October 1990

UILU-ENG-90-2248
CRHC-90-14

Center for Reliable and High-Performance Computing

AUTOMATIC DATA PARTITIONING ON DISTRIBUTED MEMORY MULTIPROCESSORS

Manish Gupta
Prithviraj Banerjee

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-90-2248 (CRHC-90-14)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research and NSF
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Washington, DC Arlington, VA 22217 20050
8a. NAME OF FUNDING / SPONSORING ORGANIZATION NSF and Office of Naval Research 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-88-K-0624 NSF MIP 86-57563 PYI
8c. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Washington, DC Arlington, VA 22217 20050	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Automatic Data Partitioning on Distributed Memory Multiprocessors	
12. PERSONAL AUTHOR(S) GUPTA, Manish and P. Banerjee	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 90-10-10 15. PAGE COUNT 40
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	distributed-memory multiprocessors, parallelizing compilers, data distribution, constraints, data reference patterns

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

An important problem facing numerous research projects on parallelizing compilers for distributed memory machines is that of automatically determining a suitable data partitioning scheme for a program. Most of the current projects leave this tedious problem almost entirely to the user. In this paper, we present a novel approach to the problem of automatic data partitioning. We introduce the notion of constraints on data distribution, and show how a parallelizing compiler can infer those constraints by looking at the data reference patterns in the source code of the program. We show how these constraints may be combined by the compiler to obtain a complete and consistent picture of the data distribution scheme, one that offers good performance in terms of the overall execution time. We illustrate our approach on an example routine, TRED2, from the EISPACK library, to demonstrate its applicability to real programs. Finally, we discuss briefly some other approaches that have recently been proposed for this problem, and argue why ours seems to be more general and powerful.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Automatic Data Partitioning on Distributed Memory Multiprocessors*

Manish Gupta and Prithviraj Banerjee

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Avenue
Urbana, IL 61801

Tel: (217) 244-7168
Fax: (217) 244-1764
E-mail: banerjee@crhc.uiuc.edu

Abstract

An important problem facing numerous research projects on parallelizing compilers for distributed memory machines is that of automatically determining a suitable data partitioning scheme for a program. Most of the current projects leave this tedious problem almost entirely to the user. In this paper, we present a novel approach to the problem of automatic data partitioning. We introduce the notion of constraints on data distribution, and show how a parallelizing compiler can infer those constraints by looking at the data reference patterns in the source code of the program. We show how these constraints may be combined by the compiler to obtain a complete and consistent picture of the data distribution scheme, one that offers good performance in terms of the overall execution time. We illustrate our approach on an example routine, TRED2, from the EISPACK library, to demonstrate its applicability to real programs. Finally, we discuss briefly some other approaches that have recently been proposed for this problem, and argue why ours seems to be more general and powerful.

KEYWORDS: distributed-memory multiprocessors, parallelizing compilers, data distribution, constraints, data reference patterns, inter-processor communication.

*This research was supported in part by the Office of Naval Research under Contract N 00014-88-K-0624, and in part by the National Science Foundation under Contract NSF MIP 86-57563 PYI.

1 Introduction

Distributed memory multiprocessors are increasingly being used for providing very high levels of performance for scientific applications. The distributed memory machines offer significant advantages over their shared memory counterparts in terms of cost and scalability, but it is a widely accepted fact that they are much more difficult to program than shared memory machines. One major reason for this difficulty is the absence of a single global address space. As a result, the programmer has to distribute code and data on processors himself, and manage communication among tasks explicitly. Clearly there is a need for parallelizing compilers that would relieve the programmer of this burden. Hence the area of parallelizing compilers for distributed memory machines has seen considerable research activity during the last few years [3, 12, 26, 4, 23, 13].

The current work on parallelizing compilers for distributed memory machines has, by and large, concentrated on automating the generation of messages for communication among processes. In this approach, the compiler accepts a program written in a sequential or an implicitly parallel language, and based on the user-specified partitioning of data, generates a parallel program to be executed on that machine. The parallel program corresponds to the SPMD (single program, multiple-data) [10] model, in which each processor executes the same program but operates on distinct data items. Usually, the source language is extended with some primitives which allow the programmer to specify how various data structures are distributed across the processors. However, the task of determining a good data partitioning scheme can be extremely difficult and tedious. In this paper, we propose a strategy which would instead allow a parallelizing compiler to come up with a suitable data distribution pattern, based on an analysis of the computation structure. We shall use the terms data distribution and data partitioning interchangeably in our discussions.

