AUTOMATIC DATA PARTITIONING ON

DISTRIBUTED MEMORY MULTICOMPUTERS

Manish Gupta, Ph.D.
Department of Computer Science,
University of Illinois at Urbana-Champaign, 1992
Prithviraj Banerjee, Advisor

Distributed-memory parallel computers are increasingly being used to provide high levels of performance for scientific applications. Unfortunately, such machines are not very easy to program. A number of research efforts seek to alleviate this problem by developing compilers that take over the task of generating communication. The communication overheads and the extent of parallelism exploited in the resulting target program are determined largely by the manner in which data is partitioned across different processors of the machine. Most of the compilers provide no assistance to the programmer in the crucial task of determining a good data partitioning scheme.

This thesis presents a novel approach, the *constraint-based approach*, to the problem of automatic data partitioning for numeric programs. In this approach, the compiler identifies some desirable requirements on the distribution of various arrays being referenced in each statement, based on performance considerations. These desirable requirements are referred to as constraints. For each constraint, the compiler determines a quality measure that captures its importance with respect to the performance of the program. The quality measure is obtained through static performance estimation, without actually generating the target data-parallel program with explicit communication. Each data distribution decision is taken by combining all the relevant constraints. The compiler attempts to resolve any conflicts between constraints such that the overall execution time of the parallel program is minimized.

This approach has been implemented as part of a compiler called PARADIGM, that accepts Fortran 77 programs, and specifies the partitioning scheme to be used for each array in the program. We have obtained results on some programs taken from the Linpack and Eispack libraries, and the Perfect Benchmarks. These results are quite promising, and demonstrate the feasibility of automatic data partitioning for a significant class of scientific application programs with regular computations.

To my parents

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Prithviraj Banerjee, for his advice, support, and constant encouragement during the course of my research. It has been a pleasure working with him. I would also like to thank the members of my committee, Professors Laxmikant Kale, David Padua, Andrew Chien and Wen-Mei Hwu, for their valuable comments and suggestions on my work. Further thanks are due to Vas Balasundaram and Jeanne Ferrante for the technical discussions during my summer work at IBM, that considerably influenced my research.

Professor Constantine Polychronopoulos and his entire team of students working on the Parafrase-2 project deserve a special thanks for making that system available to us. In particular, I wish to thank Mohammad Haghighat, not only for doing a fine job with symbolic analysis in Parafrase-2, but also for his help with my queries about the system.

My numerous friends at Urbana made my stay at this town quite pleasant and comfortable. I did not anticipate how much I would enjoy these years, when I first came here to pursue graduate studies. A special note of thanks goes to all my friends.

My sister, Aarti, has always inspired me through her achievements in academic and non-academic activities that I too have often tried to accomplish, usually with only partial success. I would like to thank her for all her guidance and affection. I also wish to thank Sharad for his help on numerous occasions.

I cannot possibly expresss in words my gratitude towards my parents for their love, support, and the sacrifices they have made for me. I consider myself extremely fortunate to have such wonderful parents. It is to them that I dedicate this thesis.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Parallel machines are increasingly being used for providing high levels of performance for numerous applications. The technological advances in VLSI circuits have contributed enormously to the affordability of such machines. There are commercial products available today that interconnect several thousand powerful microprocessors, and are capable of delivering very high performance. The importance of such computers is bound to grow as the VLSI circuit speeds driving uniprocessor performance reach their physical limits. It is widely recognized that the constant demand for increased computational power can only be met by using massively parallel computers.

Unfortunately, software support for such systems is still lagging far behind the advances in hardware. In order to exploit the computational power of such machines, the programmer is currently forced to pay attention to very low-level implementation details. This leads to low productivity on the programmer's part and lack of portability in the resulting programs. Several researchers have proposed to tackle this problem for massively parallel computers through advanced compiler technology. However, the methodology for compile time analysis still remains insufficiently developed in certain critical areas. As a result, the programmer is still burdened with tedious tasks that could well be automated. This thesis presents our efforts to automate what many researchers regard as the most challenging step in parallelizing programs for massively parallel multicomputers, namely the partitioning of data on distributed memory machines. We shall now explain more precisely this problem and its significance.

1.1 Distributed-Memory Parallel Computers

Multiprocessor systems are commonly classified into the following two categories: shared-memory and distributed-memory parallel computers. The shared-memory machines provide all processors with access to a common global memory. Examples of such machines are the Alliant FX/2800, the Sequent Balance [74], the Cedar [45], and the IBM RP3 [19] machines. On a distributed-memory machine (multicomputer), each processor has direct access to only a small part of the total memory, which is distributed among processors. Examples of such systems are the Connection Machine CM-5 [75], the Intel iPSC/860 [36], and the NCUBE/2 [51].

In the context of building massively parallel systems, distributed-memory machines offer a tremendous advantage in terms of cost and scalability. Large shared-memory machines are much more difficult to build. Unfortunately, distributed-memory machines are relatively harder to program. One major reason for this difficulty is the absence of a single global address space. As a result, the programmer has to think about how data should be partitioned among various processes, and manage communication among the processes explicitly. This communication of data among processes via message passing is usually quite expensive, forcing the programmer to pay considerable attention to saving communication costs. Since the performance characteristics of communication relative to computation vary from one machine to another, the programmer ends up hard-wiring many machine-specific details into the code. Hence, a great deal of effort is required to port such programs to new machines.

1.2 Emerging Trend in Programming Support

The last few years have seen considerable research activity towards development of compilers that alleviate the problem of machine dependence and relieve the programmer of the burden of explicit message-passing. These compilers take a program written in a sequential or shared-memory parallel language, and based on user-specified partitioning of data, generate the target parallel program for a multicomputer. For most compilers, this parallel program corresponds to the SPMD (Single Program Multiple Data) model [39], where all processors execute the same program, but operate on distinct data items, thus enabling the exploitation of data-parallelism [28]. These research efforts include the Fortran D compiler [30, 31], and the SUPERB compiler [81], both accepting Fortran 77 as the base language. The Crystal compiler [15] and the Id Nouveau compiler [62] are targeted for single assignment languages. Numerous other compilers, Dataparallel C [59], C* [63], Kali [43, 44], DINO [64, 65], AL [77], ARF [67], Oxygen [66], Pandore [4] also produce parallel code for multicomputers, but require explicit parallelism in the source program. Some of the commercially available compilers for multicomputers are MIMDIZER [69] and ASPAR [35]. Many researchers associated with the development of such compilers in the industry and academia are currently involved in defining High Performance Fortran (HPF), a new Fortran standard.

Given the data distribution, the basic rule of compilation used by most of the systems described above is the *owner computes* rule [11], according to which it is the processor owning a data item that has to perform all computations for that item. Any data values required for the computation that are not available locally have to be obtained via interprocessor communication. Therefore, with such compilers, one of the most important factors affecting program performance is the data partitioning scheme. It is the method of data partitioning that guides the scheduling of computation (and hence determines which of the independent computations are actually executed on different processors), and determines when interprocessor communication takes place.

Determining a good partitioning of data across processors is potentially a difficult task that requires careful examination of numerous considerations. Normally, interprocessor communication is much more expensive than local computation. Therefore, the data partitioning scheme should ensure that each processor performs as much of computation as possible using just local data. Excessive communication among processors can easily offset any gains made by the use of parallelism. At the same time, the data partitioning scheme should allow the workload to be distributed evenly among processors, so that full use is made of the parallelism inherent in the computation. There is often a trade-off between minimizing interprocessor communication and balancing load on processors, and a good approach must take into account both the communication and the computational costs governed by the underlying architecture of the machine.

1.3 Motivation for Automatic Data Partitioning

Most of the compilers for multicomputers currently leave the crucial task of determining the data partitioning scheme to the programmer, and do not provide the programmer with any support in this task. While we believe that the efforts to develop techniques for automating the generation of communication do represent a step in the right direction, we also feel that many of the fundamental objectives behind developing such compilers remain incomplete as long as the programmer is forced to take decisions on the distribution of data. Some of the advantages of making the compiler determine data distribution are:

- *Reduced burden on the programmer*: The programmer is free to concentrate on the high-level design of the algorithm. The current state of parallel programming on multicomputers is often regarded as comparable to assembly language programming on sequential machines. Partitioning of data is, in that sense, analogous to register allocation, a task that should be relegated to the compiler.

- *Machine-independence*: In general, the best partitioning scheme for a program depends not only on the program characteristics, but also on numerous machine-specific parameters. Thus, a scheme that performs the best on one particular machine may perform poorly on other machines. Therefore, true portability can only be achieved if the partitioning scheme is not specified as part of the program.

- *Relationship with compiler optimizations*: A compiler generating the target parallel code with explicit message-passing applies a number of program transformations, and uses numerous optimizations that affect the final program performance. A user unfamiliar with those details may not have a good idea about the implications of some data partitioning decisions on the performance of the final, compiled code. A compiler would more easily be able to incorporate that information in the process of choosing the data partitioning scheme.

Let us now examine some of the arguments in favor of letting the programmer control data partitioning:

- *High-level knowledge*: The programmer is likely to have an intuition or high-level knowledge about how various data structures are related to each other, and where parallelism lies in the program. That should enable the programmer to make reasonably good guesses about the best data distribution scheme.
- *Parameters unknown at compile time*: Certain parameters such as array sizes and branching probabilities may be unknown at compile time, even for programs with "regular" computations. That would make it hard for the compiler to estimate program performance accurately and come up with good data partitioning decisions.
- *Unstructured computations*: Programs where the data reference patterns themselves are quite irregular would introduce inaccuracies and make any compile time analysis difficult. In such cases, the only reasonable option may be to let the programmer distribute data.

We believe that irregular computations, where the compiler is unable to gather precise information regarding dependences and data referencing behavior, do represent cases where the compiler is unlikely to come up with good partitioning schemes. However, for a significant class of regular numeric computations (most compilers described in the previous section are anyway expected to work well only on such applications), a compiler should be able to handle this problem well. We believe that for such computations, whatever intuition a good programmer has about the data structures, *does* get reflected in the data referencing patterns that can be analyzed by the compiler. Regarding the parameters that are unknown at compile-time, we feel that it is better to enhance the capabilities of compiler through techniques like profiling, symbolic analysis, and through user assertions about those parameters, rather than burdening the user with the more tedious and error-prone task of data partitioning.

1.4 Research Goals

Our primary goal has been to develop a methodology for automatic data partitioning, where given a sequential or a shared-memory parallel program, the compiler determines how its data should be distributed on different processors of a multicomputer. The objective is to get the best performance from the resulting parallel program when it executes on that multicomputer. Our focus in this research has been on scientific application programs. The only data structures these programs normally use are arrays that have to be partitioned across processors. Usually, the scalar variables are either replicated on all processors, or there are privatized copies kept on various processors. We shall concern ourselves mainly with the distribution of arrays.

We have identified some requirements that any compiler-driven approach to data partitioning must satisfy in order to be successful. These requirements have served as specific goals in our research:

- *Estimation of program performance*: Any strategy for automatic data partitioning needs a mechanism for comparing different alternatives regarding the data partitioning scheme for a given program. Therefore, the compiler should have the ability to estimate the performance of the target parallel program, given just the source program and data distribution information. The compiler has to estimate (i) the extent to which data-parallelism can be exploited, and (ii) the cost of interprocessor communication in the target program. Clearly, this analysis has to be done *before* the target program is generated. Thus there has to be a close link between the determination of data partitioning scheme and the generation of target data-parallel program with interprocessor communication.

- *Heuristics to reduce the search space*: For any program with a reasonably large number of arrays, the search space consisting of different partitioning schemes is enormous. In fact, many simple versions of the problem of data partitioning have been proved to be NP-complete [50, 48]. Thus, any approach must employ effective heuristics that lead to the pruning of a considerable part of the search space.

In addition to satisfying these requirements, our research has been guided by the goals of incorporating the following desirable characteristics into our approach:

- *Machine-independence*: There is need for a basic methodology that is machine-independent. It should then be possible to simply incorporate machine-specific details, and obtain an automatic data partitioning system for a given machine.

- *Intermediate choice points in the extent of automation*: By design, a data partitioning system should allow a range of possibilities between the two extreme positions of completely automatic and completely user-specified partitioning. Thus for regular computations where the compiler can gather sufficient information about the program, the system should be able to perform the task automatically. For other programs, the system should be able to work with directives from the programmer giving partial details on the method of partitioning. For instance, the programmer may specify a certain alignment between two arrays through a directive [29], and the compiler should be able to figure out the remaining details which the programmer does not want to worry about.

With these goals in mind, we have developed an automatic data partitioning system for multicomputers [23, 25]. Our system has been built on top of the Parafrase-2 compiler [57], which is used to provide vital information about the program, such as the parse tree structure, the data dependences, and the control-flow information. Our system, in conjunction with a small machine-dependent module that provides the cost model for the target machine, can be used on almost any distributed-memory machine. To allow validation of the underlying concepts, a version has been developed for the Intel iPSC/2 hypercube [5]. The system accepts Fortran 77 programs, and outputs a description of the partitioning scheme for each array used in the program. Our system also accepts Fortran programs in which the programmer explicitly marks the parallelizable loops as *doall* loops.

Clearly, any automatic data partitioning system has to be integrated with a compiler generating the target program with explicit communication. As we observed earlier, the data partitioning system has to estimate the performance corresponding to the *target* parallel program (though, without actually generating that program) [24]. Therefore, it needs to know the methodology that the compiler would finally use to generate the program with message passing. We approached the problem from the point of view of performance estimation. In the process, we identified techniques that can be used by a compiler to generate communication in a better manner [26]. These ideas have led to the evolution of our system into a complete compiler for multicomputers, now being developed by our research group. This compiler is called PARADIGM (PARAllelizing compiler for DIstributed-memory General-purpose Multicomputers). In our description, we shall concentrate on just the data partitioning aspects. Throughout the rest of this thesis, we shall refer to our automatic data partitioning system as "PARADIGM", or simply, "the compiler".

The remainder of this thesis is organized as follows. Chapter 2 describes our approach to the problem of automatic data partitioning, and gives an overview of the PARADIGM system. It also discusses related work in this area, and offers comparisons. Chapters 3 and 4 describe the methodology of estimating the performance of the target program, given the source program and data partitioning information. For reasons that shall be explained in Chapter 2, the performance estimation tool has been organized in the form of two separate modules, one handling the computational part, and the other dealing with communication. Chapter 3 describes the methodology for estimating the computational costs, while Chapter 4 describes the estimation of communication costs. Chapter 5 describes how decisions are taken on different aspects of the data partitioning scheme, in different passes of PARADIGM. Chapter 6 presents the results obtained by PARADIGM on some real Fortran codes taken from the Linpack and Eispack libraries, and the Perfect Benchmarks [17]. Finally, Chapter 7 presents conclusions, and discusses directions for future work in this area.

CHAPTER 2

AUTOMATIC DATA PARTITIONING

In this chapter, we describe our approach to the problem of automatic data partitioning on multicomputers. We refer to it as the *constraint-based approach* [23]. In this approach, the compiler analyzes each statement in every loop of the program, and based on performance considerations, identifies some desirable requirements on the distribution of various arrays being referenced in that statement. These desirable requirements are referred to as constraints. There is a quality measure associated with each constraint that captures its importance with respect to the performance of the program. Finally, the compiler tries to combine constraints for each array in a consistent manner so that the overall execution time of the parallel program is minimized. Before explaining what these constraints are, we first need to describe how the arrays may be distributed on different processors of the machine, and how those distributions are specified.

2.1 Data Distribution

The abstract target machine we assume is a D-dimensional mesh (D is the maximum dimensionality of any array used in the program) of N_1 x N_2 x ... x N_D processors. The use of a virtual topology allows our approach to be developed in a machine-independent manner, as long as that topology is supported by the actual target machine. The mesh topology can, in fact, be easily embedded on most distributed memory machines. A processor in the mesh is represented by the tuple $(p_1, p_2, \ldots, p_D), 0 \leq p_k \leq N_k - 1$ for $1 \leq k \leq D$. The correspondence between a tuple $(p_1, p_2, \ldots, p_D)$ and a processor number in the range 0 to $N-1$ is established by the scheme which embeds the virtual processor mesh on the real target machine. To make the notation describing *replication* of data simpler, we extend the representation of the processor tuple in the following manner. A processor tuple with an X appearing in the ith position denotes all processors along the ith mesh dimension. Thus for a 2 x 2 mesh of processors, the tuple $(0, X)$ represents the processors $(0,0)$ and $(0,1)$, while the tuple (X, X) represents all the four processors.

The scalar variables and small arrays used in the program are assumed to be either replicated, or privatized on all processors. For other arrays, we use a separate distribution function with each dimension to indicate how that array is distributed across processors. We refer to the kth dimension of an array A as A_k. Each array dimension A_k is mapped to a unique dimension $map(A_k)$, $1 \leq map(A_k) \leq D$, of the processor mesh. If $N_{map(A_k)}$, the number of processors along that mesh dimension is one, the array dimension A_k is said to be *sequentialized*. The sequentialization of an array dimension implies that all elements whose subscripts differ only in that dimension are allocated to the same processor. The distribution function takes as its arguments a dimension A_k and a subscript value i. It returns the component $map(A_k)$ of the tuple representing the processor which *owns* the element $A[-,-,\ldots,i,\ldots-]$, where '–' denotes an arbitrary value, and i is the subscript appearing in the kth dimension. The array dimension A_k may either be partitioned or replicated on the corresponding mesh dimension. The distribution function is of the form

$$f(A_k, i) = \begin{cases} \lfloor \frac{i - offset}{block} \rfloor [\bmod N_{map(A_k)}] & \text{if } A_k \text{ is partitioned} \\ X & \text{if } A_k \text{ is replicated} \end{cases} \tag{2.1}$$

where the square parentheses surrounding $\bmod N_{map(A_k)}$ indicate that the appearance of this part in the expression is optional. At a higher level, the given formulation of the distribution function can be thought of as specifying the following parameters: (1) whether the array dimension is partitioned across processors or replicated, (2) method of partitioning – blocked or cyclic, (3) the mesh dimension to which the kth array dimension is mapped, (4) the block size of distribution, i.e., the number of elements residing together as a block on a processor, and (5) the displacement applied to the subscript value for mapping.

Examples of some data distribution schemes possible for a 16 x 16 array on a 4-processor machine are shown in Figure 2.1. The numbers shown in the figure indicate the processor(s) to which that part of the array is allocated. The machine is considered to be an N_1 x N_2 mesh, and the processor number corresponding to the tuple (p_1, p_2) is given by $p_1 * N_2 + p_2$. The distribution functions corresponding to the different figures are given below. The array subscripts are assumed to start with the value 1, as in Fortran.

a)	$N_1 = 4,\ N_2 = 1:$	$f(A_1, i) = \lfloor \frac{i-1}{4} \rfloor,$	$f(A_2, j) = 0$
b)	$N_1 = 1,\ N_2 = 4:$	$f(A_1, i) = 0,$	$f(A_2, j) = \lfloor \frac{j-9}{4} \rfloor$
c)	$N_1 = 2,\ N_2 = 2:$	$f(A_1, i) = \lfloor \frac{i-1}{8} \rfloor,$	$f(A_2, j) = \lfloor \frac{j-1}{8} \rfloor$
d)	$N_1 = 1,\ N_2 = 4:$	$f(A_1, i) = 0,$	$f(A_2, j) = (j-1) \bmod 4$
e)	$N_1 = 2,\ N_2 = 2:$	$f(A_1, i) = \lfloor \frac{i-1}{2} \rfloor \bmod 2,$	$f(A_2, j) = \lfloor \frac{j-1}{2} \rfloor \bmod 2$
f)	$N_1 = 2,\ N_2 = 2:$	$f(A_1, i) = \lfloor \frac{i-1}{8} \rfloor,$	$f(A_2, j) = X$

The last example illustrates how our notation allows specification of *partial* replication of data, i.e., replication of an array dimension along a specific dimension of the processor mesh. An array is replicated completely on all the processors if the distribution function for each of its dimensions takes the value X.

If the dimensionality (D) of the processor topology is greater than the dimensionality (d) of an array, we need $D - d$ more distribution functions in order to completely specify the processor(s) owning a given element of the array. These functions provide the remaining $D - d$ numbers of the processor tuple. These "functions" are restricted to constant values, or the value X if the array is to be replicated along the corresponding mesh dimension.

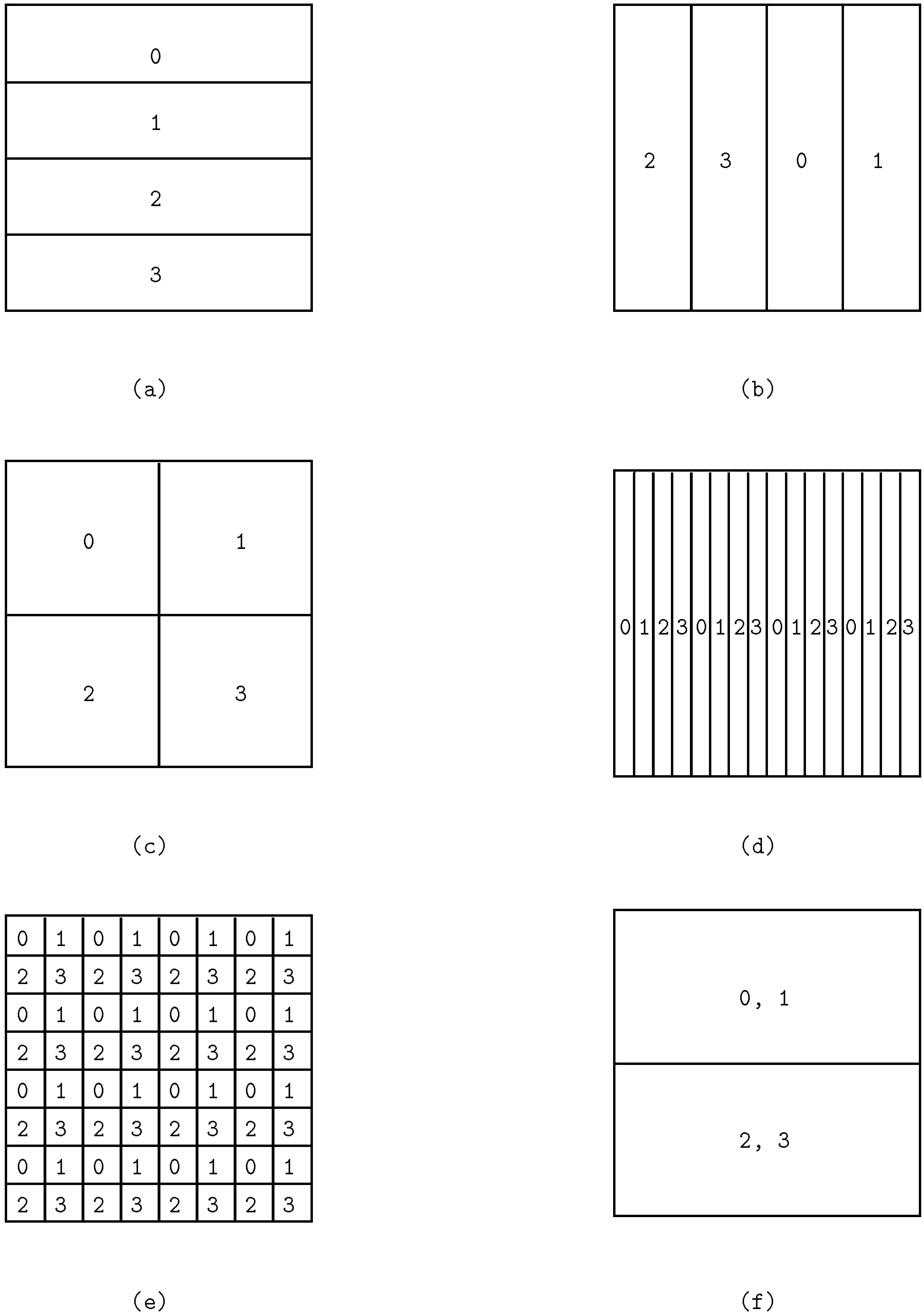

Figure 2.1: Different data partitions for a $16 * 16$ array

2.2 Constraint-Based Approach

As mentioned earlier, our approach is based on the analysis of array references in various statements inside every loop in the program. This analysis identifies some desirable restrictions on the distribution of those arrays, that are referred to as *constraints* on data distribution. Our use of this term differs slightly from its common usage in the sense that constraints on data distribution represent requirements that *should* be met, and not requirements that necessarily *have* to be met.

2.2.1 Significance of Constraints

For each statement assigning values to an array A in a parallelizable loop, and using the values of an array B, there are two kinds of constraints: *parallelization constraints* on the distribution of A, and *communication constraints* on the *relationship* between distributions of A and B (this includes the special case when the `lhs` array A is the same as the `rhs` array B). The parallelization constraints try to ensure even distribution of the array elements being assigned values in that loop, and on as many processors as possible. The objective there is to obtain good load balance, and hence performance gains through exploitation of parallelism. The communication constraints try to ensure that the data elements being read in a statement reside on the same processor as the one that owns the data element being written into. That would make the values required for a piece of computation available locally on the processor carrying out that computation, thus eliminating the need for interprocessor communication.

We showed in the previous section how the distribution of an array on the target machine (with a virtual mesh topology) is specified through separate distribution functions for each of the array dimensions. It was shown that each distribution function is characterized by distinct parameters such as the mesh dimension to which that array dimension is mapped, the method of partitioning, the number of processors, etc. A single constraint on the distribution of an array dimension usually specifies the value of only one of these parameters, and a constraint on the relationship between two array dimensions specifies how the values of a given parameter for

```
      do 20 mi = 1, morb
         xrsiq(mi, mq) = xrsiq(mi, mq) + val * v(mp, mi)
         xrsiq(mi, mp) = xrsiq(mi, mp) + val * v(mq, mi)
20    continue
      ⋮
      do 70 mq = 1, nq
         ⋮
         do 60 mj = 1, mq
60          xij(mj) = xij(mj) + val * v(mq, mj)
70    continue
```

Figure 2.2: Program segment from `trfd`

their respective distribution functions should be related. For instance, an *alignment* constraint between two array dimensions specifies that the two dimensions should be mapped to the same mesh dimension.

Intuitively, the notion of constraints provides an abstraction of the significance of each statement with respect to data distribution. The distribution of each array involves taking decisions regarding a number of parameters, and the constraints corresponding to a statement specify requirements on only the parameters that affect the performance of the given statement. This often helps combine requirements from different parts of the program. Consider the program segment shown in Figure 2.2, taken from a real life scientific application code, the `trfd` program from the Perfect Benchmarks.

The statements in the first loop lead to the following communication constraints – alignment of $xrsiq_1$ with v_2, identical distributions for $xrsiq_1$ and v_2, and the sequentialization of $xrsiq_2$ and v_1. The satisfaction of these constraints would ensure communication-free execution of those statements in the entire loop. Similarly, the statement in the second loop advocates alignment and identical distributions of xij_1 and v_2, and the sequentialization of v_1. The parallelization constraint for that statement suggests cyclic distribution of xij_1, in order to obtain better load balance (since the extent to which xij_1 is accessed in the inner loop varies in different iterations of the outer loop). All of these requirements are consistent with each other.

Together, they imply a column-cyclic distribution for v, a row-cyclic distribution for $xrsiq$, and a cyclic distribution for xij (the constraint on cyclic distribution gets passed on from xij_1 to v_2, and from v_2 to $xrsiq_1$). Thus, the use of constraints facilitates incorporating requirements from different parts of the program, when those requirements pertain to different aspects of the data partitioning scheme.

2.2.2 Conflicts among Constraints

The example shown above corresponded to an "easy" case for data partitioning. In general, different parts of the program may impose conflicting requirements on the distribution of various arrays, through constraints inconsistent with each other. In order to resolve those conflicts, the compiler records a measure of *quality* with each constraint. For the *boolean* constraints, which are finally either satisfied or not satisfied by the data distribution scheme (for example, an alignment constraint between two array dimensions), the quality measure is an estimate of the penalty paid in execution time if that constraint is not honored. For other constraints governing the value to be used for a data distribution parameter, such as the number of processors, the quality measure is an estimate of the parallel execution time as a function of that parameter. Depending on whether a constraint affects just the amount of parallelism exploited or the interprocessor communication requirement, or both, the expression for its quality measure has terms for the computation time, the communication time, or both.

One may oberve that determination of the quality measures of various constraints requires special features in the performance estimation methodology. There is a need to isolate the contribution of (all instances of) a single statement to the total program execution time. In fact, there is a further need to characterize the times spent on communication of values for each individual array referenced in a statement. For this reason, we keep separate accounts of the times spent in computation and communication for each statement associated with the recording of constraints.

One problem in determining the quality measures of constraints is that their value may depend on certain parameters of the final distribution scheme that are not known beforehand.

The problem is one of circularity: these estimates are needed in the first place to help determine a good distribution scheme, and unless the distribution scheme is known, it is difficult to come up with good estimates to guide the selection. We shall first present an overview of the design of PARADIGM, and then explain how this problem is resolved in the system.

2.2.3 Overview of the System

The structure of the overall system for automatic data distribution is shown in Figure 2.3. PARADIGM has been developed as an extension to the Parafrase-2 system [57]. Like the base system, it is organized in the form of passes through the program. Parafrase-2 builds an internal representation for the program, providing information such as the parse tree structure, the data dependences, and the control flow information. It also performs constant propagation and induction variable recognition. The internal program representation kept by Parafrase-2 has been extended to incorporate additional information that PARADIGM needs about the program, such as the count of operations in assignment statements, and the canonical representations of subscript expressions. This additional information is recorded in the `scc` and the `setup` passes, which are the first two passes to be invoked in PARADIGM following those of Parafrase-2. Further details are given along with our discussion of the methodology for estimation of computational costs in Chapter 3, and of communication costs in Chapter 4.

The decisions on data distribution scheme are taken in a number of distinct passes through the program. Each pass takes decisions on a single parameter for the distribution functions of all the arrays. The `align` pass maps each array dimension to a processor-mesh dimension, on the basis of the alignment constraints among various array dimensions. The `block-cyclic` pass determines for each array dimension, whether it should be distributed in a blocked or cyclic manner. The `block-size` pass determines the block size to be used for each dimension distributed in a cyclic manner. The `num-procs` pass determines the number of processors in each of the processor-mesh dimensions to which various array dimensions are mapped.

Each of the above passes determining a distribution parameter is organized in the form of three modules. The `detector` module detects opportunities for recording a constraint on the

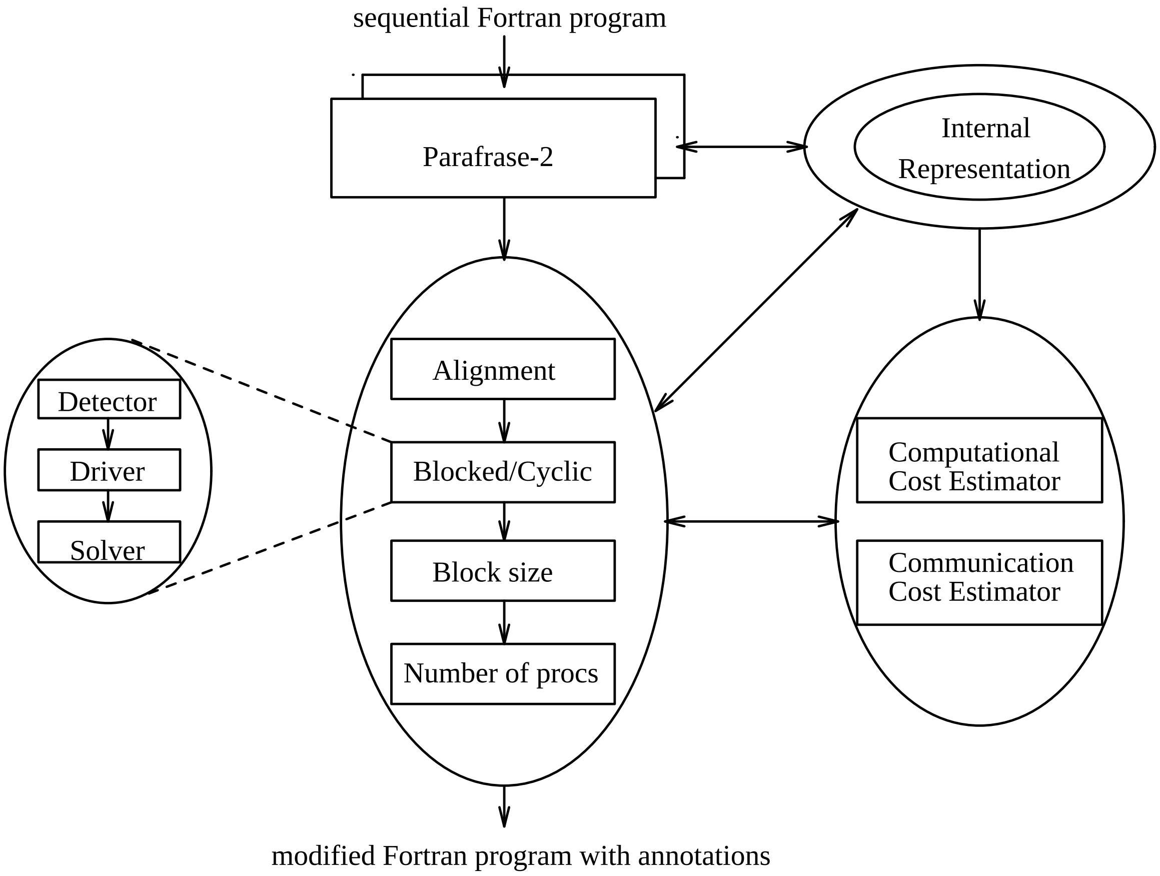

Figure 2.3: Overview of automatic data partitioning in PARADIGM

given distribution parameter for any array. The `driver` module invokes the communication cost estimator and/or the computational cost estimator to obtain the quality measure of that constraint. For example, to obtain the quality measure of an alignment constraint, the communication cost estimator is invoked twice – once to return estimates when the given array dimensions are properly aligned, and the next time to give cost estimates when those dimensions are not aligned. The quality measure recorded is the difference between these costs. Once all the constraints affecting the given distribution parameter and their quality measures have been recorded, the `solver` determines the value of that parameter, by solving the corresponding optimization problem. The details regarding each of the passes are presented in Chapter 5.

