor-geometry-dimensionality
     // This step provides candidate computation and data partitions
  1b) establish-data-computation-affinity
  
  // Phase 2
  if (candidate data partitions require re-evaluation) then
    for each array a that requires re-evaluation
     2a) finalize-data-partitions(a)
    end for
    for each loop nest l in the program
     2b) finalize-computation-partitions(l)
    end for
  end if
  
  // Phase 3
  3) processor-geometry-mapping
end

Figure 5.5: The CDP Algorithm.

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. All of the possible partitions must be evaluated to determine which
ones result in minimal execution time.

The two running examples illustrate the sequential and distribution partitioning conflicts
that can arise in a program. In general, the number of possible partitions for a program
which has partitioning conflicts is exponential, and examining all possible partitions is NP-
hard [27]. The CDP algorithm uses the affinity relationships and a cost model to reduce
the number of possible partitions.

The CDP algorithm primarily consists of three main phases. In the first phase, affinity
relationships between array dimensions and parallel loops are captured, using a graph rep-
resentation called the ALAG. Using these affinity relationships, distribution attributes and
a mapping to dimensions of the processor geometry are derived for the array dimensions
and parallel loops. In this phase, the dimensionality of the virtual processor geometry is
also determined. 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 memory contention, and the overhead of synchronization, is considered. In the last phase of the CDP algorithm, a mapping of the virtual processors of the processor geometry to physical processors is determined. Machine-specific information is also used in this phase. An outline of the CDP algorithm is shown in Figure 5.5.

5.4 Processor Geometry Dimensionality

The first step in the CDP algorithm is to determine the processor geometry dimensionality for the input program. This processor geometry dimensionality is used in the subsequent steps of the CDP algorithm. The processor geometry dimensionality for the given program is determined by the number of array dimensions that contain parallel loop iterators: it is those array dimensions that must potentially be distributed. In order to reduce fragmentation of arrays in memory, it is desirable to select a small processor geometry dimensionality, while maintaining parallelism.

We determine the processor geometry dimensionality using parallelism matrices for the arrays accessed in the input program. A parallelism matrix $M_A = [m_{ij}]$ for an array $A$ identifies the dimensions of the array associated with parallel loop iterators. The matrix 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. When the array is not referenced in a loop nest $L_k$, the entries in row $k$ of the matrix are marked X, indicating that this entry is irrelevant. All other entries in the matrix are set to 0.

We use the parallelism matrices to obtain three vectors: $\text{OR.Mask}$, $\text{AND.Mask}$, and visible parallelism ($\text{VP}$). The $\text{OR.Mask}$ is defined as the result of ORing the matrix elements along the columns, i.e., along the dimensions of the array. The $\text{AND.Mask}$ is the result of ANDing the elements along the columns. The $\text{VP}$ is defined as the sum of elements in a row of the parallelism matrix. Hence, the $\text{OR.Mask}$ indicates which dimensions of the array are accessed by a parallel loop iterator in at least one loop nest. Similarly, the $\text{AND.Mask}$ indicates which dimensions of the array are accessed by a parallel loop iterator in every loop nest in the program. Finally, the $\text{VP}$ 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 **AND Mask** of the array 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. This is because there is at least one dimension of the array that can be partitioned across the entire program. Arrays for which both the **AND Mask** and the **OR Mask** are 0 are not distributed and are replicated.

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 is chosen by considering each loop nest in which the array is referenced. The dimensionality of the processor geometry chosen for an array in a loop nest is the smaller of either the number of parallel loops or the **VP**. Note that if we choose the larger of **VP** and the number of partitionable dimensions, some of the dimensions of the processor geometry may not be assigned to loops or array dimensions. Hence, we may not exploit all the processors in performing the computation in the loop nest. The dimensionality of the processor geometry for the array is the one determined most often (i.e., statistical mode) for the array, considering all the loop nests.

False sharing can be avoided by not partitioning an array along the first dimension of the array.\(^3\) Hence, the bit corresponding to first dimension of the array in the **AND Mask** is set to 0 when the number of non-zero elements in the **AND Mask** is greater than 1. In this case, dimensions of the array can be partitioned across the entire program, and yet false sharing is avoided.

The dimensionality of the processor geometry selected for each of the arrays in the program are used to determine the dimensionality of the processor geometry for the program. It is chosen as the statistical mode of the dimensionalities of the processor geometries chosen for the arrays in the program.

The specific 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 the dimensionality of the processor geometry, then the array dimensions

\(^3\)We assume column major storage order of arrays in memory. Hence, contiguous elements in the first dimension correspond to contiguous elements in memory.
Algorithm: select-processor-geometry-dimensionality
begin
  for each array \( a \)
  
  Construct \( M_a \), the parallelism matrix for \( a \)
  
  Select \( d_a \), the processor geometry dimensionality using \( M_a \)
  
end for

\( D = \text{statistical mode} (d_a) \)
// where \( D \) is the processor geometry dimensionality for the program
end

Figure 5.6: Algorithm: select-processor-geometry-dimensionality (Phase 1a).

\[
\begin{align*}
\text{L1} & : A_1 \quad A_2 \quad \text{VP} \\
& : (1 \quad 0) \quad 1 \\
\text{L2} & : A_1 \quad A_2 \quad \text{VP} \\
& : (0 \quad 1) \quad 1 \\
\text{AND Mask} & : 0 \quad 0 \\
\text{OR Mask} & : 1 \quad 1
\end{align*}
\]

\[
\begin{align*}
\text{L1} & : C_1 \quad C_2 \quad \text{VP} \\
& : X \quad X \quad X \\
\text{L2} & : C_1 \quad C_2 \quad \text{VP} \\
& : (1 \quad 0) \quad 1 \\
\text{AND Mask} & : 1 \quad 0 \\
\text{OR Mask} & : 1 \quad 0
\end{align*}
\]

Figure 5.7: The parallelism matrix for the arrays in running example #1.

indicated by 1 in the \text{AND Mask} are chosen, and the array can potentially be distributed only along these dimensions. However, if the number of 1's in the \text{AND Mask} is less than the dimensionality of processor geometry, then all array dimensions indicated by 1 in the \text{OR Mask} of the parallelism matrix are chosen, since it is a subset of these dimensions that will potentially be distributed. The selection of a dimensionality of the processor geometry in the CDP algorithm is accomplished by the algorithm \text{select-processor-geometry-dimensionality}, shown in Figure 5.6.

The parallelism matrix for the arrays in running example #1 (Figure 5.3) is shown in Figure 5.7, where \( A_1 \) and \( A_2 \) correspond to the first and second dimension of array \( A \). The parallelism matrix for array \( A \) contains a 1 entry along \( A_1 \) dimension, and a 0 entry along \( A_2 \) dimension in the loop nest \( L_1 \). Similarly, there is a 1 entry along \( A_2 \) dimension, and a 0 entry along \( A_1 \) dimension in the loop nest \( L_2 \). The parallelism matrix for \( C \) has \( X \) entries
The parallelism matrix for the arrays in running example #2 is shown in Figure 5.8.

Figure 5.8: The parallelism matrix for the arrays in running example #2.

along both C₁ and C₂ dimensions, corresponding to the loop nest L₁. A one dimensional processor geometry will be chosen for this program. Since the AND_Mask of array A is 0, both dimensions A₁ and A₂ will be distributed based on the OR_Mask. Similarly, B₁ and C₁ will be distributed because of the AND_Masks of B and C respectively.

The parallelism matrix for the arrays in running example #2 (Figure 5.4) is shown in Figure 5.8, where A₁, A₂, and A₃ correspond to the first, second and third dimensions of A respectively. The AND_Mask of A indicates that all three dimensions are potentially distributable dimensions. In order to avoid false sharing, a two dimensional processor geometry is chosen for A, based on the AND_Mask. Similarly, a three dimensional processor geometry is chosen for B and C. Hence, a three dimensional geometry is chosen for the program. All the dimensions of A, i.e., A₁, A₂, and A₃, are distributed based on the OR_Mask. Only the three dimensions B₂, B₃, and B₄ of B, and the three dimensions C₂, C₃, and C₄ of C are distributed based on their respective AND_Masks. Recall that B₁ and C₁ are not partitioned to avoid false sharing. However, we cannot avoid partitioning A₁ because the subscript expression in the first dimension of A is a function of a parallel loop iterator.

5.5 Data-Computation Affinity

The purpose of this phase of the algorithm is to establish affinity between arrays and parallel loops in the program. The CDP algorithm uses the affinity relations to derive data and computation partitions. This step of the CDP algorithm is accomplished by the algorithm
establish-data-computation-affinity shown in Figure 5.9.

5.5.1 The ALAG

A bipartite graph, called the Array-Loop Affinity Graph (ALAG), is constructed to capture affinity relationships between arrays and parallel loops. The ALAG is an undirected bipartite graph \((V, E)\), in which each vertex or node \(v \in V\) corresponds to either a parallel loop or an array dimension that can potentially be distributed. The nodes corresponding to parallel loops are called loop nodes \((V_l)\), and the nodes corresponding to array dimensions are array dimension nodes \((V_a)\). Each edge \(e \in E\) connects a loop node \((V_l)\) to an array dimension node \((V_a)\).

An array dimension node has two sub-nodes: a forward sub-node and a reverse sub-node. A forward sub-node represents positive stride accesses along the array dimension, whereas a reverse sub-node represents negative stride accesses along the array dimension. This representation is useful in co-locating elements of two arrays when they are accessed with opposite strides. There is an edge between a loop node \(L_1\) and the forward sub-node of an array dimension if there is a reference to the array in which a subscript expression along that dimension has a positive coefficient of the iterator \(i\). Similarly, there is an edge between a loop node \(L_1\) and the reverse sub-node of an array dimension if there is a reference to the array in which a subscript expression along that dimension has a negative coefficient of the iterator \(i\). Two references to an array, one with positive and the other with negative coefficients of the same parallel loop iterator, result in two edges from the loop node to the array dimension node: one to the forward sub-node and one to the reverse sub-node.

Figure 5.10 shows the ALAG for running example #1. The nodes labeled \(L_1\) and \(L_2\) correspond to the parallel loops in the program. Since the \texttt{AND.Mask} of array \(A\) is 0, both dimensions \(A_1\) and \(A_2\) are chosen as nodes in the ALAG based on the \texttt{OR.Mask}. Nodes \(B_1\) and \(C_1\) are chosen as nodes based on the \texttt{AND.Mask} of arrays \(B\) and \(C\) respectively. For these arrays, we have to partition the cache line dimension, since that is the only dimension that we have selected to be distributable based on the array references and the parallelism in the program. An edge connects the forward node of \(A_1\) with the loop node \(L_1\) because of the references to \(A\) in \(L_1\). Similarly, an edge connects the forward node of \(A_2\) with the loop node \(L_2\) because of the references to \(A\) in \(L_2\). In contrast, an edge connects the reverse node of \(C_1\) to loop node \(L_2\) because of the negative coefficient of \(j\) in the reference \(C(N - j + 1.1)\)
Algorithm: \textit{establish-data-computation-affinity}

\begin{verbatim}
begin
1. // Construct the ALAG
   Create a loop node for each parallel loop
   Create an array dimension node for every array dimension to be distributed
   // Each array node has a forward and a reverse sub-node.
   for each array dimension node \( a \)
      Construct a set of subscript expressions \( S_a \) used for the array dimension
   end for
   for each expression \( e \) in \( S_a \)
      for each loop index \( i \) in \( e \)
         if \( i \) corresponds to a loop node \( l \) then
            if the coefficient of \( i \) in \( e \) is positive then
               Add an edge between nodes \( l \) and forward node of \( a \)
            else
               Add an edge between nodes \( l \) and reverse node of \( a \)
            end if
         end if
      end for
   end for
2. // Select distribution attributes for the nodes in the ALAG
   for each loop node
      Select initial distribution attribute
   end for
   for each forward and reverse sub-nodes
      Select initial distribution attribute
   end for
   // Modify initial distribution attributes to ensure that all the connected nodes
   // have the same distribution attribute
   \textit{propagate-resolve}
3. // Select block sizes for BlockCyclic distribution attributes
   // for the nodes in the ALAG
   for each array node with BlockCyclic attribute
      Select initial block sizes
   end for
   for each node in the ALAG with BlockCyclic attribute
      Select final block sizes
   end for
4. // Assign dimensions of the processor geometry to loop nodes and array dimension nodes
   \textit{assign-proc-dim}
end
\end{verbatim}

Figure 5.9: Algorithm: \textit{establish-data-computation-affinity} (Phase 1b).
in loop L2.

Figure 5.11 shows the ALAG for running example #2. The nodes labeled \( L_1, L_1k, L_1j, L_1i, L_2i, L_2k, L_2j, \) and \( L_2i \) correspond to the parallel loops in the program. The array dimension nodes of A is selected based on the OR\_Mask of the array. The array dimension nodes of B and C in the ALAG are selected based on the AND\_Mask of the arrays. Nodes labeled \( A_1, A_2, A_3, B_2, B_3, B_4, C_2, C_3, \) and \( C_4 \) correspond to the distributed dimensions of arrays A, B, and C respectively. Edges are introduced between array dimension nodes and loop nodes based on the array references in the program, as shown.

### 5.5.2 Deriving Distribution Attributes

The derivation of distribution attributes for the parallel loops and the distributable array dimensions is accomplished in two steps. First, loop and array nodes in the ALAG are assigned *initial* distribution attributes based on the load balancing and locality considerations for each individual node. Second, *final* distribution attributes are derived using the
initial distribution attribute of a node and the attributes of all the nodes connected to it. That is, the second step modifies the initial distribution attributes so as to account for the load balancing and locality considerations for all the connected nodes. It should be noted that non-local memory accesses are minimized when all the connected nodes have the same distribution attribute and are then mapped onto the same dimension of the processor geometry.

5.5.2.1 Initial Distribution Attributes

Loop nodes are assigned initial distribution attributes mainly to balance the workload of the corresponding parallel loop, and secondarily to preserve locality of data 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.

Traditionally, the Cyclic distribution attribute has been used to balance the load. In general, the CyclicRCyclic distribution attribute balances workloads better than the Cyclic distribution attribute, because the CyclicRCyclic distribution attribute reduces the difference in the number of iterations assigned to processors. Figure 5.12 shows the difference in partitioning a triangular iteration space using the Cyclic and CyclicRCyclic distribution attributes on two processors. With the Cyclic distribution attribute, there is a workload imbalance of \( N/2 \) iterations, where \( N = 8 \) in the example. In general, when there are \( P \) processors, the Cyclic distribution attribute has a workload imbalance of \( N/P \). With the CyclicRCyclic distribution attribute, the workload imbalance can be at most \( N \mod P \) iterations. In the example, there is a workload imbalance of 4 iterations with the Cyclic distribution attribute, and there is no workload imbalance with the CyclicRCyclic distribution attribute.

However, we opt to use CyclicRCyclic only for the outermost parallel loop, to reduce the overhead associated with CyclicRCyclic. (The loop must be split into two loops when implementing CyclicRCyclic.) Hence, the default distribution attribute for a loop node corresponding to a parallel loop in a loop nest with load imbalance is Cyclic, with the exception of the outermost parallel loop.

The CyclicRCyclic or Cyclic distribution attributes also improve locality when the parallel loop iterator is a function of enclosing loop iterator(s). For example, in Figure 5.12.
Additional iterations assigned to Processor 1 because of Cyclic distribution attribute.