The distribution of data across processors is of critical importance to the efficiency of the parallel program in a distributed memory system. Since interprocessor communication is much more expensive than computation on processors, it is essential that a processor be able to do 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. Another important consideration for a good data distribution pattern is that it should allow workload to be evenly distributed among processors so that full use is made of the parallelism inherent in the computation. There is often a tradeoff involved in minimizing interprocessor communication and balancing load on processors, and a good scheme for data partitioning must take into account both communication and computation costs governed by the underlying architecture of the machine.

The goal of automatic parallelization of sequential code clearly remains incomplete as long as the user has to think about the above mentioned issues, and come up with a suitable data partitioning scheme. However, most of the existing projects on parallelizing compilers for such machines, till very recently had chosen not

to tackle this problem at the compiler level, since it has been known to be a difficult problem. Mace [17] has shown that the problem of finding optimal data storage patterns for parallel processing, even for 1-d and 2-d arrays, is NP-complete. Another related problem, the component alignment problem has been discussed by Li et al in [15], and shown to be NP-complete. Clearly, any strategy for automatic data partitioning can work well only for applications with a regular computational structure and static dependence patterns which may be determined at compile time. There is, however, a large class of important scientific applications that do satisfy these properties, and such a strategy would be extremely useful for those applications.

In this paper we present a novel approach, which we call the *constraint-based approach*, to the problem of automatic data partitioning. The basic idea in this approach is to analyse each loop in the program, and identify the constraints it imposes on what kind of distribution various data structures being referenced in that loop should have. Associated with each constraint is a goodness measure which estimates the penalty paid in terms of execution time if that constraint is not met by the finally chosen data distribution. Once the constraints associated with all the loops in the program have been recorded, the compiler can try to identify the set of non-conflicting constraints for each data structure, so that the overall savings in execution time for the parallel program get maximized. In this paper, we shall restrict ourselves to the partitioning of arrays. Discussions on the distribution for more complex data structures, such as linked lists, are beyond the scope of this paper. The ideas presented here can be applied to most distributed memory machines, such as the Intel iPSC/2, the NCUBE, and the WARP systolic machine. We use a Fortran-like notation to represent loops in all of our examples, and present results on Fortran programs. However, the ideas developed on the partitioning of arrays can as well be applied to other programming languages, such as C.

The rest of this paper is organized as follows. Section 2 describes the kind of data distribution patterns arrays can have in our scheme. Section 3 introduces the notion of constraints on data distribution and describes the various kinds of constraints possible. In Section 4 we describe how constraints can be identified by examining loops in the source code, and how their goodness measures are estimated. In Section 5 we show how the goodness measures determined for loops are modified to reflect goodness for the entire program. Our strategy for determining the data partitioning scheme is presented in Section 6. Some experimental results are presented in Section 7. We discuss some of the related work being done by other researchers in Section 8. Finally we present conclusions, including a discussion on the significance of our results and ideas on further research, in Section 9.

2 Data Distribution

The target machine we assume conceptually is a multi-dimensional grid of processors. Such a topology can be easily embedded on almost any distributed memory machine. The number of dimensions needed in the grid topology is bounded by the maximum dimension, say D, of any array used in the program. Let N be the total number of processors in the system, and let N_k be the number of processors along the k^{th} dimension of the topology. Clearly, we have

$$N = N_1 * N_2 * \ldots * N_D$$

A processor in such a topology can be 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 the tuple $(p_1, p_2 \ldots p_D)$ and the processor number in the range $0 \ldots N-1$ is established by the scheme which embeds the virtual processor grid topology 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. An X appearing in the i^{th} position in a processor tuple means that the tuple represents a set of N_i processors, each processor in the set corresponding to a different value of p_i varying from 0 to $N_i - 1$. Thus for a $2 * 2$ grid 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 used in the program are assumed to be replicated on all processors. The distribution function for an array of data needs to specify the processor number on which a particular element of the array resides, i.e., the processor which owns that element. For an array with d dimensions, we use d such distribution functions, one for each dimension, to denote how that array is distributed across processors. For our scheme, this is much more convenient than having a single distribution function associated with a multidimensional array. We refer to the k^{th} dimension of an array A as A_k. Each dimension k of the array eventually gets mapped on to a unique dimension k', $1 \leq k' \leq D$, of the processor grid. If $N_{k'}$, the number of processors along the dimension number k' of the grid is one, we say that the array dimension A_k has been *sequentialized*, i.e., all array elements whose subscripts differ only in that dimension are allocated to the same processor. The distribution function for A_k takes as its argument an index i_k and gives the component number k' of the tuple representing the processor number which *owns* the array element $A[-, -, \ldots, i_k, \ldots -]$, where '–' denotes an arbitrary value, and i_k is the index appearing in the k^{th} dimension. The array dimension may either be partitioned, or replicated on the k'^{th} grid dimension. The distribution function is of the form