Let us now consider the problem of circularity in obtaining the measures through performance estimation. This problem is dealt with through a combination of two techniques. During the early stages, the compiler uses "reasonable" values of parameters not known at that point, based on some simplifying assumptions. For instance, while determining the quality measures of alignment constraints, the compiler assumes that each array dimension is distributed in a blocked manner, and on an equal number of processors. In successive passes, as decisions are taken on different aspects of the partitioning scheme, the amount of information available to the compiler increases, thus enabling more accurate performance estimation to guide the choice of the remaining distribution parameters. Note that the number of processors in each mesh dimension is not determined until the very end. Therefore, another technique we use is to express all performance estimates in a symbolic form, with the number of processors in different mesh dimensions appearing as parameters in those expressions. This eliminates the need for repeated program analysis to obtain different performance estimates, when the only parameters changing are the number of processors in various mesh dimensions.

2.3 Related Work

Due to the close relationship between the problem of automatic data partitioning and of generating data-parallel programs, our work is related and relevant to numerous research projects on compiling for distributed memory machines. We have described some of that work in the previous chapter. In this section, we only examine the research efforts that have addressed the problem of automatically determining a data partitioning scheme, or of providing help to the user in this task.

Mace worked on the problem of selecting memory storage patterns (shapes) [50] for vector machine environments, particularly those using memory interleaving. Using a graph-theoretic framework for computation, she shows that the problem of finding optimal data storage patterns for parallel processing, even for 1-D and 2-D arrays, is NP-complete. Those results are valid for distributed memory machines as well.

Ramanujan and Sadayappan have worked on deriving data partitions for a restricted class of programs [60]. They present a matrix notation to describe array accesses in parallel loops and derive conditions for communication-free partitioning of arrays [61]. Their approach is mainly directed towards individual loops, and they do not discuss applying these ideas to complete programs, which might have conflicting requirements on the partitioning scheme.

Hudak and Abraham [34], and Socha [71] present data partitioning techniques for sequentially iterated parallel loops, based on the access patterns inside those loops. Their work allows for more general data distributions than those described by us, but it may be difficult for most compilers to generate efficient communication for such complex distributions. These approaches have the limitation of restricted applicability, they apply only to programs that may be modeled as a single, multiply nested loop structure.

Tseng describes the AL compiler [77] that performs data mapping once the programmer chooses one dimension of each array that is to be distributed on the linear array of processors in the WARP machine. The techniques used in the AL compiler need to be extended significantly to allow the distribution of multiple dimensions, and to recognize automatically which dimensions to distribute.

Knobe, Lukas, and Steele have developed techniques for automatic data layout on SIMD machines [41, 42]. They use the concept of *preferences* between data references to guide the layout process, which is similar in spirit to our use of constraints to guide the choice of data distribution parameters. A significant feature unique to our approach is the analysis carried out to record the quality measure with each constraint, which leads to a much more precise characterization of the "weight" to be attached to each constraint.

Balasundaram, Fox, Kennedy, and Kremer discuss an interactive tool that provides assistance to the user in determining the data distribution [6, 40]. The key element in their tool is a performance estimation module, which is used to evaluate various alternatives regarding the distribution scheme. They use the method of "training sets" to help estimate the performance of a program with message passing [7]. Those techniques need to be extended to allow performance estimation, given just the source program and data partitioning information, without actually translating it to one with explicit message passing.

Our research has been influenced by the work of Li and Chen on the Crystal compiler [48, 47], and theirs is probably the most closely related to our work. In [48] they discuss the problem of alignment between array dimensions, show it to be NP-complete, and present a heuristic algorithm for that problem. However, the measures they use to capture the importance of any given alignment are somewhat simplistic. PARADIGM obtains more appropriate measures by estimating the penalty in communication costs if the array dimensions are not aligned the proper way. Li and Chen also describe how explicit communication can be synthesized and communication costs estimated by pattern-matching on data references in the source program [47]. We have introduced the notion of synchronous properties between array references that enables our compiler to perform more sophisticated analysis.

A feature common to the approaches proposed by Kremer and Kennedy [40], and Li and Chen [47] is that the data partitioning decisions (except for the decisions on alignment) are based on comparisons between almost all possible alternatives regarding those schemes. An advantage of this approach is that the performance estimation can now be done in a more accurate manner. However, as the problem size is increased, the number of possibilities to consider becomes too large, unless additional heuristics are incorporated to prune the search space.

Chapman, Herbeck and Zima describe the features of a knowledge-based interactive tool [12] to provide support for automatic data distribution. Their tool relies on program analysis and pattern-matching techniques, in conjunction with the use of a knowledge base to guide the search for a good data partitioning scheme. This tool is being developed as part of the second generation of the SUPERB compiler [81].

Wholey describes an automatic data mapping system for the ALEXI language [78]. The problem of performance estimation is simpler compared to that for Fortran 77 programs, since the ALEXI programs already have the calls to primitives to carry out high-level communication and parallel array operations. The ALEXI compiler uses a detailed cost model for performance estimation to guide data partitioning decisions. The compiler does not deal with the problem of alignment conflicts between arrays.

O'Boyle and Hedayat describe application of transformations in a Sisal compiler to achieve better data alignment and load balancing [52, 53, 54]. One of the main contributions of their work is in providing a linear algebraic framework to apply those transformations. However, they do not discuss resolving conflicts between alignment requirements of different parts of the program. While their framework is quite elegant, we believe it is not very well-suited to modeling communication costs accurately, since it ignores many relevant factors, like the nature of communication primitives.

CHAPTER 3

ESTIMATION OF COMPUTATIONAL COSTS

This chapter describes the methodology used to estimate the computational part of the parallel program execution time. By computational part, we simply mean the execution time of the parallel program if all communications take zero time. These estimates help determine the quality measures of parallelization constraints on the distribution of various array dimensions. Specifically, those measures are used to guide the choice of blocked or cyclic distributions, and the number of processors on which the array dimensions are distributed.

The basic approach we use for performance estimation of programs is to determine the contribution of each statement to the overall program execution time. Thus, to estimate the sequential execution time for a program, the compiler would estimate the following items for each statement in the program: (i) time taken to carry out the computation of a single instance of that statement, and (ii) count of the number of times that statement is executed. Extending this analysis to estimate the parallel execution time requires an understanding of how the compilers for multicomputers expose parallelism in programs.

The methodology used by compilers to generate parallel code for multicomputers relies mainly on the exploitation of data-parallelism. Logically, the unit of computation that is scheduled on a processor is a single statement (as opposed to an entire iteration, typical in the case of control-parallelism). As a consequence of the *owner computes* rule, computations for different instances of a statement (corresponding to different iterations of a loop) are partitioned according to how the `lhs` data elements (the data elements being assigned values) are distributed. The overall computation time is determined by when the last processor finishes its share of computation. If there is no flow dependence between different instances of the given statement, the compiler simply has to determine the time taken by the processor with maximum load. If there are flow dependences, the compiler has to account for the synchronization delays as well. For example, consider a flow dependence from $S(i)$ to a later instance $S(j)$, $i < j$, of the statement. If $S(i)$ and $S(j)$ are executed by different processors, the compiler has to recognize that the computation for $S(j)$ will start only after the computation for $S(i)$ is complete. In addition, there would be a further delay due to communication of the value computed by $S(i)$ that is used by $S(j)$. However, we keep a separate account for the communication costs. In this chapter, we describe estimation of only the computational part of the overall execution time. The next chapter describes our methodology for estimating the communication times.

We first present some relevant details of the internal representation kept for a program, and how the computational costs pertaining to all instances of a given statement are estimated. It is important to keep in mind that the purpose of the estimation process is not to predict the actual execution time of the program. The objective is to guide the selection of data partitioning scheme by determining the performance implications of choosing some data partitioning parameters a certain way for a given statement.

3.1 Program Information

This section describes some information regarding the source program recorded by Parafrase-2 and the PARADIGM compiler, and terms that we shall use in later discussions.

Dependence Information The `builddep` pass in Parafrase-2 builds a data dependence graph [46] that keeps the information regarding all data dependences in a program. Each dependence is labeled as a flow, anti, or output dependence. Associated with each dependence edge representing a dependence from statement S1 to statement S2, both nested in n loops, is a direction vector $(d_1, d_2, \ldots, d_n)$, where $d_i \in \{<, =, >, \leq, \geq, \neq, *\}$ [79]. The direction vector describes the direction of dependence for each loop, with d_1 describing the direction of dependence for the outermost loop, and successive components of the vector describing directions for the inner loops. The forward direction "$<$" implies that the dependence is from an earlier to a later iteration of the corresponding loop, "$=$" implies that the dependence does not cross an iteration boundary, and "$>$" means that the dependence crosses an iteration boundary backwards. The other four directions are simply combinations of these three basic directions, and are associated with imprecision in data dependence tests. In Fortran `do` loops, a backward direction can occur only if there is a forward direction in an outer loop. Another way this fact is often expressed is that every legal direction vector has to be non-negative, where the directions "$<$","$=$", and "$>$" are expressed as $+1, 0$ and -1, respectively.

Once the data dependence graph is built, the `dotodoall` pass in Parafrase-2 determines for each loop whether it can be parallelized or not. A loop at level k is regarded as sequential if there is at least one dependence edge between nodes representing statements in that loop that has a direction vector of $(d_1, d_2, \ldots, d_n)$ satisfying the following properties: (i) $d_k \in \{<, \leq, \neq, *\}$ (ii) $\forall i$ in $[1, k-1], d_i \notin \{<, >, \neq\}$. These conditions check for the existence of a cross-iteration dependence that it is not satisfied by the sequentiality of an outer loop. The significance of ignoring a dependence in the "$>$" direction at level k lies in the fact that a "$>$" direction can occur only when there is a "$<$" direction occuring at an outer level [79]. The remaining loops are marked `doall` in this step.

The information about control dependences was not yet available in our version of Parafrase-2. Hence for simplicity, the only dependences we have considered in this work are the data dependences in a program. Control dependences can be handled by converting them to data dependences [3], or by using a different program representation, the *program dependence graph* [20] instead of a data dependence graph.

Strongly Connected Components in Dependence Graph The `scc` pass in PARADIGM operates on the data dependence graph built during the `builddep` pass of Parafrase-2, and identifies the strongly connected components in the graph [72]. Initially the compiler determines `max-level`, the maximum nesting level of any loop in the program. Since the extent of parallelism has to be estimated at all loop levels, the compiler determines the strongly connected components at all levels varying from 1 to `max-level` [2]. We shall briefly describe here the notion of `level-k` dependence graph and `level-k` strongly connected components. The interested reader is referred to [2] for further details.

Given a dependence graph DG, the corresponding `level-k` graph DG_k is derived as follows. The nodes included in DG_k are those corresponding to statements nested at level k or greater. An edge from DG between these nodes is included in DG_k if and only if the associated direction vector $(d_1, d_2, \ldots, d_n)$ satisfies the following properties: (i) $d_k \neq$ "$>$" (ii) $\forall i$ in $[1, k-1], d_i \notin \{<, >, \neq\}$. These conditions lead to any dependence satisfied by the sequentiality of loops outside level k to be ignored. Thus, DG_k may be regarded as the graph derived from DG by considering only the edges at nesting level k or greater. The `scc` pass in the compiler identifies the strongly connected components (SCCs) in each such graph DG_k "built" for a loop at level k. For each statement at nesting level m, the compiler records information identifying the SCC it belongs to at each of the levels 1 through m.

Array References and Computation Times During the `setup` pass, PARADIGM traverses the expression tree corresponding to each assignment statement to record a list of all the variables (both scalar and array variables) referenced in that statement. During this traversal, the compiler also keeps a count of the number of loads and individual arithmetic operations in the computation. Given information on the times taken to carry out each of those operations on the target machine, these counts are now used to record the execution time estimates for a single instance of each statement. Our machine-specific version of PARADIGM uses simple timing figures for the Intel iPSC/2 hypercube for this purpose. For each array reference, the compiler records additional information about each subscript expression, as described below.

Subscript Types PARADIGM analyzes the expression corresponding to each subscript, and assigns it to one of the following categories:

- `constant`: if the subscript expression evaluates to a constant at compile time.

- `single-index`: if the subscript expression reduces to the form $\alpha * i + \beta$, where α, β are integer constants and i is a loop index.

- `multiple-index`: if the subscript expression reduces to the form $\alpha_1 * i_1 + \alpha_2 * i_2 + \ldots + \alpha_k * i_k + \beta$, $k \geq 2$, where $\alpha_1, \ldots, \alpha_k$ and β are integer constants, and $i_1, \ldots, i_k$ are loop indices.

- `unknown`: this is the default case, and signifies that the compiler has no knowledge of how the subscript expression varies with different iterations of a loop.

The canonical form of the subscript expression shown above is recorded if the subscript is of the type `constant`, `single-index` or `multiple-index`. For each subscript, the compiler also determines the value of a parameter called `variation-level`, which is the level of the innermost loop in which that subscript changes its value. For a subscript of the type `constant`, it is set to zero. For all other subscript types, the compiler first determines the loop level corresponding to the innermost loop index that appears in the subscript expression. The `variation-level` of a subscript of the type `single-index` or `multiple-index` is set to that value. For a subscript of the type `unknown`, the compiler also constructs a list of all the variables appearing in the subscript expression. It examines the nesting levels of all the statements from which there is a flow dependence to the given statement due to one of the variables in the list. The minimum of these levels and that of the innermost loop index (if any) appearing in the subscript expression yields the `variation-level` of that subscript. This gives the compiler additional information about the "unknown" subscript, namely, the loops with respect to which that subscript expression is an invariant.

Based on our experience with some real-life applications, we have found keeping the following additional information about subscripts to be useful. Consider a subscript expression of the form $\alpha_1 * i_1 + \ldots + \alpha_k * i_k + x$, where x is an expression of the type `unknown` with a `variation-level` smaller than that of the first part, $\alpha_1 * i_1 + \ldots + \alpha_k * i_k$. In such a case, the overall subscript is regarded as of the type `unknown`, but the compiler also records the first part of the expression. That enables the compiler to detect regularity in the variation of subscript for the innermost k

$$
\begin{array}{l}
\text{do } j_1 = i_1, f_1, s_1 \\
\quad \vdots \\
\quad \text{do } j_2 = i_2, f_2, s_2 \\
\quad\quad \ddots \\
\quad\quad\quad \text{do } j_m = i_m, f_m, s_m \\
\quad\quad\quad\quad \vdots \\
\quad\quad\quad\quad A(g_1, g_2, \ldots, g_p) = \cdots
\end{array}
$$

Figure 3.1: Assignment statement with array references

loops. Currently, Paradigm carries out this analysis only for $k = 1$, i.e., for subscripts of the form $\alpha_1 * i_1 + x$.

Iteration Counts and Reaching Probabilities for Statements Consider the statement S shown in Figure 3.1. The estimates of execution time spent on S depend on the iteration count of the surrounding loops, and on the branching probabilities of various conditionals in the program that affect the flow of control to that statement. The loops surrounding the statement need not form a perfectly nested structure. When the initial value i_l, the final value f_l, and the stride s_l of the loop index for L_l are constants, the iteration count of the loop can easily be determined as $n_l = \lfloor (f_l - i_l)/s_l \rfloor + 1$ if $f_l \geq i_l$, otherwise n_l is set to zero. Tawbi and Feautrier present an algorithm [73] that computes an approximate count of the total number of iterations when the expressions for i_l and f_l are linear functions of the surrounding loop indices. Their algorithm computes an approximate solution to the problem of counting the number of integer points in the bounded convex polyhedron corresponding to the loop nest. However, our methodology for performance estimation requires explicit estimates for the expected value of each of those loop bounds, particularly for estimating communication costs. These expected values should be chosen such that the product of the iteration counts computed for each loop is equal to the total iteration count for the entire loop nest (note that the available algorithms compute the symbolic form only for the count for the complete loop nest).

```
do i = 1, n
   do j = i, n
      S_1
   enddo
enddo
```

Figure 3.2: Variation of inner loop bounds with outer loop index

For a commonly occurring case where only one such inner loop has bounds that are linear functions of an outer loop index (say, i), the solution is quite straightforward. The compiler simply replaces each occurrence of i in the expression for loop bound by the *expected value* of i, which is set to the arithmetic mean of the lower and upper bounds of the corresponding loop. For example, in the following loop, the expected value of i is set to $(n+1)/2$. In more complex cases, and when the expressions for loop bounds involve other variables having values unknown at compile-time, the compiler queries the user for the expected values of loop bounds. A more sophisticated system can obtain these values through profiling [68].

In general, due to the presence of conditionals, the number of times a statement is executed during the program would be less than or equal to the product of iteration counts of all the loops surrounding it. For each statement S surrounded by m loops, $m \geq 1$, we define a quantity called its *reaching probability*, that models the effect of all conditionals in the program from which there a control dependence to S. The reaching probability is defined as:

$$\rho(S) = num(S)/(\prod_{l=1}^{m} n_l),$$

where $num(S)$ is the number of times S is executed during the program, and n_l denotes the expected value of the iteration count of loop L_l. Probably the best way to determine $\rho(S)$ for each relevant statement S is to use profiling [68]. The current version of PARADIGM follows a relatively simpler approach where the user supplies these values for any statement having a reaching probability significantly lower than one. For all other statements, the default value assumed is one.

3.2 Computational Cost for a Statement

Consider again the statement shown in Figure 3.1. The first step in the estimation process is to determine the loops with respect to which computations involving the given statement S can be executed in parallel. In the next step, the compiler uses data partitioning information to determine how those computations would be partitioned on processors.

3.2.1 Identification of Loop Level Parallelism

The algorithm in Figure 3.3 determines the innermost loop level $lmax$ at which the statement S is involved in a recurrence. The compiler infers that the statement S can be parallelized with respect to all the loops nested at levels greater than $lmax$, and also with respect to all the loops outside L_{lmax} that are marked `doall` during the `dotodoall` pass. (A loop at level k is referred to as L_k). Thus, for a statement S to be parallelizable with respect to a loop, the characterization of that loop as a `doall` at the end of the `dotodoall` pass is a sufficient, and not a necessary condition. In general, exploiting parallelism at all these levels requires transformations like loop distribution and loop interchange, which can be identified using this algorithm. Any `doall` loops outside the loop L_{lmax} need to be brought inside through loop interchange. Whenever the source S' of a loop-carried dependence and S belong to different SCCs in the *level-l* dependence graph DG_l (see line 7 of the algorithm), if l takes a higher value than the eventual value of $lmax$, the loop L_l should be distributed over the components corresponding to statements S' and S. This would enable parallelization of the statement S with respect to L_l, which would not be possible otherwise.

Legality of ignoring output and anti dependences We shall now explain why the compiler can safely ignore all output and anti dependences to statement S in the given algorithm. Consider an output dependence from a statement instance $S'(i)$ to a statement instance $S(j)$. By the definition of output dependence, $S'(i)$ and $S(j)$ must write into the same data element. Hence by the *owner computes* rule, the same processor must perform the computation of both

$lmax = 0$
for each dependence into S **do**
  if ((dependence-type == flow) and (dependence is loop-carried))
    S' = source statement of dependence
    determine k = (innermost) loop level at which dependence is carried
    for $(l = k; l \geq 1; l--)$ **do**
      if S and S' belong to same SCC in DG_l
        break
      endif
    endfor
    $lmax = max(l, lmax)$
  endif
endfor

Figure 3.3: Algorithm to determine loop level parallelism for S

of those statement instances. Therefore, output dependence cannot place any constraints on the relative execution order of computations assigned to different processors. In other words, output dependence does not lead to any loss of effective parallelism (beyond that caused by the *owner computes* rule). Now consider an anti dependence from $S'(i)$ to $S(j)$. Such a dependence implies that $S'(i)$ has to use the value of a data element (say, x) before $S(j)$ assigns a value to it. For this dependence to impose an ordering between computations on different processors, the two statement instances $S'(i)$ and $S(j)$ must be executed by different processors. The translation process followed by the compiler would cause the processor owning x to send that value to the processor executing $S'(i)$, and would also ensure that the correct value of x is sent before $S(j)$ overwrites that value. Hence, regardless of the relative order of execution of $S'(i)$ and $S(j)$, the underlying data dependence is honored. Thus there is no ordering constraint between computations involved in an anti dependence, that take place on different processors.

3.2.2 Contribution of Each Loop to Computational Cost

We now describe how PARADIGM estimates the contribution of all instances of statement S to the parallel execution time (excluding the time spent on communication). The estimated time is given by the following expression:

$$T(S) \quad = \quad \rho(S) * (\prod_{l=1}^{m} C_l(S)) * t(S) \tag{3.1}$$

where $\rho(S)$ is the reaching probability for S, $t(S)$ is the estimated execution time for a single instance of S, and $C_l(S)$ is the contribution term from the loop L_l.

The value of $C_l(S)$ depends on the iteration count n_l of the loop L_l, and on the extent to which the statement S is effectively parallelized in different iterations of that loop. The algorithm to determine the $C_l(S)$ terms proceeds from the innermost loop ($l = m$) to the outermost loop ($l = 1$). Each step is based on the following two cases:

1. If the computations for S are marked sequential with respect to L_l, or if no subscript in the `lhs` reference $A(g_1, g_2, \ldots, g_p)$ has a `variation-level` of l, $C_l(S)$ is set to the iteration count n_l. The first condition represents the lack of available parallelism, while the second condition represents the entire computation corresponding to n_l iterations being mapped to a single processor.

2. If the computations for S are marked parallelizable with respect to L_l, and if there is a subscript g_k with `variation-level` l, the contribution term $C_l(S)$ is set to the number of iterations (for statement S) that the processor with maximum load is assigned. The expression for that depends on the type of subscript g_k. Clearly, the subscript cannot be of the type `constant`, since a constant subscript must have a `variation-level` of zero. Thus, there are three possibilities regarding the type of subscript:

   - `single-index` : let $g_k = \alpha_1 * j_l + \beta_1$, where α_1 and β_1 are constants. Let the array dimension A_k be distributed on N_k processors, with a block size of b_k. We define the loop range r_l to be $f_l - i_l + s_l$ if $f_l \geq i_l$, and zero otherwise. Given that A_k is distributed in a blocked-cyclic manner (of which blocked and completely cyclic distributions are special cases), the term contributed by the loop L_l is given by the following expression:

$$\begin{aligned} C_l(S) \quad = \quad & (\lfloor b_k/(|\alpha_1| * s_l) \rfloor * \lfloor (|\alpha_1| * r_l)/(N_k * b_k) \rfloor) + \\ & \min(\lfloor b_k/(|\alpha_1| * s_l) \rfloor, \lceil ((|\alpha_1| * r_l) \bmod (N_k * b_k))/(|\alpha_1| * s_l) \rceil) \end{aligned} \tag{3.2}$$

Figure 3.4 illustrates the situation for $N_k = 3$, and helps explain how we arrive at this expression. The region P represents the part of the computation that is divided equally among all processors. The term $\lfloor(|\alpha_1| * r_l)/(N_k * b_k)\rfloor$ denotes the number of complete blocks covered in that region on each processor. The term $\lfloor b_k/(|\alpha_1| * s_l)\rfloor$ denotes the number of loop iterations that traverse a single block of A_k on a processor. Thus, the product of these two terms (the first additive term in the expression for $C_l(S)$) represents the contribution corresponding to the region P. The remaining part of the computation, marked Q, is not divided equally on all processors. If the extent of this region exceeds the block size of A_k, the additional number of iterations executed by the most loaded processor is $\lfloor b_k/(|\alpha_1| * s_l)\rfloor$. Otherwise, that additional number is $\lceil((|\alpha_1| * r_l) \bmod (N_k * b_k))/(|\alpha_1| * s_l)\rceil$, i.e., the total number of iterations corresponding to the region Q.

- `multiple-index` : let $g_k = \alpha_1 * j_l + \alpha_2 * j_{l+1} + \ldots + \alpha_{m-l+1} * j_m + \beta$. For the purpose of analysis, the compiler "freezes" all the loops outside L_l that contribute their index to the subscript expression. Thus, the computations for S are marked sequential with respect to each of the loops outside L_l for which the coefficient of loop index in the expression for g_k is non-zero. The given case now reduces to the previous one, and the expression for $C_l(S)$ is identical to the one shown in Equation 3.2 above.

- `unknown` : the nature of partitioning of computation is unknown at compile time in this case. Therefore, it is not possible to obtain an accurate estimate for the contribution term at compile time. The compiler assumes that the load does get distributed, but with an imbalance, causing the speedup to be reduced by an imbalance-factor of `imb` (a parameter that is set to an arbitrary small value greater than one). Clearly, when communication costs are not considered, the speedup cannot be lower than one. Thus, the speedup is estimated to be $\mathtt{max}(N_k/\mathtt{imb}, 1)$, and the estimate of the term contributed by the loop L_l is:

$$C_l(S) = \lceil (n_l / \max(N_k/\mathtt{imb}, 1)) \rceil$$

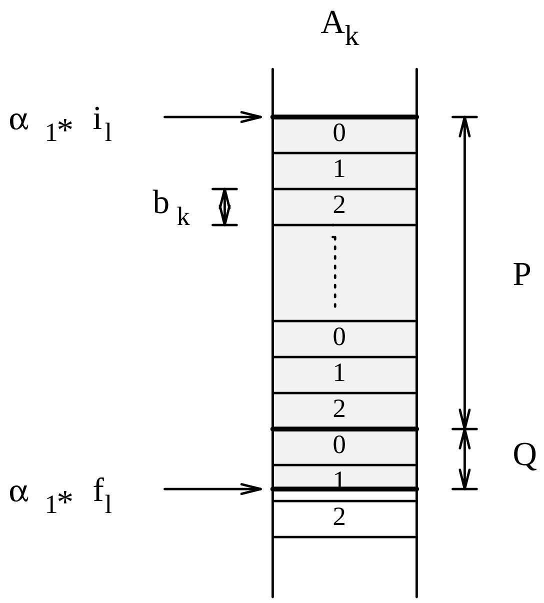

Figure 3.4: Region of A_k accessed in loop L_l

If there are two or more subscripts for which the `variation-level` is l, it implies that the computation "traverses" multiple mesh dimensions in the loop L_l. Figure 3.5 shows an example, where both the mesh dimensions are being traversed in the i-loop. Consider the terms obtained by applying the above rules individually to each subscript. Each such term represents the number of iterations in which the computation crosses a processor boundary along the mesh dimension corresponding to that subscript. In the example shown, the two terms obtained would be $n/2$ and $n/4$. Clearly, the number of iterations (for statement S) assigned to a single processor is given by the minimum of such terms (which is $n/4$ in the given example). Therefore in case of multiple subscripts varying in the same loop, the contribution term for the loop is determined by taking the minimum of all the contribution terms obtained for individual subscripts.

In actual practice, the number of processors in various mesh dimensions are not known at the time of the estimation process. Hence, the expressions for these estimates are recorded in a symbolic form, with the number of processors in different mesh dimensions appearing as parameters.

```
do i = 1, n
   A ( i, i ) = ....
```

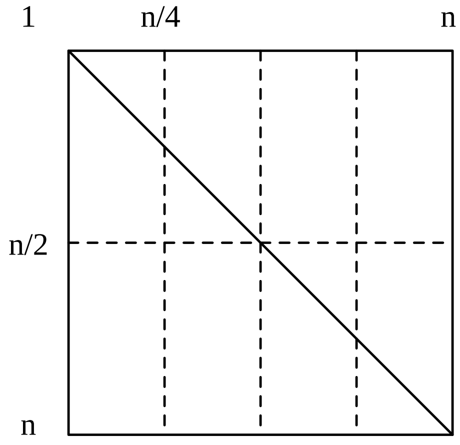

Figure 3.5: Variation of multiple subscripts in the same loop

3.3 Detection of Overlap

Even when computations for a statement S are marked sequential with respect to a loop, they may still overlap when they are distributed over more than one processor. The algorithm described in the previous section ignores that overlap. We now describe special cases when the compiler can estimate the extent of overlap, and modify the cost estimates appropriately. It may be recalled that computations for S would be marked sequential with respect to a loop L_l if there is any flow dependence to S from another statement belonging to the same SCC in the level-l dependence graph DG_l. We consider the special cases when the only flow dependence to S is from itself.

3.3.1 Case 1: OneToMany dependence

A special case of loop-carried flow dependence is one where the dependence is from a *single* instance of a statement to other instances of the same statement. Consider the example shown in Figure 3.6. If we have $1 \leq k \leq n$, there is a flow dependence from the statement instance corresponding to the kth iteration to those corresponding to later iterations. Continuing with the terminology from the previous section, let the loop have its index j_l varying from i_l to f_l in steps of s_l. When the iteration corresponding to the source of dependence can be identified

```
do i = 1, n
   A(i) = F(A(k))
enddo
```

Figure 3.6: Statement with `onetomany` dependence

at compile-time (say, $j_l = d_l$), the compiler analyzes the loop by breaking it up into three parts: the (possibly empty) initial part with loop bounds i_l and $d_l - s_l$, the single iteration d_l, and the final part with loop bounds $d_l + s_l$ and f_l. Both the initial part and the final parts can now be regarded as parallelizable loops, and the overall contribution term obtained as $C_l(S) = C_l^i(S) + 1 + C_l^f(S)$, where $C_l^i(S)$ and $C_l^f(S)$ denote contribution terms from the initial and the final parts of the loop respectively, and are obtained using Equation 3.2.

3.3.2 Case 2: Constant dependence

For a statement S, when the only flow dependence carried by loop L_l has a constant dependence distance d, there is an overlap possible between computations of S over d successive iterations of L_l. Figure 3.7 shows an example of such a statement, and illustrates the overlap possible between computations in the i-loop, for the general case of A being distributed in a blocked-cyclic manner. The expression for the overlap term (using the terminology defined in the previous section) is:

$$O_l(S) = (\min(d-1, \lfloor b_k/(|\alpha_1| * s_l)\rfloor) * (\lceil(|\alpha_1| * r_l)/b_k\rceil - 1)$$

In the above expression, $\lceil(|\alpha_1| * r_l)/b_k\rceil - 1$ represents the expected number of processor boundaries "traversed" by the computation during the loop L_l. The term $\min(d-1, \lfloor b_k/(|\alpha_1| * s_l)\rfloor)$ gives the number of iterations corresponding to the traversal of each block, that can be overlapped with the computation of elements in the preceding block.

However, on most multicomputers, the need to vectorize messages being sent in the i-loop (to offset the high start-up costs of sending messages) would make it undesirable to exploit the pipelining of computation at this level. The above analysis is now used for obtaining a balance

do $i = d + 1, n$
    $A(i) = \mathcal{F}(A(i - d))$
enddo

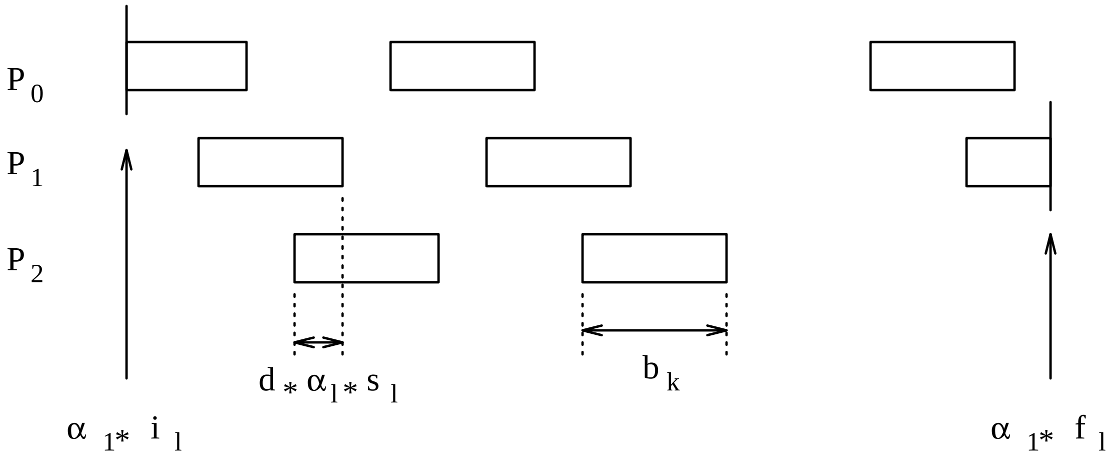

Figure 3.7: Overlap of computations in case of constant dependence

between the two goals of vectorizing messages and exploiting pipeline parallelism. That can be done by stripmining [55] the loop L_l, and the analysis we have shown helps choose the block size for stripmining in such cases. However as we shall explain in the next chapter, PARADIGM does not currently carry out this analysis for the purpose of estimating communication costs.