Figure 5.12: Example showing the effectiveness of CyclicRCyclic distribution attribute.

if the j loop is parallel and the i loop is sequential, partitioning j using the Block distribution attribute will result in poor locality across different iterations of the outer sequential loop. Hence, CyclicRCyclic distribution attribute is used to partition a loop which is a function of outer loop iterator(s) in order to improve locality. However, when the parallel loop iterator is a function of enclosing loop iterator(s) and an outer parallel loop is assigned a CyclicRCyclic distribution attribute, then the parallel loop is assigned Cyclic distribution attribute to reduce the overhead associated with CyclicRCyclic.

Loop nodes are assigned a * attribute when it is not necessary to assign Cyclic or CyclicRCyclic distribution attributes for improved load balance or improved locality. The * attribute is used as a notation to indicate that the loop node may be partitioned in any manner suitable to the connected nodes.

Array dimension nodes are assigned distribution attributes to enhance access locality for the corresponding array references. When an array is accessed in a loop with a single reference pattern where each subscript expression is a function of a single loop iterator, then there are no constraints on the selection of a distribution attribute, i.e., every possible distribution attribute is suitable for the array. Hence, array nodes introduced only because
of such references are assigned the distribution attribute: that is, the array dimension node may be partitioned in any manner suitable to connected nodes. However, if an array is referenced in an iteration of a parallel loop more than once, with the same stride of access, all elements accessed by the iteration must be co-located to enhance locality. For example, if \( A(a \cdot i) \) and \( A(a \cdot i + c) \) are two references to an array \( A \) in an iteration of a parallel loop \( i \), then both data elements accessed must be assigned to the same processor. This is achieved by assigning the corresponding array dimension nodes the \textbf{Block} distribution attribute.

When an array is accessed with differing strides as with references, \( A(a \cdot i) \) and \( A(b \cdot i) \), the two references are treated as though the references were to different arrays. In a later step of selecting initial block sizes, we consider these references together in deriving a single block size for the array.

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 \( A \) that is referenced as \( A(a \cdot i + b \cdot j) \) 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 \( A \) 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 \textbf{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 the \( j \) loop is sequential, large chunks of the array \( A \) are accessed by each processor. In this case, the array dimension node is assigned a \textbf{Block} distribution attribute. If both \( i \) and \( j \) loops are parallel, the array dimension node is assigned a \textbf{Block} distribution attribute, to favor outer loop parallelism.

The initial distribution attributes are assigned to the forward and reverse sub-nodes of an array dimension separately. Since an array can be referenced in multiple loop nests, an array dimension node may require conflicting initial distribution attributes. In such cases, a \textbf{BlockCyclic} attribute is used.

In running example \#1, the loop node \( L11 \) is assigned the \textbf{CyclicRCyclic} distribution attribute to improve data locality. Since the load is balanced in \( L2 \) and there are no implicit

\footnote{We define the array \textit{access stride} of an array dimension to be the coefficient of a parallel loop iterator in that array dimension.}
locality considerations in L2. loop node L2j is assigned the \* distribution attribute. The forward sub-node of array dimension node B1 is assigned a Block distribution attribute. Array dimension nodes A1, A2, and C1 are assigned the \* distribution attribute.

In running example #2, loop node L11 is assigned CyclicRCyclic attribute to balance the load, and L1k is assigned Cyclic attribute because of load imbalance/locality considerations in L1. All other loop nodes are assigned a \* distribution attribute, because load balancing is not an issue for these iterators. The forward sub-node of array dimension nodes B4 and A2 are assigned the Block distribution attribute. All other array dimension nodes are assigned the \* distribution attribute.

5.5.2.2 Final Distribution Attributes

The initial distribution attributes may not ensure that parallel loops and array dimensions whose subscript expressions are functions of the same parallel loop 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. This step of the CDP algorithm is accomplished by the algorithm propagate-resolve shown in Figure 5.14.

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 5.13. 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 locality of access to be maintained through the selection of an appropriate block size, and also allows the load to be balanced by distributing the blocks Cyclically. A node in the conflict resolution graph can be its own
Algorithm: *propagate-resolve*

```plaintext
begin
repeat
    // For each array node, ensure that the forward sub-node distribution attribute
    // is the complement of the reverse sub-node distribution attribute.
    1. for each array dimension node a
       D_f ← Distribution attribute of the forward sub-node
       D_r ← Distribution attribute of the reverse sub-node
       D_f ← resolve(D_f, reverse(D_r))
       D_r ← reverse(D_f)
    end for

    // Ensure that the nodes that are connected have the same distribution attribute
    2. for each loop node, forward, and reverse sub-node i
       for each loop node, forward, and reverse sub-node j connected to i
          D_i ← Distribution attribute of node i
          D_j ← Distribution attribute of node j
          D_i ← resolve(D_i, D_j)
       end for
    end for

    until distribution attributes of the nodes of ALAG do not change
end
```

Figure 5.14: Algorithm: *propagate-resolve*.

least common ancestor. A distribution attribute in conflict with the * attribute resolves to the distribution attribute itself.

For each array dimension node, the distribution attributes of the forward and reverse sub-nodes are first made consistent (using the conflict resolution graph), by having the distribution attribute of the forward sub-node be the reverse of the distribution attribute of the reverse sub-node. 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.

A special case occurs when all the nodes that are connected are assigned a * distribution attribute at the end of the algorithm *propagate-resolve*. In this case, an arbitrary forward array sub-node with a * distribution attribute is assigned a Block distribution attribute and the *propagate-resolve* algorithm is applied again. This step can be viewed as assigning the Block distribution attribute to parallel loops and array dimensions that do not have any preferred distribution attributes based on load-balancing and locality considerations.

Figure 5.15 shows the ALAG with the final distribution attributes for running example
The distribution attribute of $B_1$ is forced to be $\text{BlockCyclicRBlockCyclic(BCRBC)}$, resolving the distribution attribute $\text{Block}$ and the $\text{CyclicRCyclic}$ distribution attribute of loop node $L_1$ connected to it. The nodes $L_1$, $L_2$, $A_1$, $A_2$, $B_1$, and $C_1$ are partitioned using the $\text{BCRBC}$ distribution attribute when consensus is reached.

Figure 5.16 shows the ALAG with the final distribution attributes for running example #2. When consensus is reached, nodes $L_1$, $L_2$, $L_3$, $L_4$, $A_1$, $A_2$, $B_1$, $B_2$, $C_1$, $C_2$, and $C_3$ are partitioned using the $\text{BCRBC}$ distribution attribute, while the nodes $L_1$, $L_2$, $L_3$, $A_1$, $A_2$, $B_2$, and $C_2$ are partitioned using the $\text{Block}$ distribution attribute.
5.5.3 Selecting Block Sizes

The next step of the CDP algorithm is to select block sizes for the distribution attributes assigned to the nodes in the ALAG. The block sizes for the Cyclic, RCyclic, and CyclicRCyclic distribution attributes are set to 1. The block sizes for the Block, RBlock, and BlockRBlock distribution attributes can be determined in a straightforward manner after we determine the number of processors assigned to each of the dimensions of the processor geometry.

In this section, we describe a technique to select block sizes for the nodes of the ALAG that have the remaining distribution attributes, namely BlockCyclic, RBlockCyclic, or BlockCyclicRBlockCyclic. For these distribution attributes, we select the block sizes in two steps. First, we select initial block sizes for the array dimension nodes in the ALAG based on the array subscript expressions in each loop nest in isolation. An initial block size of 1 is selected for all the loop nodes in the ALAG. Subsequently, we use the algorithm select-block-size of Figure 5.17 to select the final block sizes for all the nodes in the ALAG.

5.5.3.1 Initial Block Size Selection

The array dimension nodes are assigned initial block sizes to ensure that a certain number of elements are co-located. In particular, we select an initial block size along a dimension of an array based on the minimum number of elements along the array dimension that must be co-located for one iteration of a parallel loop.

First, we consider a simple case where an array has a single reference in a loop nest. The initial block size along a dimension of such an array is determined by the access stride along that dimension. For example, consider a one dimensional array A, with an array reference A(c * i) and a parallel loop i. The array elements are accessed in a stride of c. Therefore, a block size of c ensures that each parallel iteration of i will access an array element from different partitions.\(^5\) Note that, in this case, there are no other references that enforce the need for co-locating all array elements accessed by a parallel loop iterator together in the same block. In addition, with a single reference, the offsets in the subscript expressions do not affect the selection of initial block sizes.

When there are two references to an array, the initial block size is not only dependent on the array access stride but also on the offsets in the array subscript expressions. It is

\(^5\)When an array subscript expression is a function of multiple loop iterators, we use the array access stride along the outermost parallel loop iterator as the block size for the array.
necessary to consider the offsets in array subscript expressions only if the two references have the same access stride. For example, consider two references \( A(c \cdot i) \) and \( A(c \cdot i + d) \) to an array \( A \) in a parallel loop \( i \). The array dimension must be partitioned in blocks of at least \((d + 1)\) elements to ensure that one iteration of the parallel loop will access both elements of the array from the same partition. However, the stride of \( c \) requires an initial block size of \( c \). Therefore, the initial block size is chosen to be the least common multiple (lcm) of the stride and the offset. In this example, an initial block size of \( \text{lcm}(c \cdot (d + 1)) \) is chosen for the array dimension node of array \( A \). Note that, when the access strides in the two references are not equal, there will be non-local accesses irrespective of the block size chosen. In such cases, we choose the maximum of the two strides as the initial block size of the array dimension node.

As a general rule, when there are more than two references to an array with the same stride, the initial block size is chosen by first selecting the block size for a pair of array references at a time. The initial block size for the array dimension is selected as the least common multiple of the block sizes selected for each pair of references. Thus, the initial block size selected ensures co-locality of array elements accessed with the same access stride. With differing strides of access, only the references with the largest access stride are considered in the block size selection.

A special case occurs when the partitioned array dimension is the cache line dimension. In this case, the initial block size is chosen to be the number of array elements in a cache line, in order to avoid false sharing.

Finally, the initial block size for an array dimension is chosen to be the least common multiple of the initial block sizes selected for the array dimension in each of the loop nests in the entire program.

In running example #1, \( B_1 \) requires a block size of 2, based on its offset and stride requirements. To avoid false sharing, the block sizes of \( A_1 \), \( B_1 \), and \( C_1 \) are selected as \( \text{CL} \) (the number of elements that can be in the cache line size of the machine). Since the cache line is a multiple of 2, \( \text{lcm}(2, \text{CL}) = \text{CL} \) is chosen as the block size for the array \( B_1 \).

In running example #2, \( B_4 \) requires a block size of 3, based on the offset requirements in loop nest \( L_1 \).
Algorithm: select-block-size

// All parallel loops in the program are assumed to be normalized
// with a step of 1.
begin
repeat
1. for each loop node $l \in V$
   for each array dimension node $a$ connected to $l$
      $b_t \leftarrow \text{lcm}(b_t, \{b_a/\text{Stride}(a,l)\})$
   end for
2. for each array dimension node $a$ connected to $l$
   $b_a \leftarrow \text{lcm}(b_a, \{b_l \cdot \text{Stride}(a,l)\})$
   end for
end for
until no block size changes in the nodes of ALAC
end

Figure 5.17: Algorithm: select-block-size.

5.5.4 Final Block Size Selection

The initial block sizes for the nodes in the ALAC are used to select final block sizes for the nodes. Algorithm select-block-size, shown in Figure 5.17, selects final block sizes for nodes in the ALAC. This algorithm repeatedly updates block sizes of loop nodes, based on the block sizes of connected array nodes. The block sizes of array dimension nodes are in turn updated, based on the block sizes of connected loop nodes, until there are no changes in the block size of the nodes in the ALAC. The algorithm will terminate because there is a number that is the least common multiple of all possible block sizes.

Note that in some rare cases, the block size of a parallel loop may turn out to be very large, and it may hide the parallelism in the program. Similarly, if the block size of an array dimension is very large, the array dimension may not be partitioned among processors. As an optimization, we limit the maximum block sizes to be the block size for the Block distribution attribute, assuming that the parallel loop or the array dimension is mapped onto all the processors in the system.

In running example #1, the array dimension nodes $A_l, B_1,$ and $C_1$ have a block size of $CL$, the number of elements contained in the cache line of the machine. In both loop nests, the stride along all the array dimensions is 1. Applying the select-block-size algorithm selects a block size of $CL$ for all the nodes in the ALAC. In running example #2, a block size of 3 is chosen for all the nodes with the BCRBC distribution attribute.
5.5.5 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 the 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.

The processor geometry dimensions are first assigned to loop nodes and are then propagated to the connected array dimension nodes in the ALAG. The number of parallel loops in a loop nest may be equal, greater than, or less than the dimensionality of the processor geometry. We first assign processor dimensions to loop nests that have parallel loops equal to the processor geometry dimensionality. This is a simple case, because we do not have to select the parallel loop that will not be assigned a processor dimension, or the processor dimension that will not be assigned to a parallel loop.

In the loop nests in which the number of parallel loops is equal to the dimensionality of the processor geometry, all parallel loops will indeed execute in parallel. These loop nests are first assigned processor geometry dimensions. We start the processor dimension assignment with one such loop nest. 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, depending on the number of loop nodes connected to it. Processor geometry dimensions are assigned from outer to inner, because this order exploits outer loop parallelism and avoids inner parallelism.

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.

When the array dimension nodes connected to a loop node have more than one processor geometry dimension, then the loop node is assigned a processor geometry dimension that is assigned to the maximal number of the connected array dimension nodes.6

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, and also from outermost loop to innermost loop in a loop nest.

Figure 5.18 shows the processor geometry dimension assignments for running example #1. In the example, the processor geometry is a linear grid. The processor geometry dimension available for assignment is {1}. In the ALAG, loop node L1 is assigned processor dimension 1. Consequently, the array dimension nodes A1 and B1 connected to L1 are also assigned the processor geometry dimension 1. In the second loop nest L2, the loop node L2j is assigned the processor geometry dimension 1, since the array dimension node B1 connected to it is assigned the dimension. Array dimension nodes A2 and C1 that are connected to L2j are also assigned the processor geometry dimension 1. For arrays B and C, no partitioning conflicts exist and static partitions are determined. However, there is a sequential conflict

6When this does not resolve conflicts, the assignment of processor geometry dimension is delayed until after all other loop nodes in the loop nest under consideration are assigned processor geometry dimensions. As a last resort, we resolve the conflict by choosing one of the possible choices arbitrarily.
Figure 5.19: Processor geometry assignment for running example #2.

for the array $A$.

Figure 5.19 shows the processor geometry assignment for running example #2. Loop nodes $L_1$, $L_{1k}$, and $L_{1j}$ are assigned geometry dimensions 3, 2, and 1 respectively. These assignments are propagated to array dimension nodes $A_2$, $A_3$, $B_2$, $B_3$, and $B_4$. Loop nodes $L_{2l}$, $L_{2k}$, and $L_{2j}$ are assigned dimensions 3, 2, and 1 respectively based on the assignments of $B_4$, $B_3$, and $B_2$. Consequently, the array dimension nodes $C_4$, $C_3$, and $C_2$ also are assigned geometry dimensions 3, 2, and 1 respectively. Because $A_3$ is connected to both $L_{1k}$ and $L_{2l}$, it gets the geometry assignment of both these loop nodes i.e., {2, 3}. Loop nodes $L_{1l}$ and $L_{2l}$ are mapped onto the non-existent processor geometry dimension {0}. Similarly, $A_1$ is also mapped onto processor geometry dimension {0}. As in running example #1, no partitioning conflicts exist for arrays $B$ and $C$, and static partitions are determined. However, there is a distribution conflict for the array $A$.