$$f_A^k(i_k) = \begin{cases} \lfloor \frac{i_k - offset}{block} \rfloor [mod N_{k'}] & \text{if } A_k \text{ is partitioned} \\ X & \text{if } A_k \text{ is replicated} \end{cases}$$

where the square parentheses surrounding the $mod\ N_{k'}$ part indicate that this part is optional. Also, if the dimension is partitioned and not replicated, the number in the processor tuple given by the above function is "wrapped around" (by adding $N_{k'}$) if it gets a negative value. At a higher level, the given formulation of

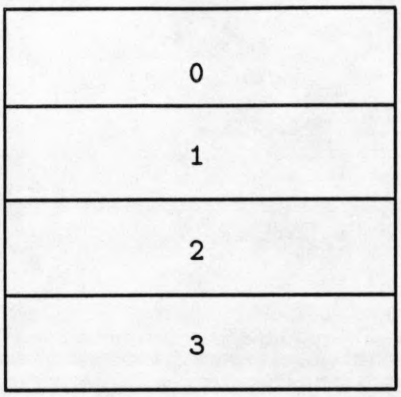
the distribution function can be thought of as specifying the following parameters :

- Whether the array dimension is partitioned across processors or replicated.
- Method of partitioning – contiguous or cyclic. If the *mod* $N_{k'}$ part is present in the expression, it implies a cyclic distribution. Otherwise it means data along that dimension is distributed in a contiguous manner.
- The grid dimension on to which the k^{th} array dimension gets mapped, and the number of processors in that grid dimension. That value is specified by $N_{k'}$ if the distribution is cyclic. In case the distribution is contiguous, the number of processors would be $\lfloor n_k/block \rfloor$, where n_k is the number of elements along the k^{th} dimension that get distributed.
- Block size, viz. the number of elements residing together as a block on a processor, specified by *block*. In case the distribution is contiguous, this parameter indicates the number of elements along the k^{th} dimension that are allocated to a processor. If, however, the distribution is cyclic, this parameter indicates the size of the sub-blocks which get distributed in a cyclic manner on processors.
- The displacement to be applied to the subscript value before mapping it to a processor in a standard way, given by $offset$.

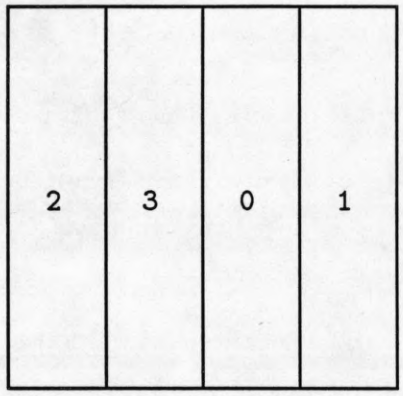
Some examples of different data distribution schemes possible for a $16*16$ array on a 4-processor machine are shown in Figure 1. The numbers shown in the figure indicate the processor(s) to which that part of the array is allocated. The machine is considered to be a $N_1 * N_2$ mesh, the processor number 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)	$N_1 = 2, N_2 = 2:$	$f_A^1(i) = \lfloor \frac{i-1}{8} \rfloor,$	$f_A^2(j) = X$

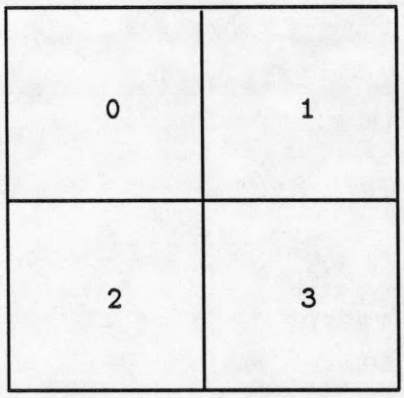
The last example illustrates how our notation allows us to specify *partial* replication of data, i.e., replication of the appropriate part of the array along a specific dimension of the processor grid. An array is replicated completely on all the processors if the distribution function for each of its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