CHAPTER 4

ESTIMATION OF COMMUNICATION COSTS

This chapter describes the methodology for estimating the times spent on communication during program execution. These estimates are obtained in a largely machine-independent manner, and are expressed in terms of times to carry out certain high-level communication primitives on the target machine. Given information regarding the performance characteristics of those primitives on a multicomputer, the actual times spent on communication can then be estimated for that specific machine.

Our primary objective for estimating communication costs is to determine the quality measures of various communication constraints, and hence guide the selection of data partitioning scheme. However in that process, the compiler also identifies better ways of generating communication. It determines when messages may be combined to reduce the communication overheads, and the loop transformations that need to be applied to enable such optimizations. Besides, the compiler also identifies opportunities for using collective communication primitives, which is useful in many ways:

- The code generated using this analysis is much more concise, as numerous calls to the `send` and `receive` primitives get replaced by fewer calls to high-level primitives.
- The performance of programs can be improved by exploiting efficient implementations of such routines on the target machine. For instance, the broadcast routine can be implemented using $log_2(N)$ steps on an N-processor machine, whereas an equivalent piece

of code using individual send and receive primitives would take up N steps. It can be seen that the performance benefits improve with an increase in the value of N. Hence, the exploitation of collective communication is especially important for massively parallel machines.

Our main motivation for performing this analysis in the context of performance estimation has been to capture the communication costs in a concise manner. Thus, our methodology for estimation of communication costs also lays down a framework for effective generation of interprocessor communication by compilers [26].

In keeping with the objective of obtaining quality measures of constraints on distribution of specific arrays, the estimation of communication costs is done separately for each array referenced in a statement. The estimates reflect the communication requirements pertaining to that array for executing all instances of the given statement. This process consists of the following steps: (i) determining the extent to which communication can be combined, and where the communication should be placed, (ii) analyzing the data movement required in each mesh dimension to recognize the best communication primitive to use, and to determine the data sizes and the number of processors involved in that communication, and (iii) determining the order in which communication primitives operating over different mesh dimensions should be invoked. Each of these steps is described in detail, following a description of the high-level communication primitives, and some results on the characterization of array references that helps carry out the second step.

4.1 Communication Primitives

This section describes the special primitives that are recognized by the compiler as appropriate for implementing the data movement in different parts of a program. Many of the early ideas on these routines came from Geoffrey Fox and his co-workers [21], in the context of developing scientific application programs on multicomputers. Many researchers have looked at the problem of developing efficient algorithms for such primitives [32, 37], and obtaining parameterized estimates for the performance of those primitives on specific machines [7, 38]. The list of primitives is given below. All of these primitives, other than Transfer, are referred to as collective communication primitives, since they represent communication over a collection of processors.

- `Transfer` : a single source processor sends a message to a single destination processor.
- `OneToManyMulticast` : a single source processor sends a message to all other processors in the given group.
- `ManyToManyMulticast` : all processors in the group send data to all other processors in that group.
- `Scatter` : a single source processor sends *different* messages to all other processors in the group.
- `Gather` : all processors send messages to a single destination processor in the group.
- `Shift` : circular shift of data among adjacent processors in a group.
- `Reduction` : reduction of data using a simple associative and commutative operator, over all of the processors in the group.

Figure 4.1 illustrates the data movement associated with each routine. Table 4.1 shows the time complexities of functions corresponding to these primitives on the hypercube architecture. The parameter m denotes the message size in words, p represents the number of processors in the group over which the collective communication routine is carried out.

The use of high-level collective communication routines is becoming increasingly popular with application programmers on multicomputers. A number of such communication libraries have been developed, in some cases, as part of a broader research effort [56, 21, 16, 67].

4.2 Characterization of Array References

The communication requirements for an `rhs` array reference in a statement depend on the relationship between processors owning the elements corresponding to the `rhs` and the `lhs`

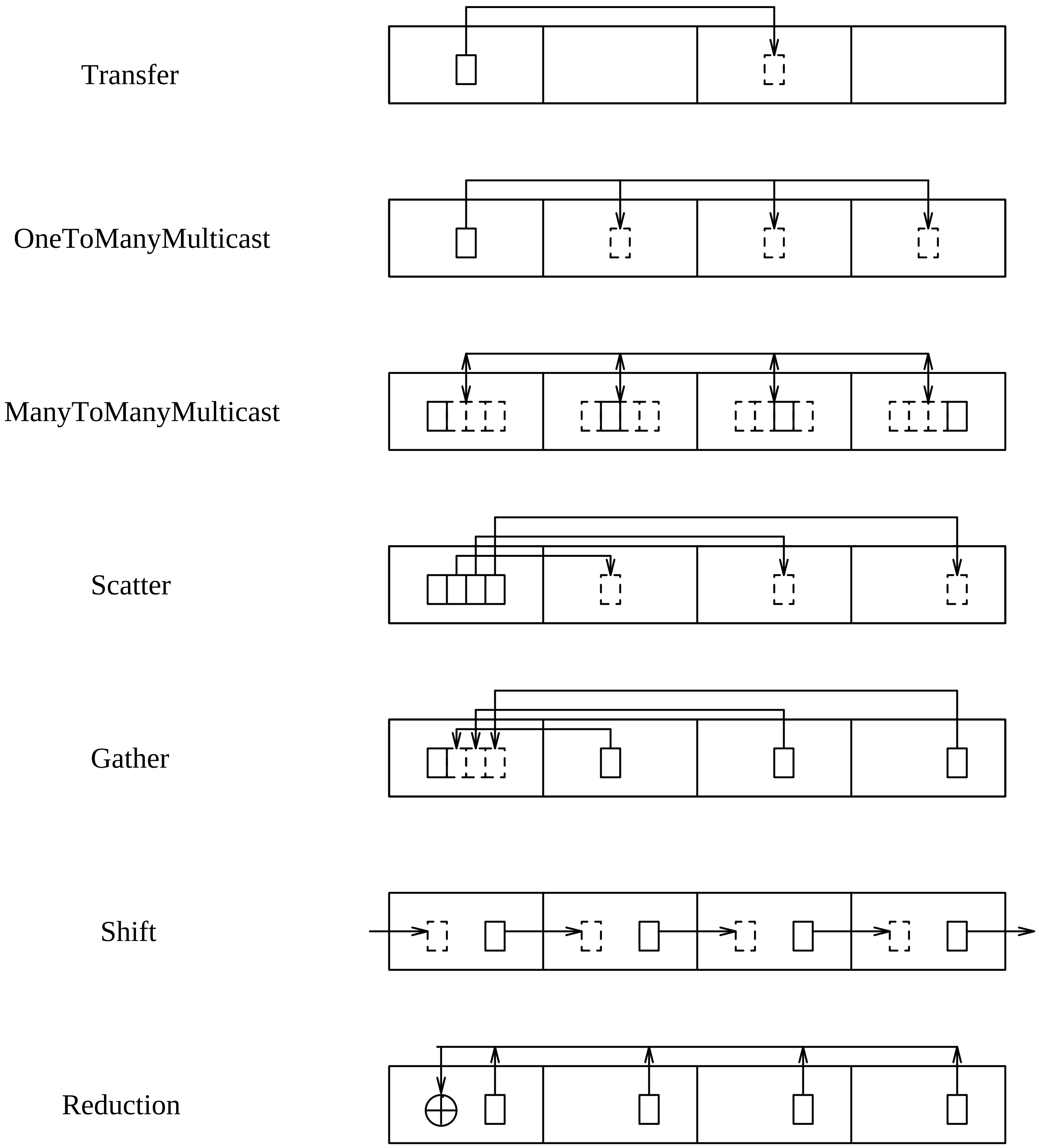

Figure 4.1: Communication primitives recognized by PARADIGM

Primitive	*Cost on Hypercube*
Transfer(m)	O(m)
OneToManyMulticast(m, p)	O($m*$ log p)
ManyToManyMulticast(m, p)	O($m*$ p)
Scatter(m, p)	O($m * p$)
Gather(m, p)	O($m * p$)
Shift(m)	O(m)
Reduction(m, p)	O($m*$ log p)

Table 4.1: Cost complexities of communication primitives on the hypercube architecture

reference for all instances of that statement. (Throughout this thesis, we shall use the term *array reference* to refer to the symbolic term appearing in the program, not the physical reference (read or write) to the element(s) represented by that term). In order to infer the pattern of communication during different iterations of a loop, the compiler has to study the variation of subscripts corresponding to distributed dimensions in those references. In this section, we define the properties regarding such variations, and present compile time tests to detect the presence of those properties. These tests enable the compiler to identify the high-level primitives that can effectively realize the data movement needed. In addition, we define quantities associated with array references that help characterize the extent of communication in terms of data sizes and the number of processors involved. The results presented in this section shall be used in the algorithms described in Section 4.4 to obtain the communication cost terms.

4.2.1 Definitions

We refer to a subscript in an array reference, corresponding to a particular distributed dimension as a *sub-reference*. A sub-reference can be represented by a tuple $\langle r, d \rangle$, where r is an array reference, and d is the position ($d \geq 1$) of the subscript in that reference. A sub-reference varying inside a loop can be seen as traversing a sequence of elements distributed on different processors along a mesh dimension. Figure 4.2 shows the traversal of two sub-references, $\langle A(i, c_1), 1 \rangle$ and $\langle B(c_2, i), 2 \rangle$, by solid lines.

do $i = 1, n$

  $A(i, c_1) = B(c_2, i)$

enddo

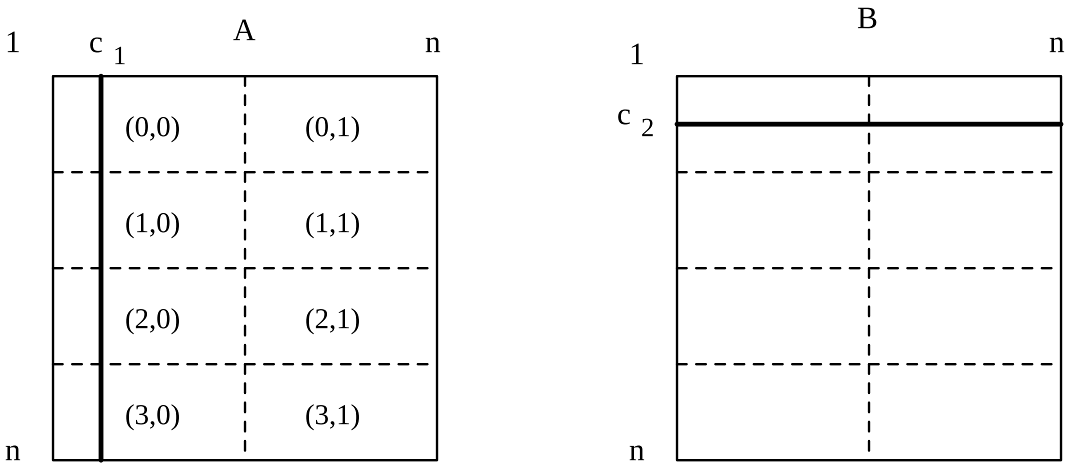

Figure 4.2: Example program segment

The sequence of processor positions in a mesh dimension traversed by a sub-reference sr in a loop L is referred to as its *external sequence* with respect to the loop, denoted as $\texttt{es}(sr, L)$. Similarly, the sequence of memory locations traversed on a processor at position p in the mesh dimension is referred to as its *internal sequence*, denoted as $\texttt{is}(sr, L, p)$. The number of processor positions traversed by the sub-reference, i.e., the length of its external sequence is referred to as $\texttt{les}(sr, L)$. Correspondingly, the number of memory locations traversed on a particular processor is referred to as $\texttt{lis}(sr, L, p)$. Another useful quantity the compiler often needs to compute is the maximum length of any internal sequence of a sub-reference with respect to a loop, denoted as $\texttt{mlis}(sr, L)$. It is defined as the maximum value of $\texttt{lis}(sr, L, p)$ for all p belonging to $\texttt{es}(sr, L)$. For the sub-reference $sr = \langle A(i, c_1), 1\rangle$ in Figure 4.2, it can be seen that $\texttt{les}(sr, L) = 4$, and $\texttt{mlis}(sr, L) = \lceil n/4 \rceil$.

Each point in the loop (identified by the value of loop index) at which the sub-reference crosses a processor boundary in that mesh dimension is called a *transition point* of the loop for that sub-reference. We now define some properties describing the relationships between sub-references varying inside the same loop, that help characterize the data movement for that loop.

Property 1 A sub-reference sr_1 is said to be *k-synchronous* (k is a positive integer) with another sub-reference sr_2, with respect to a loop L, if (i) every transition point of L for sr_2 coincides with a transition point of L for sr_1, and (ii) between every pair of consecutive transition points of L for sr_2, there are exactly $k-1$ transition points of L for sr_1.

Example: In Figure 4.2, the sub-reference $\langle A(i, c_1), 1\rangle$ is 2-synchronous with $\langle B(c_2, i), 2\rangle$, with respect to the i-loop.

Property 2 A sub-references sr_1 is said to be *strictly synchronous* with another sub-reference sr_2, with respect to a loop L, if (i) sr_1 is 1-synchronous with sr_2, with respect to L (i.e., every transition point of L for sr_1 is also a transition point of L for sr_2, and vice versa), and (ii) the coinciding transition points represent the cross-over points between the same processor numbers in the respective mesh dimensions for those sub-references.

Example: In Figure 4.2, if B_2 were distributed on four processors (rather than two), the sub-reference $\langle A(i, c_1), 1\rangle$ would be strictly synchronous with $\langle B(c_2, i), 2\rangle$, with respect to the i-loop.

A useful convention adopted regarding the k-synchronous property is that k is also allowed to be a reciprocal of positive integer. In that case, a statement that sr_1 is k-synchronous with sr_2 is really to be interpreted as conveying that sr_2 is $1/k$-synchronous with sr_1.

4.2.2 Static Determination of Characteristics

We now present some results that enable determination of the above characteristics of sub-references at compile time. The expressions obtained pertaining to the traversal of sub-references help determine the data sizes and the number of processors involved in corresponding communications. The tests on synchronous properties between pairs of sub-references help identify the communication primitive that is suitable for implementing the required data movement.

All of the results presented in this section are for sub-references of the type `single-index` (we shall often use the terms *subscript* and *sub-reference* interchangeably). The corresponding analysis for sub-references of the type `constant` is trivial, for those of the type `multiple-index` is based on these results, and for sub-references of the type `unknown` cannot be done precisely at compile-time. We shall describe how each of those cases is handled, when we describe the procedure for estimating communication costs, based on these results.

Traversal of Sub-references

Consider a sub-reference sr of the type `single-index`, with subscript expression $e = \alpha * j_l + \beta$. We shall continue with the terminology introduced in Chapter 3 (the corresponding dimension A_k is distributed on N_k processors in a blocked-cyclic manner, with a block size of b_k, and the loop L_l has a range of $r_l = \max(f_l - i_l + s_l, 0)$. The symbolic expression used for the estimated length of the external sequence of sr with respect to L_l is:

$$\texttt{les}(sr, L_l) \quad = \quad \min(N_k, \lceil (|\alpha| * r_l)/b_k \rceil) \tag{4.1}$$

The expression for $\texttt{mlis}(sr, L_l)$ has already been shown in Chapter 3, under a different guise. If sr appears in the `lhs` array reference of a statement S (and is the only one among the `lhs` sub-references varying in the loop L_l), the maximum number of instances of S (corresponding to iterations of L_l) assigned to a single processor is given by $\texttt{mlis}(sr, L_l)$. This is precisely what was determined as $C_l(S)$, the contribution term of loop L_l for the computational cost of statement S. Thus, using Equation 3.2, we have:

$$\begin{aligned} \texttt{mlis}(sr, L_l) \quad = \quad & (\lfloor b_k/(|\alpha| * s_l) \rfloor * \lfloor (|\alpha| * r_l)/(N_k * b_k) \rfloor) + \\ & \min(\lfloor b_k/(|\alpha| * s_l) \rfloor, \lceil ((|\alpha| * r_l) \bmod (N_k * b_k))/(|\alpha| * s_l) \rceil) \end{aligned} \tag{4.2}$$

Synchronous Properties

We now present tests to detect the synchronous properties for a pair of sub-references. In order to prove our results regarding the synchronous properties, we need the following two lemmas.

Lemma 4.1 *Given any integer x, and positive integers y and z,*

$$\lfloor x/(y*z)\rfloor = \lfloor \lfloor x/y \rfloor /z\rfloor$$

Proof

$$\begin{array}{llcll}
\text{Let} & x & = & q*y+r, & \text{where } 0 \le r < y,\ q \text{ and } r \text{ are integers.} \\
\text{Let} & q & = & q'*z+r', & \text{where } 0 \le r' < z,\ q', \text{ and } r' \text{ are integers.} \\
\text{We have} & \lfloor x/(y*z)\rfloor & = & \lfloor (q*y+r)/(y*z)\rfloor & \\
 & & = & \lfloor q/z + r/(y*z)\rfloor & \\
 & & = & \lfloor (q'*z+r')/z + r/(y*z)\rfloor & \\
 & & = & \lfloor q' + r'/z + r/(y*z)\rfloor & \\
 & & = & q' + \lfloor (r'*y+r)/(y*z)\rfloor & \\
\text{Now} & r'*y+r & < & (r'+1)*y & \qquad (\text{since } r < y) \\
 & r'+1 & \le & z & \qquad (\text{since } r' < z) \\
\text{Hence,} & r'*y+r & < & z*y & \\
 & \lfloor (r'*y+r)/(y*z)\rfloor & = & 0 & \\
\text{Therefore,} & \lfloor x/(y*z)\rfloor & = & q' & \\
 & & = & \lfloor q/z\rfloor & \\
 & & = & \lfloor \lfloor x/y\rfloor /z\rfloor &
\end{array}$$

□

Lemma 4.2 *Given any integer x, and positive integers y and z,*

$$\lfloor x_1/y_1\rfloor = \lfloor x_2/y_2\rfloor \quad \Rightarrow \quad \lfloor x_1/(z*y_1)\rfloor = \lfloor x_2/(z*y_2)\rfloor$$

Proof

$$
\begin{array}{rlcll}
 & \lfloor x_1/y_1 \rfloor & = & \lfloor x_2/y_2 \rfloor & \\
\Rightarrow & \lfloor \lfloor x_1/y_1 \rfloor /z \rfloor & = & \lfloor \lfloor x_2/y_2 \rfloor /z \rfloor & \\
\Rightarrow & \lfloor x_1/(z*y_1) \rfloor & = & \lfloor x_2/(z*y_2) \rfloor & \text{(by Lemma 4.1)}
\end{array}
$$

□

Strictly synchronous sub-references Consider two sub-references sr_1 and sr_2, corresponding to array dimensions that are distributed in the same manner – blocked or cyclic, and on an equal number of processors (say, N_k), in case of cyclic distribution. Let their subscript expressions be $e_1 = \alpha_1 * i + \beta_1$, and $e_2 = \alpha_2 * i + \beta_2$. Let b_1 and b_2 be the block sizes, and o_1 and o_2 be the offsets of distribution of the corresponding array dimensions. During loop iteration i, the positions in their mesh dimensions to which the two sub-references get mapped are $\lfloor(\alpha_1 * i + \beta_1 - o_1)/b_1\rfloor[\text{mod} N_k]$ and $\lfloor(\alpha_2 * i + \beta_2 - o_2)/b_2\rfloor[\text{mod} N_k]$ respectively. Clearly, sr_1 is strictly synchronous with sr_2, with respect to the i-loop L, if

$$\lfloor(\alpha_1 * i + \beta_1 - o_1)/b_1\rfloor = \lfloor(\alpha_2 * i + \beta_2 - o_2)/b_2\rfloor$$

The following theorem gives sufficient conditions under which the above equation holds, and hence, gives sufficient conditions for the strictly synchronous property.

Theorem 4.1 *Let sr_1 and sr_2 be two sub-references, as described above. The sub-reference sr_1 is strictly synchronous with sr_2, with respect to loop L if either of the following sets of conditions are satisfied:*

- *(i) $\alpha_1/b_1 = \alpha_2/b_2$, and (ii) $(\beta_1 - o_1)/b_1 = (\beta_2 - o_2)/b_2$,*
  or
- *(i) $b_1 = m * \alpha_1$, (ii) $b_2 = m * \alpha_2$, and (iii) $\lfloor(\beta_1 - o_1)/\alpha_1\rfloor = \lfloor(\beta_2 - o_2)/\alpha_2\rfloor$,*
  where m is a positive integer.

Proof The first set of conditions implies that

$$
\begin{array}{llll}
 & (\alpha_1 * i)/b_1 + (\beta_1 - o_1)/b_1 & = (\alpha_2 * i)/b_2 + (\beta_2 - o_2)/b_2, & \forall i \in I \\
\Rightarrow & \lfloor(\alpha_1 * i + \beta_1 - o_1)/b_1\rfloor & = \lfloor(\alpha_2 * i + \beta_2 - o_2)/b_2\rfloor & \text{(Q.E.D. for case 1)}
\end{array}
$$

The condition (iii) from the second set implies that

$$
\begin{array}{llll}
 & \lfloor(\beta_1 - o_1)/\alpha_1\rfloor + i & = \lfloor(\beta_2 - o_2)/\alpha_2\rfloor + i, & \forall i \in I \\
\Rightarrow & \lfloor(\beta_1 - o_1)/\alpha_1 + i\rfloor & = \lfloor(\beta_2 - o_2)/\alpha_2 + i\rfloor, & \\
\Rightarrow & \lfloor(\alpha_1 * i + \beta_1 - o_1)/\alpha_1\rfloor & = \lfloor(\alpha_2 * i + \beta_2 - o_2)/\alpha_2\rfloor & \\
\Rightarrow & \lfloor(\alpha_1 * i + \beta_1 - o_1)/(m * \alpha_1)\rfloor & = \lfloor(\alpha_2 * i + \beta_2 - o_2)/(m * \alpha_2)\rfloor & \text{(by Lemma 4.2)} \\
\Rightarrow & \lfloor(\alpha_1 * i + \beta_1 - o_1)/b_1\rfloor & = \lfloor(\alpha_2 * i + \beta_2 - o_2)/b_2\rfloor & \text{(from conditions (i) and (ii))}
\end{array}
$$

□

k-synchronous sub-references The conditions we check to see whether sr_1 is k-synchronous with sr_2 are obtained in a similar manner, and are shown below:

Theorem 4.2 *Let sr_1 and sr_2 be two sub-references as described above. The sub-reference sr_1 is k-synchronous (k being an integer) with sr_2, with respect to loop L if either of the following sets of conditions are satisfied:*

- *(i) $\alpha_1/b_1 = k * (\alpha_2/b_2)$, and (ii) $(\beta_1 - o_1)/b_1 = k * ((\beta_2 - o_2)/b_2) + l$, where l is an integer, or*
- *(i) $b_1 = m * \alpha_1$, (ii) $b_2 = k * m * \alpha_2$, and (iii) $\lfloor(\beta_1 - o_1)/\alpha_1\rfloor = \lfloor(\beta_2 - o_2)/\alpha_2\rfloor + m * l$, where m and l are integers, and $m > 0$.*

Proof Omitted, similar to the proof of Theorem 4.1 shown above.

Other Methods of Characterization

During the analysis of communication requirements, if the compiler is unable to establish a k-synchronous property between two sub-references of the type `single-index`, it often needs other measures to characterize the relationship between the two sub-references. One such measure is the *speed-ratio*, defined as:

$$\texttt{speed-ratio}(sr_1, sr_2, L) \quad = \quad \lceil (\alpha_1 * b_2)/(\alpha_2 * b_1) \rceil \tag{4.3}$$

This captures roughly, the "speed" with which the sub-reference sr_1 crosses processor boundaries in the loop L, relative to the sub-reference sr_2. This measure is a generalization of the notion of k-synchronous property, and it may be observed that a k-synchronous property between two sub-references for a given loop implies a `speed-ratio` of k.

Boundary-communication The "boundary-communication" is a specialized test performed between sub-references corresponding to aligned dimensions, and related by a `speed-ratio` of one. This test helps detect data movement taking place across boundaries of regions allocated to neighboring processors. It checks for the following conditions:

1. $\alpha_1/b_1 = \alpha_2/b_2$.
2. $|(\beta_1 - o_1)/b_1 - (\beta_2 - o_2)/b_2| \leq 1$.

If the above conditions are satisfied, the amount of data transfer across the boundary of each participating processor is estimated as:

$$\texttt{bd}(sr_1, sr_2, L) = (\lceil |(\beta_1 - o_1)/(\alpha_1 * s_l) - (\beta_2 - o_2)/(\alpha_2 * s_l)| \rceil) * (\lceil (\alpha_2 * r_l)/(N_k * b_2) \rceil) \tag{4.4}$$

where the symbols s_l, r_l, N_k have their usual meaning. The first term, $\lceil |(\beta_1 - o_1)/(\alpha_1 * s_l) - (\beta_2 - o_2)/(\alpha_2 * s_l)| \rceil$ gives the number of elements transferred across a single block of the array dimension. The second term in the product gives the number of blocks (among those traversed by the sub-reference sr_2) that are held by a single processor.

$$
\begin{array}{l}
\text{do } j_1 = i_1, f_1, s_1 \\
\quad \vdots \\
\quad \text{do } j_2 = i_2, f_2, s_2 \\
\qquad \ddots \\
\qquad\quad \text{do } j_m = i_m, f_m, s_m \\
\qquad\qquad \vdots \\
\qquad\qquad A(g_1, g_2, \ldots, g_p) = \mathcal{F}(B(h_1, h_2, \ldots, h_q)) \qquad (S)
\end{array}
$$

Figure 4.3: Statement requiring communication of data

4.3 Placement of Communication

Consider a statement S shown in Figure 4.3. Prior to the execution of any instance of that statement, the value of $B(h_1, h_2, \ldots, h_q)$ has to be sent to the processor owning $A(g_1, g_2, \ldots, g_p)$ if the two elements are owned by different processors. In the absence of any optimization, the communication required to implement this data movement for array B is placed just before that statement. However, that may lead to a serious problem on most multicomputers, where the start-up cost of sending a message is much greater than the per-element cost. Hence combining messages, particularly those being sent in different iterations of a loop, is an important optimization goal for most compilers for multicomputers [30, 22, 15]. During the `setup` pass, PARADIGM determines the outermost possible level at which communication for each array reference can legally be placed. Later, other considerations such as regularity of data movement and exploitation of pipeline parallelism further influence the actual placement of communication.

4.3.1 Moving Communication Outside Loops

The algorithm in Figure 4.4 shows how the compiler determines the outermost loop level at which communication involving array B in statement S can be placed. The existence of a flow dependence edge due to array B from S' to S with a nesting level of k implies that the values of some elements of B that may be involved in communication are computed in the loop L_k (the nesting level of a dependence is the innermost loop level with respect to which

```
lmax = 0
for each dependence into S do
   if ((dependence-type == flow) and (dependence due to array B))
      S' = source statement of dependence
      determine k = nesting level of dependence
      for (l = k; l ≥ 1; l − −) do
         if S and S' belong to same SCC in DG_l
            break
         endif
      endfor
      lmax = max(l, lmax)
   endif
endfor
Identify the doall loops outside loop L_lmax.
```

Figure 4.4: Algorithm to determine communication level for reference to B in S

the dependence is carried, or if the dependence is loop independent, the common nesting level of the two statements involved in that dependence). Further, if that edge comes from a node belonging to the same SCC in DG_l (defined earlier in Chapter 3), the communication has to be done within the loop L_l to honor the data dependence. However, if the edge comes from a node belonging to a different SCC, by distributing the loop over these components, the compiler can take the communication out of the loop.

PARADIGM infers that communication can be combined with respect to all the loops inside L_{lmax} *and* with respect to all the `doall` loops outside L_{lmax}. In order to ensure this combining, the compiler has to perform some loop transformations which can be identified by simple extensions to this algorithm. Whenever the source S' of a loop-carried dependence and S belong to different SCCs in the *level-l* dependence graph DG_l, if l takes a higher value than the eventual value of $lmax$, the loop L_l has to be distributed over the SCCs corresponding to S' and S to allow communication to be combined with respect to that loop. Even though communication has to be placed inside loop L_{lmax}, it can be combined with respect to any `doall` loop outside L_{lmax} by bringing the `doall` loop inwards. Note that an outer `doall` loop can be brought inside L_{lmax} even if there are intervening loops, and even if the loops are not part of a perfectly nested

$$
\begin{array}{lr}
\text{do } i = 1, n & \\
\quad \text{do } j = 2, n & \\
\quad\quad A(i,j) = \mathcal{F}(A(i,j-1), B(i,j-1)) & \text{(S1)} \\
\quad\quad B(i,j) = \mathcal{F}(A(i,j), A(i,j-1)) & \text{(S2)} \\
\quad \text{enddo} & \\
\quad D(i) = \mathcal{F}(A(i,n), D(i-1)) & \text{(S3)} \\
\text{enddo} &
\end{array}
$$

Figure 4.5: Original program segment

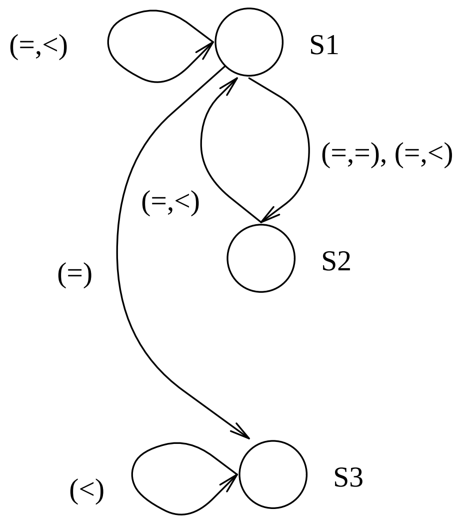

Figure 4.6: Dependence graph for the example program segment

structure. This can be achieved by applying a sequence of the following transformations, which are always valid:

- Distribution of a `doall` loop over statements constituting its body.
- Loop permutation [8] (a special case of that is loop interchange) that brings a `doall` loop inwards.

The application of these transformations is illustrated by the example shown in Figure 4.5. The dependence graph for this program segment is shown in Figure 4.6. Consider the process of determining the level for communication involving array B for statement $S1$. There is a flow dependence from statement $S2$, carried by the j-loop at level 2. Also, $S2$ belongs to the same

```
do j = 2, n
    do i = 1, n
        A(i, j) = F(A(i, j − 1), B(i, j − 1))        (S1)
    enddo
    do i = 1, n
        B(i, j) = F(A(i, j), A(i, j − 1))             (S2)
    enddo
enddo
do i = 1, n
    D(i) = F(A(i, n), D(i − 1))                       (S3)
enddo
```

Figure 4.7: Transformed program segment

SCC as $S1$ in DG_2. Hence communication has to take place inside the j-loop, and the value of $lmax$ is set to 2. The examination of loops outside the j-loop reveals that the i-loop is a `doall`. Hence the i-loop is brought inside by distribution over the j-loop and $S3$, followed by a loop interchange with the j-loop. Since the flow dependence to $S1$ is from a different statement $S2$, the i-loop is further distributed over $S1$ and $S2$. The transformed program segment is shown in Figure 4.7.

In general, the loop structure surrounding a statement with array references can finally be transformed to the form shown in Figure 4.8. All of the loops L_{l+1} through L_m are those from which communication can be taken outside (we shall refer to them as type-1 loops), while the loops L_1 through L_l are those which must have communication taking place inside them due to dependence constraints (we shall refer to such loops as type-2 loops).

The characterization of a loop as type-1 or type-2 is always with respect to a particular `rhs` reference, since it depends on the outermost level at which communication for that reference can legally be placed. While a parallelizable loop is always type-1 (or can be transformed to become type-1) with respect to all `rhs` references in the statements inside it, an inherently sequential loop may also be type-1 with respect to a given `rhs` reference. Throughout the remainder of this chapter, we shall refer to a loop simply as a type-1 or type-2 loop, where it is clear which `rhs` reference is being considered.

$$
\begin{array}{l}
\text{do } j_1 = i_1, f_1, s_1 \quad \text{(type-2)} \\
\quad \ddots \\
\qquad \text{do } j_l = i_l, f_l, s_l \quad \text{(type-2)} \\
\qquad\quad \langle \text{ communication for } (A(g_1, \ldots, g_p), B(h_1, \ldots, h_q)) \rangle \\
\qquad\quad \text{do } j_{l+1} = i_{l+1}, f_{l+1}, s_{l+1} \quad \text{(type-1)} \\
\qquad\qquad \ddots \\
\qquad\qquad\quad \text{do } j_m = i_m, f_m, s_m \quad \text{(type-1)} \\
\\
\qquad\qquad\qquad A(g_1, g_2, \ldots, g_p) = \mathcal{F}(B(h_1, h_2, \ldots, h_q))
\end{array}
$$

Figure 4.8: Statement involving communication

4.3.2 Limits on Combining of Messages

While the combining of messages is desirable for amortizing the high start-up costs of communication, it may not always be optimal to place communication at the outermost possible loop level. One reason for that is a possible reduction of overlap between computations when messages are combined. For instance, consider the program segment and the associated data movement shown in Figure 4.9, where A is distributed by rows. If the messages are combined for all iterations of the j-loop, each processor sends data to its neighboring processor only after completing its entire share of computation. A greater amount of overlap is possible between those computations if the j-loop is stripmined (corresponding to the dotted vertical lines), and the communication of values is done between execution of successive strips. Once all the data partitioning parameters are known, PARADIGM can provide the performance estimates to help choose an appropriate block size that balances the benefits of combining messages and exploiting pipeline parallelism [31]. However during the process of selection of data partitioning parameters, PARADIGM avoids the complexity of adding one more variable (block size of stripmining) to the symbolic expressions for estimated performance by giving priority to combining of messages, and ignoring the possibility of stripmining.