5.6 Dynamic Array Partitions

The partitions derived so far by the algorithm attempt to preserve the 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
Algorithm: assign-proc-dim
begin
  for each loop nest L
  // where L is ordered according to the number of parallel loops in its body
  // such that L in which the number of parallel loops equal the processor geometry are first.
  // Both forward and reverse sub-node of an array dimension node are assigned
  // the same processor geometry dimension. Hence, we do not distinguish forward
  // and reverse in processor geometry dimension assignment.
  // PD is the set of all possible possible processor geometry dimensions.
  PD ← { 1, 2, ... , k } // where k is the processor geometry dimensionality
  for each loop l ∈ L
    for each array dimension node a connected to l
      if a is assigned processor geometry dimension d then
        assign l processor geometry dimension d: L ← L − l
        PD ← PD − d
        end if
    end for
    propagate(d,l)
  end for
  for each l ∈ L
    if PD is not empty then
      assign l processor geometry dimension d from PD
      L ← L − l: PD ← PD − d
    end if
    propagate(d,l)
  end for
end

Subprogram: propagate(d,l)
begin
  for each node n
    if n is an array dimension node with an edge from l then
      add d to n’s list of dimensions
    end if
  end for
end

Figure 5.20: Algorithm: assign-proc-dim.
of the processor geometry exists, static partitions make accesses to the arrays local across all loop nests in the program. For arrays in which a one-to-one mapping cannot be used, remote memory accesses occur in some or all of the loop nests. The geometry dimension assignment was made for such arrays in order to satisfy multiple locality requirements in the different loop nests. A parallel loop is partitioned along only a single dimension of the processor geometry. Hence, non-local memory accesses will occur for an array if the subscript expression along a dimension contains the parallel loop iterator and the array dimension is partitioned along multiple dimensions of the processor geometry. The CDP algorithm ensures that one of the dimensions of the processor geometry onto which the array dimension is mapped will be same as the processor geometry dimension that the parallel loop is mapped to.

Similarly, partitioning multiple dimensions of the array along a single dimension of the processor geometry also results in non-local accesses. If the subscript expression along one of the dimensions of such an array is a loop iterator that corresponds to a parallel loop mapped onto the same processor dimension, then non-local memory accesses will occur because of the latin-square-type partitioning of the array.

Hence, the partitions of arrays that do not have a one-to-one mapping are re-evaluated to determine if dynamic partitioning may result in better program performance. In addition, we have to only consider the loop nests in which the array without the one-to-one mapping is referenced.

The assignment of geometry dimensions to the array dimension is explored in selecting dynamic 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 running example #1, the 2-dimensional array A is partitioned on a linear processor geometry using $A(\text{BCRBC}\{1\}, \text{BCRBC}\{1\})$, and its partitioning should be re-evaluated because two dimensions of the array are mapped onto a single dimension of the processor geometry. The set of all possible data partitions includes $A(\text{BCRBC}\{0\}, \text{BCRBC}\{0\})$, which replicates the

---

7 This is a result of the distribution conflict that we described in Section 5.3.
8 This is a result of the sequential conflict that we described in Section 5.3.
9 When the subscript expressions along the different dimensions of the array are functions of a loop iterator corresponding to a parallel loop that is mapped onto the same processor dimension as the array, then the partitioning of the array will result not only in non-local accesses but also in memory contention. This is the parallelism conflict described in Section 5.3.
array: $A(BCRBC\{1\}, BCRBC\{0\})$, which results in local accesses to the array in the first loop nest; $A(BCRBC\{1\}, BCRBC\{0\})$, which results in local accesses to the array in the second loop nest; and $A(BCRBC\{1\}, BCRBC\{1\})$, which results in non-local accesses in both loops.

When sequential conflict exists for an array in a loop nest, two computation partitioning choices must be evaluated in addition to the data partitioning choices: (i) the computation partition in which the candidate data partition incurs contention overhead. This candidate data partition is same as the partition that was determined using the ALAG, where only parallel loops are partitioned; (ii) a pipelined computation partition that incurs synchronization overhead. This computation partition is a result of partitioning a sequential loop that is the source of the sequential conflict. The computation partition changes only when pipelining is chosen for a loop nest when evaluating the candidate data partitions for an array.

In running example #2. 2 dimensions of the three dimensional array $A$ are partitioned on a three dimensional processor geometry using $A(*.Block\{1\}, BCRBC\{2,3\})$; hence, its partitioning should be re-evaluated. In this example, the first dimension is not partitioned, the second dimension is mapped onto the first dimension of the processor geometry, and the third dimension of the array is mapped onto both second and third dimensions of the processor geometry. The set of possible data partitions includes $A(*.Block\{0\}, BCRBC\{0\})$, which replicates the array completely; $A(*.Block\{1\}, BCRBC\{0\})$, $A(*.Block\{0\}, BCRBC\{2\})$. $A(*.Block\{0\}, BCRBC\{3\})$, and $A(*.Block\{0\}, BCRBC\{2,3\})$, which partially replicate the array: $A(*.Block\{1\}, BCRBC\{2\})$, which results in local accesses to the array in the first loop nest, but not the second; $A(*.Block\{1\}, BCRBC\{3\})$, 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. Although the second dimension of the array does not have any conflicts and is mapped onto a single dimension \{1\} of the processor geometry. we consider that dimension too in the re-evaluation, in order to evaluate the benefit of replication. Note that the group of data partitions that are being re-evaluated for the array is not the exhaustive set of possible data partitions, but a subset of data partitions obtained by exploiting the affinity relationship that has been established using data-computation affinity. For example, by referring to data-computation affinity, we neither partition the first dimension of the array, nor consider data partitions like $A(*.Block\{2\}, BCRBC\{3\})$.

We define a cost matrix for an array, where each element of the matrix represents the
cost of accessing the array in a loop nest using a particular candidate data partitions. The cost matrix for an array is a matrix with as many rows as the number of loop nests in the program, and as many columns as the number of candidate data partitions selected for re-evaluation. An entry in the cost matrix, \( C(l_i, d_j) \), corresponding to a loop nest \( l_i \) and a data partition \( d_j \), contains the cost of accessing the array with the data partition \( d_j \) and the computation partition derived using the ALAG for the loop nest.

In order to determine the cost of accessing the elements of an array in a loop nest with a given candidate partition, machine-specific information, such as cache, local, and remote memory accesses latencies, cost of synchronization, penalty for memory contention, and cost of redistribution is required. The cost estimation heuristics used by the CDP algorithm are described in Chapter 6 (algorithm ArrayAccessCost on page 91). For the ease of presentation, in this chapter the algorithm for cost estimation is treated as a “black box”, assumed to provide a metric of the estimated execution time of an input loop nest with a candidate data partition.

The cost matrix for the array \( A \) in running example \#1 on Hector and Convex multiprocessors is shown in Figures 5.21 and 5.22, respectively (for a problem size of \( N=256 \)). The entries with an “\(^*\)” in the cost matrix correspond to the maximum of the cost of accessing the array with contention or with pipelined computation partition incurring synchronization overhead. The cost matrix for the array \( A \) in running example \#2 is similar and is not shown here.
In order to determine the points in the program where an array repartition should occur, we use a *reaching matrix* for the array, which is constructed using flow graph analysis [39] (see Section 6.4.3.2). The reaching matrix of an array is a square matrix with a dimension equal to the number of loop nests in the program. An element, \( \text{Reach}(k, l) \), of the reaching matrix of an array \( A \) is 1 if loop nest \( k \) follows loop nest \( l \) in the program execution and both loop nests \( k \) and \( l \) access the array \( A \); otherwise, the element is 0. Figure 5.23 shows the reaching matrix for array \( A \) in running example #1. Since the running example has only two loop nests, and control flows from loop nest 1 to 2, there is only one non-zero entry in the reaching matrix.

The candidate data partitions of an array are searched to determine the data partitions that minimize the overall execution time. The objective is to derive for each loop nest \( i \), one data partition \( d_j \) for the array such that

\[
\sum_{i=1}^{n} C(i, d_j) + \sum_{m=1}^{n} \sum_{p=1}^{n} \text{Reach}(p, m) \times \text{RP}(D(m), D(p))
\]

is minimized, where \( D(m) \) corresponds to the data partitioning \( d_j \) in loop nest \( m \) and \( \text{RP}(d_i, d_j) \) is the cost of changing the partitions of the array from \( d_i \) to \( d_j \). The cost of repartitioning is 0 when the data partitions are the same. Bixby et al. have shown that this optimization problem is NP-Complete [26]. However, since only the arrays that lack one-to-one mapping between array dimensions and processor geometry dimensions are considered, and each array is considered by itself, the size of the search space is limited in most applications, allowing an exhaustive search to be practical. A depth-first search with pruning is used to further reduce the size of the search space. The cost of executing the program with the best static data partition is used to prune the depth-first search. The algorithm *finalize-data-partitions* accomplishes the depth-first search to determine the final data partition for each loop nest.

Figure 5.26 shows the search space of possible data partitions for array \( A \) in running
Algorithm: finalize-data-partitions(A)

// Returns candidate data partition in each loop nest that maximizes
// program performance
begin
1. for each candidate data partition $i$ for array $A$
   for each loop nest $j$
     $C(i, j) \leftarrow$ ArrayAccessCost($A, j, i$)
   end for
   $C_i \leftarrow \sum_j C(i, j)$
 end for
2. $BestStaticCost \leftarrow \min(C_i)$
3. for each loop nest $j$
   $MinCost_j \leftarrow \min(C(i, j))$
 end for
4. $BestDynamicCost \leftarrow \sum_j MinCost_j + (Min.Redist.Cost)$
5. if ($BestDynamicCost < BestStaticCost$) then
   // Repartitioning could be beneficial
6.   for each candidate data partition $j$
     $PATH \leftarrow \{(1, j)\}$
     $Cost \leftarrow C(1, j)$
     $BestCost \leftarrow \infty$
     $BestPath \leftarrow NULL$
     DFS_Repartition(1, j, Cost, PATH, BestCost, BestPath)
 end for
7. if ($BestCost < BestStaticCost$) then
   $PATH$ contains the loop nests and the data partitions for the array $A$
 else
   Partition for array $A \leftarrow i$ with $\min(C_i)$
 end if
else
   Partition for array $A \leftarrow i$ with $\min(C_i)$
end if
end

Figure 5.24: Algorithm: finalize-data-partitions (Phase 2a).
DFS_Repartition(PrevNest, PrevDimAsst, Cost, PATH, BestCost, BestPath)
begin
    if (Cost >= BestStaticCost or Cost >= BestCost) then
        return // Pruned Search
    end if
    if (PATH contains all the loop nests) then
        Update Cost ensuring all 1's in reaching matrix are accounted
        NewCost ← Cost + RepartitioningCost(l, j)
        if (NewCost < BestCost) then
            BestCost ← NewCost
            BestPath ← PATH
        end if
    else
        l ← loop nest not in PATH
        for each candidate data partition CurrAsst
            PATH ← PATH + (l, CurrAsst)
            NewCost ← Cost + C(l, CurrAsst) + RP(PrevDimAsst, CurrAsst)
            DFS_Repartition(l, CurrAsst, NewCost, PATH, BestCost, BestPath)
        end for
    end if
end

Figure 5.25: Subprogram DFS_Repartition.

eample #1. There are four possible data partitioning choices for the array in each of the two loop nests. Hence, we must consider 16 possible data partitioning choices for the program. Running example #2 will also result in a similar search space for array A.

5.7 Determining Final Computation Partitions

Computation partitions have been determined by the data computation affinity phase of the CDP algorithm using the ALAG. Subsequently, for each array, the finalize-array-partitions algorithm determines a final set of data partitions. In this process, the algorithm also determines whether a loop nest should be pipelined or not. Pipelining results in partitioning a sequential loop instead of a parallel loop in the loop nest. Similarly, array replication also leads to a change in computation partitioning for the loop nests in which the replicated array is written. Hence, pipelining and array replication lead to a change in the computation partition, and final computation partitions must be determined.

If the re-evaluation of data partitions for all the arrays in a loop nest results in a pipelined
Figure 5.26: Search space of possible partitions for array A in running example #1.

Algorithm: finalize-computation-partitions

begin
1. // Fix up computation partitions due to the selection of pipelining.
P ← number of arrays referenced in l selecting pipelining
N ← number of arrays referenced in l
if (P > N - P) then
    Select pipelined computation partition for l
end if

2. // Fix up computation partitions due to replication of arrays.
for each array a referenced in the loop nest l
    D_a ← data partition of array a in l
    if (D_a is replicated and a is written in l) then
        Update computation partitions for l
    end if
end for end

Figure 5.27: Algorithm: finalize-computation-partitions (Phase 2b).
Algorithm: processor-geometry-mapping

begin
1. // Select number of processors along each dimension of the processor geometry
   \( P_i \) ← possible processor geometries for a given number of processors \( P \)
   for each processor geometry \( P_i \)
       Estimate the cost of executing the program with chosen data and computation partitions
   end for
   Select a processor geometry with minimal cost
2. // Map virtual processors \( n \)-dimensional geometry onto linear physical processor geometry
   \((p_1, p_2, \ldots, p_n)\) maps onto \( \sum_{i=1}^{n} p_i \prod_{j=1}^{i-1} P_j \)
end

Figure 5.28: Algorithm: processor-geometry-mapping (Phase 3).

Computation partition for the loop nest, then the loop nest is pipelined. However, a conflict arises when a pipelined computation partition is chosen for a loop nest for some arrays and a partition with no pipelining is chosen for others. In this case of conflict within a loop nest, the decision to pipeline the loop nest is based on whether the majority of arrays require a pipelined partition for the loop nest.

Computation partition is also altered when a replicated array is written in a loop nest. In this case, all the processors must update their local copies. The computation partition is altered to contain the entire iteration space of a loop nest, so that the array can be computed by all the processors. However, we require other techniques when a loop nest accesses some arrays that are replicated and others that are partitioned. When legal, loop distribution \([40]\) can be used to split the loop nest into two, with one loop nest containing replicated array writes, and the other loop nest containing writes to the partitioned arrays. When loop distribution is not legal, we use conditional guards to protect the partitioned array writes, adhering to the owner-computes rule.\(^\text{10}\)

5.8 Processor Geometry Mapping

The final step in 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 two dimensional processor geometries are \((1, 8), (2,\)

\(^{10}\)The prototype compiler does not implement loop distribution.
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 processor geometry. The geometry 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 because of a limited cache size, are discussed in Chapter 6.

The physical processors are viewed as a linear array, and virtual processor \((p_1, p_2, \ldots, p_n)\) is assigned to the physical processor numbered \(\sum_{i=1}^{n} p_i \prod_{j=1}^{i-1} P_j\). Thus, for a two-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 nearby processors is lower.

5.9 Summary

We have described the CDP algorithm for deriving computation and data partitions that take into account the shared memory effects on SSMMs. The computational complexity of the algorithm is reduced by restricting the re-evaluation of partitions to those arrays that may benefit from such re-evaluation, and by selecting a dimensionality of the processor geometry based on the parallelism in the program. We experimentally evaluate the computational efficiency, and present the effectiveness, of the data and computation partitions derived by the CDP algorithm in Chapter 7. In the next chapter, we describe the static estimation of a metric for the execution time of a loop nest used by the CDP algorithm.
The CDP algorithm described in Chapter 5 requires a static estimate of execution times of input loop nests in order to compare alternative candidate computation and data partitions. Determining the absolute execution time of a loop nest is non-trivial, given the need for a detailed model of machine architecture and of compiler optimizations. Therefore, our objective in this chapter is to approximate the execution time of a loop nest, for the purpose of comparing the relative effectiveness of candidate computation and data partitions. The metric that we use to express the execution time is the approximate number of processor cycles. We first show how to derive approximate execution time for the target machine, assuming infinite caches. Subsequently, we take into account the cache size on the target machine.