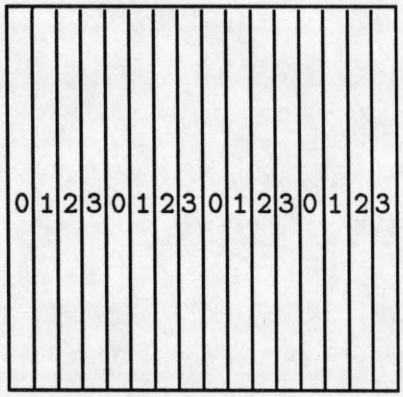
(a)



(b)



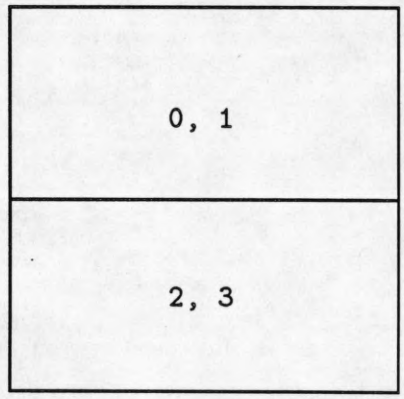
(c)



(d)

0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3

(e)



(f)

t

6

Figure 1: Different data partitions for a $16 * 16$ array

In case the dimensionality of the processor topology, D, is greater than the array dimensionality, d, we need $D - d$ more distribution functions to completely specify on which processor(s) a particular element of the array resides. These functions provide the remaining $D - d$ numbers of the processor tuple. We restrict these "functions" to have just constant values. If the constant corresponding to a grid dimension i takes the special value X, it means that the array is replicated along the i^{th} grid dimension.

We regard an array element as being *truly* replicated if every time its value is modified due to some computation, all processors on which it is replicated carry out that computation. Clearly this definition regards all arrays whose elements are computed only once and then broadcast to all the processors, as being replicated. This is different from the situation where a compiler might *tag* an array element (actually, all elements varying along a certain array dimension) as being "replicated" for a part of the program, so that during the execution of that part, all processors on which that element is "replicated" use their local copy of the element. In the other parts of the program, there is a single processor which owns that element. By our definition, we do not refer to such elements as being replicated, even though we do advocate this technique of using valid local copies of elements received through a broadcast, to eliminate repeated communications of the same value.

Most of the arrays used in real scientific programs, such as routines from LINPACK and EISPACK libraries and most of the Perfect Benchmark programs [5], have fewer than three dimensions. We believe that even for programs with higher dimensional arrays, restricting the number of dimensions that can be distributed across processors to two usually does not lead to any loss of effective parallelism. Consider the loop shown in Figure 2. Even though the loop has parallelism at all three levels, a two-dimensional grid topology in which Z_1 and Z_2 are distributed and Z_3 is sequentialized would give the same performance as a three-dimensional topology with the same number of processors, in which all of Z_1, Z_2, Z_3 are distributed. In fact the former topology has an edge over the latter one with regard to exploiting parallelism and minimizing communication for loops involving two-dimensional arrays. Hence the underlying target topology we shall always assume is a *two-dimensional mesh*. For the sake of notation describing the distribution of an array dimension on a grid dimension, we shall still regard the target topology conceptually as a D-dimensional grid, with the restriction that the values of $N_3, \ldots N_D$ are set to one.

3 Constraints on Data Distribution

The data reference patterns associated with each loop in the program suggest some desirable properties that the final distribution for various arrays should have. We formulate these desirable characteristics as *constraints* on the data distribution functions. In the following discussion, we shall refer to the arrays whose

```
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 2: Fully parallel nested loop

elements are assigned values in a loop are referred to as the left hand side (LHS) arrays, and the ones whose values are used, as the right hand side (RHS) arrays.

Corresponding to a parallelizable loop, there are two kinds of constraints, parallelization constraints and communication constraints. The former kind gives constraints on the distribution of the LHS arrays. The distribution should be such that the array elements which can be written into in parallel reside on different processors, and get distributed evenly on all processors so that there is good balancing of load. The communication constraints associated with a statement inside a loop basically try to ensure that the data elements getting written into and the ones being referenced in a statement, all reside on the same processor, thus internalizing communication wherever possible. They fall into one of the following categories :

- constraints on the *relationship* of distribution of an RHS array with that of a different LHS array.
- constraints on the distribution of an LHS array variable which also appears on the RHS.
- constraints on the distribution of an RHS array to which there are multiple references on the RHS of the assignment statement.

The constraints on the distribution of an array may specify any of the relevant parameters, such as, whether the distribution is contiguous or cyclic, the number of processors, the block size, or the offset. There are two kinds of constraints on the relationship between distribution of arrays. One kind specifies the *alignment* between two dimensions of the two arrays. The alignment of two dimensions means that the distribution functions associated with those dimensions are linked together, i.e., they determine distribution along the same dimension of the processor grid. The other kind of constraint formulates one distribution function in terms of the other for aligned dimensions. For example, the reference pattern shown in Figure 3 suggests that A_1 should be aligned with B_2, and A_2 with B_1. Secondly, it suggests the following distribution functions for B, in terms of those for A.

$$f_B^1(j) = f_A^2(j) \tag{1}$$

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

Figure 3: Example 1 illustrating the relationship between distributions