Another situation when it may be better not to combine communication with respect to a type-1 loop is when that data movement is not "regular enough" (or known at compile-time

```
do  i  =  2,  n
  do  j  =  1,  n
     A(i, j)  =  ...  A(i-1, j) ..
  enddo
enddo
```

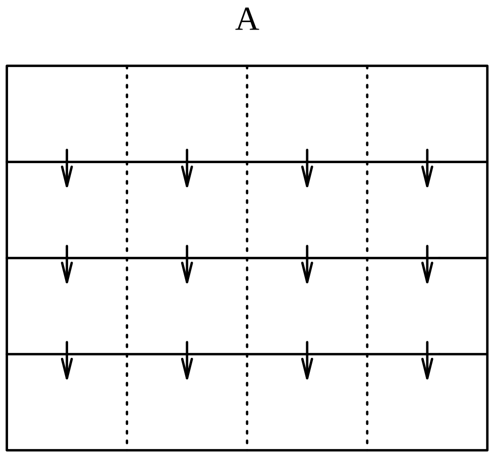

Figure 4.9: Combining of messages versus exploitation of pipeline parallelism

do $i = 1, n$
    $A(i) = \mathcal{F}(B(D(i)))$
enddo

Figure 4.10: Statement with irregular data movement

to be regular enough) in the loop. This notion of regularity will be made precise in the next section. In such cases, the use of collective communication necessarily involves communication of extra values. Therefore, it may be better to carry out communication *inside* the type-1 loop, i.e., use repeated calls to the Transfer primitive during different iterations of that loop, rather than a single call to collective communication primitive for the entire loop. For instance, consider the type-1 loop shown in Figure 4.10. The use of collective communication would involve each processor (on which B is distributed) sending the entire section of array B that it owns to all the processors on which A is distributed. This primitive is carried out only once, before entering the loop. However, it involves communication of larger amounts of data than necessary, and also requires each processor to allocate a greater amount of space to receive the incoming data. The other alternative that we mentioned is to carry out communication inside the loop. During each iteration, the owner of $B(D(i))$ is determined, and if it is different from the owner of $A(i)$, the value of $B(D(i))$ is communicated using a Transfer operation. Yet another alternative is to use the run-time compilation techniques developed by Saltz et al. [67]

and Koelbel et al. [44]. The compiler generates an *inspector* that pre-processes the loop body at run-time to determine the communication requirements for each processor.

The best method to use amongst these usually depends on the nature of the problem and the target machine characteristics. If the given loop itself appears inside another loop and the values of elements of D do not change inside that outer loop, using the inspector method is likely to be the best, since the overhead of inspector gets amortized over different iterations of the outer loop. Otherwise, if the target machine has a large set-up cost for sending messages, and has enough memory (given the data sizes used) on each node, it may be better to use collective communication. On a massively parallel machine tackling a large-sized problem, where the memory limitation on each node is more severe, the use of Transfer operations inside the loop may be the best, or the only choice.

Ideally, a compiler generating communication should choose amongst these alternatives only after evaluating these trade-offs, and taking into account the resource constraints. For the purpose of estimation of costs, Paradigm assumes that communication would be combined with respect to every type-1 loop, even if that leads to communication of some extra values.

4.4 Identification of Communication Terms

Consider again the two array references shown in Figure 4.8. The compiler analyzes each pair of sub-references corresponding to aligned array dimensions in those references, and obtains a term representing the cost of communication in that mesh dimension. Each term gives a "one-dimensional" view of the data movement in a specific dimension. For instance, in the data movement shown in Figure 4.11, the individual terms corresponding to the movement in the two mesh dimensions are Scatter($m_1, 3$), and OneToManyMulticast($m_2, 3$). These terms are composed to give an overall communication cost estimate of Scatter($m_1 * m_3, 3$) + OneToManyMulticast($m_1 * m_2, 3$). In this section, we show how terms are obtained for each pair of sub-references. The next section describes the procedure for obtaining the overall cost estimates based on these terms.

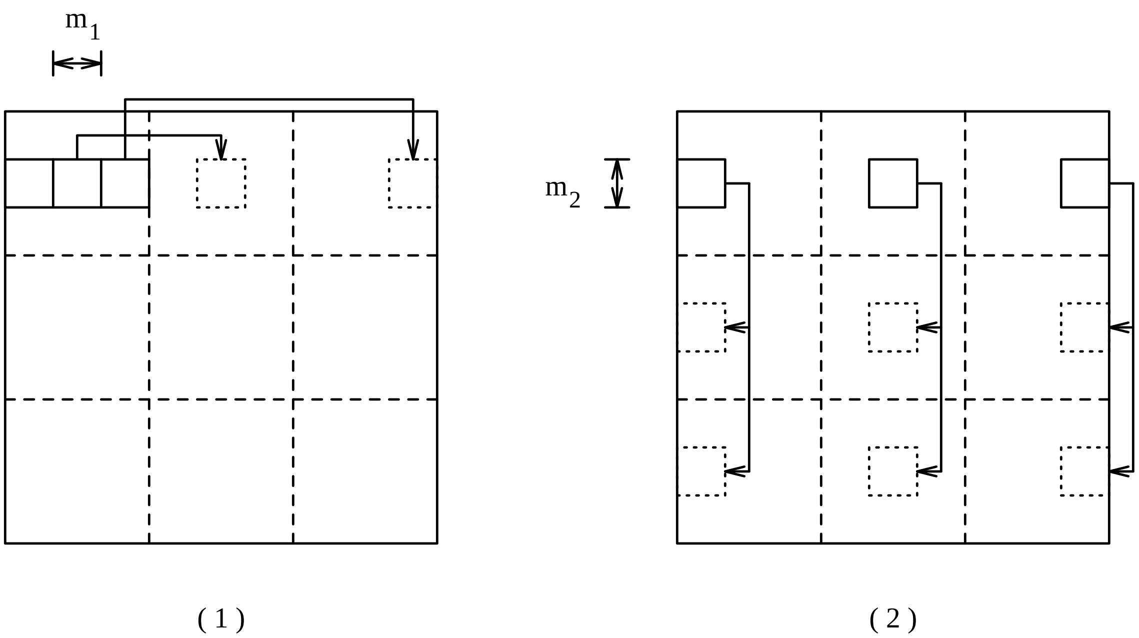

Figure 4.11: Different data movements in the two mesh dimensions

4.4.1 Overview of Algorithm

Given the `lhs` and the `rhs` reference, the compiler first matches pairs of sub-references corresponding to aligned array dimensions. If the `lhs` and the `rhs` arrays differ in the number of dimensions, the sub-references corresponding to the "extra" distributed dimensions of one array are paired up with sub-references corresponding to the "missing" dimensions of the other array. The "missing" sub-reference is regarded as being of the type `constant`. For each pair of sub-references, the larger of their values of `variation-level` identifies the innermost loop in which data movement for that mesh dimension takes place.

The compiler now steps through each of the loops surrounding the given statement, from the innermost to the outermost loop. For each loop, it identifies the matched pair(s) of sub-references, if any, for which the value of variation-level is equal to the current loop level. Depending on whether the current loop is a type-1 or a type-2 loop, the compiler invokes the appropriate procedure to obtain the term(s) representing the cost of communication in the corresponding mesh dimension(s). Initially, the compiler assumes that communication would be placed at the outermost possible level l, determined earlier, as shown in Figure 4.8. Hence, the loops $L_{l+1} \dots L_m$ are regarded as type-1 loops, and all others as type-2 loops. However, as the algorithm proceeds from inner loop onwards, the analysis for data movement in the current loop may designate an outer loop previously marked type-1 as a type-2 loop (this shall be discussed further along with the detailed description of the analysis performed). Therefore at each step, the compiler has to check whether the current loop is to be regarded as a type-2 or a type-1 loop.

If there is no pair of sub-references identified in the above step for a given loop, there is no communication term obtained for that loop. If the given loop is a type-1, the significance of this case is that any communication taking place for the `rhs` reference need not be repeated inside this loop, since the values communicated once (before executing this loop) can be reused. On the other hand, if it is a type-2 loop, any communication taking place simply gets repeated inside this loop. The iteration count of the loop only modifies the value of other communication cost terms in this case. That modification is done during the step performing the composition of all the individual terms.

Finally, the compiler analyzes the remaining pairs of sub-references that do not vary inside any loop. For each such pair, both the sub-references are of the type `constant`. If the elements corresponding to those sub-references are mapped to different positions in the mesh dimension, there is a need for Transfer primitive to make up that position difference. A symbolic expression is obtained for the communication cost in terms of N_k (the number of processors in the mesh dimension), based on the following simplifying assumption. Any two arbitrary elements along the given array dimension are assumed to belong to the same processor with a probability of $1/N_k$. Hence, the probability that a Transfer is needed is $1 - 1/N_k$, and the cost term is obtained as $(1 - 1/N_k) * \text{Transfer}(1)$. This elegantly takes care of the important boundary case $N_k = 1$, since the communication cost term does evaluate to zero when the array dimension is sequentialized.

We now describe the analysis performed by the compiler to obtain communication cost terms for the pairs of sub-references varying in a loop. For the variation of sub-references in a type-2 loop, the term represents the data movement taking place in a single loop iteration. For the variation in a type-1 loop, the term obtained represents the data movement for the entire loop.

4.4.2 Data Movement in Type-2 Loops

Since any communication in a type-2 loop, L, has to be realized with a separate message during every iteration, the primitive used is Transfer, with a data size of one. The only analysis needed is to estimate how often the elements corresponding to the two sub-references sr_1 and sr_2 are mapped to different processors. The compiler first checks for the following special cases:

1. *Both sr_1 and sr_2 are of the type single-index, and sr_1 is strictly synchronous with sr_2, with respect to L.* This implies that there is no interprocessor communication required. Hence, no communication cost term is returned in this case.

2. *Both sr_1 and sr_2 are of the type single-index, and satisfy the boundary-communication test.* In this case, the number of iterations during which a Transfer primitive has to be used is given by $\texttt{bd}(sr_1, sr_2, L) * (\texttt{les}(sr_2, L) - 1)$. The term $\texttt{les}(sr_2, L) - 1$ denotes the number of processor boundaries across which communication takes place, while $\texttt{bd}(sr_1, sr_2, L)$ denotes the number of data items that are transferred (in different iterations) across each processor boundary. The expressions for these terms have been shown earlier in Equations 4.1 and 4.4 respectively. Thus, the cost term returned is $\texttt{bd}(sr_1, sr_2, L) * (\texttt{les}(sr_2, L) - 1) * \text{Transfer}(1)$. This is the only case in which a term for communication in a type-2 loop represents the cost for all the loop iterations, hence this term is marked for special handling during the composition step.

3. *Both sr_1 and sr_2 have an identical expression.* In this case, the elements corresponding to both the sub-references are expected to be mapped to the same position in the mesh dimension. Hence, no communication cost term is returned.

In all other cases, the compiler assumes that a Transfer is needed during every iteration with a probability of $1 - 1/N_k$. Hence, the communication cost term returned is $(1 - 1/N_k) *$

Transfer(1). If there are multiple pairs of sub-references varying in the same type-2 loop, the above analysis is repeated for each pair to obtain separate cost terms.

4.4.3 Data Movement in Type-1 Loops

The data movements required in different iterations of a type-1 loop can legally be combined. Therefore, the compiler attempts to recognize the communication primitive that best realizes the collective movement, and obtains the term(s) representing its cost. The techniques we have developed allow this analysis to be carried out for arbitrary kinds of sub-references, and for any arbitrary number of pairs of sub-references varying in a loop. For ease of presentation, we shall describe different aspects of the analysis separately.

Single pair of varying sub-references

Table 4.2 lists the communication term(s) obtained if there is a single pair of sub-references varying in such a loop L. This table enumerates the cases corresponding to only the "basic" categories of the subscripts for the `lhs` and `rhs` sub-references. The results for the other cases (when the two subscripts of the type `single-index` have different values of `variation-level`, or when one of the subscripts is of the type `multiple-index`) are derived in terms of these results, and are presented later in this section. The column marked *conditions tested* lists the tests performed by the compiler to obtain further information about the nature of data movement. The entry *reduction op* represents the test to see if the `rhs` reference is involved in a reduction operation (such as addition, max, min) inside the loop.

The case corresponding to both the sub-references being of the type `single-index` is perhaps the most commonly occurring pattern inside various loops in the scientific application programs. If the two sub-references satisfy one of the five tests shown in the table, it means that the processors can be partitioned into mutually disjoint groups over which the communication primitives indicated by the cost terms may be carried out in parallel. Figure 4.12 shows the different kinds of data movement corresponding to those cases. The term IDM(m) (internalized

LHS (sr_1)	*RHS (sr_2)*	*Conditions Tested*	*Communication Term*
single-index	constant	default	OneToManyMulticast$(1, \texttt{les}(sr_1, L))$
constant	single-index	1. reduction op 2. default	Reduction$(1, \texttt{les}(sr_2, L))$ Gather$(\texttt{mlis}(sr_2, L), \texttt{les}(sr_2, L))$
single-index	single-index	1. sr_1 strictly synch sr_2 2. sr_1 1-synch sr_2 3. boundary-commn. 4. sr_1 k-synch sr_2, $k > 1$ 5. sr_1 k-synch sr_2, $k < 1$	IDM$(\texttt{mlis}(sr_2, L))$ $(N_I > 1)*$ Transfer$(\texttt{mlis}(sr_2, L))$ (i) $(N_I > 1)*$ Shift$(\texttt{bd}(sr_1, sr_2, L))$ (ii) IDM$(\texttt{mlis}(sr_2, L) - \texttt{bd}(sr_1, sr_2, L))$ D-Scatter$(\texttt{mlis}(sr_2, L)/k, k)$ D-Gather$(\texttt{mlis}(sr_2, L), 1/k)$

Table 4.2: Communication terms for a pair of sub-references varying in a type-1 loop

data movement of m elements) by itself does not represent any communication cost, it affects the data sizes of other communication cost terms during the composition step, as shall be discussed later. The symbol *D-Scatter* represents a Scatter operation over a "different" group, i.e., a group that does not include the source processor sending data values. A D-Scatter operation involves data being sent to p other processors, rather than $p - 1$ (this distinction is important for accuracy when p takes a small value). Thus, the term D-Scatter(m, p) is simply short-hand for the following two terms: (i) $(N_I > 1) *$ Transfer(m), and (ii) $(N_I > 1) *$ Scatter(m, p), where N_I is the number of processors on which the aligned array dimensions are distributed. The term D-Gather(m, p) is defined in a similar manner.

If none of the five tests indicated in the table are satisfied, the compiler computes the value of `speed-ratio`(sr_1, sr_2, L) using Equation 4.3. Similar to the last two entries for the k-synchronous property, a D-Scatter or a D-Gather term is used depending on the whether the value of `speed-ratio` is greater than or less than one. However, in this case, the Scatter or the Gather operations corresponding to different groups do not take place entirely in parallel, since there is an overlap of processors in those groups. This overlapping between groups, and the resulting contention during the process of communication is modeled by multiplying each of those terms (represented by D-Scatter or D-Gather) by a factor of two.

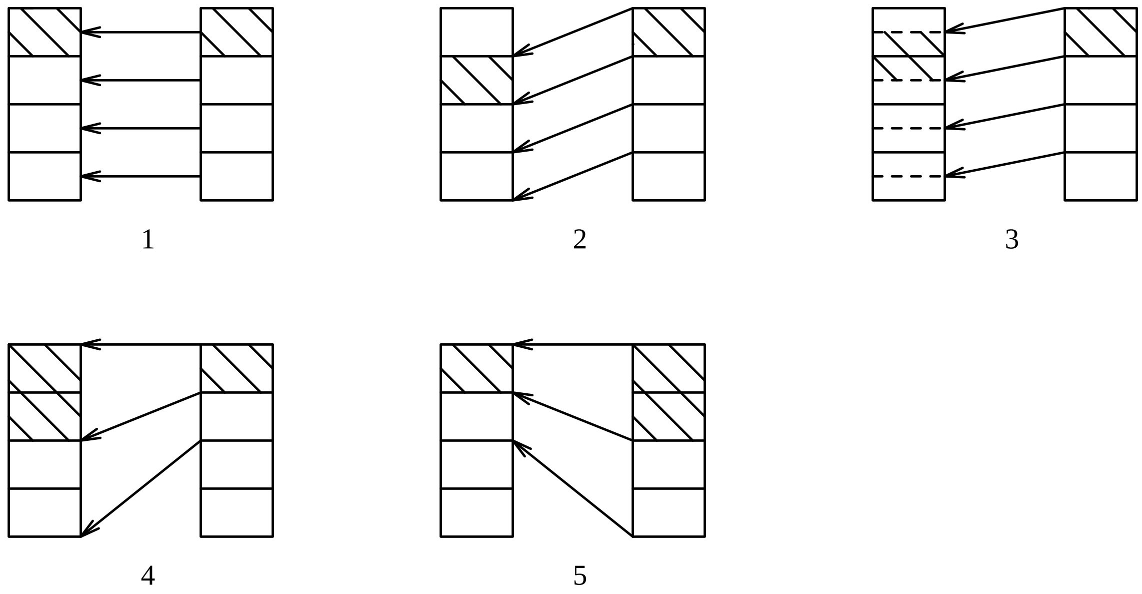

Figure 4.12: Data movements for sub-references of the type `single-index`

Multiple pairs of varying sub-references

The compiler analyzes all of the sub-references varying in a loop together, to infer the relationship between simultaneous traversals of those sub-references in different mesh dimensions. Table 4.3 presents some of the communication cost terms when there are two pairs of sub-references varying in a loop L. The unnumbered properties listed under the *conditions tested* column are those which must be satisfied, before an appropriate cost term is chosen, based on the numbered condition.

Some of the terms used in the table need to be explained. The function `fv` chooses the "faster-varying" sub-reference between two given sub-references. Stated more formally, if a sub-reference s_1 is k-synchronous with s_2 (with respect to L), $\texttt{fv}(s_1, s_2)$ is set to s_1 if $k \geq 1$, and otherwise to s_2. The symbol k_m appearing in various cost terms refers to the value of $\max(k_1, k_2)$. The symbols N_f and N_s refer to the number of processors in the two mesh dimensions corresponding to the "faster-varying" and the "slower-varying" `rhs` sub-references respectively.

LHS	*RHS*	*Conditions Tested*	*Communication Term*
single-index (s_1) single-index (s_3)	single-index (s_2) constant (s_4)	s_1 k_1-synch s_2, s_3 k_2-synch s_2, s_1 k_3-synch s_3. 1. $max(k_1, k_2) > 1$ 2. $max(k_1, k_2) = 1$ 3. $max(k_1, k_2) < 1$	 D-Scatter($\texttt{mlis}(s_2, L)/k_m, k_m$) $(N_f > 1) *$ Transfer($\texttt{mlis}(s_2, L)$) D-Gather($\texttt{mlis}(s_2, L), 1/k_m$)
single-index (s_1) single-index (s_3)	single-index (s_2) single-index (s_4)	s_1 k_1-synch s_2, s_3 k_2-synch s_4, s_1 k_3-synch s_3, s_2 k_4-synch s_4. 1. s_1 strictly synch s_2, s_3 strictly synch s_4. 2. $max(k_1, k_2) > 1$ 3. $max(k_1, k_2) = 1$ 4. $max(k_1, k_2) < 1$	 IDM($\texttt{mlis}(\texttt{fv}(s_2, s_4), L)$) D-Scatter($\texttt{mlis}(\texttt{fv}(s_2, s_4), L)/k, k$) $(N_f > 1) *$ Transfer($\texttt{mlis}(\texttt{fv}(s_2, s_4), L)$) D-Gather($\texttt{mlis}(\texttt{fv}(s_2, s_4), L), 1/k$)
single-index (s_1) constant (s_3)	single-index (s_2) single-index (s_4)	s_1 k_1-synch s_2, s_1 k_1-synch s_4, s_2 k_3-synch s_4. 1. $max(k_1, k_2) > 1$ 2. $max(k_1, k_2) = 1$ 3. $max(k_1, k_2) < 1$	 D-Scatter($\texttt{mlis}(\texttt{fv}(s_2, s_4), L)/k_m, k_m$) $(N_f > 1) *$ Transfer($\texttt{mlis}(\texttt{fv}(s_2, s_4), L)$) D-Gather($\texttt{mlis}(\texttt{fv}(s_2, s_4), L), 1/k$)
single-index (s_1) single-index (s_3)	constant (s_2) constant (s_4)	s_1 k-synch s_3	(i) $(1 - 1/N_s) *$ Transfer(1) (ii) OneToManyMulticast(1, $\texttt{les}(\texttt{fv}(s_1, s_3), L)$)
single-index (s_1) constant (s_3)	constant (s_2) single-index (s_4)	s_1 k-synch s_4. 1. $k > 1$ 2. $k = 1$ 3. $k < 1$	 D-Scatter($\texttt{mlis}(s_4, L)/k, k$) $(N_f > 1) *$ Transfer($\texttt{mlis}(s_4, L)$) D-Gather($\texttt{mlis}(s_4, L), 1/k$)
constant (s_1) constant (s_3)	single-index (s_2) single-index (s_4)	s_2 k-synch s_4 1. reduction op 2. default	(i) $(1 - 1/N_s) *$ Transfer(m) (ii) Reduction($1, \texttt{les}(\texttt{fv}(s_2, s_4), L)$) (ii) D-Gather($\texttt{mlis}(\texttt{fv}(s_2, s_4), L)$, $\texttt{les}(\texttt{fv}(s_2, s_4), L)$)

Table 4.3: Communication terms for two pairs of sub-references varying in a type-1 loop

Intuitive Explanation Most of the entries in the table correspond to the faster-varying sub-reference between the `lhs` sub-references pitted against the faster-varying sub-reference between the `rhs` ones, and selecting a D-Scatter, D-Gather, or Transfer term depending on whether that `lhs` sub-reference varies at a faster, slower, or the same rate as the `rhs` sub-reference. (These results can thus be extended to handle any arbitrary number of pairs of sub-references varying in a loop). The only entries that are different correspond to the cases where all of the `lhs` or the `rhs` sub-references are of the type `constant`. In those two cases, the appropriate choices of communication primitive are Reduction/Gather, and OneToManyMulticast respectively. In the first set of cases, the unnumbered conditions ensure that all the processors in the two mesh dimensions can be partitioned into mutually disjoint groups over which the indicated primitives can be carried out in parallel. In fact, the conditions tested in all of those cases can be seen as equivalent, if one observes that a sub-reference of the type `constant` is always (trivially) k-synchronous with another sub-reference of the type `single-index`.

If the given sub-references do not satisfy the conditions shown in the table, the compiler first selects the `lhs` sub-reference with the higher `speed-ratio`, and similarly, the faster varying `rhs` sub-reference. Now, the `speed-ratio` is determined for those two sub-references, and the communication cost term obtained, as described earlier for the case of a single pair of sub-references varying in the loop.

Example Consider the example shown in Figure 4.13, where both the arrays A and B are distributed in a blocked manner on a mesh with N_1 x N_2 processors. Depending on the relative values of N_1 and N_2, the best choice of communication primitive may be Scatter, Transfer, or Gather. It is not possible to determine the appropriate communication cost term just by analyzing the two pairs of sub-references individually. For the cases $N_1 = 4, N_2 = 2$, and $N_1 = 2, N_2 = 4$, Figures 4.14 and 4.15 show the use of parallel Scatter and Gather primitives respectively, over groups of 2 processors each. As shown in Table 4.3, the choice of communication cost term is governed by the test for k-synchronous property between sub-references $\langle A(i, c_1), 1\rangle$, and $\langle B(c_2, i), 2\rangle$. In those two cases, $\langle A(i, c_1), 1\rangle$ is determined to be 2-synchronous and 1/2-synchronous, respectively, with $\langle B(c_2, i), 2\rangle$, with respect to the i-loop. Hence, the

do $i = 1, n$
$$A(i, c_1) = \mathcal{F}(B(c_2, i))$$
enddo

Figure 4.13: Variation of multiple pairs of sub-references in a loop

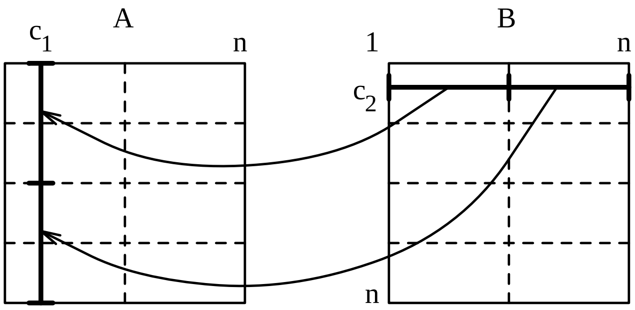

Figure 4.14: Choice of D-Scatter term for the given example

compiler chooses the terms D-Scatter($\lceil n/4 \rceil, 2$) and D-Gather($\lceil n/4 \rceil, 2$) respectively in those cases.

Cyclic distributions

All of the analysis we have described above is valid for both kinds of array distributions, blocked and cyclic. For cyclic distributions, however, another condition is added to the tests for regularity of data movement. For every sub-reference with a subscript of the form $e = \alpha_1 * j_l + \beta_1$, the compiler checks if b_1 is a multiple of $\alpha_1 * s_l$, where b_1 is the block size of distribution of the

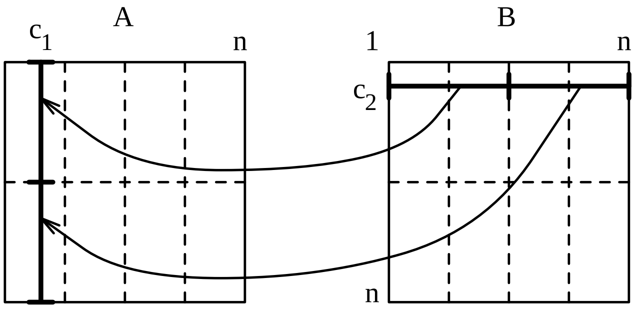

Figure 4.15: Choice of D-Gather term for the given example

given dimension, and s_l is the stride of the loop L_l. The satisfaction of this condition ensures that the data elements involved in any collective communication corresponding to the given sub-reference can be accessed on the local memories of involved processors with a constant stride. Otherwise, the compiler uses cost terms corresponding to communication with a degraded performance.

Different/multiple loop indices

When the subscripts corresponding to a matched pair of sub-references involve different loop indices, or when one of the subscripts is of the type `multiple-index`, one simple way to analyze the data movement in terms of our earlier results would be to "freeze" (i.e., regard as type-2) all relevant loops except for the innermost one, for the purpose of analysis. This would involve *not* combining communication with respect to the frozen loops, and treating the corresponding loop indices as constants. PARADIGM uses an extension of this idea, with *tiling* [80] instead of freezing of one of the outer type-1 loops, so that communication may be combined with respect to at least the tiles of that loop.

Sub-references with different loop indices Let us first describe the analysis for a pair of sub-references of the type `single-index`, but with different values of `variation-level`. Consider sub-references sr_1 and sr_2 with subscript expressions of the form $e_1 = \alpha_1 * j_l + \beta_1$, $e_2 = \alpha_2 * j_m + \beta_2$, where $l > m$. If L_m is a type-2 loop, then sr_2 is regarded as a sub-reference of the type `constant`, and the data movement is analyzed for loop L_l in a normal manner. However, if L_m is a type-1 loop, it is assumed to be tiled suitably by the compiler. (The compiler generating communication would tile it such that the starting points of the tiles are precisely the transition points of the loop L_m for the sub-reference sr_2). In the context of this thesis, we shall limit our discussion to just the part dealing with cost estimation.

The value of j_m is regarded as a constant for the purpose of analysis of communication requirements, as shown in Table 4.2. However, the cost term obtained is modified in the following manner to model the effect of tiling of the loop L_m. There are two cases:

1. *The sub-reference sr_2 appears on the* `rhs`: The data size of the cost term is multiplied by $\texttt{mlis}(sr_2, L_m)$ to model the effect of combining of communication with respect to the tiles of the loop L_m. Further, the term itself is multiplied by $\texttt{les}(sr_2, L_m)$ to account for the repetitions corresponding to the number of tiles. In fact, in this case, it is known that the cost term obtained initially is OneToManyMulticast$(1, \texttt{les}(sr_1, L_l))$. It is changed to $\texttt{les}(sr_2, L_m) * \text{OneToManyMulticast}(\texttt{mlis}(sr_2, L_m), \texttt{les}(sr_1, L_l))$.

2. *The sub-reference sr_2 appears on the* `lhs`: The cost term obtained is multiplied by $\texttt{les}(sr_2, L_m)$ in this case too. However, the data size of the original term is left unchanged (the data corresponding to the `rhs` sub-reference is re-used in different iterations that are part of the same tile).

Some additional analysis is required when there other pairs of sub-references that vary inside the loop L_m. Consider another pair of sub-references sr_3 and sr_4, both of the type `single-index` with a `variation-level` of m. This time, the analysis proceeds along the lines of that shown in Table 4.3. The compiler checks for the k-synchronous property between sr_2 and sr_3, and between sr_2 and sr_4, with respect to L_m. Tiling is used only if the k-synchronous property holds between each of the above pairs. The fastest-varying sub-reference among sr_2, sr_3, and sr_4 is selected to determine the tile size. The communication cost term is now modified using the method shown above. The following example illustrates this process.

Example Consider the statement and the associated data movement for a 4 x 2 mesh shown in Figure 4.16. The first pair of aligned dimensions have subscripts i and j varying in different loops. The j-loop is tiled, and j is regarded as a constant for the purpose of obtaining the initial term, OneToManyMulticast(1,4). There is another pair of sub-references varying in the j-loop (L_1). The sub-reference $\langle B(j,j), 1\rangle$ is 2-synchronous with both $\langle B(j,j), 2\rangle$ and $\langle A(i,j), 2\rangle$, with respect to the j-loop. Hence, the tile size is set to $\texttt{mlis}(\langle B(j,j), 1\rangle, L_1) = \lceil n/4 \rceil$. The number of tiles is given by $\texttt{les}(\langle B(j,j), 1\rangle, L_1) = 4$. Therefore, the communication term is changed to $4 * \text{OneToManyMulticast}(\lceil n/4 \rceil, 4)$.

do $j = 1, n$
    do $i = 1, n$
        $A(i, j) = B(j, j)$
    enddo
enddo

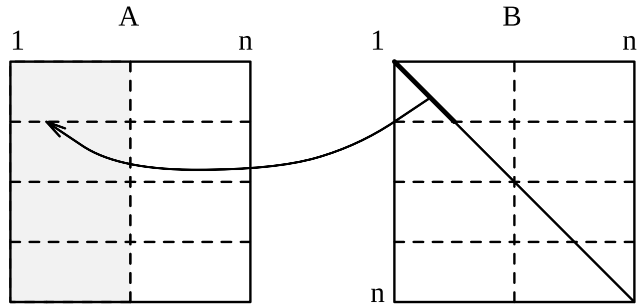

Figure 4.16: Example to illustrate tiling of outer loop

Sub-references with multiple loop indices For all the loops whose indices appear in a sub-reference of the type `multiple-index`, the compiler first determines those that are type-1 loops. If there are more than two such loops, the sub-reference is handled the same way as one of the type `unknown`. If there is only one such loop, the indices corresponding to the type-2 loops are regarded as constants, and the sub-reference reduces to one of the type `single-index`. Thus, in the following discussion, a sub-reference of the type `multiple-index` has exactly two indices corresponding to type-1 loops in the subscript expression.

Given such a sub-reference sr_1 in a pair, if the other sub-reference sr_2 is of the type `constant` or `unknown`, then the cost term is obtained by regarding sr_1 as of the type `unknown` (the analysis for dealing with a sub-reference of the type `unknown` is discussed next). Essentially, special analysis for sr_1 is applied only when sr_2 is of the type `single-index` or `multiple-index` as well. The handling of these cases is again based on the idea of tiling.