We present the estimation of the approximate execution time of a loop nest in five parts. First, we identify the parameters that are most relevant in estimating the execution time of a loop nest. We classify these parameters as machine-dependent and program-dependent parameters. Second, we provide the mathematical formulation for obtaining the execution time of a given loop nest. Third, we describe techniques that are used to derive the machine- and program-dependent parameters. Fourth, we show how these parameters are combined to obtain the execution time of a given loop nest. Finally, we describe how the execution time is affected by the actual size of the cache on the target machine.

6.1 Performance Estimation Parameters

The parameters for estimating the execution time of a loop nest are summarized in Table 6.1. These parameters are classified as either machine-dependent or program-dependent. Machine-dependent parameters depend on the architecture of the target machine and are
Table 6.1: Parameters for performance estimation.

<table>
<thead>
<tr>
<th>Machine dependent parameters</th>
<th>$C_i$</th>
<th>cache access latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$M_l$</td>
<td>local memory access latency</td>
</tr>
<tr>
<td></td>
<td>$M_r$</td>
<td>remote memory access latency</td>
</tr>
<tr>
<td></td>
<td>$c_f$</td>
<td>memory contention factor</td>
</tr>
<tr>
<td></td>
<td>$S$</td>
<td>cost of synchronization</td>
</tr>
<tr>
<td></td>
<td>$OCP$</td>
<td>overhead for partitioning the cache line dimension</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program dependent parameters</th>
<th>$N_c$</th>
<th>number of elements accessed from the cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$N_l$</td>
<td>number of cache lines accessed from local memory</td>
</tr>
<tr>
<td></td>
<td>$N_r$</td>
<td>number of cache lines accessed from remote memory</td>
</tr>
<tr>
<td></td>
<td>$aw$</td>
<td>number of arrays written in the loop nest</td>
</tr>
<tr>
<td></td>
<td>$Ns$</td>
<td>number of synchronization operations if the loop nest is pipelined</td>
</tr>
</tbody>
</table>

The first machine-dependent parameter is the cache access latency denoted by $C_i$. It is the number of processor cycles taken to load an array element from the cache into a register of a processor on the target machine. The next two machine-dependent parameters are the local memory access latency, denoted by $M_l$, and the remote memory access latency, denoted by $M_r$. They are the number of processor cycles taken in the absence of contention to load a cache line into the processor's cache from the local and from the remote memories respectively.

The memory contention factor, denoted by $c_f$, is used to model the percentage increase in local and remote memory access latencies due to contention. When candidate data and computation partitions do not result in contended memory accesses, the memory contention factor is 1. The cost of synchronization, $S$, is the number of processor cycles taken for $P$ processors to execute one barrier synchronization operation.

The $OCP$ factor is used to model the percentage increase in the latencies for accessing the cache, local, and remote memories when the cache line dimension of an array is parti-
tioned. Candidate data partitions that partition the array along the cache line dimension incur an overhead in accessing the elements of the array from the different levels of memory hierarchy. This is because implementing these data partitions on SSMMs artificially increases the number of array dimensions by 1, thus increasing the index computations for each array reference.

6.1.2 Program-Dependent Parameters

The program-dependent parameters are the number of elements accessed from the cache. $N_C$: the number of cache lines accessed from the local memory. $N_I$: the number of cache lines accessed from the remote memory. $N_v$: the number of arrays that are written in a loop nest. $aw$: and the number of synchronization operations that occur in a loop nest. $Vs$. The parameters $N_C$, $N_I$, and $N_v$ are computed separately for each array that is referenced in the given loop nest. $N_I$ and $N_v$ are the numbers of cache lines that will be transferred from the local and remote memory, respectively, assuming an infinite cache. $N_C$ is the number of elements of the array that will be accessed from the cache due to spatial and temporal locality. The estimation of $N_C$, $N_I$, and $N_v$ requires additional compiler analysis techniques that will be described in the following sections.

The number of arrays that are written, $aw$, in the given loop nest is important in estimating the synchronization overhead, as will be seen later in the model formulation.

The number of synchronization operations, $Vs$, is the number of barriers that occur when the given loop nest is pipelined. Irrespective of which candidate data partition is used, sequential loops in the loop nest give rise to a number of synchronization operations, even when a loop nest is not pipelined. Consequently, the synchronization operations that are introduced because of sequential loops are ignored, i.e., the number of synchronization operations is considered to be 0 when a loop nest is not pipelined.

6.2 Model Formulation

A block diagram of the performance estimator is shown in Figure 6.1. The inputs to the performance estimator are the machine- and program-dependent parameters. The output of the performance estimator is a metric of the estimated execution time of the loop nest in

---

1 False sharing is minimized by the selection of an appropriate block size.
2 The implementation of data partitions is explained in Appendix A.
terms of the number of processor cycles. In this section, we mathematically formulate the performance estimator.

We split the execution time of a loop nest into two components, namely the computation component and the memory access component.\(^3\) We assume that the computation in the loop nest can be divided equally among the processors for all the data and computation partitions.\(^4\) Hence, the computation time is immaterial when the candidate computation and data partitions are compared. However, with certain candidate data partitions, the processors require additional execution time due to synchronizations needed for the partitions. Therefore, only the increase in computation time due to synchronization overhead is included in the execution time.

The memory access component includes the time to access scalar data and array elements that are referenced in the loop nest. However, we ignore the cost of accessing scalar data in our model, because most of these tend to be either in registers or cache. Hence, the primary contributor to the memory access component in a loop nest are array references. If there are \(m\) different arrays that are accessed in a loop nest, the estimated memory access time, \(T_L\), can be written as:

\[
T_L = \sum_{k=1}^{m} \text{AAC}(k)
\]  

---

\(^3\)Although today's super-scalar processors allow the memory operations to occur concurrently with the arithmetic operations, we do not account for this overlap.

\(^4\)When a replicated array is written in a loop nest, then all the processors redundantly perform additional computations that write to the replicated array. However, the increase in execution time due to additional computations can be ignored, when comparing replicated data partitions with non-replicated data partitions, because the number of memory accesses that occur for the replicated array will be higher.
where $AAC'(k)$ represents the cost of accessing the elements of array $k$ that are referenced in the loop nest.

The array access cost for a given array can be expressed as a sum of the costs of accessing array elements from the cache, local, and remote memories:

$$AAC'(k) = N_c \cdot C_l + N_l \cdot M_l + N_r \cdot M_r$$

This estimate of array access cost must be adjusted to account for contention, synchronization, and cache line partitioning. The formulation of the CDP algorithm requires that the cost of synchronization be expressed per array. Hence, all the arrays that are written in a pipelined loop nest share the synchronization overhead equally. The synchronization overhead for the read-only arrays in the loop nest are not included, because they can be replicated. Hence, the access time of an array, including these overheads, can be expressed as:

$$AAC'(k) = \{N_c \cdot C_l + (N_l \cdot M_l + N_r \cdot M_r) \cdot cf\} \cdot OCP + N_s \cdot S/aw$$

### 6.3 Estimating Machine-Dependent Parameters

The number of processor cycles for cache, local, and remote memory access latencies is obtained from the architectural specification of the target machine. For example, the latency for accessing data from the cache, local, and remote memories on the Convex Exemplar multiprocessor are 1.50, and 200 processor cycles respectively.

When available, we use analytical models of the target machine for estimating the machine-dependent parameters. In cases where such analytical models are not available, we use a “training set”-based approach to estimate the machine-dependent parameters. A “training set” is a collection of kernel loop nests designed to measure machine-dependent parameters. Balasundaram et. al. introduced and used this approach to model distributed memory machines, and showed that it is both accurate and portable across different machines [41]. The set of kernels we use are included in Appendix B.

The memory contention factor for a multiprocessor is determined by measuring the performance of training set loop nests. Recall that contention occurs when all processors access the same memory module. Since, the effect of contention is highly machine-dependent, analytical models that model contention tend to be complex. Hence, to estimate the memory
correction factor, we use a training set that consists of two loop nests. In the first loop nest, all the processors access a certain number of array elements from the same memory module. In the second loop nest, all the processors access the same number of array elements as in the first loop, but from different remote memory modules. The memory contention factor, \( cf \), is \( t_1/t_2 \), which is the ratio of the execution time of the first loop nest, \( t_1 \), and the execution time of the second loop nest, \( t_2 \).

Unlike memory contention, the cost of synchronization can be analytically modelled for most machines. For example, the cost of a barrier synchronization on the Convex SPP1000 is approximately \((8 + P) \mu\sec \) [42], where \( P \) is the number of synchronizing processors. In the absence of analytical models, the synchronization cost for a given number of processors can also be estimated by using training set loop nests. The training set for estimating the cost of synchronization has two loop nests. In the first loop nest, a loop nest is pipelined. The second loop nest is the same as the first loop nest, but without all the synchronization operations. The cost of synchronization, \( S \), is \((t_1 - t_2)/N_s \), where \( N_s \) is the number of synchronizations that were performed in the first loop nest, and \( t_1 \) and \( t_2 \) are the execution times of the first and second loop nests, respectively.

The \( OCP \) factor is also determined by using training set loop nests. The training set loop nests are executed on a single processor to eliminate the overheads due to contention and synchronization. The training set consists of two loop nests in which a 2-dimensional array is accessed. The cache line dimension is partitioned in the first loop nest, and the second dimension is partitioned in the second loop nest. The \( OCP \) factor is \( t_1/t_2 \), which is the ratio of the execution time of the first loop nest, \( t_1 \), and the execution time of the second loop nest, \( t_2 \).

The machine-dependent parameters used in modeling the Hector and Convex multiprocessors with 16 processors are shown in Figure 6.2.

<table>
<thead>
<tr>
<th>Multiprocessor</th>
<th>( C_f ) (cycles)</th>
<th>( M_f ) (cycles)</th>
<th>( M_r ) (cycles)</th>
<th>( S ) (cycles)</th>
<th>( cf )</th>
<th>( OCP )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hector</td>
<td>1</td>
<td>10</td>
<td>17</td>
<td>1600</td>
<td>2</td>
<td>1.1</td>
</tr>
<tr>
<td>Convex</td>
<td>1</td>
<td>50</td>
<td>200</td>
<td>2400</td>
<td>1.2</td>
<td>1.4</td>
</tr>
</tbody>
</table>
6.4 Estimating Program-Dependent Parameters

In order to estimate the execution time of a loop nest, the program-dependent parameters must also be estimated. Of the program-dependent parameters, it is straightforward to determine the number of arrays written in the loop nest, \textit{aw}, and the number of synchronization operations, \textit{Vs}. Depending on the computation and data partitions chosen for the loop nest, and the loop bounds for each of the loops in the loop nest, the number of synchronization operations that occur when the loop nest is pipelined can also easily be determined. The next task is to estimate \textit{Ve}, \textit{Vf}, and \textit{Vr}. In order to estimate these parameters, we have to (i) summarize sections of the arrays accessed in the loop nest, and (ii) determine if array elements are accessed from the cache.

In this section we describe (i) how we summarize the sections of the arrays that are accessed in a loop nest, (ii) how the number of cache lines accessed from local and remote memory is estimated, and (iii) the \textit{Array Flow Graph} (AFG) that is used to estimate the number of elements accessed from the cache across loop nests.

6.4.1 Estimating Array Access Ranges

We use \textit{bounded regular sections} \cite{14} to summarize the array access information. A bounded regular section is denoted using triplets along every dimension of an array access being summarized, where a triplet consists of a lower bound, an upper bound, and a stride. Using the triplet notation to summarize accesses along each dimension of an array, we can represent dense regions such as columns, rows, and blocks. The stride can be modified suitably to represent certain sparse regions.

The following example illustrates the use of \textit{bounded regular sections} to represent array access regions.

\begin{verbatim}
for i = L,, U,, S,
    B(2 * i + 1) = A(i) + A(i + 1)
end for
\end{verbatim}

It is convenient to represent ranges of the loop iterators using triplet notation as well. The range of \textit{i}, the loop iterator, is represented as the triplet \([L_i:U_i:S_i]\), where \(L_i\) is the lower bound, \(U_i\) is the upper bound, and \(S_i\) is the stride. The array section is obtained by using the range of the loop iterator, and the subscript expressions. Hudak and Kennedy present
an algorithm for computing the array sections accessed, based on the subscript expressions and the loop iterators [43]. For example, the array section for the reference \( A(i) \) in the loop nest is \([L_i:C_i:S_i]\). However, the array section for the reference \( A(i+1) \) in the above loop nest is the triplet \([L_i+1:C_i+1:S_i]\), because of the constant +1 in the subscript expression. The array section for the reference \( B(2 \times i + 1) \) in the above loop nest is \([2 \times L_i+1:2 \times C_i+1:2 \times S_i]\).

When there are two or more references to the same array in a loop nest, then the access ranges of an array are merged together using an union operation [43]. For instance, if the value of the stride for the iterator \( i \), \( S_i \), is 1, then the array section of \( A \) is \([L_i:C_i+1:1]\), which is obtained by merging the array references \( A(i) \) and \( A(i+1) \), i.e., \( A(i) \cup A(i+1) \). The value of the stride plays a key role in the merging of array sections. In the absence of stride information, the number of array elements accessed in a loop nest can be overestimated.

### 6.4.2 Estimating Local and Remote Accesses

The bounded regular section representation is used to estimate the amount of data accessed from cache, local, and remote memory, given candidate data and computation partitions for a loop nest. It is assumed that no data is in the cache when the execution of the loop nest starts. In the next section, we relax this assumption in order to take into account the effect of cache reuse across loop nests.

The section of array \( A \) accessed in the loop nest, \( S_a \), can be determined by analyzing the input loop nest. Some of the elements in the array section \( S_a \) will be from local memory and others will be from remote memory. Using the data partition of the array, we can estimate the array section that is mapped onto the local memory of a processor, \( S_1 \). The elements that are common to the two array sections \( S_a \) and \( S_1 \) will be accessed from local memory. Thus, the elements accessed from local memory are \( S_a \cap S_1 \). The elements that remain in \( S_1 \) and which are not accessed from local memory will be accessed from remote memory, i.e., \( S_1 - S_a \cap S_1 \). The number of cache lines transferred from the local and remote memories is estimated based on the stride information along the cache line dimension of the array. The analysis in this section takes into account only the data reuse within a loop nest, i.e., data reuse across loop nests has been ignored. In the next section, the analysis is

---

3. The array section is a synthesis of the triplets along every dimension of an array. Array range is used to refer to a triplet along a dimension. For simplicity, we do not distinguish between array sections and array ranges.

4. Finding the elements that are common between two sections is called the intersection operation [43].
extended so as to include the effect of temporal data reuse in the estimation of the number of local and remote memory accesses.

6.4.3 Including Temporal Reuse Across Loops in the Estimation of the Cache Accesses

In order to determine the section of an array that is in the cache at the start of execution of a loop nest, it is necessary to know the total number of array elements accessed by each loop nest in the program. The section of a given array that remains in the cache at the start of the execution of a loop nest is estimated in a two-step process. First, a cache data flow graph (CDFG) is constructed for the input program. Second, the CDFG is used as a template to construct an array flow graph (AFG) for each array in the program. The AFG represents the flow of control between loop nests that access the array.

6.4.3.1 The Cache Data Flow Graph

The CDFG is similar to the flow graphs used in optimizing compilers to portray the basic blocks and their successor relationships [39]. The CDFG is a directed graph \((V, E)\), in which the nodes \(v \in V\) represent loop nests and the edges \(e \in E\) represent the flow of control between pairs of loop nests. Each node in the CDFG contains the amount of data accessed by the corresponding loop nest.\(^7\) There is a directed edge from loop nest \(T_1\) to loop nest \(T_2\) if the execution of the loop nest \(T_2\) can immediately follow the execution of the loop nest \(T_1\). Depending on the flow of control in the program, the CDFG can be a linear, cyclic, or an acyclic graph.

The CDFG for the program in Figure 6.2 is shown in Figure 6.3. The CDFG has 4 nodes, \(T_1, T_2, T_3,\) and \(T_4\), corresponding to the four loop nests, \(L_1, L_2, L_3,\) and \(L_4\), in the program. The edges represent the flow of control between the loop nests.

The total number of array elements accessed by each loop nest depends on the computation and data partitions. The candidate computation and static data partitions derived during the data-computation affinity phase of the CDP algorithm are used to estimate the number of array elements accessed in each loop nest.

In the CDFG of Figure 6.3, each node is annotated by the total amount of data accessed in the corresponding loop nest by any processor, assuming \(N = 256\) on a 16 processor system.

\(^7\)The array access ranges are used to estimate the amount of data accessed in each loop nest.
real * 8 A(N,N), B(N,N), C(N,N)
forall j = 1 to N
forall i = 1 to N
$L_1 (T_1)$: A(i,j) = B(i,j) + C(i,j)
end for
end for
100: if (condition1) then
  return
end if
if (condition2) then
forall j = 1 to N
for i = 1 to N - 1
$L_2 (T_2)$: B(i,j) = A(i,j) * B(i+1,j)
end for
end for
else
forall j = 1 to N
for i = 2 to N
$L_3 (T_3)$: B(i,j) = B(i-1,j) + C(i,j)
end for
end for
end if
for j = 1 to N
forall i = 1 to N
$L_4 (T_4)$: C(i,j) = B(i,j) + C(i,j-1)
end for
end for
goto 100

Figure 6.2: An example to illustrate the CDFG.

In $T_1$, 96K bytes of data will be accessed by each processor. However, in $T_2$, $T_3$, and $T_4$, only 64K bytes of data will be accessed by each processor.

The algorithm to build a CDFG is shown in Figure 6.4.3.1. An edge is added between $T_i$ and $T_j$ when:

1. a loop nest $T_j$ follows the loop nest $T_i$.
2. an unconditional branch following $T_i$ leads to $T_j$, or
3. a condition branch following $T_i$ leads to $T_j$.

---

There are 3 arrays, each of size 256 x 256. Each element is assumed to be 8 bytes long. In this example, because of the array accesses and the one dimensional processor geometry determined by the CDP algorithm, the arrays will be divided equally among processors irrespective of the data partitions (excluding replication).
Algorithm: build-CDFG

begin
1. Create a node for each loop nest in the program
2. for each loop nest \(T_i\)
   add the statement that follows the loop nest to the List to evaluate \(L\)
   while \(L\) is not empty
      remove an entry in the list which is the statement \(S\) to be analyzed
   3. if \(current \text{ statement}(S) = \text{ loop nest } (T_j)\) then
      Add an edge from \(T_i\) to \(T_j\)
   4. else if \(current \text{ statement}(S) = \text{ conditional branch}\) then
      add the first statement of the paths to the List
   5. else if \(current \text{ statement}(S) = \text{ unconditional branch}\) then
      add the first statement of target of branch to the List
   else
   6. if \(current \text{ statement}(S) \) is not return or exit then
      add the statement following the current statement to the List
   end if
   end if
end while
end

Figure 6.4: Algorithm: build-CDFG.

6.4.3.2 The Array Flow Graph

In order to determine the section of the array that is present in the cache at the start of execution of each loop nest, we construct an array flow graph (AFG) for each array. The AFG of an array is similar to the CDFG, but it only contains nodes that correspond to loop nests in which the array is referenced. In addition, the nodes corresponding to loop nests that do not contain any parallel loops are not included in the AFG. The directed edges of the AFG have weights of 0 or 1. An edge between two nodes \(T_1\) and \(T_2\), corresponding to loop nests \(L_1\) and \(L_2\), has a weight of 1 if the total number of array elements accessed by the loop nest \(L_1\) is less than the size of the cache. The assumption here is that when the total number of array elements accessed by the loop nest is less than the size of the cache, all the data accessed by the loop nest \(L_1\), obtained from the CDFG, will remain in the cache for the loop nest \(L_2\) at the end of execution of loop nest \(L_1\). All other edges have a weight of 0. An edge is added between all predecessors and successors of a node that is removed.
These edges are given a weight of 0.\footnote{We assume that data might not remain in the cache because of the execution of an intervening loop nest which will access different data.}

We assume that the array elements are reused from the cache in a loop nest corresponding to a node $T_i$, if and only if all the edges reaching the node $T_i$ have a weight of 1. This is because a weight of 0 on a path indicates that it is possible array elements might not be retained in the cache. The intersection of the array access ranges of all the predecessors $T_j$ of a node $T_i$ is used to estimate the array access range that is retained in the cache when $T_i$ begins execution.

The AFG for arrays A, B, and C in the program shown in Figure 6.2 are shown in Figure 6.5. The AFG shows the array sections that are contained in the cache of processor number 4, at the end of execution of each of the loop nests. The AFG for array $A$ contains only two nodes: $T_3$ and $T_4$ are not present in the AFG because $A$ is not accessed in loop nests 3 and 4. The AFG for array $C$ contains three nodes: $T_2$ is not present in the AFG because $C$ is not accessed in loop nest 2. However, the AFG for array B in the program is the same as the CDFG for the program, with the edges marked appropriately. For purposes of illustration, assuming that the cache size of the target multiprocessor is greater than 96K, the edges are marked 0 or 1, based on all the predecessors for each loop nest.

The reaching matrix required by the CDP algorithm in the dynamic array partitions phase can be calculated using the AFG. If there is an edge in the AFG connecting the node $l$ to node $m$, then there will be a 1 in $(m, l)$ of the reaching matrix.
6.5 Putting It All Together

Algorithm: \textit{ArrayAccessCost} (Array A, Loop Nest T, AFG, Array Partition)
// For a given Array Partition with a processor dimension assignment.
// \textit{ArrayAccessCost} returns the cost of cache, local and remote
// access of the partitioned array in a loop nest T.

begin
1. Given processor geometry (G), computation partition (C),
   identify the iterations (I) to be executed on a processor
2. access range AccessRange(A_i) along a dimension i of array
   \( \leftarrow \bigcup_{\text{all access of } A} f(I) \)
3. CacheRange(A_i) \( \leftarrow \cap_{\text{parent}(T)} \text{AccessRange}(A_i) \ast \text{Edge}(\text{Parent}(T), T) \)
4. CacheAccess(A_i) \( \leftarrow \text{AccessRange}(A_i) \cap \text{CacheRange}(A_i) \)
5. LocalAccess(A_i) \( \leftarrow (\text{LocalPartition}(A_i) \cap \text{AccessRange}(A_i)) \)
   LocalAccess(A_i) \( \leftarrow \text{LocalAccess}(A_i) - \text{LocalAccess}(A_i) \cap \text{CacheAccess}(A_i) \)
6. Total Access(A) \( \leftarrow \prod_{i=1}^{n} \text{AccessRange}(A_i) \)
   CacheAccess(A) \( \leftarrow \prod_{i=1}^{n} \text{CacheAccess}(A_i) \)
   LocalAccess(A) \( \leftarrow \prod_{i=1}^{n} \text{LocalAccess}(A_i) + \prod_{i \subseteq S} \prod_{j \in K} \text{LocalAccess}(A_j) \ast \text{CacheAccess}(A_j) \)
   where \( S \subseteq \{1, \ldots, n\} \) and \( K = \{1, \ldots, n\} \)
   RemoteAccess(A) \( \leftarrow \text{TotalAccess}(A) - \text{LocalAccess}(A) \ast \text{CacheAccess}(A) \)
7. Cost \( \text{T} \leftarrow \text{CacheAccess}(A) \ast \text{CacheCost} + \text{LocalAccess}(A) \ast \text{LocalLatency} + \text{RemoteAccess}(A) \ast \text{RemoteLatency} \)
8. if contented access then
   Cost_1 \( \leftarrow \text{Cost}_T \ast cf \)
   if loop can be pipelined then
     Cost_2 \( \leftarrow \text{Cost}_T + S \ast Ns/aw \)
     Cost_1 \( \leftarrow \min(\text{Cost}_1, \text{Cost}_2) \)
   end if
   Cost_T \( \leftarrow \text{Cost}_1 \)
end if
9. if partitioned along cache line dimension then
   Cost_T \( \leftarrow \text{Cost}_T \ast QCP \)
end if
end for

Figure 6.6: Algorithm: \textit{ArrayAccessCost}.

The algorithm to estimate the cost of accessing an array in a loop nest, given a data
partition for the array, is shown in Figure 6.6. In defining loop nests in the CDP algorithm,
we have ignored sequential loops that enclose a loop nest and the selection statements.
Hence, the sum of the execution time of the loop nests does not reflect the true program
execution cost. If a loop nest is enclosed within a sequential loop, then the array access cost
is increased, based on the number of iterations of the enclosing sequential loop. Similarly,
we reduce the array access cost in a loop nest when the loop nest is enclosed within selection
statements, based on the probability of execution.
6.6 Cache Utilization Factor

In Section 6.4.2, we have assumed that the data accessed within a loop nest will be retained in the cache across the iterations of a loop in the loop nest and that no capacity misses will result. Such misses can be ignored while selecting dynamic data partitions in the CDP algorithm\(^{10}\), but they are critical in determining processor geometry mapping. For example, in Chapter 4, we saw that capacity misses play an important role in determining the processor geometry. Hence, we introduce a factor, called the *cache utilization factor*, to reflect the percentage increase in the array access time, based on the cache misses that will occur in a loop nest.

We use a cache utilization factor of 1 when the amount of data accessed by a loop nest is less than the size of the cache. A cache utilization factor greater than 1 indicates that data does not fit in the cache, and cache misses can occur due to limited cache capacity.

We estimate the cache utilization factor for a loop nest by using the number of cache misses that can occur at each loop level. For a given loop level, the number of cache misses that can occur is estimated as the ratio of the total number of array elements accessed in the body of the loop and the cache size. The product of all the cache misses of all the loops in the loop nest forms the cache utilization factor.

The cache utilization factor is used for selecting the number of processors assigned to each of the dimensions of the processor geometry. The execution time of a loop nest is estimated by adding the cost of each array accessed for a given processor geometry. By including the cache utilization factor as a part of the array access cost, we model cache affinity in the selection of computation and data partitions. The cache utilization factor is an indication of the number of capacity misses. This is only an approximation, since detailed cache behaviour is hard to predict at compile time.

6.7 Summary

We have presented a technique for estimating the execution times of loop nests. The CDP algorithm uses these estimates to assess the effectiveness of candidate computation and data partitions. We have divided the parameters for performance estimation into machine-

\(^{10}\) We chose a fixed processor geometry for dynamic array partition selection because of the interdependence between processor geometry mapping and dynamic array partition selection.
dependent and program-dependent categories. The parameters that characterize the machine are derived using a combination of analytical and empirical evaluations of the target architecture. We have also shown how we use the AFG and the sections of the array accessed in a loop to estimate the program-dependent parameters. This cost model is simple, yet realistic enough to guide the CDP algorithm in selecting suitable data and computation partitions for an application, as will be illustrated in Chapter 7.
CHAPTER 7
Experimental Results

This chapter presents experimental results to show that the CDP algorithm improves the performance of a suite of eight benchmark applications, compared to the reliance on the native operating system page placement policies for data management. This chapter is organized as follows. First, we discuss the characteristics of benchmark applications and describe the experimental platforms. Subsequently, we analyze the performance of the suite of benchmark applications on the experimental platforms using the data partitions derived by the CDP algorithm. In addition, the performance of the prototype compiler in analyzing the benchmarks is discussed. Finally, we empirically compare our algorithm with two other automatic data partitioning techniques to show the efficiency and effectiveness of our algorithm.

7.1 Benchmark Applications

Eight benchmark applications representative of scientific programs are used to evaluate the CDP algorithm. They are the vpenta, mxm, and fft2d kernels from nasa7 in the SPEC92 floating-point benchmark suite, tomcatv, also from the SPEC92 floating-point benchmark suite, the ADI stencil computation, svd from eispack, psinv subroutine in multigrid from the NAS suite of benchmarks, and fft3d from the ARCO suite of benchmarks. Table 7.1 summarizes the characteristics of these applications.

The vpenta kernel inverts three pentadiagonal matrices. There are 8 loop nests in the program, of which 7 have parallel loops. This application uses seven 2-dimensional arrays and two 3-dimensional arrays.

The mxm kernel repeatedly multiplies two matrices for a fixed number of iterations. This kernel operates on three 2-dimensional matrices. Two of the three arrays are read-only
Table 7.1: Application characteristics.

<table>
<thead>
<tr>
<th>Application</th>
<th>Lines of Code</th>
<th>No. of Procedures</th>
<th>No. of parallel loop nests</th>
<th>No. of Arrays</th>
<th>Array dimensionality</th>
<th>Brief Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpenta</td>
<td>128</td>
<td>2</td>
<td>7</td>
<td>9</td>
<td>2/3</td>
<td>inverts three pentadiagonal matrices</td>
</tr>
<tr>
<td>mxm</td>
<td>50</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>matrix multiplication</td>
</tr>
<tr>
<td>tomcav</td>
<td>195</td>
<td>1</td>
<td>9</td>
<td>7</td>
<td>2</td>
<td>mesh generation program</td>
</tr>
<tr>
<td>ADI</td>
<td>35</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>stencil computation</td>
</tr>
<tr>
<td>fft2d</td>
<td>250</td>
<td>2</td>
<td>11</td>
<td>4</td>
<td>1/2</td>
<td>2-D fast fourier butterfly computation</td>
</tr>
<tr>
<td>svd</td>
<td>319</td>
<td>1</td>
<td>16</td>
<td>5</td>
<td>1/2</td>
<td>decomposes a 2-D array</td>
</tr>
<tr>
<td>psinv</td>
<td>20</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>computes 3-D potential field</td>
</tr>
<tr>
<td>fft3d</td>
<td>70</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3/1</td>
<td>3-D fast fourier butterfly computation</td>
</tr>
</tbody>
</table>

and one of the arrays is written. This application has two initialization loop nests and two computational loop nests. One of the two computational loop nests primarily reinitializes an array, while the other loop nest performs matrix multiplication. In the latter loop nest, the two outer loops are parallel. Only one of the three arrays is updated, while the other two arrays are read-only. Executing both the outer loops in parallel requires replication of both read-only arrays. If only one of the parallel loops is executed in parallel, then only one of the read-only arrays requires replication. An important aspect of this application is that when the problem size is small, all the data fit into the cache, and no remote or local memory accesses occur during the repeated multiplication process.