Let the current loop level being examined be l. Consider first the case of sr_1 with subscript expression $e_1 = \alpha_1 * j_l + \alpha_1' * j_m + \beta_1$ ($l > m$), and sr_2 with $e_2 = \alpha_2 * j_l + \beta_2$. Similar to the earlier case of sub-references with different loop indices, j_m is regarded as a constant for the purpose of obtaining an initial cost term, and the term obtained is modified to model the effect to tiling

LHS	*RHS*	*Communication Term*
unknown	constant	OneToManyMulticast($1, N_I$)
constant	unknown	Gather($\lceil n/N_I \rceil, N_I$)
unknown	single-index multiple-index unknown	ManyToManyMulticast($\lceil n/N_I \rceil, N_I$)
single-index multiple-index	unknown	ManyToManyMulticast($\lceil n/N_I \rceil, N_I$)

Table 4.4: Collective communication for sub-references involving unknowns

of loop L_m. Consider now the case of both sr_1 and sr_2 being of the type `multiple-index`, with subscript expressions $e_1 = \alpha_1 * j_l + \alpha'_1 * j_m + \beta_1$, and $e_2 = \alpha_2 * j_l + \alpha'_2 * j_m + \beta_2$ (if the two sub-references together involve more than two indices corresponding to type-1 loops, they are again handled like sub-references of the type `unknown`). The compiler obtains the value of `speed-ratio`(sr_1, sr_2, L_m), and determines the tile size of L_m based on the faster varying sub-reference. Again, communication cost term is first obtained assuming j_m is a constant, and then modified to account for tiling of L_m.

Sub-references of the type `unknown`

If either of the pair of sub-references varying in a type-1 loop is of the type `unknown`, the compiler in unable to infer the precise pattern of data movement in that loop. As discussed in Section 4.3, there is a trade-off in this case between using collective communication with larger data size and fewer messages, or using repeated Transfers with potentially more messages, but smaller amounts of data being communicated. For the purpose of estimating communication costs, PARADIGM currently assumes that the compiler would use collective communication. Table 4.4 shows the cost term obtained for the loop when at least one of the sub-references is of the type `unknown`. In the entries for cost terms, n denotes the size of the corresponding array dimension, and N_I denotes the number of processors in the mesh dimension. In this case, there is no special analysis needed for multiple pairs of sub-references varying in a loop. Those pairs are analyzed independently, and the resulting terms are composed in the normal manner.

Term1	*Term2*	*Resultant Term*
IDM(m_1)	IDM(m_2)	IDM($m_1 * m_2$)
Gather(m_1, p_1)	Gather(m_2, p_2)	Gather($m_1 * m_2, p_1 * p_2$)
IDM(m_1), ($N_1 > 1$) Shift(c_1)	IDM(m_2), ($N_2 > 1$) Shift(c_2)	IDM($m_1 * m_2$), ($N_1 > 1$) Shift($c_1 * m_2$), ($N_2 > 1$) Shift($c_2 * m_1$), ($N_1 > 1$ & $N_2 > 1$) Shift($c_1 * c_2$)
Reduction(m_1, p_1)	Reduction(m_2, p_2)	Reduction($m_1 * m_2, p_1 * p_2$)
ManyToManyM'cast(m_1, p_1)	ManyToManyM'cast(m_2, p_2)	ManyToManyM'cast($m_1 * m_2, p_1 * p_2$)
OneToManyM'cast(m_1, p_1)	OneToManyM'cast(m_2, p_2)	OneToManyM'cast($m_1 * m_2, p_1 * p_2$)
$(1 - 1/p_1) *$ Transfer(m_1)	$(1 - 1/p_2) *$ Transfer(m_2)	$(1 - 1/(p_1 * p_2)) *$ Transfer($m_1 * m_2$)

Table 4.5: Combining of terms with identical communication primitives

4.5 Composition of Communication Terms

Once the communication terms corresponding to all the mesh dimensions have been obtained, they are composed together to obtain the overall estimate of communication cost. This process consists of the following steps: (i) combining terms with the same communication primitive, (ii) determining the order in which different primitives are invoked, and modifying the data sizes of the terms suitably, and (iii) multiplying the terms by the expected number of repetitions of that communication during the program.

4.5.1 Communication Terms with Same Primitive

The terms involving the same primitive in different mesh dimensions are combined as shown in Table 4.5. The combined term represents the primitive carried out over a bigger group, spanning multiple mesh dimensions.

4.5.2 Ordering of Primitives

Since the primitives corresponding to different terms implement the data movement in distinct mesh dimensions, they can legally be composed in any order. However, the order in which they are invoked is important because the position of each primitive affects the message sizes

$$
\begin{array}{l}
\text{do } i = 1, n \\
\qquad A(n,n) = \mathcal{F}(B(i,1)) \\
\text{enddo}
\end{array}
$$

Figure 4.17: Statement requiring Gather and Transfer in different dimensions

and the number of processors involved (in parallel) in subsequent primitives. It is desirable to obtain an ordering that leads to fewer processors being involved and smaller messages handled by each processor, but sometimes, there is a trade-off between the two.

For example, consider the statement shown in Figure 4.17, where the arrays A and B are distributed in an identical manner on a 2-D mesh. The primitives required are: Gather in the first dimension, and Transfer in the second dimension. Figure 4.18 illustrates the two possible orderings for a 3 x 3 mesh. If Gather is invoked first, it is carried out with a data size of $n/3$ words, over 3 processors, followed by a single Transfer of n words of data. If this ordering is reversed, there are 3 parallel Transfers that take place, each involving $n/3$ words, followed by a Gather operation, also with a data size of $n/3$ words, over 3 processors.

The second ordering in the above example leads to the use of parallelism in implementing communication, and would yield better performance if there were no other communications being carried out on the mesh of processors. This suggests resolving the trade-off in favor of reducing the message sizes handled by processors. When there is no trade-off involved, the compiler should use an ordering that reduces the message sizes and/or the number of processors involved. These considerations suggest the ordering shown in Table 4.6.

Determination of message sizes

The data size associated with each communication term initially represents just one "edge" of the overall volume of data being communicated. In accordance with the ordering of primitives shown above, the compiler determines the actual message size for each primitive corresponding to a communication term.

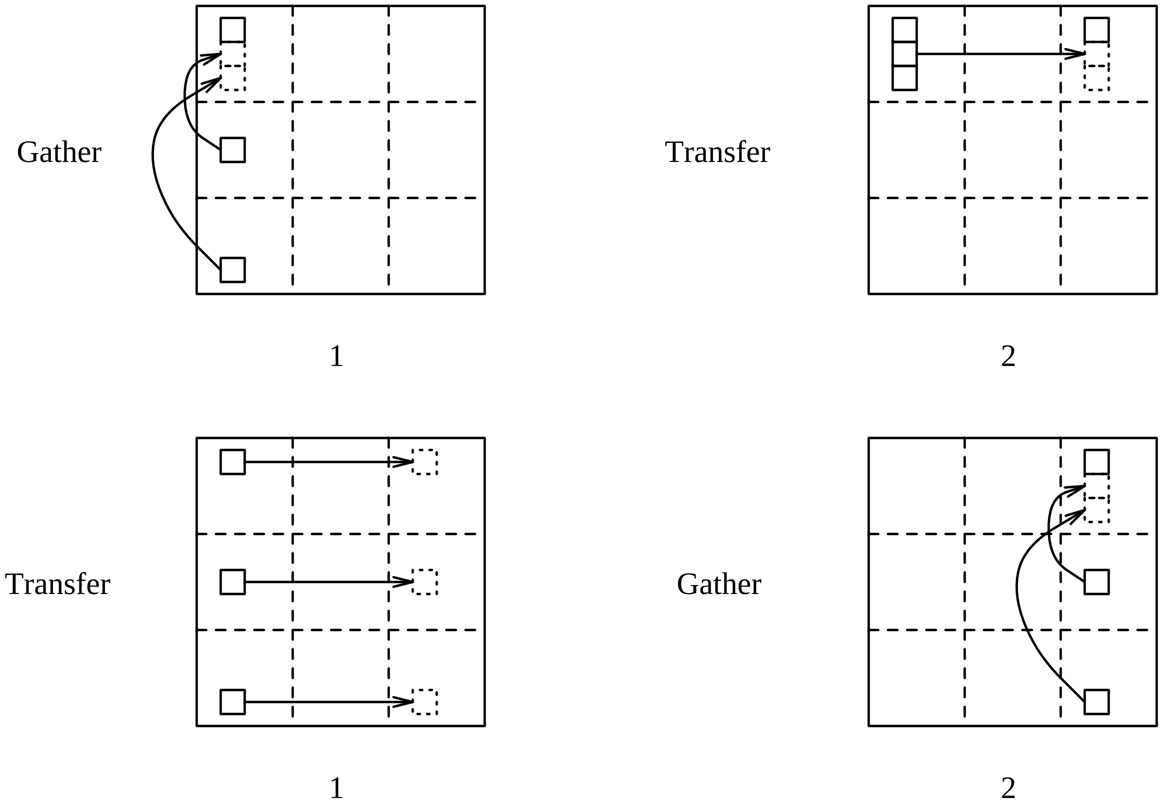

Figure 4.18: Possible compositions of Gather and Transfer

Rank	*Primitive*	*Message Size*	*No. of Processors*
1	Reduction	reduced	reduced
2	Scatter	reduced	increased
3	Shift, Transfer	preserved	preserved
4	OneToManyMulticast	preserved	increased
5	Gather	increased	reduced
6	ManyToManyMulticast	increased	increased

Table 4.6: Ordering of communication primitives for composition

do $j = 1, n$
    do $i = 1, n$
        $A(i, j, c_1) = \mathcal{F}(B(i, c_2, c_3))$
    enddo
enddo

Figure 4.19: Example of composition of communication terms

First, the compiler determines the product of individual data sizes associated with each communication term, including an IDM term. This product represents the complete volume of data to be communicated, and serves as the message size for each term corresponding to any of the first five primitives listed in Table 4.6. (Note that the data sizes used in terms for the Reduction and Scatter primitives already reflect the "reduction" in message size caused by those primitives, as can be seen from the entries shown in Tables 4.2 and 4.3). In case there are terms present for both Gather and ManyToManyMulticast primitives, the data size for the ManyToManyMulticast is multiplied further by the number of processors shown in the Gather term, to account for the increased amount of data participating in that primitive following the Gather operation.

Example Consider the statement shown in Figure 4.19, where the arrays, A and B are distributed in an identical manner on a N_1 x N_2 x N_3 mesh. The communication terms obtained are: IDM($\lceil n/N_1 \rceil$), OneToManyMulticast($1, N_2$), and $(1 - 1/N_3) *$ Transfer(1). These terms are composed together to give the communication cost estimate as $(1 - 1/N_3) *$ Transfer($\lceil n/N_1 \rceil$) + OneToManyMulticast($\lceil n/N_1 \rceil, N_2$).

4.5.3 Number of Repetitions

The sequence of communication primitives implementing the data movement for the given `rhs` array reference is placed at a certain level l, as shown in Figure 4.8. This communication is repeated during various iterations of the surrounding type-2 loops, L_1 through L_l. The effect of various conditionals in the program that influence the flow of control to the given statement is modeled by the reaching probability $\rho(S)$. Hence, the compiler multiplies each communication cost term by $\rho(S) * (\prod_{k=1}^{l} n_k)$, where n_k represents the iteration count of the loop L_k. As we had noted earlier, a boundary-Transfer term (corresponding to boundary-communication in a type-2 loop) is treated in a slightly different manner. Such a term already represents the communication cost for all iterations of the given type-2 loop. Hence, the product shown above with which that cost term is multiplied, is modified to exclude the iteration count of the corresponding type-2 loop.

The above analysis assumes strict sequentiality in the process of carrying out communication inside the type-2 loops. In practice, the generation of messages may sometimes get pipelined over different iterations of some type-2 loop(s). For an important case, we now present an algorithm to detect if communication would be overlapped during different iterations of a surrounding type-2 loop L_{k-1}, and obtain better cost estimates if there is an overlap. This special case corresponds to boundary communication taking place during a type-2 loop L_k. We believe that any generalizations of this analysis would be helpful in refining the process of estimation of communication costs.

Detection of Pipelining

Consider a pair of sub-references sr_1 and sr_2 with subscript expressions $e_1 = \alpha_1 * j_k + \beta_1$, and $e_2 = \alpha_2 * j_k + \beta_2$, such that the corresponding communication term is a boundary-Transfer. Let the `rhs` reference be to an array A, and the corresponding dimension $A_{k'}$ be distributed in a blocked manner with a block size of b_2. If sr_2 is the only sub-reference in the `rhs` array reference varying in loop L_k, the following algorithm determines whether communications are pipelined during different iterations of L_{k-1}.

1. Examine all dependence edges due to A coming into the given statement from statements inside L_{k-1}. Let $(d_1, d_2, \ldots, d_m), (m \geq k)$ denote the direction vector associated with such an edge. If for every edge, either d_k = "=", or all dependences corresponding to that edge have a constant distance vector, conclude that pipelining takes place.

2. If pipelining is detected, for all backward flow dependences (flow dependences in the ">" direction) at level k examined in Step 1, determine the maximum dependence distance at level k, and call it $bdist_k$. If there is no backward flow dependence at level k, set the value of $bdist_k$ to 0.

In case of pipelining, the precedence constraints indicate that communications for successive iterations of the loop L_{k-1} may be started after waiting for those of $bdist_k + 1$ iterations of L_k (corresponding to the previous iteration of L_{k-1}) to finish. The only resource constraint assumed is that a processor can participate in no more than one communication primitive at a time.

The various communication cost terms for the given reference are modified as follows. For the sake of illustration, let us ignore all loops outside L_{k-1}, and let all the loops inside L_k be type-1 loops. Consider a cost term T_1 corresponding to data movement in a type-1 loop. Without considering the overlap, the contribution of this term to the overall cost estimate (ignoring the conditionals and the loops outside L_{k-1}) would be $n_{k-1} * n_k * T_1$. The term corresponding to the Transfer taking place in L_k is of the form $\delta * (\eta - 1) * T_2$, where $\delta = \mathtt{bd}(sr_1, sr_2, L)$, $\eta = \mathtt{les}(sr_2, L)$, and $T_2 = \text{Transfer}(m)$. The estimate obtained for this term (without considering the overlap) would be $n_{k-1} * \delta * (\eta - 1) * T_2$. With the detection of pipelining, these estimates are modified as follows:

1. The cost estimate of the terms corresponding to data movement in type-1 loops is changed from $n_{k-1} * n_k * T_1$ to $(n_k + max(\lfloor b_2/(|\alpha_2| * s_k)\rfloor, bdist_k + 1) * (n_{k-1} - 1)) * T_1$.

   Explanation The communications associated with the inner loop L_k for the first iteration of L_{k-1} see a full latency of $n_k * T_1$. Due to precedence constraints, each successive iteration of L_{k-1} contributes to a further latency of at least $(bdist_k + 1) * T_1$. Consider now the resource constraints. Each processor along the mesh dimension traversed by L_k participates in $\lfloor b_2/(|\alpha_2| * s_k)\rfloor$ of those communications in a single iteration of L_{k-1} ($\lfloor b_2/(|\alpha_2| * s_k)\rfloor$ represents the number of iterations of L_k that cover a distributed block of $A_{k'}$). The resource constraints dictate that each processor has to complete its participation in all those communications associated with an iteration of L_{k-1} before starting one for

```
do j = 2, n1
    do i = 2, n2
        A(i) = F(A(i − 1))              (S1)
        C(i, j) = F(B(i, j), A(i))      (S2)
    enddo
enddo
```

Figure 4.20: Example to illustrate pipelining

the next iteration of L_{k-1}. Thus, each of the remaining $n_{k-1} - 1$ iterations of that loop contributes to a latency of $\max(bdist_k + 1, \lfloor b_2/(|\alpha_2| * s_k)\rfloor) * T_1$.

2. The cost estimate of the term corresponding to Transfer in the loop L_k is changed from $n_{k-1} * \delta * (\eta - 1) * T_2$ to $\delta * (\eta - 1) * T_2 + max(\delta, \lfloor(\delta * (bdist_k + 1) * |\alpha_2| * s_k)/b_2\rfloor) * (n_{k-1} - 1) * T_2$.
Explanation For the first iteration of L_{k-1}, the full latency corresponding to all $\delta * (\eta - 1)$ Transfers taking place during loop L_k is seen. For the remaining $n_{k-1} - 1$ iterations of L_{k-1}, the resource constraints dictate that each processor has to complete all of its δ Transfers associated with one iteration of L_{k-1} before starting those for the next iteration. Also, precedence constraints require that the processor has to wait for the Transfers taking place during the $bdist_k + 1$ iterations of L_k (corresponding to the previous iteration of L_{k-1}) to finish. These $bdist_k + 1$ iterations of L_k involve approximately $\lfloor(\delta * (bdist_k + 1) * |\alpha_2| * s_k)/b_2\rfloor$ Transfers.

Consider the statement S1 shown in Figure 4.20. Let N_1 denote the number of processors over which the array A is distributed. The communication term associated with sub-reference $\langle A(i-1), 1\rangle$ is $(N_1 - 1) *$ Transfer(1) (using the terminology described above, $\alpha_2 = 1$, $s_k = 1$, $\delta = 1$, $\eta = N_1$, $b_2 = \lceil n/N_1\rceil$). Without taking pipelining into account, the overall communication cost estimate for the reference $A(i-1)$ would be $n_1 * (N_1 - 1) *$ Transfer(1).

Let us consider the application of the algorithm described above to this case. All of the dependence edges involving array A do satisfy the conditions of Step 1, and the algorithm determines that pipelining takes place. There is no backward flow dependence at level 2, hence

$bdist_2$ is set to 0. The modified procedure now estimates the communication costs for the given reference to be $(n_1 - 1 + N_1 - 1) * \mathrm{Transfer}(1)$, rather than a conservative $n_1 * (N_1 - 1) * \mathrm{Transfer}(1)$.

CHAPTER 5

DATA DISTRIBUTION PASSES

One of the basic features of our approach to data partitioning is the decomposition of this problem into a number of sub-problems, each dealing with a different kind of distribution parameter for all the arrays. These sub-problems are solved in different passes in PARADIGM. An overview of the system was given in Chapter 2. In this chapter, we present details on each of those passes.

As shown in Figure 5.1, each data distribution pass is logically composed of three modules, the `detector`, the `driver`, and the `solver`. The `detector` is a complete pass through the program in which the compiler detects the need for imposing a specific kind of constraint on the distribution of arrays. The compiler looks for conditions in various computations that clearly suggest that better performance will be obtained if the distributions of the referenced arrays satisfy a certain property.

Given that a constraint is to be recorded, the next task is to obtain the quality measure of that constraint. Depending on whether the constraint affects only the communication costs or computational costs, or both, the `driver` invokes the appropriate cost estimator(s) for this purpose. This seemingly simple task is complicated by the fact that performance estimation requires complete information about the data partitioning scheme, which is not available at this stage. Therefore, the `driver` module has to specify some appropriate, default values for data distribution parameters that are unknown when the given pass is being run. The extent of data partitioning information available to the compiler increases during successive passes.

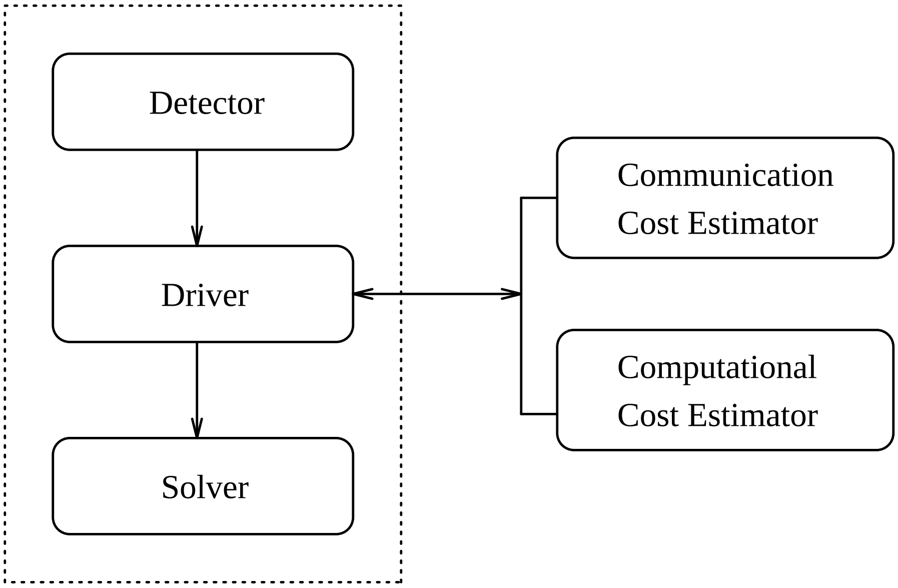

Figure 5.1: Structure of a data distribution pass

Hence, the use of separate passes allows the data distribution decisions taken during the later passes to be guided by more accurate quality measures.

Once all the constraints relevant to a given distribution parameter, and their quality measures have been recorded for the entire program, the `solver` determines the value of that parameter for all the arrays. Essentially, the `solver` obtains (an approximate) solution to an optimization problem, where the objective is to minimize the execution time of the target data-parallel program. The optimization process relies on the resolution of any conflicts between constraints on the basis of their quality measures. That is why quality measures are defined carefully in our approach to capture the performance implications of the corresponding constraints.

We now describe the design of each of the four passes in which decisions are taken on different aspect of the partitioning scheme, namely, (i) alignment of array dimensions, (ii) method of partitioning (blocked/cyclic), (iii) block sizes of distributions, and (iv) number of processors on which each array dimension is distributed.

5.1 Alignment of Array Dimensions

The `align` pass identifies the constraints on alignment among various array dimensions, and groups those array dimensions into classes that would be mapped to the same processor-mesh dimension. In the special case when the maximum dimensionality of any array in the program (and hence the dimensionality of the virtual processor mesh) is one, the problem becomes trivial, as all array dimensions are mapped to the same mesh dimension. The problem in its general form was first discussed and formulated in graph-theoretic terms by Li and Chen [48]. The idea of *conformance preference*, introduced by Knobe et al. [41] in the context of SIMD machines is also similar, but directed towards individual array elements instead of array dimensions.

We use the *Component Affinity Graph* (CAG) framework [48] for the alignment problem. The CAG constructed for the program has nodes representing dimensions of arrays. For every constraint on the alignment of two dimensions, an edge having a weight equal to the quality measure of the constraint is generated between the corresponding two nodes. Finally, the compiler solves the component alignment problem, which is defined as partitioning the node set of the CAG into D (D being the maximum dimensionality of arrays in the program) disjoint subsets such that the total weight of edges across nodes in different subsets is minimized. Each subset identifies the array dimensions that are to be mapped to the same mesh dimension. The weights on edges across the nodes belonging to different subsets denote communication costs that should be avoided if possible. There is an obvious restriction on the above partitioning, that no two nodes corresponding to the same array can be in the same subset. Our main contribution to this specific problem lies in the method of obtaining edge weights that reflect the communication costs saved by aligning two array dimensions.

5.1.1 Detection of Constraints

The detection of alignment constraints between array dimensions is quite straightforward. The compiler performs a pairwise analysis of the `lhs` array reference with each `rhs` array reference in every assignment statement inside a loop. The `rhs` references corresponding to the same

```
do j = 1, n
   do i = 1, n
      A(i, j) = F(B(j, 3 * i))
   enddo
enddo
```

Figure 5.2: References suggesting alignment constraints

array as the `lhs` array are ignored in this analysis. The compiler scans through the information kept on the subscript expressions for both the references. The presence of a pair of subscripts (one each from the `lhs` and `rhs` references) of the type `single-index`, and with the same value of `variation-level` suggests an alignment constraint on the corresponding array dimensions. For example, the program segment shown in Figure 5.2 leads to alignment constraints being recorded for A_1 and B_2, and for A_2 and B_1.

The significance of checking for the above condition is that in such cases, the alignment of the two dimensions can help save a great deat deal of communication costs. Normally, the easiest way to eliminate communication represented by a pair of subscripts is to sequentialize the corresponding dimensions, but that also implies giving up on parallelism. The above condition represents a case where even if those dimensions are partitioned on more than one processor, their remaining distribution parameters can be chosen such that interprocessor communication is reduced, or eliminated completely. In the above example, communication can be eliminated by further choosing identical distributions for A_2 and B_1, and distributions that give A_1 thrice the block size of B_2 but are identical in other regards (blocked/cyclic).

There may be situations when there are multiple candidates for alignment with a given dimension. For example, in the statement shown in Figure 5.3, either of A_1 or A_2 may be chosen for alignment with D_1. The compiler arbitrarily picks one of those dimensions for alignment. In such cases, the process of determining quality measures usually takes care of "reducing" the importance of those constraints. In the above example, the compiler would end up assigning a weight of zero to the edge corresponding to that alignment constraint.

$$
\begin{array}{l}
\text{do } i = 1, n \\
\quad D(i) = \mathcal{F}(A(i, i)) \\
\text{enddo}
\end{array}
$$

Figure 5.3: Multiple candidates for alignment

5.1.2 Determination of Quality Measure

Given an edge corresponding to the alignment constraint between two array dimensions, the method of quality measure determination should supply a weight to that edge that reflects the extra communication cost incurred if those dimensions are not aligned. Unfortunately, even with the availability of a communication cost estimator, there is no satisfactory solution to this problem in general. Consider two arrays A and B with m and n dimensions respectively, and an alignment constraint between dimensions A_i and B_j, $1 \leq i \leq m, 1 \leq j \leq n$. The communication cost can only be estimated corresponding to a given mapping of the array dimensions to mesh dimensions. For arbitrary values of m and n, we cannot identify two unique alignment configurations under which the communication costs may be estimated and compared to determine the penalty in case A_i and B_j are not aligned. Exhaustively enumerating all possible cases will only add to the complexity of the problem, and will not be feasible.

We overcome this difficulty in PARADIGM by solving the following approximate problem: for any pair of references, the alternate alignment configurations evaluated are those corresponding to the distribution of only two dimensions of each array at a time. We introduce the notion of *primary* and *secondary* pairs of dimensions for this purpose. Given a pair of array references, two array dimensions are said to form a *primary pair* if there is an alignment constraint on those dimensions. The dimensions are referred to as forming a *secondary pair* if there is no alignment constraint between them, but they get mapped to the same mesh dimension if the alignment constraint is honored for another primary pair. For example, in the program segment shown in Figure 5.4, A_1 and B_1 form a primary pair, while A_2 and B_2 form a secondary pair.

The compiler examines all alignment constraints for a given pair of references together. The basic idea underlying the determination of quality measure(s) is to first identify two pairs of

```
do i = 1, n
   do j = 1, n
      A(i, j) = F(B(i, k))
   enddo
enddo
```

Figure 5.4: Identification of primary and secondary pairs

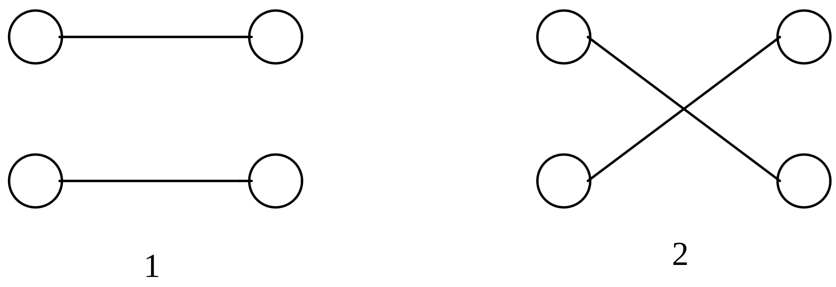

Figure 5.5: The two alignment configurations to obtain quality measures

dimensions, at least one of which forms a primary pair. The communication cost estimator is invoked to return the cost estimate, say, t_1, when the array dimensions are mapped so as to honor the alignment constraint(s). Next, as shown in Figure 5.5, the alignment of the two pairs of dimensions is swapped in the data partitioning information supplied to the estimator. The cost estimate obtained now, say, t_2 corresponds to the case when the alignment constraint(s) is (are) not satisfied. If there is only one primary pair of dimensions, the quality measure of that alignment constraint is set to $t_2 - t_1$. If both the pairs of dimensions are primary pairs, the quality measure of each of those constraints is set to $(t_2 - t_1)/2$. This accounts for the fact that under the given assumptions about which array dimensions are distributed, if one alignment constraint is not satisfied, the other will also not be satisfied. Hence, their quality measures are assigned values that will add up to give the performance penalty of $t_2 - t_1$.

In order to implement the above procedure, whenever a pair of references identifies only one primary pair, the compiler has to identify a secondary pair of dimensions. If the number of dimensions in the two arrays are different, the one with fewer dimensions is regarded as having extra "missing" dimensions, as was done for estimating communication costs. The identification

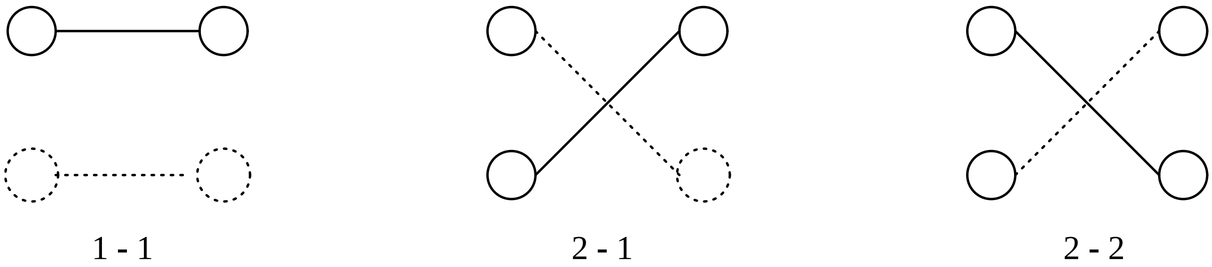

Figure 5.6: Identification of the secondary pair of dimensions

of the secondary pair is trivial when both the arrays have two or fewer dimensions, as shown in Figure 5.6. The solid lines in the figure link primary pairs, and the dotted lines link secondary pairs. The solid circles denote actual array dimensions, while the dotted circles denote missing dimensions.

If any array in the given pair of references has three or more dimensions, and if there is only one primary pair, there is more than one choice available for the array dimension that may form part of the secondary pair. In that case, the compiler selects one of the dimensions (other than the one constituting the primary pair) which "exhibits" parallelism somewhere in the program. A dimension is said to exhibit parallelism if there is at least one reference that traverses the particular dimension in some parallelizable loop. In order to help in this task, the `setup` pass in PARADIGM identifies all such array dimensions. During that pass, whenever the compiler detects that a statement can be parallelized with respect to a loop (as discussed in Chapter 3), all array dimensions for which the `variation-level` of the subscript identifies the same loop, are marked as exhibiting parallelism. Now if there are two or more candidates to form part of the secondary pair, the compiler chooses one of them arbitrarily (it picks the first such dimension). In the rare case when the two arrays have more than three dimensions each, and also at least three primary pairs, the above procedure is carried out for two dimension pairs at a time.

Another step to be performed in our procedure for obtaining the quality measures is to supply the remaining partitioning parameters for the two pairs of dimensions that are assumed to be distributed. Note that determining the alignment is the very first step, so all other parameters are unknown at this stage. The compiler obtains the cost estimates under the assumption that all array dimensions are distributed in a blocked manner. While the estimates are obtained as functions of the number of processors in the two mesh dimensions, the numerical values for the purpose of resolving conflicts are obtained assuming that both pairs of array dimensions are distributed on an equal number of processors. Thus in this step, the number of processors in each dimension is assumed to be $\sqrt{N}$, where N is the total number of processors in the system. Our experimental results (described in the next chapter) suggest that the resulting cost estimates, in spite of the inaccuracy due to these assumptions, usually guide the alignment decisions in the right direction.

Example Consider again the statement shown in Figure 5.2, where the arrays A and B have sizes n x n and n x $3n$ respectively. There are two primary pairs corresponding to alignment constraints: (i) A_1 and B_2, and (ii) A_2 and B_1. The communication cost estimator is first invoked for the case when these dimensions are aligned appropriately, and distributed in a blocked manner on $N_1 = \sqrt{N}$ processors each. The cost estimator finds each pair of sub-references satisfying the `strictly synchronous` property. Hence the data movement is detected as being internalized, and the cost estimate returned is zero.

Next, the relative alignment of the two pairs of dimensions is swapped in the data partitioning information supplied to the communication cost estimator. This time, the cost estimator finds each pair of sub-references (corresponding to A_1 and B_1, and to A_2 and B_2) varying in the same loop, the i-loop. The expression j represents a subscript of the type `single-index`, but with a different value of `variation-level` than the subscript expression i, and hence is regarded as a constant in the i-loop. As shown in Table 4.3, the cost estimator tests for the k-synchronous property between the sub-references $\langle A(i,j), 1\rangle$ and $\langle B(j, 3*i), 2\rangle$. The two sub-references are found to be 1-synchronous, and hence the communication term obtained is $\text{Transfer}(n/N_1)$. To account for the tiling of the j-loop, this term is modified to give the final communication cost estimate as $N_1 * \text{Transfer}((n*n)/(N_1*N_1))$. The quality measure assigned to each of the two alignment constraints is $0.5 * N_1 * \text{Transfer}((n*n)/(N_1*N_1))$.

For each alignment constraint on a pair of dimensions, the compiler introduces an edge between the corresponding nodes in the CAG. The quality measure obtained for that constraint is recorded as the weight on the edge. If an edge already exists corresponding to the alignment constraint to be recorded, the quality measure value is simply added to the existing value of the weight on that edge. Once the entire program has been analyzed in this manner, the compiler obtains a solution to the alignment problem.