The tomcav is a mesh generation program that operates on seven 2-dimensional arrays. The application has 14 loop nests, of which only 9 have parallel loops. All the 9 loop nests with parallel loops have a two-dimensional iteration space. Some of these loop nests do not have any dependence and are fully parallel, while the rest of the loop nests carry dependences across columns of the arrays and thus contain only inner loop parallelism.

The altering direction integration (ADI) program is used to solve partial differential equations. The application was described in Chapter 4. It operates on three 2-dimensional arrays and has two phases: the first phase sweeps along the columns of the arrays and the second phase sweeps along the rows.

The fft2d kernel performs a fast fourier transform (FFT) butterfly computation on a two dimensional array. The computation is performed repeatedly for a number of iterations. This application accesses a 2-dimensional complex array, two 1-dimensional complex arrays,
and a 1-dimensional integer array. The integer array is used to index various elements in the 2-dimensional array. The Fourier transformation is first performed on columns of the 2-dimensional array followed by computation along rows of the array.

The **svd** routine is one of a collection of Fortran subroutines that compute eigenvalues and eigenvectors of classes of matrices. The **svd** subroutine determines the singular value decomposition ($A = U \Sigma V^T$) of a real rectangular matrix. This program operates on three 2-dimensional arrays ($U$, $V$, and $A$). There are 23 loop nests in the routine, of which 16 have parallel loops. The amount of parallelism in the program is small because most of the parallel loop nests operate on only one of the dimensions of the arrays, i.e., $N$ elements of the $N \times N$ matrices. In addition, the sweeps across the columns and rows of the array are interleaved. A number of scalar values are computed based on the arrays. These scalar values introduce synchronizations to ensure program correctness.

The **psinv** subroutine is from the **multigrid** application, which has been described in Chapter 4. It is used to illustrate the impact of the choice of processor geometry on performance.

The **fft3d** is a program from the ARCO suite of benchmarks. The program performs a 3-D complex-to-complex FFT on a 3-dimensional array. The application performs a FFT on each of the dimensions of the 3-dimensional array. In each loop nest in the application, parallelism is along the remaining two dimensions on which the FFT is not performed. In each loop nest, the dimension on which the FFT is performed is first copied into a temporary 1-dimensional array. All the computations are performed on this temporary array. The temporary array is written back into the original array. This application is compute-intensive.

### 7.2 Experimental Platforms

The three multiprocessor architectures: Hector, Convex SPP1000, and the KSR-1, are considered to evaluate the CDP algorithm. Each of the three multiprocessor architectures has unique characteristics in terms of the ratio of remote memory access time to local memory access time, cache capacity, and memory contention and synchronization penalties. The characteristics of these multiprocessors are summarized in Table 7.2.
Table 7.2: Multiprocessor characteristics.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Cache Access</th>
<th>Local Access</th>
<th>Remote-1/Remote-2</th>
<th>ratio</th>
<th>Cache Size</th>
<th>Memory Contention Penalty</th>
<th>Synchronization Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hector</td>
<td>1</td>
<td>10</td>
<td>19/29</td>
<td>1.9/2.9</td>
<td>16K</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Convex</td>
<td>1</td>
<td>50</td>
<td>200</td>
<td>4</td>
<td>1M</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>KSR-1</td>
<td>2</td>
<td>18</td>
<td>170/530</td>
<td>9.4/29.4</td>
<td>256K</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>

7.2.1 Hector

Hector [3] is a scalable shared memory multiprocessor that consists of a set of processing modules connected by a hierarchical ring interconnection network. Four processing modules are connected by a bus to form a station. Stations are connected by a local ring to form a cluster. Clusters are then connected by a central ring to form the system. Each processing module consists of a Motorola 88000 processor with 16 Kbytes of fast cache and a 4-Mbyte local portion of shared memory. A processing module accesses non-local portions of shared memory over the ring interconnect. Hector has no hardware mechanisms for cache consistency. Cache coherence is maintained by software [44]. Experimental results presented in this thesis have been conducted on a 16-processor cluster consisting of 4 stations on a single local ring.

7.2.2 Convex Exemplar

The Convex SPP1000 [6] consists of up to 16 hypernodes, each containing 8 processors with a crossbar connection to a 512 Mbytes portion of shared memory. The crossbar provides uniform access to the local memory for processors within a hypernode. Each processor is a Hewlett-Packard PA7100 RISC microprocessor running at 100 MHz with separate 1-Mbyte instruction and data caches. Hypernodes are connected together with the Coherent Toroidal Interconnect (CTI), a system of rings based on the SCI standard interconnect. The cache access latency is 1 clock cycle, or 10 nsec, and the cache line size is 32 bytes. Cache misses to retrieve data from the local hypernode memory incur a latency of 50 cycles, or 500 nsec. Misses to data on remote hypernode memory incur a latency of 200 cycles, or 2000 nsec.
7.2.3 KSR-1

The KSR-1 [5] consists of 32 to 1024 processing cells interconnected by a two-level ring hierarchy. Thirty-two processing cells are connected to a first-level ring, which is referred to as ring:0. Up to 34 ring:0 rings can be connected to a second-level ring, which is referred to as ring:1. The system on which we conducted our experiments consists of 16 processors in the single 32-processor ring:0. The processing cell consists of a 64-bit superscalar processor, a 512-Kbyte first-level cache referred to as the subcache, and a 32-Mbyte second-level cache, which serves as the cell’s main memory, and is referred to as the local cache. The KSR-1 is a Cache Only Memory Access (COMA) architecture. Data is replicated and migrated from one portion of the shared memory to another on demand.

Due to the inherent migration of data, the COMA architecture makes the implementation of data partitions somewhat different. The data partitions in a program are dictated by the computation partitioning. COMA permits data partitions only with one-to-one mapping of array dimension to processor geometry dimension, because there is no fixed memory in which a data item can reside. In order to implement data partitions according to computation partitions, data transformations are used. Data transformation is the process of moving the partitioned dimensions to higher dimensions of the array, using dimension permutation operations. This not only minimizes cache line false sharing, but also ensures that the data accessed by a processor is allocated contiguously to minimize cache conflicts.

7.2.4 Comments on Machine Characteristics

The characteristics of the machines dictate the data and computation partitions derived using the CDP algorithm. The Hector multiprocessor, the Convex SPP1000, and the KSR-1 represent a spectrum of machine characteristics with respect to cache size, local and remote memory access latencies, and cost of memory contention and synchronizations (see Table 7.2). The processors on Hector have a relatively small cache of 16K bytes; the processors on Convex SPP1000 have a relatively large cache of 1024K bytes; and the processors on KSR-1 have a medium-sized (primary) cache of 256K bytes.

The ratio of remote memory access time to local memory access time is relatively small on the Hector multiprocessor (1.9 for remote memory on the same station and 2.9 for remote memory off-station): the ratio on the Convex multiprocessor is relatively high (4.0): and
the ratio on the KSR-1 multiprocessor is quite high (9.4 for remote memory on the same ring and 29.4 for remote memory off-ring).

The Hector multiprocessor is representative of an SSMM where the penalty for contention is very high; the penalty for contention on the Convex multiprocessor is low; and the penalty for contention on the KSR-1 multiprocessor is moderate.

The cost of synchronization on the Convex multiprocessor is relatively low, and both the Hector multiprocessor and the KSR-1 multiprocessor have a relatively high cost of synchronization.

The diverse characteristics of these machines present an opportunity to experimentally determine the adaptability and effectiveness of the CDP algorithm.

7.3 The Compiler Prototype

The CDP algorithm has been implemented as a part of the Jasmine compiler prototype. The overall structure of Jasmine is shown in Figure 7.1. The compiler accepts a Fortran 77 program and generates a parallel program. It consists of four major phases: parallelism detection, cache locality enhancement, memory locality enhancement, and code generation. The KAP preprocessor from Kuck and Associates and the Polaris compiler are used for the parallelism detection phase. The KAP preprocessor and the Polaris compiler also per-
form procedure inlining. The CDP algorithm primarily constitutes the memory locality enhancement phase of the Jasmine compiler, and is used to automatically determine data and computation partitions for applications. The output of the prototype compiler is a C++ program with run-time calls to NUMACROS [45] for implementing data and computation partitions.

### 7.4 Overall Performance

The data partitions derived by the compiler for the major\(^1\) arrays in the applications on the multiprocessors is shown in Table 7.4. For all the applications, the processor geometry determined by the CDP algorithm was a linear array of processors, unless otherwise indicated.

Table 7.3 shows the serial execution times of the benchmark applications on the three machines considered. These execution times are used for calculating the speedups. In the table, the numbers in parenthesis indicate the problem size for the respective benchmark application.

The performance of the applications on the Hector multiprocessor without data partitions, compared to their performance with the computation and data partitions derived using the CDP algorithm, is shown in Figure 7.2. The performance of the applications without data partitions delegates data management to the operating system. The two page placement policies used by operating system to manage data are “first-hit” and “round-robin.”

\(^1\)The largest-sized arrays in the program are considered major arrays in this thesis.
Furthermore, when operating system data management is used, only the parallel loops at the coarsest granularity that would be chosen by a parallelizing compiler are partitioned among processors. The performance of the applications with data partitions outperform both the operating system data management policies.

We first comment on the salient features that are responsible for improving the performance of the benchmark applications on the Hector multiprocessor. The memory locality and contiguous allocation of data using data partitions in each processor’s memory module improves the performance of *vpenta*. The replication of one of the matrices being multiplied was the key to improving the performance of the *mm* application. The performance of *tomcatv* is improved due to the reuse of data across the loop nests in the program. In this application there are two types of loop nests. The first type of loop nests has both outer and inner loop parallelism and the second type of loop nests has only inner loop parallelism.

The parallelizing compiler strategies exploit outer loop parallelism when available and do not use inner loop parallelism. Across different loop nests, each processor accesses different data elements. In contrast, the CDP algorithm selects inner loop parallelism across all the loop nests, increasing the data reuse. The performance of *tomcatv* with and without data partitions is poor, however, because of the overhead due to synchronization between the loop nests [34]. Still the application with data partitions outperforms the operating system policies. In the *ADI* application, the operating system data management techniques result in contention. The compromise data partition selected by the CDP algorithm eliminates

<table>
<thead>
<tr>
<th>Application</th>
<th>Hector (16 Processors)</th>
<th>Convex (30 Processors)</th>
<th>KSR (16 Processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>vpenta</em></td>
<td>F(•.Block{1}.•)</td>
<td>F(•.Block{1}.•)</td>
<td>F(•.Block{1}.•)</td>
</tr>
<tr>
<td><em>mm</em></td>
<td>A(•.•)</td>
<td>A(•.•)</td>
<td>A(•.•)</td>
</tr>
<tr>
<td><em>tomcatv</em></td>
<td>C(•.Block{1})</td>
<td>C(•.Block{1})</td>
<td>C(•.Block{1})</td>
</tr>
<tr>
<td><em>ADI</em></td>
<td>RX(Block{1}.•)</td>
<td>RX(Block{1}.•)</td>
<td>RX(Block{1}.•)</td>
</tr>
<tr>
<td><em>fft2d</em></td>
<td>X(Block{1}.Block{1})</td>
<td>X(Block{1}.Block{1})</td>
<td>X(Block{1}.Block{1})</td>
</tr>
<tr>
<td><em>svd</em></td>
<td>X(•.Block{1})</td>
<td>X(•.Block{1})</td>
<td>X(•.Block{1})</td>
</tr>
<tr>
<td><em>psinv</em></td>
<td>/X(Block{1}.•)</td>
<td>/X(Block{1}.•)</td>
<td>/X(Block{1}.•)</td>
</tr>
<tr>
<td>proc. geom.</td>
<td>(16.1)</td>
<td>(16.1)</td>
<td>(16.1)</td>
</tr>
<tr>
<td><em>fft3d</em></td>
<td>Z(•.Block{1}.Block{2})</td>
<td>Z(•.Block{1}.Block{2})</td>
<td>Z(•.Block{1}.Block{2})</td>
</tr>
<tr>
<td>proc. geom.</td>
<td>(1.16)</td>
<td>(1.30)</td>
<td>(1.16)</td>
</tr>
</tbody>
</table>

Table 7.4: Data partitions derived by the CDP algorithm.
contention. Repartitioning of the arrays improves the performance of the \texttt{fft2d} application by eliminating contention. In contrast, in the \texttt{fft3d} application, a static data partition that incurs contention performs well because the three dimensional array being transformed is read and written only once in the contended loop nest. The performance of \texttt{svd} on the Hector is poor because of the overhead involved in partitioning both dimensions of an array in the application. Besides, the application characteristics are poor for parallelization. This is an application for which the operating system data management performs slightly better than the derived data partitions, because of the overhead involved in partitioning data.

The performance of the applications on a 30-processor Convex Exemplar multiprocessor without data partitions is compared with the performance of the computation and data partitions derived by the CDP algorithm in Figure 7.3. The only page placement policy available on Convex is “round-robin.” The data partitions derived by the algorithm outperform the operating system page placement policy on the Convex Exemplar multiprocessor. The \texttt{vpenta} application exhibits super-linear speedup because of improved cache utilization with data partitions. For the \texttt{fft3d} application, the performance without data partitions is slightly better than the partition derived by the algorithm. The performance of the application with data partitions is lower because of the overhead involved in partitioning the arrays. The application is computation-intensive, and the remote or local memory access cost incurred in the program is small and cannot offset the data partitioning overhead.
The performance of applications on the KSR-1 multiprocessor with outer loop parallelism and the computation partitions derived by the CDP algorithm are shown in Figure 7.4. The performance of ADI results in super-linear speedup because of repartitioning, which results in better spatial locality in the loop nests in the application. Overall, data partitions derived by the CDP algorithm enhance the performance of the applications.

The main machine characteristics that affect the choice of computation and data partitions are summarized in Table 7.5. An "X" in the table indicates that the corresponding machine characteristic tends to influence the selection of computation and data partitions for the benchmark. Note that the computation and data partitions for the remaining three

---

2Both fft2d and fft3d were not executed on the KSR-1 multiprocessor because of the lack of a C++ compiler for implementing the complex class corresponding to the complex data type in Fortran.
Table 7.5: Main machine characteristics that affect the choice of computation and data partitions on the chosen suite of benchmarks.

<table>
<thead>
<tr>
<th>Application</th>
<th>Ratio of remote to local memory access</th>
<th>Cache size</th>
<th>Contention</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADI</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>fft2d</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>svd</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>psinv</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft3d</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.6: Impact of cache on data partition selection: performance of psinv application.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Hector</th>
<th>Convex</th>
<th>KSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data partition</td>
<td>U[*·Block{1}·Block{2}]</td>
<td>(16, 1)</td>
<td>(4, 4)</td>
</tr>
<tr>
<td>Speedup</td>
<td>14.5</td>
<td>9.6</td>
<td>16.2</td>
</tr>
<tr>
<td>Cache size</td>
<td>16K</td>
<td>1M</td>
<td>256K</td>
</tr>
</tbody>
</table>

benchmarks, namely vpenta, mmx, and tomcatv, are not affected by the machine-specific characteristics. It is not necessary to consider machine-specific characteristics for these applications because the data partitions derived using data-computation affinity have a one-to-one mapping between the array dimensions and processor geometry dimensions, and are hence static.

The ratio of remote memory access latency to the local memory access latency influences the selection of data and computation partitions in all the benchmark applications. Among the benchmarks, pipelining was possible only in ADI, and hence, only in this application did the cost of synchronization affect the choice of computation and data partitions.