5.1.3 Determination of Distribution Parameter

The component alignment problem has been shown to be NP-complete, and a heuristic algorithm is given in [48]. PARADIGM uses a similar algorithm, so we shall just give a brief description here.

1. Sort all the columns (each column refers to the collection of nodes corresponding to different dimensions of a single array) of CAG in a non-decreasing order of their number of dimensions. Let the number of columns (i.e., the number of arrays) be num.

2. For i varying from 2 to num do

   (a) Find the optimal weighted matching between the bipartite graph corresponding to columns C_1 and C_i. Since the number of nodes in each column is usually less than four, this step can be done by exhaustively considering all possible matchings.

   (b) Merge the columns C_i and C_1 into a single column C_1 by combining the matched nodes according to the optimal matching. Clean up the CAG by replacing each edge from a node in the remaining columns to a node originally in C_i by an edge to the corresponding merged node in C_1, and by replacing multiple edges between nodes by a single edge with weight equal to the sum of their weights.

All the merges performed between columns represent alignment constraints honored between dimensions of the corresponding arrays. At the end of the algorithm, each node in C_1 represents a class of array dimensions that are mapped to the same processor mesh dimension. Thus, the information on mapping of array dimensions to processor mesh dimensions becomes known at the end of the `align` pass.

5.2 Method of Partitioning: Blocked/Cyclic

The `block-cyclic` pass determines for each array dimension, whether it should be distributed in a blocked or cyclic manner. It is important to note that "blocked" and "cyclic" distributions are simply two extremes of a general distribution, referred to as blocked-cyclic distribution. The blocked distribution represents the special case when the block size is set to S/P, where S is the dimension size and P is the number of processors. The term "cyclic" distribution normally refers to the special case of blocked-cyclic distribution, with a block size of one. In this section, we shall describe the procedure for choosing between these two extreme positions. Later, we shall describe how further adjustments may be made on the block size of a dimension given a cyclic distribution.

First, let us clarify the terminology in yet another fuzzy situation. In real programs, the arrays are often statically declared to have bigger sizes than their actual sizes at run time. For example, a single dimensional array may be declared to be of the size 1000, but at run time may have only 800 elements. If it is to be distributed in a blocked manner on 10 processors, it would be desirable to distribute all the 800 elements equally among those processors. This can be accomplished by specifying a blocked-cyclic distribution for the array with a block size of 80, so that the actual 800 elements are distributed in a blocked manner, and the remaining "non-existent" elements are mapped to the first three processors by the wrap-around. We regard such distributions also as blocked distributions. For simplicity, the current version of Paradigm requires the programmer to supply the actual sizes of all arrays, so that there is no disparity between the declared and the actual sizes.

We shall now examine the conditions under which there are constraints recorded on individual array dimensions for blocked or cyclic distributions. After describing the procedure to obtain quality measures for those constraints, we shall describe how collective decisions are taken on the method of partitioning for groups within each class of aligned array dimensions.

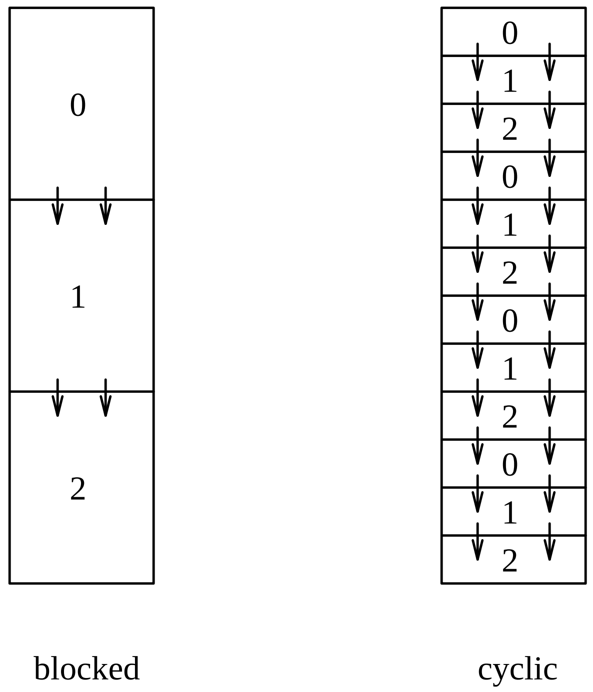

Figure 5.7: Need for blocked distribution

5.2.1 Detection of Constraints

Blocked Distribution

If the distribution of any array dimension on more than one processor leads to communication between nearest neighbors in that mesh dimension, as shown in Figure 5.7, it clearly indicates the need for blocked distribution. The use of blocked distribution allows the communication to be restricted to the elements lying at the boundaries of regions assigned to processors. If the dependence constraints allow these communications to be carried out independently, usually the savings are in the associated data sizes of messages. Cyclic distribution of such a dimension would lead to larger amounts of data being transferred than blocked distribution. If these communications take place in type-2 loops (due to dependence constraints), the potential advantages of blocked distribution are even greater. In that case, the choice of cyclic distribution leads to a larger number of messages to be sent. Given the high start-up costs of sending messages, the penalties of such a choice can be tremendous.

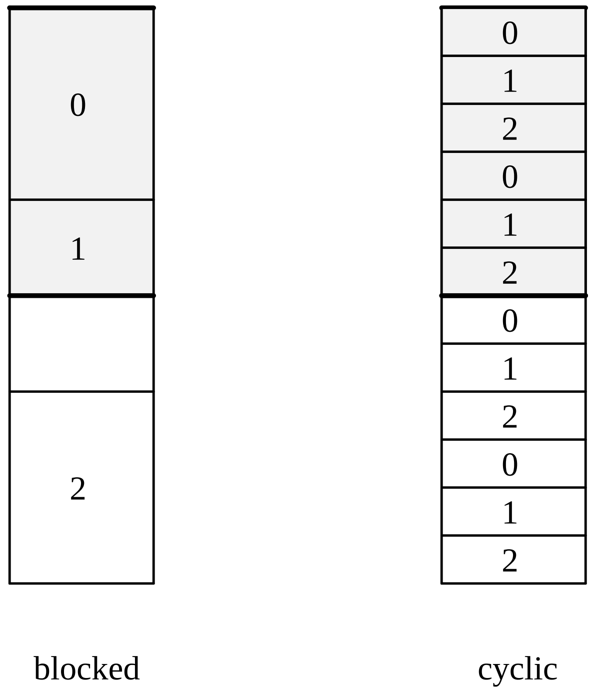

Figure 5.8: Need for cyclic distribution

Hence, during the detection phase, the compiler analyzes pairs of sub-references corresponding to aligned dimensions in each assignment statement. If the corresponding subscripts are of the type `single-index` and the sub-references satisfy the boundary-communication test described in Chapter 4, a constraint favoring blocked distribution is recorded for the `rhs` array dimension.

Cyclic Distribution

Usually, the main motivation for distributing an array dimension in a cyclic manner is to get better load balance for parallelizable computations. Consider an assignment statement in which the `lhs` array gets only partially traversed, as indicated by the filled region in Figure 5.8. Since the computations are partitioned according to the *owner computes* rule, using cyclic distribution with a small block size leads to a more even distribution of computation than using blocked distribution. That leads to a reduction in the execution time if those computations are parallelizable.

$$\begin{array}{l} \text{do } i = 2, n \\ \quad D(i) = \mathcal{F}(D(i-1)) \\ \text{enddo} \end{array}$$

Figure 5.9: References suggesting constraint for blocked distribution

The detection of constraints for cyclic distribution proceeds in the same phase as that for blocked distribution. For each `lhs` array sub-reference in an assignment statement, the compiler examines the extent to which the subscript varies in the loop corresponding to its `variation-level`. The compiler looks for a subscript of the type `single-index`, i.e., one of the form $\alpha_1 * j_l + \beta_1$. Using the terminology from Chapter 3 again, let the corresponding loop index j_l vary from i_l to f_l in steps of s_l, with the range of the loop given by $r_l = f_l - i_l + s_l$. The extent of traversal of the subscript along the array dimension is given by $|\alpha_1| * r_l$. If the ratio of the extent of traversal to the dimension size is less than a certain threshold (the value of this threshold is set to 2/3 in the current implementation), *and* if the given statement is parallelizable with respect to the loop L_l, the compiler introduces a constraint on cyclic distribution for the given array dimension.

5.2.2 Determination of Quality Measures

The quality measure of a constraint for blocked (cyclic) distribution is an estimate of the penalty incurred in execution time if the array dimension is instead given a cyclic (blocked) distribution.

Blocked Distribution

Consider the statement shown in Figure 5.9. The analysis of the given pair of references shows a need for boundary Transfers inside the i-loop, and hence suggests a constraint for blocked distribution of D_1. Given the general blocked-cyclic distribution of D_1, the communication cost for the `rhs` reference is estimated by the compiler as:

$$\text{Cost} = (\lceil (n-1)/b_1 \rceil - 1) * \text{Transfer}(1),$$

do $i = 1, n$
   do $j = 1, i$
      $D(j) = \cdots$

Figure 5.10: Reference suggesting constraint for cyclic distribution

where b_1 is the block size of distribution of D_1. Let N_1 denote the number of processors on which D_1 is distributed. The quality measure for the constraint is obtained as the difference in the communication costs when the block size is 1 (corresponding to cyclic distribution), and when the block size is $\lceil n/N_1 \rceil$ (corresponding to blocked distribution). Thus, the compiler obtains the following value:

$$\text{Quality measure} = (n-2) * \text{Transfer}(1) - (\lceil (n-1)/(\lceil n/N_1 \rceil) \rceil - 1) * \text{Transfer}(1)$$

Cyclic Distribution

As an example of a constraint favoring cyclic distribution, consider the program segment shown in Figure 5.10. Let the array D consist of n elements. The compiler regards the range of the j-loop as $\lceil n/2 \rceil$. The extent of traversal of the subscript j over D_1 during the j-loop is accordingly determined as $\lceil n/2 \rceil$. Hence, the compiler recognizes the need for cyclic distribution of D_1 in order to obtain better load balance. Let t denote the estimated computational time for executing one instance of the given statement. As shown by Equations 3.1 and 3.2 in Chapter 3, the overall computational cost estimate for the statement is obtained as:

$$\text{Cost} \quad = \quad t * n * (b_1 * \lfloor \lceil n/2 \rceil / (N_1 * b_1) \rfloor + \min(b_1, \lceil n/2 \rceil \bmod (N_1 * b_1)))$$

The quality measure of the constraint is given by the difference in computational cost when the block size is $\lceil n/N_1 \rceil$ (for blocked distribution), and when the block size is 1 (for cyclic distribution). The estimated cost amounts to $t * n * \lceil n/N_1 \rceil$ for blocked distribution, and to $t*n*\lceil n/(2*N_1) \rceil$ for cyclic distribution. This corresponds to PARADIGM estimating the speedup roughly as $N_1/2$ for blocked distribution, and N_1 for cyclic distribution, and hence recording a constraint favoring cyclic distribution, with an appropriate quality measure.

5.2.3 Determination of Distribution Parameter

So far we have discussed only the requirements of individual array dimensions with regard to blocked or cyclic distribution. For arrays that cross-reference each other, it is also important to make sure that their aligned dimensions are given the same kind of partitioning: blocked or cyclic. Otherwise, the alignment of array dimensions would fail to ensure the intended alignment of array elements, leading to excessive communication costs that defeat the purpose of aligning those dimensions.

However, not all the array dimensions that are mapped to the same mesh dimension need be constrained to have identical methods of partitioning. To determine which array dimensions must satisfy that property, PARADIGM keeps track of the transitive closure of the cross-referencing (CR) relationships between arrays. Two arrays A and B are said to be CR-related if there is an assignment statement in the program that has A (B) as its `lhs` array, and B (A) as its `rhs` array. The `setup` pass in PARADIGM records this information in the form of disjoint CR sets, each set consisting of arrays that are directly or transitively CR-related. This is done using the well-known *union* and *find* algorithms [18]. Initially, each array is in a separate CR set by itself. For each pair of `lhs` and `rhs` arrays A and B in an assignment statement, the compiler first determines the CR sets containing A and B using the *find* algorithm. If the two CR sets are different, they are combined using the *union* algorithm.

Given the above information, PARADIGM takes collective decisions on the method of partitioning for all the array dimensions that are mapped to the same mesh dimension, and that correspond to arrays in the same CR set. For each such group of array dimensions, the compiler compares the sum of quality measures of constraints advocating blocked distribution and those favoring cyclic distribution, and chooses the one with the higher value. In case of a tie (for instance, if there are no constraints either way), the compiler chooses blocked partitioning. Thus, decisions in this pass are considerably influenced by alignment considerations as well.

5.3 Block Sizes of Distributions

While the compiler makes a choice between the extreme positions of blocked and cyclic distribution during the `block-cyclic` pass, further adjustments on the block size are made during the `block-size` pass. The load-balancing considerations that lead to cyclic distribution often favor using a block size of one (the smallest possible value) to get a more even distribution of computation. In this section, we show how the compiler detects and attempts to satisfy other requirements on the block sizes of distributions. These requirements include relationships between block sizes of aligned array dimensions, that ensure the proper alignment of the underlying array elements.

We believe that this analysis is sometimes needed when arrays are used to simulate certain data structures like records, not supported directly by Fortran, or when lower-dimensional arrays play the role of higher-dimensional arrays. For instance, a linear array of records may sometimes be represented simply as a 1-D array, where the programmer "knows" that a certain number of successive elements in the array represent a single logical entity. An example of that is seen in the `mdg` program of the Perfect Benchmarks [17], where certain items such as the velocities and momenta of water molecules are stored in 1-D arrays that contain the relevant information for each of the three atoms making up every molecule. The block sizes of distributions of those arrays need to be given values that are thrice the corresponding values for arrays that store just a single piece of information for each molecule.

The current version of PARADIGM carries out this analysis only for cyclic distributions. The same ideas can also be used to choose appropriate block sizes for "blocked" distributions, discussed in the previous section, where the static declarations show bigger dimension sizes than the actual sizes at run time. However, presently the compiler expects values of the actual dimension sizes to be supplied by the programmer. The block size for a dimension distributed in a blocked manner is then determined only by the dimension size and the number of processors in the corresponding mesh dimension.

5.3.1 Detection of Constraints

The compiler examines the sub-references corresponding to aligned dimensions for each pair of `lhs` and `rhs` array references in an assignment statement. For sub-references corresponding to different arrays, if both of the corresponding subscripts are of the type `single-index`, and have the same value of `variation-level`, block-size constraints are recorded as follows. Let the two subscripts be of the form $\alpha_1 * j_l + \beta_1$, and $\alpha_2 * j_l + \beta_2$, with $\omega_1 = |\alpha_1|$, and $\omega_2 = |\alpha_2|$ The block-size constraint induced by the given pair of sub-references suggests two related requirements, namely: (i) the block sizes of (distributions of) the two array dimensions should respectively be multiples of ω_1 and ω_2, and (ii) the ratio of block sizes of those dimensions should be ω_1/ω_2. The first requirement represents one of the conditions for regularity of data movement in case of interprocessor communication. The second requirement is essential for proper alignment of the elements of the two arrays. Both of these requirements are satisfied if the block sizes chosen for the two dimensions are $k * \omega_1$ and $k * \omega_2$, where k is an integer, preferably as small as possible.

The block-size constraints and their quality measures are represented by means of an undirected graph, the *Block-size Constraint Graph* (BCG), that is similar in some regards to the CAG used for alignment constraints. The BCG has nodes representing array dimensions, and edges representing the block-size constraints between aligned dimensions. Each edge is marked with two items of information: (i) the coefficients ω_1 and ω_2 associated with the given constraint (our representation identifies the node that each coefficient corresponds to, even though the edge itself is undirected), and (ii) the value of weight, that is equal to the quality measure of that constraint.

5.3.2 Determination of Quality Measures

The quality measure of a block-size constraint between two array dimensions is an estimate of the extra communication cost incurred if the conditions associated with the given constraint are not met. To obtain estimates for the case when the conditions are satisfied, the compiler sets the block sizes of the two dimensions to the values ω_1 and ω_2 respectively, and invokes

do $i = 1, n$
    $A(i) = \mathcal{F}(B(3 * i))$
enddo

Figure 5.11: References suggesting block-size constraint

the communication cost estimator. However, for the situation when the conditions are not satisfied, there are numerous possibilities regarding the block sizes of those dimensions. The compiler needs to make a "reasonable" choice among those possibilities to model the effect of the constraint not being honored.

The default block size given to dimensions in the absence of any block size constraint is 1. Hence if either of ω_1 or ω_2 is not equal to 1, the compiler sets the block size of both dimensions to 1 while obtaining cost estimates for the case when the constraint is not satisfied. If both ω_1 and ω_2 are equal to 1, the compiler sets the block size of the `lhs` dimension to 1, and uses an arbitrary value (two) for the block size of the `rhs` dimension to model the situation when the constraint is not satisfied.

Consider the statement shown in Figure 5.11. Let the sizes of the dimensions A_1 and B_1 be n and $3 * n$ respectively, and let them be distributed in a cyclic manner on N_1 processors. There is a constraint recorded between A_1 and B_1 that requires their respective block sizes to be $k * 1$ and $k * 3$ for a small integer k. To obtain the quality measure, the communication cost estimator is first invoked with the block size of A_1 set to 1, and of B_1 set to 3. The data movement is completely internalized in that case, and the communication cost returned is zero. The situation when the constraint is not satisfied is created by setting both the block sizes to 1. In this case, the cost estimator is unable to detect any regularity in the data movement, due to the block size of B_1 not being a multiple of 3 (as discussed in Chapter 4). Therefore, the cost estimator assumes the use of repeated Transfers inside the i-loop, and returns an estimate of $n * (1 - 1/N_1) *$ Transfer(1). Since the cost estimate in the other case is zero, this term also gives the quality measure of the constraint.

Each new block-size constraint corresponding to a pair of dimensions is recorded as an edge in the BCG. The edge is marked with the associated coefficient values ω_1 and ω_2, and the value of quality measure is recorded as the weight on that edge. If at a certain point in the program, an edge already exists in the BCG corresponding to a constraint to be recorded, the compiler examines the relationship between the coefficients ω_1^o and ω_2^o on the existing edge, and the coefficients ω_1^n and ω_2^n associated with the present constraint. If $\omega_1^o/\omega_2^o = \omega_1^n/\omega_2^n$, the two constraints are consistent. In this case, the compiler computes the value of $lcm(\omega_1^o, \omega_1^n)$ (the lowest common multiple of ω_1^o and ω_1^n). The coefficients on the edge are changed to $lcm(\omega_1^o, \omega_1^n)$ and $(\omega_2^o * lcm(\omega_1^o, \omega_1^n))/\omega_1^o$ respectively, and the value of quality measure for the present constraint is added to the weight on the existing edge. If on the other hand, $\omega_1^o/\omega_2^o \neq \omega_1^n/\omega_2^n$, the two constraints are not consistent with each other. In that case, a separate edge with its own values of coefficients and weight is introduced between the same pair of nodes.

5.3.3 Determination of Distribution Parameter

We now describe the algorithm that takes into account all block-size constraints, and obtains the block sizes for all the array dimensions with cyclic distributions. The objective is to select the block sizes such that the sum of quality measures of constraints that are not satisfied is minimized. The following theorem indicates a sufficient condition for the absence of conflicts among constraints.

Theorem 5.1 *The conditions associated with all of the block-size constraints in the program can be satisfied if there are no cycles in the BCG.*

Proof: The absence of cycles implies that there are no multiple edges between any pair of nodes in the BCG. Hence, by our method of construction of BCG, there are no direct conflicts between block-size requirements on a pair of dimensions due to any two constraints. We shall now prove the absence of indirect conflicts as well, by presenting an algorithm that constructs a solution for such a BCG.

Each component (connected component) of the BCG forms a tree, since there are no cycles in the graph [1]. For each tree, the following algorithm selects the block sizes corresponding to all its nodes (i.e., the array dimensions associated with those nodes) such that every block-size constraint between them is satisfied.

1. Choose an arbitrary node r in the tree, and any edge incident on that node. Set the block size of the node r to the value of the coefficient ω_1 marked for r on the edge.
2. For each node v visited in a *preorder* traversal starting from node r do
   (a) Let b_1 be the block size of the previous node p (keep track of the last node visited). Let ω_1 and ω_2 be the values of coefficients associated with the nodes p and v respectively on the current edge.
   (b) If $b_1 * \omega_2$ is perfectly divisible by ω_1, set the block size b_2 of node v to $(b_1 * \omega_2)/\omega_1$, and go to the next step. Otherwise, set l to $lcm(b_1 * \omega_2, \omega_1)$; set the block size of v to l/ω_1; for each of the node visited so far (from r to p), multiply its value of block size by $l/(b_1 * \omega_2)$.

It can be seen that the assignment(s) made to block size(s) in Step 2 (b) at the time of traversal of each edge ensure that the following condition is satisfied for every edge traversed so far: the block sizes of the two nodes connected by the edge have values that can be expressed as $k*\omega_1$ and $k*\omega_2$ respectively, where k is an integer, and ω_1 and ω_2 are the values of the coefficients recorded for that edge. Thus, the application of the above algorithm to each component of the graph BCG results in an assignment of block sizes that satisfies all of the block-size constraints. $\square$

In general, the BCG for a program may have cycles that introduce conflicts between block-size constraints, as shown in Figure 5.12. These conflicts may be (i) direct, as shown by the presence of multiple edges between a given pair of nodes, or (ii) indirect, as shown by cycles involving more than two nodes (the presence of such cycles is a necessary, and not a sufficient condition for an indirect conflict). The first problem is dealt with by retaining only one edge

[1] The BCG has at least as many components as the number of mesh dimensions.

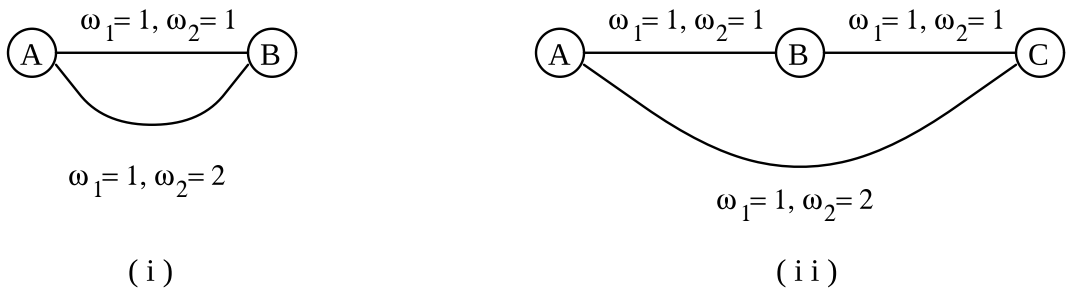

Figure 5.12: Conflicts between block-size constraints

between any pair, the one with the highest weight, and deleting all other edges between that pair.

Let us explain how PARADIGM deals with the second problem. The objective of the conflict resolution strategy is to disregard those conflicting constraints that have the smallest value of quality measure associated with them. In the context of BCG, that translates to removing those conflicting edges from each cycle that have the smallest weight on them. We have already shown that if each component of a BCG is a tree, all of the block-size constraints corresponding to the edges in that tree can be satisfied. This suggests that the compiler should identify trees that include the "heavy" edges. Hence, for each component of BCG, the compiler finds a *maximum cost spanning tree*, i.e., one for which the sum of weights on the edges is maximum. The algorithm used for this purpose is one obtained by a trivial modification of Prim's algorithm [58, 18] for finding a *minimum* cost spanning tree. Once the spanning tree is obtained, the block sizes for (the dimensions corresponding to) the nodes in the graph are determined by applying the algorithm shown above in the proof of Theorem 5.1. The block size for each other dimension with a cyclic distribution is set to 1, by default.

5.4 Number of Processors in Mesh Dimensions

The only data partitioning parameters that are left unknown at the beginning of the `num-procs` pass are the number of processors in different mesh dimensions. The `num-procs` pass attempts to determine these parameters such that the expected parallel execution time of the program is minimized.

The structure of this pass is slightly different from that of other passes, due to the different nature of the optimization problem to be solved. In this case, the constraints do not suggest definite values of the distribution parameters (i.e., the number of processors in various mesh dimensions) for getting good performance. Hence, the notion of selecting certain constraints to be honored, is absent from the optimization process. Correspondingly, the quality measures of these constraints are defined in terms of the expected execution time, rather than the performance penalty, and are expressed as functions of the number of processors in different mesh dimensions. The quality measures of *sequentialization constraints*, that recommend sequentializing an array dimension to eliminate interprocessor communication, are expressed as functions that evaluate to zero if the number of processors is set to one.

5.4.1 Recording of Constraints and Quality Measures

In this phase, the compiler analyzes each assignment inside a loop, to an array element, or to a scalar involved in a reduction operation. Both the computational and the communication cost estimators are invoked, and the sum of those estimate terms gives the contribution of the statement to the expected program execution time. Since all other data partitioning parameters are known at this point, these terms are functions only of the number of processors in different mesh dimensions.

The sum of contribution terms from all statements examined by the compiler yields the relevant part of the expected program execution time, that drives the selection of the number of processors. For the most part, these terms are determined only once. They are simply re-evaluated with different values for number of processors, in different steps of the optimization process. However for estimating the communication costs of statements with certain kinds of references as described in Table 4.3 of Chapter 4, the precise value of the block size, and hence, of the number of processors (in case the dimension is distributed in a blocked manner) is needed.

For such references, the communication cost terms are determined repeatedly in different steps of the optimization process described below.

5.4.2 Determination of Distribution Parameter

PARADIGM currently determines an appropriate data partitioning scheme for *exactly* the number of processors, N, specified as available in the system. It is well-known that for any program operating on a fixed data size, increasing the number of processors (on which the program is partitioned) beyond a certain point causes a degradation in performance due to higher communication overheads. Hence, another flavor of the data partitioning problem would be one where given a value of N, the compiler also has to determine N', the number of processors on which data should be distributed. For that problem, the algorithms described below can easily be extended to explore the search space over N' that varies from 1 to N.

Reduction of Number of Mesh Dimensions

In the preceding passes, each dimension of an array is assumed to be distributed on a distinct mesh dimension, and hence, potentially on more than one processor. Clearly, mesh dimensions in which excessive communication take place, and those in which not much parallelism is exhibited during the program, need to be collapsed into a single processor. In order to reduce the amount of search space, PARADIGM sequentializes all except for two (at most) dimensions of each array. Thus, the target topology is finally configured as a 1-D or a 2-D mesh.

We believe that for most scientific application programs, restricting the number of distributed dimensions of a single array to two does not lead to any loss of effective parallelism. In a comprehensive study of Fortran programs [70], Shen et al. reported finding only 8% of the array references with more than two dimensions, and only 1% with more than three dimensions. Even when higher-dimensional arrays show parallelism in each dimension, restricting the number of distributed dimensions does not necessarily limit the extent to which parallelism can be exploited. For instance, the statement shown in Figure 5.13 has references to three-dimensional

```
do k = 1, n
    do j = 1, n
        do i = 1, n
            Z(i, j, k) = c * Z(i, j, k) + Y(i, j, k)
        enddo
    enddo
enddo
```

Figure 5.13: References to higher-dimensional arrays in parallelizable loops

arrays, and appears in a completely parallelizable loop nest. However with a fixed number of processors, roughly the same speedup would be obtained whether we distribute two or three dimensions of the arrays Y and Z.

If the value of D (the maximum dimensionality of any array in the program) is greater than two, the compiler selects $D-2$ of the mesh dimensions to be collapsed. This is done through the following steps:

1. For a given set of aligned array dimensions, the compiler examines the field recorded for each array dimension during the `setup` pass, that indicates whether the array dimension exhibits parallelism somewhere in the program. If none of the array dimensions in the set exhibits parallelism, the mesh dimension to which they are mapped is collapsed into a single processor. For the purpose of obtaining performance estimates during the remainder of this pass, the parameter representing the number of processors in that dimension is set to one. This step is repeated for all of the D sets of aligned array dimensions.

2. Let the number of dimensions that have not been collapsed at the end of the previous step be D'. This step is required only if $D' > 2$. The estimates for times spent on computation and on communication are available to the compiler as functions of the parameters $N_1, N_2, \ldots N_{D'}$, representing the numbers of processors along the corresponding mesh dimensions. The compiler evaluates the expected execution times of the program for $C_2^{D'}$ cases ($C_2^{D'}$ is the number of ways of choosing 2 items from D' items), each case corresponding to two different N_i variables set to $\sqrt{N}$, and the other $D'-2$ variables set

to 1. The case which yields the smallest value for expected execution time is chosen, and the corresponding $D' - 2$ dimensions are collapsed.

Search for Mesh Configuration

At this point, there are at most two mesh dimensions that have not been collapsed. If there is only one such dimension left, the number of processors in that dimension is set to N, and that marks the end of the `num-procs` pass.

If the number of mesh dimensions is two, the only parameters left unknown in the partitioning scheme are N_1 and N_2, the number of processors in those dimensions. The two parameters are related by $N_1 * N_2 = N$. The compiler evaluates different mesh configurations corresponding to values of N_1 varying from 1 to N, and being doubled in each step (assuming N is a power of two). The expression for the expected execution time is evaluated for each configuration, and the one that leads to the smallest expected execution time is selected.

CHAPTER 6

EXPERIMENTAL RESULTS

The methodology we have described for automatic data partitioning has been implemented as part of the PARADIGM system. Our system accepts Fortran 77 programs and determines the partitioning scheme to be used for all the arrays in them. In this chapter, we present the results obtained by PARADIGM on some Fortran codes taken from the Linpack and Eispack libraries, and the Perfect Benchmarks [17].

PARADIGM has been designed to be largely machine-independent. There is a small machine-dependent part that supplies the cost metrics of various operations used in the local computations, and of high level communication primitives supported on the target machine. Clearly, the validation of the ideas underlying our approach has to be done on a real machine. The testbed used for the implementation and evaluation of our system is the Intel iPSC/2 hypercube [5].

The final measure of success of any automatic data partitioning scheme is the performance of the resulting compiler-generated parallel program on the target multicomputer. However, most of the compilers that carry out the task of generating data-parallel programs, given the sequential program and data partitioning information, are still under development. Hence currently, the validation of results through actual program performance is a very tedious process. It requires manual development of different versions of the parallel program with message passing, corresponding to different data partitioning schemes.

For the sake of evaluation of our approach, we have gone through this effort for three of the programs. We have developed multiple parallel program versions corresponding to different data partitioning schemes. For those programs, we shall show the actual performance results of different versions on the iPSC/2. For the remaining programs, we shall only show the data distribution scheme determined by the compiler, and explain the rationale behind some of the choices of data distribution parameters. A more rigorous evaluation would be possible once sophisticated compilers for multicomputers become available. For the present, we believe even these results are quite significant, since the compilers have widely been regarded as incapable yet of taking good data partitioning decisions.

6.1 Methodology

The results we report on data partitioning are obtained using the version of PARADIGM supplied with information on the cost metrics for the iPSC/2. All of the results are obtained for a 16-processor system. For some of the programs in which the choice of partitioning scheme is influenced considerably by the data size, we present results for different data sizes used. The information on the size of each array is added to each program, in case the original program does not have that information.

The current version of PARADIGM does not perform interprocedural analysis. Therefore, programs with procedure calls are first transformed through in-line expansion of procedure calls. In order to obtain results for some of the larger programs, we have selected the routines, based on profiling information, in which the program spends the maximum amount of time. These routines are then analyzed by PARADIGM as separate entities.

The source listing of the final version of each program is being provided for the sake of clarity, and to allow further evaluation of these results in future. Two of the programs are shown in Figures 6.1 and 6.4. The remaining programs are listed in Appendix I.

6.1.1 Application Programs

Results are presented on six Fortran programs of varying complexity. The smallest program is `Jacobi`, a simplified version of a relaxation code that performs Jacobi iterations in a loop. The second program is `tred2`, a routine taken from the Eispack library. It reduces a real symmetric matrix to a symmetric tridiagonal matrix, using and accumulating orthogonal similarity transformations. The next program is `dgefa`, taken from the Linpack library. It factors a real matrix using gaussian elimination with partial pivoting. This program makes calls to some other Linpack routines. Hence, the version we use is a transformed one where procedure in-lining has been done by hand.

The remaining three programs are individual procedures taken from the Perfect Benchmarks. `Olda` is the dominant procedure in the `trfd` program that simulates the computational aspects of a two-electron integral transformation. A profile of the sequential version showed the `trfd` program spending more than 98% of its time in the `olda` routine. The routines `dflux` and `eflux` are two of the three most important procedures (in terms of time spent) of the `flo52` program. `Flo52` is a two-dimensional code that analyzes the transonic inviscid flow past an airfoil by solving the unsteady Euler equations. The other important procedure in that program, `psmoo` is a much simpler piece of code, and uses only one array. Hence, we have chosen to illustrate our methodology for the above, more interesting routines.

6.1.2 Machine-Specific Information

We now describe the information assumed regarding the performance characteristics of the iPSC/2 machine, as supplied to the compiler. The following approximate function [33] is used to estimate the time taken, in microseconds, to complete a `Transfer` operation on m bytes :

$$Transfer(m) = \begin{cases} 350 + 0.15 * m & \text{if } m < 100 \\ 700 + 0.36 * m & \text{otherwise} \end{cases}$$

Note that our examples usually express the message sizes in terms of the number of words. A double-precision floating point number occupies eight bytes of storage.

Primitive	*Cost on iPSC/2*
Reduction(m, p)	$\lceil \log_2(p) \rceil *$ Transfer(m)
Shift(m)	$2*$ Transfer(m)
OneToManyMulticast(m, p)	$\lceil \log_2(p) \rceil *$ Transfer(m)
ManyToManyMulticast(m, p)	$(p-1)*$ Shift(m)
Scatter(m, p)	$(p-1)*$ Transfer(m)
Gather(m, p)	$(p-1)*$ Transfer(m)

Table 6.1: Costs of collective communication primitives on the iPSC/2

The cost metrics used for the collective communication primitives are shown in Table 6.1, where they are expressed in terms of the time to carry out a Transfer primitive. The parameter p denotes the number of processors over which the primitive is carried out. Both Reduction and OneToManyMulticast take $\log_2(p)$ steps with tree-based algorithms. A Shift operation involves each processor sending data to its neigboring processor, and receiving data from another neighboring processor in the other direction, and hence is modeled as taking time equivalent to two Transfers. A ManyToManyMulticast over a group of p processors can be done via $p-1$ Shifts using a ring-based algorithm. A Scatter operation is implemented as a sequence of Transfers, where the source processor sends different pieces of data individually to the $p-1$ other processors. Similarly, a Gather primitive involves $p-1$ processors sending data to one processor. In case a different algorithm is used for any of these primitives, the cost measures can be modified appropriately.

The double-precision floating point performance of a single node of iPSC/2 has been reported at 0.2 MFlops [5]. Hence for the computational costs, each double precision floating point add or multiply operation is assumed to take 5 microseconds. The floating point division is assumed to be thrice as expensive. To obtain the costs of referencing variables, the compiler uses a simple model, and does not attempt to estimate the extent of usage of registers or cache memory. The compiler assumes that each load and store operation takes 0.5 microseconds each. The timing overhead associated with various control instructions is ignored.

6.2 Data Partitioning Schemes

We now describe the data distribution schemes determined by PARADIGM for different programs. We have chosen the `tred2` program to illustrate the operation of different data distribution passes in greater detail, since `tred2` is a small yet reasonably complex program that defies easy determination of the "best" data partitioning scheme by simple inspection.

6.2.1 Application 1: TRED2

The source code of `tred2` is listed in Figure 6.1. The program uses four different arrays. The arrays A and Z are two-dimensional, while D and E are one-dimensional.

Alignment The first step relating to data distribution is the determination of alignment between array dimensions. The Component Affinity Graph (CAG) built by PARADIGM based on the alignment constraints in different parts of the program is shown in Figure 6.2. The symbolic expressions corresponding to the edge weights on CAG are shown below:

$$
\begin{aligned}
c_1 &= [0.5 * (N_1/2) * \text{Transfer}((n * n)/N) - 0] \text{ (line 3)} \\
c_2 &= [0.5 * (N_1/2) * \text{Transfer}((n * n)/N) - 0] \text{ (line 3)} \\
c_3 &= [\text{Transfer}(n/N_1) - (1 - 1/N_1) * \text{Transfer}(n/N_1)] \text{ (line 4)} \\
c_4 &= [(n-1) * (n/2) * \text{Transfer}(n/N_1) - (n-1) * (n/2) * (1 - 1/N_1) * \text{Transfer}(n/N_1)] \\
&\quad \text{(line 58)} + [(n-1) * \text{Transfer}(n/N_1) - (n-1) * (1 - 1/N_1) * \text{Transfer}(n/N_1)] \text{ (line 71)} + \\
&\quad [(n-1) * (\lfloor N_1/4 \rfloor + 1) * \text{Transfer}(n/N_1) - \\
&\quad (n-1) * \text{OneToManyMulticast}(n/N_1, (\lfloor N_1/4 \rfloor + 1))] \text{ (line 77)} \\
c_5 &= [(n-1) * (n/2) * (1 - 1/N) * \text{Transfer}(1) - (n-1) * (n/2) * (1 - 1/N_1) * \text{Transfer}(1)] \\
&\quad \text{(line 18)} + [(n-1) * (n/2) * (1 - 1/N) * \text{Transfer}(1) - \\
&\quad (n-1) * (n/2) * (1 - 1/N_1) * \text{Transfer}(1)] \text{ (line 59)} + [\text{Transfer}(n/N_1) - \\
&\quad (1 - 1/N_1) * \text{Transfer}(n/N_1)] \text{ (line 83)} \\
c_6 &= [(n-1) * (1 - 1/N) * \text{Transfer}(1) - (N_1 - 1) * \text{Transfer}(1)] \text{ (line 16)} +
\end{aligned}
$$

```
DO 5 I = 1, N
    DO 3 J = I, N
      Z(J,I) = A(J,I)
    D(I) = A(N,I)
CONTINUE
IF (N .EQ. 1) GO TO 82
DO 63 II = 2, N
    I = N + 2 - II
    L = I - 1
    H = 0.0D0
    SCALE = 0.0D0
    IF (L .LT. 2) GO TO 16
    DO 14 K = 1, L
      SCALE = SCALE + DABS(D(K))
    IF (SCALE .NE. 0.0D0) GO TO 23
    E(I) = D(L)
    DO 21 J = 1, L
        D(J) = Z(L,J)
        Z(I,J) = 0.0D0
        Z(J,I) = 0.0D0
    CONTINUE
    GO TO 62
    DO 25 K = 1, L
        D(K) = D(K) / SCALE
        H = H + D(K) * D(K)
    CONTINUE
    F = D(L)
    G = -DSIGN(DSQRT(H),F)
    E(I) = SCALE * G
    H = H - F * G
    D(L) = F - G
    DO 33 J = 1, L
      E(J) = 0.0D0
    DO 45 J = 1, L
        F = D(J)
        Z(J,I) = F
        G = E(J) + Z(J,J) * F
        JP1 = J + 1
        IF (L .LT. JP1) GO TO 43
        DO 43 K = JP1, L
            G = G + Z(K,J) * D(K)
            E(K) = E(K) + Z(K,J) * F
        CONTINUE
        E(J) = G
    CONTINUE
    F = 0.0D0
    DO 50 J = 1, L
        E(J) = E(J) / H
        F = F + E(J) * D(J)
    CONTINUE
    HH = F / (H + H)
    DO 53 J = 1, L
      E(J) = E(J) - HH * D(J)
    DO 61 J = 1, L
        F = D(J)
        G = E(J)
        DO 58 K = J, L
            Z(K,J) = Z(K,J) - F * E(K) - G * D(K)
        D(J) = Z(L,J)
        Z(I,J) = 0.0D0
    CONTINUE
    D(I) = H
CONTINUE
DO 81 I = 2, N
    L = I - 1
    Z(N,L) = Z(L,L)
    Z(L,L) = 1.0D0
    H = D(I)
    IF (H .EQ. 0.0D0) GO TO 78
    DO 71 K = 1, L
      D(K) = Z(K,I) / H
    DO 78 J = 1, L
        G = 0.0D0
        DO 75 K = 1, L
          G = G + Z(K,I) * Z(K,J)
        DO 78 K = 1, L
          Z(K,J) = Z(K,J) - G * D(K)
    CONTINUE
    DO 80 K = 1, L
      Z(K,I) = 0.0D0
CONTINUE
DO 85 I = 1, N
    D(I) = Z(N,I)
    Z(N,I) = 0.0D0
CONTINUE
Z(N,N) = 1.0D0
E(1) = 0.0D0
END
```

Figure 6.1: Source code of tred2 routine

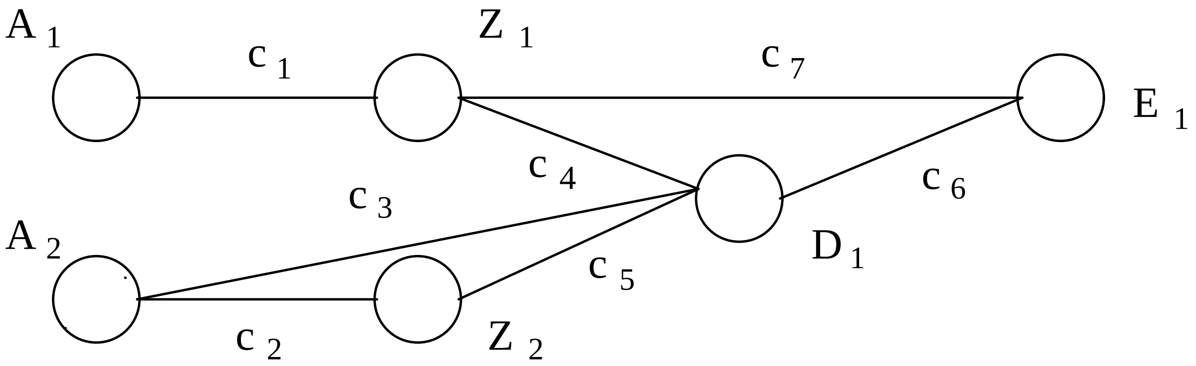

Figure 6.2: Component Affinity Graph for `tred2`

$$
\begin{aligned}
& [(n-1) * \mathrm{Transfer}(n/N_1) - 0] \text{ (line 53)} \\
c_7 \quad = \quad & [(n-1) * (n/2) * \mathrm{Transfer}(n/N_1) - (n-1) * (n/2) * (1 - 1/N_1) * \mathrm{Transfer}(n/N_1)] \\
& \text{(line 42)} + [(n-1) * (\lfloor N_1/4 \rfloor + 1) * \mathrm{Transfer}(n/N_1) - \\
& (n-1) * \mathrm{OneToManyMulticast}(n/N_1, (\lfloor N_1/4 \rfloor + 1))] \text{ (line 58)}
\end{aligned}
$$

Along with each term, we have indicated the line number in the program that leads to the corresponding constraint. The total number of processors is denoted by N. At this stage, the mesh of processors is assumed to be configured as N_1 x N_1 mesh, where $N_1 = \sqrt{N}$.

The CAG for `tred2` shows conflicting requirements on alignment from different parts of the program. For example, the references in statements 58, 71 and 77 favor alignment of D_1 with Z_1, while those in statements 18, 59 and 83 favor alignment of D_1 with Z_2. This clearly illustrates the need for good quality measures to guide the compiler in taking such decisions. In fact, in this case, the best alignment to use may not be immediately obvious to even an experienced programmer. The algorithm for component alignment groups the array dimensions into the following classes – class 1 consisting of A_1, Z_1, D_1, E_1, and class 2 consisting of A_2, Z_2. The array dimensions in these classes are mapped to dimensions 1 and 2 respectively of the processor mesh.

Method of Partitioning There are numerous statements throughout the program that lead to constraints favoring cyclic distribution being recorded for array dimensions mapped to both the mesh dimensions. For example, statement 53 suggests a constraint on cyclic distribution of

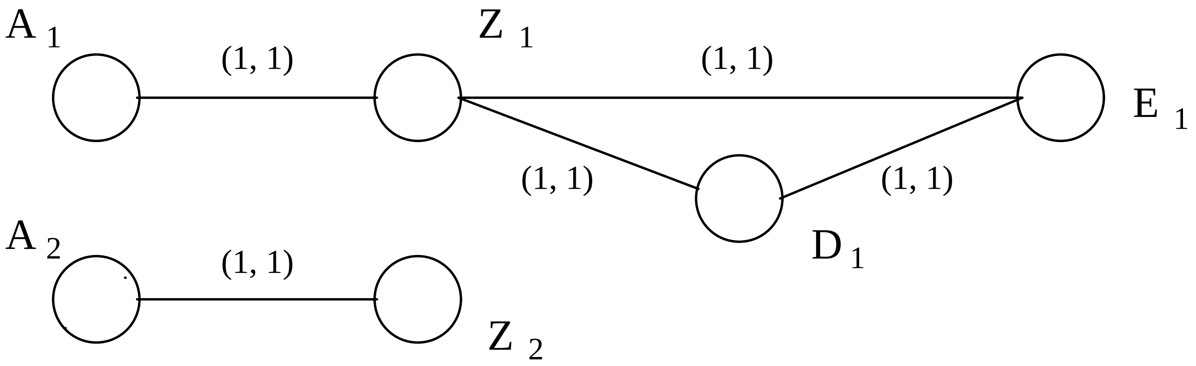

Figure 6.3: Block-size Constraint Graph for `tred2`

E_1, and statement 60 suggests a similar constraint for Z_2. There is only one statement in the program that favors blocked distribution of any array dimension, namely, statement 16. Note that the given assignment $E(i) = D(l)$ on that statement appears as $E(n+2-ii) = D(n+1-ii)$, as a result of the induction variable recognition performed by Parafrase-2 [27]. That enables detection of the constraint favoring blocked distribution of D_1.

All of the arrays in the program belong to the same set of CR-related arrays, since they cross-reference each other in various statements. Hence collective decisions are taken on the method of partitioning for entire classes of aligned array dimensions. For the array dimensions mapped to mesh dimension 1, the sum of quality measures of constraints for cyclic distribution is 316.5 seconds (for $n = 512$, and for $N = 16$). The quality measure of the lone constraint favoring blocked distribution in that case is 0.134 seconds. Therefore all those array dimensions are given cyclic distribution. For array dimensions mapped to mesh dimension 2, the only constraints recorded are those favoring cyclic distribution. Hence all the remaining array dimensions too are given cyclic distribution.

Block Size The Block-size Constraint Graph (BCG) constructed for the program is shown in Figure 6.3. For each edge, we have shown the values of the two coefficients that identify the block-size requirements of the given constraint. It can be seen that all block-size constraints in the program have identical requirements that both the lhs and the **rhs** dimensions be given block sizes of the form $k * 1$. This is due to the fact that every subscript of the type **single-index**, i.e., of the form $\alpha_1 * j_l + \alpha_2$, has the coefficient α_1 equal to +1 or -1. Thus, even though there is a cycle in the BCG, there are no conflicts between any block-size constraints. The algorithm used in the `block-size` pass assigns a block size of 1 to each array dimension.

Number of Processors Since the maximum dimensionality of arrays in the `tred2` program is two, the machine is initially assumed to be configured as a 2-D, N_1 x N_2 mesh, with $N_1 * N_2 = 16$. Therefore, the first step of collapsing extra dimensions for dealing with higher-dimensional meshes is not required. The compiler evaluates the expressions for (the relevant part of) the expected program execution time for different mesh configurations, varying N_1 from 1 to 16, doubling its value in each step. The expected execution time consistently drops with an increase in the value of N_1, its value goes down from 1854.9 seconds for a 1 x 16 mesh to 515.7 seconds for a 16 x 1 mesh. The compiler selects 16 x 1 as the final mesh configuration, for $n = 512$.

The final distribution functions determined for all the array dimensions are:

$$\begin{aligned} f(A_1, i) \quad &= \quad f(Z_1, i) \quad = \quad f(D_1, i) \quad = \quad f(E_1, i) \quad = \quad (i-1) \bmod 16 \\ f(A_2, j) \quad &= \quad f(Z_2, j) \quad = \quad 0 \end{aligned}$$

This corresponds to the distribution of arrays A and Z by *rows* in a *cyclic* manner, and of arrays D and E also in a cyclic manner, on all the 16 processors.

6.2.2 Application 2: JACOBI

The source code of the `Jacobi` program is shown in Figure 6.4. This simplified version repeatedly carries out relaxation computations followed by copying of array elements to store the elements computed in the previous step. The two assignment statements involving arrays suggest identical alignment constraints that lead to A_1 being aligned with B_1, and A_2 being aligned with B_2. The data movement for the first of those statements shows nearest-neighbor communication in the form of Shift operations. The resulting constraints lead to blocked distribution of all the array dimensions. The final step of determining the values of N_1 and N_2, namely, the number of processors in each mesh dimension, shows an interesting variation in results with change in the size of arrays. The contribution of communication costs to the program

```
        parameter (np2 = 514, ncycles = 100)
        double precision A(np2,np2), B(np2,np2)

        np1 = np2 - 1
        do 10 k = 1, ncycles
           do 20 i = 2, np1
              do 30 j = 2, np1
                 B(i,j) = 0.5 * A(i,j) + 0.125 * (A(i-1,j) + A(i+1,j) +
                                                  A(i,j-1) + A(i,j+1))
30            continue
20         continue
           do 40 i = 2, np1
              do 50 j = 2, np1
                 A(i,j) = B(i,j)
50            continue
40         continue
10      continue
```

Figure 6.4: Source code for `Jacobi` program

execution time is estimated by the compiler as:

$$\text{Communication cost} \quad = \quad ncycles * (2 * (N_1 > 1) * \text{Shift}(n/N_2) + 2 * (N_2 > 1) * \text{Shift}(n/N_1))$$

Let us consider two mesh configurations, 1 x 16, and 4 x 4. The first of these corresponds to a column partitioning of the arrays. It leads to two Shift operations being carried out in every cycle, with a data size of n words each. The second configurations corresponds to a "2-D" partitioning of the arrays, where both the rows and the columns are distributed on 4 processors each. That leads to four Shift operations in every cycle, each with a data size of $n/4$ words. For smaller values of n, the first scheme is better, since the start-up costs of sending messages dominate the communication cost. As the data size is increased, the second scheme starts becoming more attractive, since the total amount of data transferred is lower under that scheme. Supplied with different values of n that are doubled in each step, PARADIGM starts choosing the second scheme at $n = 1024$.

6.2.3 Application 3: DGEFA

The source code of the `dgefa` routine is shown in Appendix I. This code is a transformed version of the one appearing in the Linpack library. The calls made to other Linpack routines have been removed by function-inlining, performed by hand. Some of the important loops where Parafrase-2 is currently unable to infer the absence of loop-carried dependence, have been explicitly marked as *doall*. The `dgefa` program uses two arrays, A and $IPVT$, which do not cross-reference each other. In fact, there is no constraint recorded on the distribution of $IPVT$, and it is given the default, blocked distribution. There are constraints favoring cyclic distribution for both dimensions of A, and none suggesting any need for blocked distribution.

The analysis of the program for obtaining the expected execution time shows parallelism in both the mesh dimensions. The first dimension has a relatively greater amount of interprocessor data movement taking place due to the determination of the pivot element along each column, multicasting of that pivot element, and the exchange of rows required if the pivot element belongs to a different row than the one being zeroed in the current step. The two-level loop performing the update of array elements in each step requires a significant amount of communication in both the mesh dimensions. In that step, a section of the row is multicast to all the processors along the first mesh dimension, and a section of the column multicast along the second mesh dimension. This data movement is shown pictorially in Figure 6.5. Both the array dimensions are shown to have blocked distribution purely for the ease of illustration, actually they are given cyclic distributions.

The data partitioning scheme for `dgefa` was obtained for three different data sizes (n taking the values 128, 256 and 512). For $n = 128$, the cost terms corresponding to the "extra" communications in the first mesh dimension lead to the configuration being biased in favor of a smaller number of processors in the first dimension. The mesh configuration chosen for that data size is 2 x 8. However, for larger data sizes, a different effect becomes more dominant. Consider the multicasts being carried out along the second dimension. as shown in Figure 6.5. All of these multicasts, corresponding to different positions along the first dimension, take place in parallel. Given a fixed array size, reducing the number of processors in the first dimension

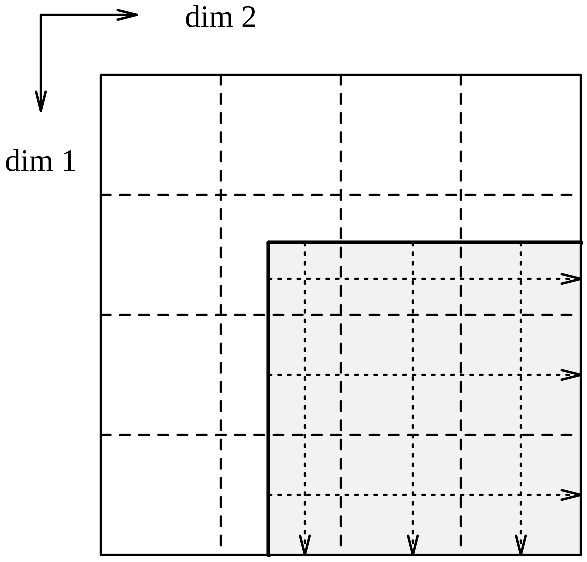

Figure 6.5: Data movement for updates of array elements in each step in `dgefa`

leads to an increase in the cost of this data movement, due to fewer parallel multicasts that now involve a bigger data size. Thus, at larger data sizes, minimizing the sum of costs of multicasts along both the dimensions requires the number of processors to be evenly balanced between both the dimensions. The configuration chosen for both $n = 256$ and $n = 512$ is a 4 x 4 mesh, though in the first case, the expected excution time is only marginally higher than that for a 2 x 8 configuration. In summary, the distribution functions chosen for the array A are:

$$f(A_1, i) = (i-1) \bmod 2, \qquad f(A_2, j) = (j-1) \bmod 8, \qquad \text{for } n = 128$$

$$f(A_1, i) = (i-1) \bmod 4, \qquad f(A_2, j) = (j-1) \bmod 4, \qquad \text{for } n > 128$$

6.2.4 Application 4: OLDA

`Olda` is the dominant routine in the `trfd` program of Perfect Benchmarks. The original routine uses nine arrays, some of which are aliases of each other (due to identical actual arguments being used for the corresponding formal parameters in the only call to `olda` in the program). We have factored in that information by directly modifying the source code, replacing multiple names corresponding to aliased arrays by a single name. The final version of the code as supplied to the compiler is shown in Appendix I. The distribution functions for various arrays are determined by the compiler as follows:

$$\begin{aligned} f(XRSIQ_1, i) &= f(V_2, i) = f(XIJ_1, i) = f(XRSPQ_1, i) = (i-1) \bmod 16 \\ f(XRSIQ_2, j) &= f(V_1, j) = 0 \end{aligned}$$

Some pertinent sections of this program have been presented earlier in Section 2.2, where we introduced the notion of constraints on data distribution. In that discussion, we described some of the references that lead to alignment of $XRSIQ_1$ with V_2, and of XIJ_1 with V_2, We also explained why the dimensions $XRSIQ_1$, V_2, and XIJ_1 are given cyclic distribution.

Here we report on some additional experiences with this program, that have led to improvements in the handling of subscripts of the type `unknown` by PARADIGM. Consider the statement, $XRSPQ(mrsij) = XIJ(mj)$, marked by the label '80' in the program. If the compiler treats $mrsij$ as a subscript of the type `unknown`, the communication primitive obtained for the data movement is ManyToManyMulticast. A similar statement appears in the program, marked by the label '280'. The resulting communication costs for those statements represent a significant part of the program execution time.

With an improved characterization of subscripts (described in Section 3.1), the compiler actually performs a better analysis. The pass for induction variable recognition in Parafrase-2 recognizes the following relationship:

$$mrsij = (-mi + mi * mi + 2 * mj + 1640 * mrs - 1640)/2$$

Hence, the `setup` pass in PARADIGM finds the `lhs` reference with $mrsij$ replaced by the above expression. Note that mj, mi, and mrs are loop indices at levels 3, 2, and 1 respectively. Thus, the subscript is of the form $\alpha_1 * mj + x$, where $\alpha_1 = 1$, and x is an expression with a `variation-level` of 2. The subscript is regarded as being of the type `single-index` in the mj-loop. The communication is placed outside the mj-loop, and the compiler now recognizes the possibility of using Transfers to implement the data movement. That leads to much lower communication costs for the given statement.

Array	*Size*	*Distribution Functions*
FS	193 x 33 x 4	$\lfloor(i-1)/16\rfloor$, 0, 0
W	194 x 34 x 4	$\lfloor(i-1)/16\rfloor$, 0, 0
FW	193 x 33 x 4	$\lfloor(i-1)/16\rfloor$, 0, 0
DW	194 x 34 x 4	$\lfloor(i-1)/16\rfloor$, 0, 0
DP	195 x 35	$\lfloor(i-1)/16\rfloor$, 0
$RADJ$	194 x 34	$\lfloor(i-1)/16\rfloor$, 0
DTL	194 x 34	$\lfloor(i-1)/16\rfloor$, 0
P	194 x 34	$\lfloor(i-1)/16\rfloor$, 0
$RADI$	194 x 34	$\lfloor(i-1)/16\rfloor$, 0
VOL	194 x 34	$\lfloor(i-1)/16\rfloor$, 0
EP	193 x 33	$\lfloor(i-1)/16\rfloor$, 0
$DIS4$	193 x 33	$\lfloor(i-1)/16\rfloor$, 0
$DIS2$	193 x 33	$\lfloor(i-1)/16\rfloor$, 0

Table 6.2: Distribution functions for arrays in `dflux`

6.2.5 Application 5A: DFLUX

The source code of the `dflux` routine, taken from the `flo52` program, is shown in Appendix I. The distribution functions for various arrays in the program, as determined by PARADIGM, are shown in Table 6.2. All of the arrays are distributed by rows in a blocked manner on 16 processors. There are numerous constraints that lead to mutual alignment of the first dimensions of all the arrays, and also of their second dimensions. There are no conflicts between any two alignment constraints. Similarly, all constraints on the method of partitioning favor blocked distribution of the first two dimensions of all the arrays. The step reducing the number of mesh dimensions collapses the one to which the third dimensions of arrays FS, W, FW and DW are mapped. The analysis of the expected execution time for the 2-D mesh shows a number of Shift operations taking place along both the mesh dimensions, at different points in the program. The cost estimates predict that the best performance would be achieved by sequentializing the second dimension (which has fewer elements) of all the arrays. We expect that an increase in the data size and the number of processors in the system would lead to cases where distributing both the dimensions of each array becomes better, as the communication costs begin to get dominated by data transfer times rather than the start-up times for messages.

However, the best mesh configuration would still have fewer processors in the second dimension than the first, as the first dimension of each array has significantly more elements than the second dimension.

6.2.6 Application 5B: EFLUX

`Eflux` is the second routine chosen from the `flo52` program. The distribution functions selected for all the arrays are shown below:

$$\begin{array}{lllllll} f(X_1,i) & = f(W_1,i) & = f(DW_1,i) & = f(FS_1,i) & = f(P_1,i) & = \lfloor (i-1)/16 \rfloor \\ f(X_2,j) & = f(W_2,j) & = f(DW_2,j) & = f(FS_2,j) & = f(P_2,j) & = 0 \\ f(X_3,k) & = f(W_3,k) & = f(DW_3,k) & = f(FS_3,k) & = 0 & \end{array}$$

As can be seen from the program listing, the `eflux` routine performs computations quite similar to those in the `dflux` routine. Again, there are no conflicts seen between any alignment constraints, and all constraints on the method of partitioning favor blocked distribution of array dimensions. The arrays with identical names in the `eflux` and `dflux` routines in fact correspond to the same global arrays in the `flo52` program. The fact that both the routines choose the same distribution functions for them is an encouraging sign for the performance of `flo52`. It means that between those two routines, there would be no re-distribution required for the given arrays.

6.3 Performance Results

In this section, we present results on the evaluation of data partitioning schemes selected for some of the programs by PARADIGM. This involves developing multiple parallel program versions corresponding to different partitioning schemes, and comparing the actual performance of those versions on the iPSC/2.

6.3.1 Application 1: TRED2

We now describe the results obtained for different data-parallel versions of the `tred2` program. Starting with the sequential program, each version was obtained by hand-simulating the compilation process (corresponding to a sophisticated compiler) on that program, under the given data partitioning scheme.

As described earlier, the data distribution scheme selected by PARADIGM is – distribute arrays A and Z by *rows* in a *cyclic* fashion, distribute D and E also in a cyclic manner, on all the N processors. The first version corresponds to this scheme, referred to as *row cyclic*. The presence of conflicts in the CAG for `tred2` suggests that another reasonable scheme would be one where the arrays D and E are aligned with the second, rather than the first dimension of arrays A and Z. If these dimensions are further distributed on all the N processors (to satisfy constraints to sequentialize those dimensions of A and Z not aligned with D and E), we obtain a scheme where A and Z are distributed by columns, again in a cyclic manner. That forms the basis for the second version.

The remaining versions correspond to "bad" choices (according to compiler-generated estimates) that might be made on certain data distribution parameters. The third version is based on a variant of the row-cyclic scheme, where both the rows and the columns of A and Z are distributed on more than one processor. All other characteristics of the first scheme are retained, D and E are still aligned with the first dimension of A and Z, and all dimensions are distributed in a cyclic manner. It is quite possible for a human programmer to choose such a scheme, which we have referred to as *2-D cyclic*. The fourth version is also a variant on the preferred row-cyclic scheme, where the rows of A and Z (and the arrays D and E) are distributed in a blocked, rather than cyclic manner.

The programs were run for two different data sizes corresponding to the values 256 and 512 for n. The plots of performance of various versions of the program are shown in Figures 6.6 and 6.7. The sequential time for the program is not shown for the case $n = 512$, since the program could not be run on a single node due to memory limitations. The data partitioning scheme selected by PARADIGM performs much better than other schemes for that data size, as shown

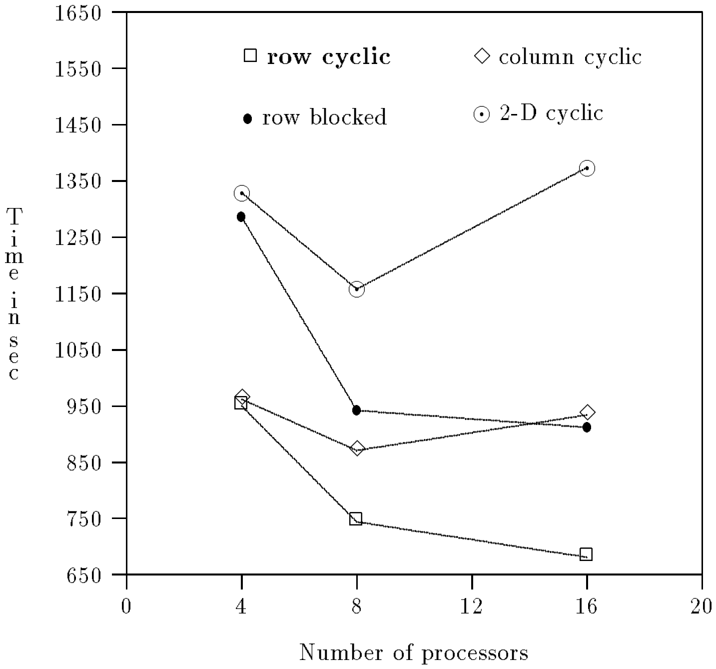

Figure 6.6: Performance of `tred2` on Intel iPSC/2 for data size $n = 512$

in Figure 6.6. For smaller data size (Figure 6.7), and for fewer than 16 processors, the column-cyclic scheme performs slightly better. Based on the estimates generated by PARADIGM, this is not entirely unexpected, given the close conflict between alignment constraints. When all of the 16 processors are being used, the row-cyclic scheme still performs the best.

All other data distribution decisions too are validated by the results. To recall the decision process leading to the partitioning of array dimensions in a cyclic manner, the sum of quality measures of constraints favoring cyclic distribution was about three orders of magnitude higher than that for blocked distribution. That is confirmed by the relatively poor performance of the *row-blocked* scheme. The estimates of program execution times guiding the final selection of mesh configuration showed the 16 x 1 mesh performing much better than the 4 x 4 mesh, where both the dimensions of A and Z were distributed. That decision is again confirmed by the higher execution times obtained by the 2-D cyclic scheme as compared to the row-cyclic scheme.

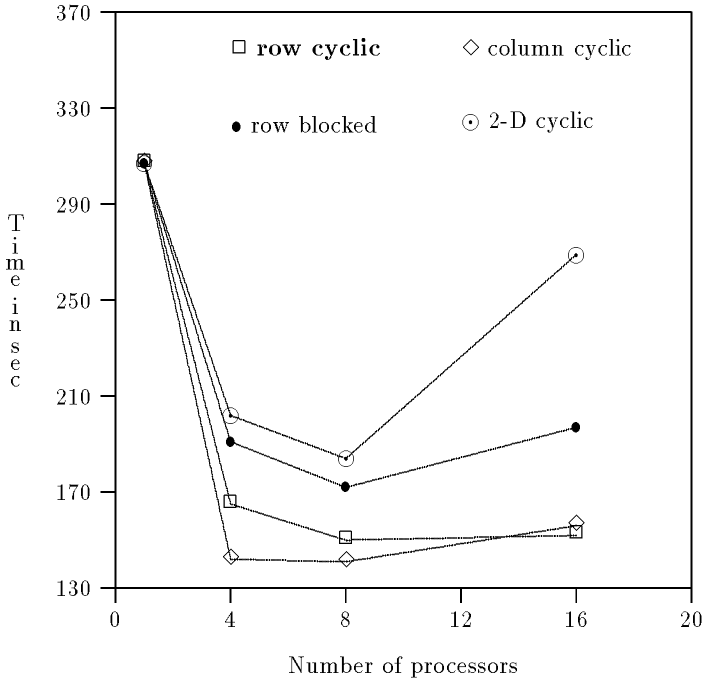

Figure 6.7: Performance of `tred2` on Intel iPSC/2 for data size $n = 256$

6.3.2 Application 2: JACOBI

We now report results on the performance of different versions of the Jacobi program on the iPSC/2. Our compiler selects a column partitioning for smaller data sizes, and a 2-D partitioning (where both rows and columns are distributed on the same number of processors) for larger data sizes. In each case, the compiler chooses a blocked method of partitioning for all distributed array dimensions. The first two versions developed by us correspond to these partitioning schemes. The remaining two versions are based on variants of the above schemes, where the array dimensions are instead distributed in a cyclic manner.