The processor geometry mapping is only required in psinv and fft3d applications because only in these applications is the dimensionality of the processor geometry greater than 1. The size of the cache plays a major role in determining the processor geometry mapping in order to improve cache affinity, as described in Chapter 4.

The impact of the size of the cache in the selection of data partitions can be demonstrated using the psinv application. Using 16 processors, the processor geometries selected by the algorithm to enhance cache affinity are (16, 1), (4, 4), (8, 2) for the Hector, the Convex, and the KSR-1 multiprocessors, respectively. The performance of the application
Figure 7.5: Comparing the performance of different data partitions with the partitions derived by the CDP algorithm: Applications- vpenta, mxm, tomatv, and ADI.

on the three multiprocessors with 16 processors is summarized in Table 7.6. On the Convex multiprocessor, a (4, 4) processor geometry is selected because of the large cache: the (4, 4) processor geometry minimizes remote memory access. In contrast, to improve the cache affinity, a (8, 2) processor geometry is chosen on the KSR multiprocessor, in spite of a higher remote memory access cost, as was discussed in Chapter 4. A (16, 1) processor geometry is chosen on the Hector multiprocessor, because of the very small cache size, to enhance cache locality.

Figures 7.5 and 7.6 compare the speedups of benchmark applications when using the computation and data partitions derived by the CDP algorithm with the speedups of the benchmark applications when using several other possible data partitions. For convenience, the figures show this comparison only on the Convex multiprocessor.
Figure 7.6: Comparing the performance of different data partitions with the partitions derived by the CDP algorithm: Applications - fft2d, svd, psinv, and fft3d.

7.5 Compiler Performance

This section describes how the eight benchmark applications are analyzed by the algorithm. The time taken for each of the different phases and the criterion for choosing specific phases of the algorithm for the specific applications are discussed.

7.5.1 Data-Computation Affinity Phase

The execution time and data partitions selected by the data-computation affinity phase of the algorithm are shown in Table 7.7. The execution time of the compiler is measured on a SUN SPARCstation 10. It is clear that the time taken for the data computation-affinity phase of the algorithm for all the benchmark applications is small (less than 40 mSec).
Table 7.7: Measurements from data-computation affinity phase.

<table>
<thead>
<tr>
<th>Application</th>
<th>Time (mS)</th>
<th>Processor Geometry Dimensionality</th>
<th>Number of Array nodes in the ALAG</th>
<th>Number of Loop nodes in the ALAG</th>
<th>Data partition for major arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpenta</td>
<td>31</td>
<td>1</td>
<td>9</td>
<td>9(7)</td>
<td>F(<em>.Block{1}.</em>), X(*.Block{1})</td>
</tr>
<tr>
<td>mxm</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4(2)</td>
<td>A(<em>.</em>)</td>
</tr>
<tr>
<td>tomcatv</td>
<td>14</td>
<td>1</td>
<td>7</td>
<td>11(5)</td>
<td>X(Block{1}, Block{1})</td>
</tr>
<tr>
<td>ADI</td>
<td>9</td>
<td>1</td>
<td>6</td>
<td>5(5)</td>
<td>X(Block{1}.Block{1})</td>
</tr>
<tr>
<td>fft2d</td>
<td>20</td>
<td>1</td>
<td>4</td>
<td>11(11)</td>
<td>X(Block{1}.Block{1})</td>
</tr>
<tr>
<td>svd</td>
<td>18</td>
<td>1</td>
<td>5</td>
<td>16(16)</td>
<td>U(BCRBC{1}. BCRBC{1})</td>
</tr>
<tr>
<td>psinv</td>
<td>18</td>
<td>2</td>
<td>4</td>
<td>3(2)</td>
<td>U(*.Block{1}.Block{2})</td>
</tr>
<tr>
<td>fft3d</td>
<td>16</td>
<td>2</td>
<td>3</td>
<td>6(6)</td>
<td>2(Block{1}.Block{1,2}.Block{2})</td>
</tr>
</tbody>
</table>

The CDP algorithm selected a one dimensional processor geometry for six of the eight benchmark applications. Only psinv and fft3d require a processor geometry dimensionality of 2, because there are two levels of parallelism that can be exploited in each loop nest in these applications. Although there are two levels of parallelism in the mxm application, a one dimensional processor geometry was selected to avoid cache line false sharing.

The number of dimensions of the arrays that are selected for partitioning in the ALAG is a subset of all the dimensions of the arrays in the program. For example, in the vpenta application, though there are two 3-dimensional arrays and seven 2-dimensional arrays, resulting in 20 possible dimensions that must be considered, only 9 of these array dimensions form array dimension nodes in the ALAG. Similarly, many of the array dimensions in other applications are also not considered. This filtering is possible because of data-computation affinity. Using data-computation affinity, we have selected the potentially distributable dimensions of the array based on the array subscript expression(s) and the parallelism in the program. Thus, the number of data partitions that must be considered is minimized.

The number of loop nodes in the ALAG for each application is shown in the table. Although all the parallel loops in the program become loop nodes in the ALAG, only a subset of these nodes are assigned processor dimensions; hence, the corresponding parallel loop will be executed in parallel; all other loops are executed sequentially.3 For example,

3The number of parallel loop nodes that are assigned a processor geometry dimension is shown in the parenthesis.
in the psinv application, only two of the three loops are selected for parallel execution. and only these two loops have to be mapped onto the processor geometry. Thus, the number of computation partitions that must be considered for processor geometry mapping is minimized.

The data partitions derived by the data-computation affinity phase for major arrays in the applications are also shown in Table 7.7. The partitioning attribute for all the array dimension nodes and loop nodes in all the applications except svd is Block. Load balancing and locality considerations in svd force a BlockCyclicRBlockCyclic partitioning for the array dimension and loop nodes in the ALAG.

In four out of the eight applications (vpenta, mxm, tomatv, and psinv), the data-computation affinity phase derives static data and computation partitions for the program. In three of these four applications (i.e., all but psinv), the selected processor geometry is linear, and only one dimension of the arrays is partitioned. Thus, these applications do not require any further analysis.

In four of the eight applications (ADI, fft2d, svd, and fft3d), dynamic data partitioning may be beneficial, because some of the arrays in these applications do not have a one-to-one mapping between the distributed array dimensions and the dimension of the processor geometry.

Thus, only a subset of the applications require machine-dependent analysis. Indeed, in all applications, only 6 out of 35 arrays and 30 out of 54 loop nests are considered for dynamic array partitioning, and 4 arrays and 4 loop nests are considered for processor geometry mapping. In cases where static data partitions are possible, the CDP algorithm derives these partitions using just the data-computation affinity phase.

7.5.2 Dynamic Array Partition Phase

The Hector multiprocessor is chosen to be a representative platform to illustrate compilation analysis for the selection of dynamic array partitions. Only the arrays that may benefit from dynamic partitioning are considered in the re-evaluation for the dynamic array partition selection. The execution time and the number of data partitions analyzed by this phase of the compiler for the four applications are shown in Table 7.8. The execution time and the number of nodes searched are comparable for the other two multiprocessor
Table 7.8: Measurements from dynamic array partition phase.

<table>
<thead>
<tr>
<th>Application</th>
<th>Time (mS)</th>
<th>Number of arrays re-evaluated</th>
<th>Number of loops analyzed</th>
<th>Number of partitions</th>
<th>Total combination of choices</th>
<th>Number of nodes searched</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADI</td>
<td>271</td>
<td>3 out of 3</td>
<td>5</td>
<td>4</td>
<td>3*1024</td>
<td>116</td>
</tr>
<tr>
<td>fft2d</td>
<td>5164</td>
<td>1 out of 4</td>
<td>9</td>
<td>4</td>
<td>262144</td>
<td>1591</td>
</tr>
<tr>
<td>svd</td>
<td>1975</td>
<td>1 out of 5</td>
<td>13</td>
<td>4</td>
<td>67108864</td>
<td>240</td>
</tr>
<tr>
<td>fft3d</td>
<td>504</td>
<td>1 out of 2</td>
<td>3</td>
<td>16</td>
<td>4096</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 7.9: Measurements from processor geometry mapping phase.

<table>
<thead>
<tr>
<th>Application</th>
<th>Time (mS)</th>
<th>Number of processor geometry choices (16 procs.)</th>
<th>Number of loops analyzed</th>
</tr>
</thead>
<tbody>
<tr>
<td>psinv</td>
<td>179</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>fft3d</td>
<td>177</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

platforms.

In all four applications, the time required for the selection of appropriate data partitions is small (a maximum of about 5 seconds). The CDP algorithm minimizes the time required by (i) minimizing the number of arrays re-evaluated, (ii) minimizing the number of loops to be analyzed, and (iii) using pruning to limit the search space.

The number of arrays that are considered for re-evaluation for dynamic array partition selection is a subset of the arrays in the application. For example, in the fft2d application, only one of the two arrays is re-evaluated. In contrast, in the ADI application, all of the three arrays in the application require re-evaluation.

Only the loop nests in which the array is accessed are considered in the re-evaluation process. For example, in the svd application, only 13 of the 16 loop nests are considered. The remaining three loop nests in this application do not have a reference to the array that is being re-evaluated.

The depth-first search with pruning in the CDP algorithm determines a solution by evaluating only a small subset of the search space. For example, the search space that must be considered for the svd application is \(4^{13} = 67108864\). However, the CDP algorithm only considers 240 nodes.

7.5.3 Processor Geometry Mapping Phase

Psinv and fft3d are the two applications in which the selected processor geometry
is greater than 1, and thus require a selection of an appropriate processor geometry. The CDP algorithm selects a processor geometry for the entire program by analyzing the possible geometry choices. On a 16-processor machine, with two dimensional processor geometry, five processor geometry choices are evaluated. The execution time for processor geometry selection and the number of processor geometry choices are shown in Table 7.9.

7.5.4 Comments on the Characteristics of Applications

Each of the eight applications exercises a different feature of the CDP algorithm, as illustrated in Table 7.10. The vpenta application illustrates the effectiveness of the CDP algorithm on a simple application in deriving static data partitions. The mxm application illustrates the case in which the algorithm replicates an array to enhance performance. Although the CDP algorithm tries to avoid partitioning the cache line dimension, in the tomcatv application the cache line dimension is partitioned to exploit cache affinity across loop nests.

The ADI application is an example of an application that requires dynamic array partition selection. The computation and data partitions selected across the three multiprocessors are different because of the machine characteristics. The compromise data partition is selected on the Hector multiprocessor because of the high penalty for contended access and the high cost of synchronization. A data partition with pipelined computation partitioning is selected on the Convex multiprocessor because of its large cache and the low cost of synchronization. Repartitioning is selected on the KSR-1 multiprocessor because of the high cost of remote memory access. Fft2d also requires dynamic array partition selection. In contrast to the ADI application, the data partitions selected for the fft2d application across the three multiprocessors are the same, and require repartitioning of the arrays.

Table 7.10: Features of the algorithm applications exercise.

<table>
<thead>
<tr>
<th>Application</th>
<th>Salient feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpenta</td>
<td>Effectiveness in a simple application</td>
</tr>
<tr>
<td>mxm</td>
<td>Replication of arrays</td>
</tr>
<tr>
<td>tomcatv</td>
<td>Cache line dimension partitioning</td>
</tr>
<tr>
<td>ADI</td>
<td>Re-evaluation. Different data partitions on machines</td>
</tr>
<tr>
<td>fft2d</td>
<td>Re-evaluation. Same data partitions on machines</td>
</tr>
<tr>
<td>svd</td>
<td>Load balancing</td>
</tr>
<tr>
<td>psinv</td>
<td>Cache affinity, processor geometry selection</td>
</tr>
<tr>
<td>fft3d</td>
<td>Re-evaluation, processor geometry selection</td>
</tr>
</tbody>
</table>
Table 7.11: Bixby, Kremer, and Kennedy's approach.

<table>
<thead>
<tr>
<th>Application</th>
<th>Program Phases</th>
<th>Template dimensionality</th>
<th>Candidate data partitions 16 processors</th>
<th>Number of variables</th>
<th>Number of constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpenta</td>
<td>8</td>
<td>3</td>
<td>15</td>
<td>2595</td>
<td>338</td>
</tr>
<tr>
<td>mxm</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>170</td>
<td>59</td>
</tr>
<tr>
<td>tomcatv</td>
<td>14</td>
<td>2</td>
<td>10</td>
<td>1940</td>
<td>354</td>
</tr>
<tr>
<td>ADI</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>175</td>
<td>65</td>
</tr>
<tr>
<td>fft2d</td>
<td>16</td>
<td>2</td>
<td>5</td>
<td>-480</td>
<td>176</td>
</tr>
<tr>
<td>svd</td>
<td>23</td>
<td>2</td>
<td>16</td>
<td>9840</td>
<td>1175</td>
</tr>
<tr>
<td>psinv</td>
<td>1</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>fft3d</td>
<td>3</td>
<td>3</td>
<td>15</td>
<td>495</td>
<td>63</td>
</tr>
</tbody>
</table>

The svd application illustrates the choice of the BlockCyclicRBlockCyclic distribution attribute to balance the load among processors and to avoid false sharing. The psinv application illustrates how the processor geometry mapping varies with the size of the cache on a multiprocessor. The fft3d application requires the selection of dynamic array partitions and the processor geometry mapping. It is the only application that exercises all the different phases of the CDP algorithm.

7.6 Comparison with Other Approaches

In this section, the CDP algorithm is compared to the 0-1 integer programming formulation of Bixby et al. [20, 26] and to Anderson and Lam's algebraic framework [19, 30]. The metrics used to compare the different algorithms are the problem sizes evaluated at compile time, and the quality of the solutions obtained.

The sizes of the 0-1 integer programming problems that are solved for deriving data partitions for the applications are shown in Table 7.11. The number of program phases (i.e., one or more loop nests) here represents all the loop nests in the program. It must be noted that the phases include all loop nests in the program, even if there are no parallel loops in a loop nest. For example, in the mxm application, there are four loop nests, of which two do not have parallel loops. The template dimensionality and load balancing requirements dictate the number of candidate data partitions that are evaluated.

In the implementation of Bixby et al., the time taken for executing a 2386x715 0-1 integer programming problem using CPLEX [46], a general purpose integer program solver, is measured to be an average 3.8 seconds. However, some 0-1 integer programming problems
took longer than 4 hours [20]. The CDP algorithm partitions the applications depending on their characteristics. So, for applications like \texttt{vpenta}, \texttt{mmx}, and \texttt{tomcatv}, the CDP algorithm can derive computation and data partitions merely by analyzing data-computation affinity, without evaluating all possible data partition choices. This is in contrast to the approach of Bixby et al., which has to solve 0-1 integer programming problems in order to determine the data partitions for these applications. In the four applications in which dynamic partitions are considered, the CDP algorithm determines a solution efficiently, and in a reasonable time, using a pruned depth-first search. Furthermore, in evaluating data partitions in the presence of load imbalances, Bixby et al. consider only the \texttt{Cyclic} and \texttt{Block} distribution attributes. However, the CDP algorithm includes several additional distribution attributes to balance the load in the application.

Anderson and Lam use profiling to limit the search space of data and computation partitions. The size of the search space analyzed to determine data and computation partitions is shown in Table 7.12. In the applications shown, we determined the frequently executed loops by analyzing the applications manually. The search space comprises data and computation partitions selected for each loop nest individually, as well as possible changes to these partitions. It must be noted that the number of data and computation partitions analyzed is very small. Hence, this technique has a better compile time performance than the CDP algorithm. However, this technique only determines the dimension of the array that is to be partitioned. Heuristics are used to select a data partitioning attribute. Issues of load balancing, determining block-size for \texttt{BlockCyclic} distribution attribute, and determining the processor geometry to partition data and computation are not addressed. In contrast, our CDP algorithm determines data and computation partitions, including the block sizes for the \texttt{BlockCyclic} distribution attributes and the processor geometry mapping that will maximize cache affinity in the program.

Although the algebraic approach has a better compile time performance than the CDP algorithm, the quality of solutions obtained by this approach may not be as good. The algebraic approach is dependent on the profiling information to determine the frequently executed loop nests. Depending on which loop nest is analyzed first, data partitions derived by the algebraic approach may be different. For example, in the \texttt{ADI} application, the data partition selected for the program could be \((\texttt{*}, \texttt{Block\{1\}})\) or \((\texttt{Block\{1\}}, \texttt{*})\) (with pipelined execution of some loop nests), depending on the starting loop nest. The execution
Table 7.12: Anderson and Lam's approach.

<table>
<thead>
<tr>
<th>Application</th>
<th>Search space (Data partitions)</th>
<th>Search space (Computation partitions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpenta</td>
<td>51</td>
<td>10</td>
</tr>
<tr>
<td>mxm</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>tomcatv</td>
<td>36</td>
<td>10</td>
</tr>
<tr>
<td>ADI</td>
<td>22</td>
<td>8</td>
</tr>
<tr>
<td>fft2d</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>svd</td>
<td>27</td>
<td>19</td>
</tr>
<tr>
<td>psinv</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fft3d</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 7.13: Quality of solutions obtained by Anderson and Lam's algebraic solution on the Hector multiprocessor.

<table>
<thead>
<tr>
<th>Application</th>
<th>Data Partitions</th>
<th>Execution time</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADI</td>
<td>(.*.Block{1})/</td>
<td>47.865 mSec</td>
<td>1.31</td>
</tr>
<tr>
<td></td>
<td>Pipelining</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Block{1}.*)/</td>
<td>63.055 mSec</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pipelining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft3d</td>
<td>Z[.*.Block{1}.Block{2}]</td>
<td>61.345 mSec</td>
<td>1.48</td>
</tr>
<tr>
<td></td>
<td>Proc. Geom = (1.16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z[.*.Block{1}]/</td>
<td>91.125 mSec</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z[.<em>.Block{1}.</em>]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The algebraic approach favours a dynamic partition even when a static data partition with communication can result in better performance because of shared memory effects. This case is illustrated by the fft3d application. Data partitions with repartitioning is the only choice offered by the algebraic approach. However, a static data partition would improve the performance of the application by 48%, as shown in Table 7.13. The CDP algorithm determines that the static solution is better when it evaluates the possible dynamic

---

1Additional data transformations are applied by the algebraic approach to avoid partitioning the cache line dimension.
array partitioning choices.

7.7 Summary

In this chapter, the performance of a suite of eight benchmark applications using the partitions derived by the CDP algorithm on three different multiprocessors was analyzed. Our results show that the computation and data partitions derived by the CDP algorithm improve the performance of the benchmark applications, compared to the reliance on the native operating system page placement policies for data management. We have presented empirical evidence that the CDP algorithm is efficient and effective in deriving computation and data partitions for applications. We have found that data-computation affinity relationships are useful abstractions in effectively deriving computation and data partitions. Using the data-computation affinity relationships, the CDP algorithm effectively classifies the applications into two categories: (i) applications where all arrays require static partitions, and (ii) applications where some arrays require dynamic partitions. We have also shown that the CDP algorithm can obtain solutions of better quality than other techniques that automatically derive data and computation partitions.
In this thesis, we have shown that computation and data partitions are effective in enhancing data locality in regular (i.e., dense-matrix) scientific applications on SSMMs. We have described a new and efficient algorithm, called CDP, to automatically derive computation and data partitions on SSMMs. We have presented experimental evidence that indicates shared memory effects, such as cache affinity, memory contention, synchronization overhead, and false sharing, must be taken into account, in addition to remote memory accesses, in deriving data and computation partitions on SSMMs.

The CDP algorithm described in this thesis derives computation and data partitions in three main phases: (i) establishing data-computation affinity, (ii) deriving dynamic data partitions, and (iii) determining the virtual processor geometry and mapping it onto physical processors. The problem of deriving computation and data partitions is computationally hard. Dividing the solution to the problem into phases resulted in a relatively efficient algorithm. The first phase determines affinity relationships and uses them to select initial computation and data partitions, using a simple iterative technique. These initial partitions are the final partitions for many applications, for which it is sufficient to partition data statically on a linear array of processors. In the second phase, only the subset of the arrays that will potentially benefit from dynamic repartitioning are considered for further analysis. In this phase, a parameterized cost model of the target multiprocessor is used to evaluate the effectiveness of the candidate partitioning choices. Finally, in the third phase, the number of processors in each dimension of the virtual processor geometry is determined.

The CDP algorithm was implemented as a part of the Jasmine compiler prototype at the University of Toronto. Our experiments demonstrate that the parallel performance of a suite of benchmarks on three multiprocessor platforms using derived data and computa-
tion partitions compares favourably to the performance of the applications using operating system policies. Indeed, the performance of the application is improved, on an average by a factor of 2.15 on the Hector multiprocessor, 1.91 on the Convex multiprocessor, and 2.04 on the KSR-1 multiprocessor.

8.1 Contributions

This thesis made two main contributions:

- **Effectiveness of computation and data partitions on SSMMs**
  
  We have shown that partitioning of both computation and data significantly enhances the performance of applications on SSMMs. We have also shown that the selection of data and computation partitions on SSMMs is not only dependent on the cost of remote memory accesses, but is also influenced by shared memory effects such as cache affinity, memory contention, synchronization, and false sharing.

- **The CDP algorithm**
  
  In this thesis, we have described the CDP algorithm for efficiently deriving data and computation partitions that take into account the shared memory effects on SSMMs. Experimental evaluation of the CDP algorithm on three SSMMs demonstrates the effectiveness of derived partitions in improving the performance of a suite of benchmark applications, both when compared to the reliance on the native operating system for data management, and when compared to the use of data partitions derived by algorithms for distributed memory multiprocessors.

8.2 Directions for Future Research

This work can be extended in a number of ways.

8.2.1 Interprocedural Analysis

The current implementation of the CDP algorithm analyzes programs within procedural boundaries. Hence, the implementation requires all the procedures in the program to be inlined. However, with large applications, inlining is expensive, or even infeasible. How the
information from interprocedural analysis should be accounted for in the data-computation affinity relationships remains an open problem.

8.2.2 Dynamic Processor Geometries

The CDP algorithm derives a fixed processor geometry for the entire program. In some programs, it may be desirable to alter the processor geometry between loop nests. Determining the processor geometries for the individual loop nests and the cost of repartitioning (because of change in processor geometry) between the loop nests is another optimization problem for which integer programming or heuristics can be utilized. Analysis on large programs and problem sizes is required to determine the extent of benefits produced by dynamic changes in processor geometry.

8.2.3 Performance Estimation

The cost estimation heuristic will be accurate only when the number of iterations of loops and the size of arrays can be determined at compile time. However, in the absence of compile time information, default values are used by the prototype implementation. Like HPF compilers, the CDP algorithm and its prototype implementation require that the number of processors executing the application be known at compile time. In the absence of such information at compile time, threshold values can be used to determine a small set of data partitions, one of which can be selected at run-time. Further studies are required to analyze the benefit of such an approach.

8.2.4 Hybrid Multiprocessor Systems

The author envisions that large-scale systems will be built in the future by amalgamating distributed memory multiprocessor and SSMM technologies. Several SSMMs will be connected together with an interconnection network to form a large scale multiprocessor with several hundred processors. Exploiting parallelism on these machines will require not only data partitioning techniques designed for distributed memory multiprocessors but also the SSMM specific compilation techniques. The applicability and extensibility of the CDP algorithm for determining data and computation partitions on such hybrid machines requires further investigation.
Bibliography


This appendix outlines a technique to implement data partitions on SSMMs. This technique is utilized by the Jasmine compiler to map distributed array data onto memory modules.

On a sequential machine, accessing an array element requires the base address of the array and an offset from the base to the data element. Thus, a multidimensional array is stored as a contiguous linear segment with one base address. The offsets are computed based on array indices at run-time using compiler-generated expressions. This typical technique of accessing the array element with a base and an offset is illustrated in Figure A.1(a).

Data partitions are implemented on a SSMM using multiple base addresses. This produces several segments of the array: the base address of each segment is assigned to a processor according to the partitioning of data. All the segments accessed by a processor are grouped together and mapped onto pages that will be allocated in the local memory of the processor. This is depicted in Figure A.1(b). The array index functions for the different dimensions of the array are split into two sets: one set contributes to the base address and the other set is used to compute the offset from the base.

In general, the number of dimensions that contribute to defining the base address of an array can be reduced by adding more dimensions to the offset calculation. When all the dimensions are used to calculate the offset, this technique becomes the typical array indexing strategy used on sequential machines.

In our implementation, all dimensions except the cache line dimension are used to define the base address. The cache line dimension is used to compute the offset in each segment of the array. The partitioning of the data in the shared address space is realized by the allocation of the segments in the memory modules based on the distribution attributes.

Some data partitions do indeed partition the cache line dimension, and these data parti-
double a[N][N]

forall i = 1 to N (Cyclic)
    for j = 1 to N
        . . . a[j-1][i-1]

base = address(a)
offset = [(j-1)*N + (i-1)]

// P PROCESSORS

double* a[N]

forall i = 1 to N (Cyclic)
    for j = 1 to N
        . . . a[j-1][i-1]

base = address(a[j-1])
offset = [i-1]

(a) Operating System
(b) Explicit

Figure A.1: Data management in SSMM.
tions are implemented using one of the following two techniques. First, data transformations such as array transpose are applied so that the transformed array is partitioned along a dimension other than the cache line dimension. However, data transformations do not suffice when all dimensions of an array are partitioned. In such cases, the array is re-defined to have one additional dimension. by converting the cache line dimension of the array into the last two dimensions of the re-defined array. The last dimension of the re-defined array is chosen to be at least the size of a cache line. All dimensions except the last dimension of the re-defined array are partitioned.

On the KSR-1 machine, the partitioning of data takes on a slightly different meaning, because there is no fixed home location for array elements. In this case, data transformations are used to partition the array. All the partitioned dimensions are moved to higher dimensions of the array, using dimension permutation operations such as transpose, to ensure that data accessed by a processor are allocated contiguously.

Data partitions obtained using many-to-one, one-to-many, and many-to-many mappings of array dimensions to processor geometry dimensions cannot be implemented on the KSR-1 multiprocessor, because data is always copied into the local cache of the processor accessing the data. Automatic copying of data by the hardware makes it impossible to fix the location of data, which is necessary to implement the many-to-one, one-to-many, and many-to-many mappings of array dimensions to processor geometry dimensions. Accordingly, it is possible to implement only the one-to-one mapping of array dimensions to processor geometry dimensions on the KSR-1.

Data repartitioning of an array in the program represents the presence of different data partitions of the array. A copy of the array is created for each of the data partitions. Only one copy of the array is valid at any instance in the program. Data repartitioning entails copying the array from one data partition representation to another data partition representation.
This appendix describes the training set kernels that we used to estimate the program-dependent parameters described in Chapter 6. The kernel loop nests for estimating the memory contention factor, cost of synchronization, and the OCP factor are presented here.

### B.1 The Memory Contention Factor

The kernel for estimating the memory contention factor is shown below: it consists of two loop nests. In the first loop nest, all the processors access array elements that are assigned to processor #1. Memory contention results because all processors access data assigned to a single memory module. In the second loop nest, all the processors access array elements that are assigned to different processors. The contention factor is the ratio of the execution time of the first loop nest and the execution time of the second loop nest. A 2-dimensional array of size 512x512 is used so that the number of array elements accessed is large enough to reduce the errors due to timing.

```c
C$CDP A(*, Block{1})
C$CDP B(*, Block{1})
C$CDP C(*, Block{1})

// PROCS is the maximum number of processors in the system
// myPid is my processor id which ranges from 0..PROCS-1
// fromProc is the processor that will be contended for

fromProc = 1

// Do_Block is a macro that takes the loop iterator, start, end and
// the processor id of the iterations that the current processor
// must execute.
```
// End_Block is a macro for barrier synchronization

start = TIME()
Do_Block(i, 1, 512, fromProc)
   forall j = 1, 512
   end for
End_Block
end = TIME()

t1 = end - start

// cluster-size is the number of processors in a cluster that share
// the same memory module
// myNewPid is the remote processors data that myPid processor will access

myNewPid = (myPid + cluster_size) % PROCS

start = TIME()
Do_Block(i, 1, 512, myNewPid)
   forall j = 1, 512
   end for
End_Block
end = TIME()

t2 = end - start

contention_factor = t1/t2

B.2 The Cost of Synchronization

The kernel for estimating the cost of synchronization is shown below: it also consists of two
loop nests. The cost of synchronization is measured by timing two loops that are executed
in a pipelined manner. The barrier synchronization operations are present in the first loop
nest, but are removed in the second loop nest.

// For given PROCS, this kernel estimates the cost of synchronization

C$CDP A(*, Block[1])
C$CDP B(*, Block[1])
C$CDP C(*, Block[1])
NP = 512/PROCS

// Pipelined computation is used here because that's the
// synchronization that we want to measure
start = TIME()
for s = 1, 512+PROCS
    cur_start = max(s - 512, 1)
    cur_end = min(s, PROCS)
    for r = cur_start, cur_end
        j = s - r;
        if (r-1 .eq. myPid) then
            for i = (r-1)*NP+1, r*NP
                C(j, i) = B(j, i) + A(j, i)
            end for
        end if
    end for
BARRIER_SYNCHRONIZATION()
end for
end = TIME()

t1 = end - start

start = TIME()
for s = 1, 512+PROCS
    cur_start = max(s - 512, 1)
    cur_end = min(s, 512)
    for r = cur_start, cur_end
        j = s - r;
        if (r-1 .eq. myPid) then
            for i = (r-1)*NP+1, r*NP
                C(j, i) = B(j, i) + A(j, i)
            end for
        end if
    end for
end for
end = TIME()

t2 = end - start

Cost_of_synchronization = (t1 - t2)/(512+PROCS)
B.3 The OCP Factor

The kernel for estimating the overhead for partitioning the cache line dimension (OCP factor) is shown below: it too consists of two loop nests. This kernel is executed on a single processor to eliminate the overheads due to contention and synchronization. In the first loop nest, the arrays in which the cache line dimension is partitioned are accessed. Subsequently, in the second loop nest, the arrays in which the cache line dimension is not partitioned are accessed. The OCP factor is the ratio of the execution time of the first loop nest and the execution time of the second loop nest.

```
// PROCS = 1

start = TIME()
Do_Block(i, 1, 512, myPid)
    forall j = 1, 512
        F(i, j) = E(i, j) + D(i, j)
    end for
End_Block
end = TIME()
t1 = end - start

start = TIME()
Do_Block(i, 1, 512, myPid)
    forall j = 1, 512
        C(j, i) = B(j, i) + A(j, i)
    end for
End_Block
end = TIME()
t2 = end - start

OCP_factor = t1/t2
```