The execution times obtained for each of those versions running on the 16-processor iPSC/2 are shown in Table 6.3. These results confirm that the 2-D partitioning starts performing better than the column partitioning for larger array sizes. The excessive communication requirements resulting from the naive decision to partition the array dimensions in a cyclic manner are reflected in the poor performance of the last two versions. Those versions also require much more space to hold the non-local data received from other processors through collective communi-

Data Size n	*Column Blocked Time (s)*	*2-D Blocked Time (s)*	*Column Cyclic Time (s)*	*2-D Cyclic Time (s)*
64	1.28	1.45	1.66	2.15
128	4.10	4.12	5.82	7.33
256	15.39	14.87	23.29	28.72
512	64.43	59.66	96.64	113.66
1024	257.84	243.24	386.94	

Table 6.3: Performance of different versions of Jacobi on iPSC/2

cation. In fact, the program with 2-D cyclic partitioning could not be run for the data size $n = 1024$ due to memory limitations on each processor.

While these results confirm the compiler's prediction regarding the suitability of the 2-D partitioning at larger data sizes, the actual data size at which that scheme starts performing better than column partitioning is not predicted very accurately. We believe that this difference between the predicted value ($n = 1024$) and the observed value ($n = 256$) is due to the cost function used for the Shift operation being slightly inaccurate. The primary focus of our work in estimating communication costs has been to obtain the estimates in terms of cost functions of various communication primitives (which is performed satisfactorily in this case). Given more accurate performance characteristics of such primitives, obtained by approaches proposed in the literature, such as the "training set" method [7], we believe our compiler would do an even better job of selecting good data partitioning schemes.

6.3.3 Application 3: OLDA

The data partitioning scheme chosen by PARADIGM for the `olda` program has been described in the previous section. The array $XRSIQ$ is distributed by rows in a cyclic manner, V is distributed by columns in a cyclic manner, and the arrays XIJ and $XRSPQ$ are also distributed in a cyclic manner. The first parallel program version we developed corresponds to this partitioning scheme, referred to as *1-D cyclic*. The other version is based on a variant of

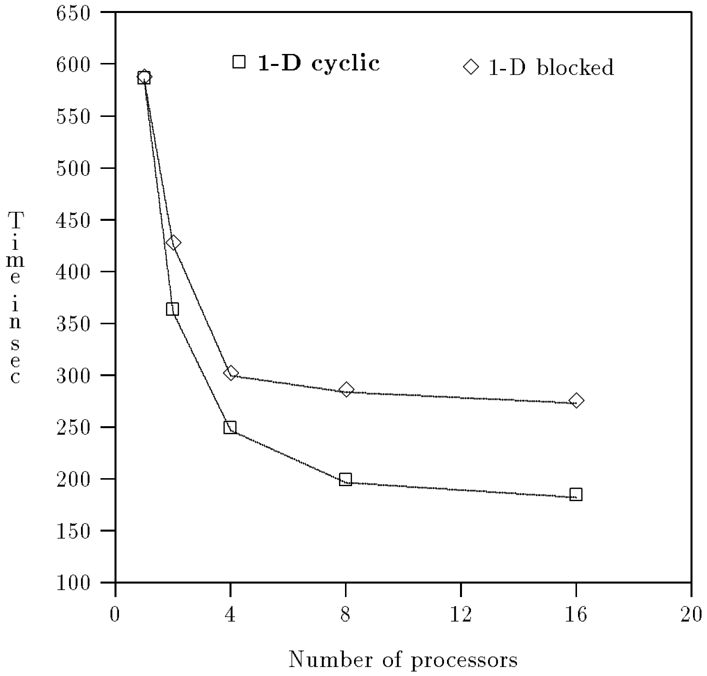

Figure 6.8: Performance of `olda` on Intel iPSC/2

the first scheme, where all the distributed array dimensions are given a blocked distribution instead. This method of partitioning is referred to as *1-D blocked*.

The performance of the two parallel program versions is shown in Figure 6.8. The sizes of the arrays are shown in the source listing of `olda` in Appendix I. The method of partitioning chosen by PARADIGM, 1-D cyclic, leads to a significantly better performance than the other method. This confirms the desirability of the decision taken by PARADIGM to distribute all dimensions in a cyclic manner. There are numerous other methods of data partitioning possible for the `olda` program, such as those in which both the dimensions of $XRSIQ$ and V are distributed on more than one processor, and those in which the alignment between the dimensions of $XRSIQ$ and V is reversed ($XRSIQ_1$ being aligned with V_1, and $XRSIQ_2$ with V_2). A manual inspection of the program shows that each of those schemes would lead to much higher communication overheads, and worse performance. These results show the success of PARADIGM in obtaining a good data partitioning scheme for the `olda` program too.

CHAPTER 7

CONCLUSIONS

In this thesis, we have presented a new approach, the constraint-based approach, to the problem of automatic data partitioning of programs on multicomputers. We have validated these ideas through the development of a compiler called PARADIGM, that takes data partitioning decisions on Fortran 77 programs, to be parallelized and executed on distributed memory machines. Our approach is quite general, and applicable to a large class of programs having references that can be analyzed at compile time.

7.1 Contributions

Our main contributions to the problem of automatic data partitioning are:

- *Analysis of the entire program*: Our approach looks at data distribution from the perspective of performance of the entire program, not just that of some individual program segments. The notion of constraints makes it easier to capture the requirements imposed by different parts of the program on the overall data distribution. Since constraints associated with different statements specify only the relevant aspects of requirements on data distribution, the compiler is often able to combine constraints affecting different parameters relating to the distribution of the same array. Our studies on numeric programs confirm that situations where such a combining is possible arise frequently in real programs.

- *Balance between parallelization and communication considerations*: Both communication *and* computational costs are taken into account during the selection of data partitioning scheme. Each data distribution parameter affecting both components is determined by an algorithm that is driven by the minimization of the overall program execution time.

- *Pruning of the search space*: The distribution functions used for arrays allow for a rich variety of data distributions to be expressed. But, that also leads to a large space of possible data partitioning schemes, that cannot be searched exhaustively for the optimal solution. Our approach is significantly different from all others that have so far been proposed in the extent to which this search space is pruned via heuristics that lead to independent decisions on numerous distribution parameters.

- *General methodology for static performance estimation*: To enable the compiler to be guided by performance estimates in the process of taking data distribution decisions, we have developed a machine-independent methodology for performance estimation. This methodology allows estimation of the extent of data-parallelism exhibited, and also the amount of communication costs incurred by a program with a given data partitioning scheme, *without* actually generating the data-parallel program. Such an analysis can be used not only by a different automatic data partitioning system, but also by a compiler generating the SPMD program, to evaluate the expected benefits of different competing optimizations.

- *Applicability to compiler-directed generation of communication*: As part of estimating the communication costs of a program, the compiler detects opportunities for optimizations in generating communication, like combining messages, and using collective communication primitives. In particular, the techniques we have developed for exploiting collective communication represent a significant advance over other currently known methods, and can contribute to substantial improvements in program performance on massively parallel systems.

- *Results on real-life scientific application programs*: Finally, we observe that the PARADIGM compiler has been successfully used to obtain data partitioning schemes for real-life programs. To the best of our knowledge, these are the first set of results demonstrating the success of automatic data partitioning on a significant class of Fortran programs.

Our approach to data partitioning has its limitations too. The permitted set of data distribution functions is well-suited only to numeric programs with a regular structure. Irregular computations, such as those involving sparse matrices and unstructured grids are not adequately supported. During the process of obtaining quality measures relating to distributions of specific arrays, the compiler often ignores possible optimizations like combining messages corresponding to different arrays. Also, due to the underlying complexity of the problem of estimating performance, the compiler uses a number of simplifying assumptions. For example, it ignores the delays due to congestion in the interconnection network, and does not attempt to characterize the locality of data within a processor, that affects cache performance.

7.2 Future Directions

In this research, we have developed a basic framework for dealing with the problem of automatic data partitioning on multicomputers. There are a number of directions in which this work can be extended, some of which are described below:

- *Interprocedural analysis* : So far, we have used in-line expansion of procedure calls, or restricted ourselves to individual procedures while analyzing real application codes with Paradigm. Clearly, there is a need to develop techniques for interprocedural analysis. A number of researchers have worked on this problem for improving the effectiveness of parallelizing compilers [9, 10, 76, 49, 1]. Those ideas need to be extended to allow determination of constraints and their quality measures across procedure boundaries, and to summarize such information for data accessed in various procedures.

- *Redistribution of data* : Currently, Paradigm assigns a fixed distribution to each array, that remains unchanged during the entire program. For some programs, it may be desirable to partition the data one way for a particular segment, and then repartition it before moving to the next segment. This apparently tougher problem (where the distribution of data is allowed to change) can be mapped to the original problem of obtaining a fixed distribution, through transformations like array renaming. However, techniques need to be developed to apply those transformations, where needed, in an automated manner.

- *Other distribution parameters* : Currently, PARADIGM does not attempt to evaluate the benefits of replicating an array as compared to partitioning it across processors. The option of replication has so far been restricted to scalars and small arrays, i.e., those with sizes less than a certain threshold. For many programs, replicating bigger arrays, particularly the read-only arrays, can lead to considerable savings in communication costs. However, the compiler would have to take such decisions keeping in mind the memory limitations on each processor. Thus, it would be interesting to extend the compile-time analysis to determine which arrays (or array dimensions) to replicate, given a certain amount of memory.

  Another parameter, the *offset* in the distribution of an array dimension is currently fixed, it is given a constant value of one for all the dimensions. In some programs, there are likely to be situations where the desired alignment of elements in two arrays requires different values to be given to offsets in their distributions. We expect that the techniques we have developed to determine the block sizes of array dimensions can be extended to obtain the desirable offset values as well.

- *Irregular problems* : Our research has mainly been directed towards applications with a regular structure, that are amenable to static analysis. Many researchers have started developing systems that provide compiler and runtime support for the task of partitioning computation and generating communication for irregular problems [67, 44, 13, 14]. It would be interesting to explore possibilities of similar support for decisions on data partitioning through extensions of our approach.

  As an example of a simple extension to our approach, consider the problem of various array sizes and loop bounds being unknown at compile time. The symbolic expressions that PARADIGM obtains for the times spent on communication and computation can be stored, and code can be generated to evaluate those expressions at run time, and take decisions on data partitioning accordingly. However, in order to handle irregularities like unknown data access patterns *well*, more sophisticated techniques need to be developed.

BIBLIOGRAPHY

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, 1988.

[2] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. 10th Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[4] F. Andre, J. Pazat, and H. Thomas. Pandore: A system to manage data distribution. In *Proc. 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[5] R. Arlauskas. iPSC/2 system: A second generation hypercube. In *Proc. The 3rd Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, January 1988.

[6] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proc. Fifth Distributed Memory Computing Conference*, April 1990.

[7] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proc. Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Williamsburg, VA, April 1991.

[8] U. Banerjee. Unimodular transformations of double loops. In *Proc. Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, California, August 1990.

[9] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proc. SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, June 1986.

[10] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. *ACM*, pages 152–161, 1986.

[11] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, October 1988.

[12] B. Chapman, H. Herbeck, and H. Zima. Automatic support for data distribution. In *Proc. 6th Distributed Memory Computing Conference*, Portland, Oregon, April 1991.

[13] C. Chase, A. Cheung, A. Reeves, and M. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. In *Proc. 1991 Conference on Parallel Processing*, St. Charles, IL, August 1991.

[14] C. Chase, K. Krowley, J. Saltz, and A. Reeves. Compiler and runtime support for irregularly coupled regular meshes. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[15] M. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 2:171–207, October 1988.

[16] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Yale University, December 1989.

[17] The Perfect Club. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, Fall 1989.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1989.

[19] G. Pfister et al. The IBM Research parallel processor prototype (RP3): introduction and architecture. In *Proc. 1985 International Conference on Parallel Processing*, 1985.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren. Program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[21] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall, 1988.

[22] M. Gerndt. Updating distributed variables in local computations. *Concurrency - Practice & Experience*, 2(3):171–193, September 1990.

[23] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proc. Sixth Distributed Memory Computing Conference*, Portland, Oregon, April 1991.

[24] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proc. 6th International Parallel Processing Symposium*, Beverly Hills, California, March 1992.

[25] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[26] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication on multicomputers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[27] M. R. Haghighat and C. D. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In *Proc. Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[28] W. Hillis and G. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[29] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[30] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proc. Supercomputing '91*, Albuquerque, NM, November 1991.

[31] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[32] C. T. Ho. *Optimal Communication Primitives and Graph Embeddings on Hypercubes.* PhD thesis, Yale University, 1990.

[33] J. M. Hsu and P. Banerjee. A message passing coprocessor for distributed memory multicomputers. In *Proc. Supercomputing 90*, New York, NY, November 1990.

[34] D. E. Hudak and S. G. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proc. 1990 International Conference on Supercomputing*, pages 187–200, Amsterdam, The Netherlands, June 1990.

[35] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proc. Fifth Distributed Memory Computing Conference*, April 1990.

[36] Intel Corporation. *iPSC/2 and iPSC/860 User's Guide*, June 1990.

[37] S. L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4(2), April 1987.

[38] S. L. Johnsson. Performance modeling of distributed memory architectures. *Journal of Parallel and Distributed Computing*, pages 300–312, August 1991.

[39] A. H. Karp. Programming for parallelism. *Computer*, 20(5):43–57, May 1987.

[40] K. Kennedy and U. Kremer. Automatic data alignment and distribution for loosely synchronous problems in an interactive programming environment. Technical Report TR91-155, Rice University, April 1991.

[41] K. Knobe, J. Lukas, and G. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[42] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Proc. Third Symposium on the Frontiers of Massively Parallel Computation*, October 1990.

[43] C. Koelbel and P. Mehrotra. Compiler transformations for non-shared memory machines. In *Proc. 1989 International Conference on Supercomputing*, May 1989.

[44] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[45] D. Kuck, E. Davidson, D. Lawrie, and A. Sameh. Parallel computing today and the Cedar approach. *Science*, 231:967–974, February 1986.

[46] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Proc. Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[47] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proc. Supercomputing '90*, New York, NY, November 1990.

[48] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[49] Z. Li and P. C. Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *ACM SIGPLAN PPEALS*, pages 85–99, 1988.

[50] M. Mace. *Memory Storage Patterns in Parallel Processing*. Kluwer Academic Publishers, Boston, MA, 1987.

[51] NCUBE Corporation. *nCUBE 2 Processor Manual*, 1990.

[52] M. O'Boyle and G. A. Hedayat. A transformational approach to compiling Sisal for distributed memory architectures. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[53] M. O'Boyle and G. A. Hedayat. Data alignment: Transformations to reduce communication on distributed memory architectures. In *Proc. SHPCC'92, The Scalable High Performance Computing Conference*, Williamsburg, April 1992.

[54] M. O'Boyle and G. A. Hedayat. Load balancing of parallel affine loops by unimodular transformations. In *Proc. The European Workshop on Parallel Computing*, Barcelona, Spain, March 1992.

[55] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[56] Parasoft Corporation. *Express User's Manual*, 1989.

[57] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling

programs on multiprocessors. In *Proc. 1989 International Conference on Parallel Processing*, August 1989.

[58] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, pages 1389–1401, 1957.

[59] M.J. Quinn and P. J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7:69–76, September 1990.

[60] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proc. Supercomputing '89*, pages 637–646, November 1989.

[61] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–481, October 1991.

[62] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.

[63] J. Rose and G. Steele Jr. An extended C language for data parallel programming. In *Proc. Second ACM International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[64] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[65] M. Rosing and R. P. Weaver. Mapping data to processors in distributed memory computations. In *Proc. Fifth Distributed Memory Computing Conference*, Charleston, S. Carolina, April 1990.

[66] R. Ruhl and M. Annaratone. Parallelization of Fortran code on distributed-memory parallel processors. In *Proc. 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[67] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[68] V. Sarkar. Determining average program execution times and their variance. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[69] R. Sawdayi, G. Wagenbreth, and J. Williamson. MIMDizer: Functional and data decomposition; creating parallel programs from scratch, transforming existing Fortran programs to parallel. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, 1991.

[70] Z. Shen, Z. Li, and P.-C. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[71] D. G. Socha. An approach to compiling single-point iterative programs for distributed memory computers. In *Proc. Fifth Distributed Memory Computing Conference*, Charleston, S. Carolina, April 1990.

[72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[73] N. Tawbi and P. Feautrier. Processor allocation and loop scheduling on multiprocessor computers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[74] S. Thakkar, P. Gifford, and G. Fielland. The Balance multiprocessor system. *IEEE Micro*, pages 57–69, February 1988.

[75] Thinking Machines Corporation. *The Connection Macine CM-5 Technical Summary*, October 1991.

[76] R. Triolet, F. Irigion, and P. Feautrier. Direct parallelization of call statements. In *Proc. SIGPLAN '86 Symposium on Compiler Construction*, pages 176–185, June 1986.

[77] P. S. Tseng. A parallelizing compiler for distributed-memory parallel computers. In *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[78] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[79] M. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, April 1987.

[80] M. J. Wolfe. More iteration space tiling. In *Proc. Supercomputing 89*, Reno, Nevada, November 1989.

[81] H. Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

APPENDIX I

BENCHMARK PROGRAMS

I.1 DGEFA

```
      PROGRAM DGEFA
      PARAMETER(N=512)
      DOUBLE PRECISION A(N,N)
      INTEGER IPVT(N)
      DOUBLE PRECISION T
      INTEGER J,K,L,NM1,INFO
      INFO = 0
      NM1 = N - 1
      IF (NM1 .LT. 1) GO TO 70
      DO 60 K = 1, NM1
      L = 1
      DMAX = DABS(A(K,K))
      DO 30 I = K+1,N
      IF(DABS(A(I,K)).LE.DMAX) GO TO 30
      L = I
      DMAX = DABS(A(I,K))
30    CONTINUE
      IPVT(K) = L
```

```
      IF (A(L,K) .EQ. 0.0D0) GO TO 40
      IF (L .EQ. K) GO TO 10
      T = A(L,K)
      A(L,K) = A(K,K)
      A(K,K) = T
10    CONTINUE
      DO 15 I = K,N
      A(I,K) = (-1.0D0/A(K,K))*A(I,K)
15    CONTINUE
      CDOALL 80 J = K+1, N
      T = A(L,J)
      A(L,J) = A(K,J)
      A(K,J) = T
      CDOALL 35 I = K+1,N
      A(I,J) = A(I,J) + A(K,J)*A(I,K)
35    CONTINUE
80    CONTINUE
      GO TO 50
40    CONTINUE
      INFO = K
50    CONTINUE
      60  CONTINUE
70    CONTINUE
      IPVT(N) = N
      IF (A(N,N) .EQ. 0.0D0) INFO = N
      STOP
      END
```

I.2 OLDA

```
      PROGRAM OLDA
      PARAMETER(NUM=32,NORB=32,MORB=32,NDIM=32,NUM2=278784)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DOUBLE PRECISION V(NDIM,NDIM)
      DOUBLE PRECISION XRSPQ(NUM2)
      DOUBLE PRECISION XRSIQ(MORB,MORB),XIJ(NUM)
      DATA ZERO /0.0D+00/
      NRS=(NUM*(NUM+1))/2
      NP=NUM
      NQ=NUM
      MRSIJ0=0
      MRSPQ=0
      DO 100 MRS=1,NRS
      DO 10 MQ=1,NQ
      DO 10 MI=1,MORB
10    XRSIQ(MI,MQ)=ZERO
      DO 40 MP=1,NP
      DO 30 MQ=1,MP
      MRSPQ=MRSPQ+1
      VAL=XRSPQ(MRSPQ)
      IF(VAL.EQ.ZERO) GO TO 30
      DO 20 MI=1,MORB
      XRSIQ(MI,MQ)=XRSIQ(MI,MQ)+VAL*V(MP,MI)
      XRSIQ(MI,MP)=XRSIQ(MI,MP)+VAL*V(MQ,MI)
20    CONTINUE
30    CONTINUE
40    CONTINUE
      MRSIJ=MRSIJ0
      DO 90 MI=1,MORB
```

```
      DO 50 MJ=1,MI
50    XIJ(MJ)=ZERO
      DO 70 MQ=1,NQ
      VAL=XRSIQ(MI,MQ)
      IF(VAL.EQ.ZERO) GO TO 70
      DO 60 MJ=1,MI
60    XIJ(MJ)=XIJ(MJ)+VAL*V(MQ,MJ)
70    CONTINUE
      DO 80 MJ=1,MI
      MRSIJ=MRSIJ+1
80    XRSPQ(MRSIJ)=XIJ(MJ)
90    CONTINUE
      MRSIJ0=MRSIJ0+NRS
100   CONTINUE
      NIJ=(MORB*(MORB+1))/2
      MRSIJ=0
      DO 120 MRS=1,NRS
      MIJRS=0
      MAX=MRS
      IF(MAX.GT.NIJ) MAX=NIJ
      DO 110 MIJ=1,MAX
      DUM=XRSPQ(MRSIJ+MIJ)
      XRSPQ(MRSIJ+MIJ)=XRSPQ(MIJRS+MRS)
      XRSPQ(MIJRS+MRS)=DUM
110   MIJRS=MIJRS+NRS
120   MRSIJ=MRSIJ+NRS
      NR=NUM
      NS=NUM
      MIJKL=0
      MIJRS=0
```

```
      MIJ=0
      MLEFT=NRS-NIJ
      DO 300 MI=1,MORB
      DO 300 MJ=1,MI
      MIJ=MIJ+1
      DO 210 MS=1,NS
      DO 210 MK=MI,MORB
210   XRSIQ(MK,MS)=ZERO
      DO 240 MR=1,NR
      DO 230 MS=1,MR
      MIJRS=MIJRS+1
      VAL=XRSPQ(MIJRS)
      IF(VAL.EQ.ZERO) GO TO 230
      DO 220 MK=MI,MORB
      XRSIQ(MK,MS)=XRSIQ(MK,MS)+VAL*V(MR,MK)
      XRSIQ(MK,MR)=XRSIQ(MK,MR)+VAL*V(MS,MK)
220   CONTINUE
230   CONTINUE
240   CONTINUE
      LMIN=MJ
      LMAX=MI
      DO 290 MK=MI,MORB
      DO 250 ML=LMIN,LMAX
250   XIJ(ML)=ZERO
      DO 270 MS=1,NS
      VAL=XRSIQ(MK,MS)
      IF(VAL.EQ.ZERO) GO TO 270
      DO 260 ML=LMIN,LMAX
260   XIJ(ML)=XIJ(ML)+VAL*V(MS,ML)
270   CONTINUE
```

```
      DO 280 ML=LMIN,LMAX
      MIJKL=MIJKL+1
280   XRSPQ(MIJKL)=XIJ(ML)
      LMIN=1
      LMAX=MK+1
290   CONTINUE
      MIJKL=MIJKL+MIJ+MLEFT
300   CONTINUE
      STOP
      END
```

I.3 DFLUX

```
      PROGRAM DFLUX
      PARAMETER (I2=194,J2=34,IL=193,JL=33)
      PARAMETER (I2P1=195,J2P1=35)
      REAL RADI(I2,J2),RADJ(I2,J2),FW(IL,JL,4),RFIL
      REAL FS(IL,J2,4),DW(I2,J2,4)
      INTEGER ISYM
      REAL W(I2,J2,4),P(I2,J2),VOL(I2,J2),DTL(I2,J2)
      REAL DP(I2P1,J2P1),EP(IL,JL),DIS2(IL,JL),DIS4(IL,JL)
      SFIL = 1. -RFIL
      FIS2 = .5*RFIL*VIS2
      FIS4 = RFIL*VIS4/64.
      DO 30 J=2,JL
      DO 10 I=2,IL
      DP(I,J) = ABS((P(I+1,J) -2.*P(I,J) +P(I-1,J))/
     . (P(I+1,J) +2.*P(I,J) +P(I-1,J)))
10    CONTINUE
      IF (ISYM.LT.0) GO TO 11
```

```
      DP(2,J) = 0.
      DP(IL,J) = 0.
11    DP(1,J) = DP(IL,J)
      DP(IL+1,J) = DP(2,J)
      DP(IL+2,J) = DP(3,J)
      DO 12 I=2,IL
      EP(I,J) = MAX(DP(I-1,J),DP(I,J),DP(I+1,J),DP(I+2,J))
12    CONTINUE
      IF (ISYM.GE.0) EP(IL,J) = 0.
      EP(1,J) = EP(IL,J)
      IF (VIS0.LT.0.) GO TO 15
      DO 14 I=1,IL
      FIL = VOL(I+1,J)/DTL(I+1,J) +VOL(I,J)/DTL(I,J)
      DIS2(I,J) = FIL*FIS2*EP(I,J)
      DIS4(I,J) = FIL*FIS4
      DIS4(I,J) = DIM(DIS4(I,J),DIS2(I,J))
14    CONTINUE
      GO TO 17
15    DO 16 I=1,IL
      FIL = RADI(I+1,J) +RADI(I,J)
      DIS2(I,J) = FIL*FIS2*EP(I,J)
      DIS4(I,J) = FIL*FIS4
      DIS4(I,J) = DIM(DIS4(I,J),DIS2(I,J))
16    CONTINUE
17    DO 18 N=1,4
      DO 18 I=1,IL
      FS(I,J,N) = W(I+1,J,N) -W(I,J,N)
18    CONTINUE
      DO 20 I=1,IL
      FS(I,J,4) = FS(I,J,4) +P(I+1,J) -P(I,J)
```

```
20    CONTINUE
      DO 30 N=1,4
      IF (ISYM.LT.0) GO TO 25
      FS(1,J,N) = FS(2,J,N)
      FS(IL,J,N) = FS(IL-1,J,N)
25    FS(I2,J,N) = FS(2,J,N)
      DO 26 I=2,IL
      DW(I,J,N) = FS(I+1,J,N) -2.*FS(I,J,N) +FS(I-1,J,N)
26    CONTINUE
      IF (ISYM.LT.0) GO TO 27
      DW(IL,J,N) = 0.
27    DW(1,J,N) = DW(IL,J,N)
      DO 28 I=1,IL
      FS(I,J,N) = DIS2(I,J)*FS(I,J,N) -DIS4(I,J)*DW(I,J,N)
28    CONTINUE
      DO 30 I=2,IL
      FW(I,J,N) = SFIL*FW(I,J,N) -FS(I,J,N) +FS(I-1,J,N)
30    CONTINUE
      DO 38 J=3,JL
      DO 38 I=2,IL
      DP(I,J) = ABS((P(I,J+1) -2.*P(I,J) +P(I,J-1))/
     . (P(I,J+1) +2.*P(I,J) +P(I,J-1)))
38    CONTINUE
      DO 40 I=2,IL
      DP(I,1) = 0.
      DP(I,2) = 0.
      DP(I,JL+1) = 0.
      DP(I,JL+2) = 0.
40    CONTINUE
      DO 42 J=2,JL
```

```
      DO 42 I=2,IL
      EP(I,J) = MAX(DP(I,J-1),DP(I,J),DP(I,J+1),DP(I,J+2))
42    CONTINUE
      EP(I,1) = EP(I,2)
      IF (VIS0.LT.0.) GO TO 45
      DO 44 J=1,JL
      DO 44 I=2,IL
      FIL = VOL(I,J+1)/DTL(I,J+1) +VOL(I,J)/DTL(I,J)
      DIS2(I,J) = FIL*FIS2*EP(I,J)
      DIS4(I,J) = FIL*FIS4
      DIS4(I,J) = DIM(DIS4(I,J),DIS2(I,J))
44    CONTINUE
      GO TO 47
45    DO 46 J=1,JL
      DO 46 I=2,IL
      FIL = RADJ(I,J+1) +RADJ(I,J)
      DIS2(I,J) = FIL*FIS2*EP(I,J)
      DIS4(I,J) = FIL*FIS4
      DIS4(I,J) = DIM(DIS4(I,J),DIS2(I,J))
46    CONTINUE
47    DO 48 N=1,4
      DO 48 J=2,JL
      DO 48 I=2,IL
      FS(I,J,N) = W(I,J+1,N) -W(I,J,N)
48    CONTINUE
      DO 50 J=2,JL
      DO 50 I=2,IL
      FS(I,J,4) = FS(I,J,4) +P(I,J+1) -P(I,J)
50    CONTINUE
      DO 60 N=1,4
```

```
      DO 52 I=2,IL
      FS(I,1,N) = FS(I,2,N)
      FS(I,JL+1,N) = FS(I,JL,N)
      52  CONTINUE
      DO 54 J=2,JL
      DO 54 I=2,IL
      DW(I,J,N) = FS(I,J+1,N) -2.*FS(I,J,N) +FS(I,J-1,N)
54    CONTINUE
      DO 56 I=2,IL
      DW(I,1,N) = 0.
56    CONTINUE
      DO 58 J=1,JL
      DO 58 I=2,IL
      FS(I,J,N) = DIS2(I,J)*FS(I,J,N) -DIS4(I,J)*DW(I,J,N)
58    CONTINUE
      DO 60 J=2,JL
      DO 60 I=2,IL
      FW(I,J,N) = FW(I,J,N) -FS(I,J,N) +FS(I,J-1,N)
60    CONTINUE
      STOP
      END
```

I.4 EFLUX

```
      SUBROUTINE EFLUX
      PARAMETER(I2=194,J2=J2,IL=193,JL=33)
      REAL DW(I2,J2,4)
      REAL FS(IL,J2,4)
      REAL W(I2,J2,4),P(I2,J2),X(I2,J2,2)
      DO 10 J=2,JL
```

```
      DO 10 I=1,IL
      XY = X(I,J,1) -X(I,J-1,1)
      YY = X(I,J,2) -X(I,J-1,2)
      PA = P(I+1,J) +P(I,J)
      QSP = (YY*W(I+1,J,2) -XY*W(I+1,J,3))/W(I+1,J,1)
      QSM = (YY*W(I,J,2) -XY*W(I,J,3))/W(I,J,1)
      FS(I,J,1) = QSP*W(I+1,J,1) +QSM*W(I,J,1)
      FS(I,J,2) = QSP*W(I+1,J,2) +QSM*W(I,J,2) +YY*PA
      FS(I,J,3) = QSP*W(I+1,J,3) +QSM*W(I,J,3) -XY*PA
      FS(I,J,4) = QSP*(W(I+1,J,4) +P(I+1,J)) +QSM*(W(I,J,4) +P(I,J))
10    CONTINUE
      DO 20 N=1,4
      DO 20 J=2,JL
      DO 20 I=2,IL
      DW(I,J,N) = FS(I,J,N) -FS(I-1,J,N)
20    CONTINUE
      DO 25 I=2,IL
      XX = X(I,1,1) -X(I-1,1,1)
      YX = X(I,1,2) -X(I-1,1,2)
      PA = P(I,2) +P(I,1)
      FS(I,1,1) = 0.
      FS(I,1,2) = -YX*PA
      FS(I,1,3) = XX*PA
      FS(I,1,4) = 0.
25    CONTINUE
      DO 30 J=2,JL
      DO 30 I=2,IL
      XX = X(I,J,1) -X(I-1,J,1)
      YX = X(I,J,2) -X(I-1,J,2)
      PA = P(I,J+1) +P(I,J)
```

```
      QSP = (XX*W(I,J+1,3) -YX*W(I,J+1,2))/W(I,J+1,1)
      QSM = (XX*W(I,J,3) -YX*W(I,J,2))/W(I,J,1)
      FS(I,J,1) = QSP*W(I,J+1,1) +QSM*W(I,J,1)
      FS(I,J,2) = QSP*W(I,J+1,2) +QSM*W(I,J,2) -YX*PA
      FS(I,J,3) = QSP*W(I,J+1,3) +QSM*W(I,J,3) +XX*PA
      FS(I,J,4) = QSP*(W(I,J+1,4) +P(I,J+1)) +QSM*(W(I,J,4) +P(I,J))
30    CONTINUE
      DO 40 N=1,4
      DO 40 J=2,JL
      DO 40 I=2,IL
      DW(I,J,N) = DW(I,J,N) +FS(I,J,N) -FS(I,J-1,N)
40    CONTINUE
      STOP
      END
```

VITA

Manish Gupta received the B.Tech. degree in Computer Science and Engineering from the Indian Institute of Technology, Delhi, in 1987. He received the M.S. degree in Computer and Information Science from the Ohio State University, Columbus, in 1988. He is currently a candidate for the Ph.D. degree in Computer Science at the University of Illinois at Urbana-Champaign.

After completing his doctoral dissertation, Mr. Gupta will take up a position as a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY. His research interests include parallel programming environments and parallel computer architectures.