```
do i = 1, n
    X(a * i + b) = Y(c * i + d)
enddo
```

Figure 4: Example 2 illustrating the relationship between distributions

$$\begin{aligned} f_B^2(3*i) &= f_A^1(i) \\ f_B^2(i) &= f_A^1(\lfloor i/3 \rfloor) \end{aligned} \tag{2}$$

To illustrate a more general case of how relationships between distributions can be determined, consider the loop shown in Figure 4, given that a,b,c,d are integer constants. The constraint implied on the relationship is :

$$\begin{aligned} f_Y^1(c*i+d) &= f_X^1(a*i+b) \\ f_Y^1(i) &= f_X^1(\lfloor \frac{a*i+(b-d)}{c} \rfloor) \end{aligned}$$

Given that we have

$$f_X^1(i) = \lfloor \frac{i - offset}{block} \rfloor [mod N_1],$$

we get

$$f_Y^1(i) = \lfloor \frac{\lfloor \frac{a*i+(b-d)}{c} \rfloor - offset}{block} \rfloor [mod N_1]$$

Thus, given the parameters about distribution of X, such as the block size and offset, we can obtain values for those parameters for Y, and choose the same kind of distribution (contiguous/cyclic) as that of X.

Since different parts of the program are likely to impose conflicting requirements on the distribution of various arrays, we need to associate some goodness measure with each constraint, so that an important constraint can take precedence over a less important one, in case of a conflict. Some constraints, such as,

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

Figure 5: Determining a goodness measure

whether an array along a given dimension is distributed in a cyclic manner on processors, are of the nature that they would either be satisfied or not satisfied by the final data distribution scheme. The goodness associated with such a constraint is an estimate of the penalty in execution time that would have to be paid if that constraint is not honored. Some other constraints, such as those governing the number of processors over which an array getting written into in a parallel loop is distributed, cannot be modelled in this manner. The goodness measure for such a constraint could be viewed as an estimate of the saving in execution time caused by it. However, for such constraints, in actual practice we keep information about the time taken to execute that part of the program, expressed as a simple mathematical function of the number of processors. As we shall see later, for constraints involving the number of processors along different dimensions, we try to come up with a solution directly by optimizing the expression for execution time expressed as a function of the number of processors.

One problem with estimating the penalty in execution time because of a constraint not getting satisfied is that the amount of penalty may depend on the actual distribution of various arrays, which is not known beforehand. In fact the estimate is needed in the first place to help take a decision on the distribution scheme for arrays. We resolve this problem by taking the view that whenever we have alternate constraints to fall back on, the goodness measure of a constraint that is not satisfied represents the *minimum* penalty that has to be paid assuming all array distributions, including those affected by the constraint, have the otherwise most favorable form. Thus we assume that the best of the alternate constraints would be chosen in case that particular constraint is not satisfied. For instance consider the program segment shown in Figure 5. One of the constraints suggested by this loop is to sequentialize B_2. The goodness measure for that is computed using the otherwise best possible distribution, namely, B_2 distributed in a contiguous manner (so that communication occurs only across boundaries of blocks allocated to processors), and A_1 and A_2 aligned with B_1 and B_2 respectively.

4 Estimating the Goodness of Constraints for Loops

The success of our strategy for data partitioning depends greatly on the ability of the compiler to recognize data reference patterns in various parts of the program, and record the constraints implied by those patterns, along with the estimates of their goodness measures. Li et al. [16] have shown how explicit communication can be synthesized by analyzing data reference patterns. In our discussion, we use communication primitives similar to the ones they use. The ideas on these primitives have come from a number of researchers such as Fox et al. [6], Ho [7] among others, and these can be implemented efficiently on most distributed memory machines, such as the iPSC/2. In this paper, we are concerned more with the cost of those primitives rather than showing the source and destination processor numbers for them. We shall use the following functions to represent the costs associated with the underlying communication primitives.

- $Transfer(m)$: cost of sending a message of size m words from a single source processor to a single destination processor.
- $OneToAllMulticast(m, p)$: cost of multicasting along a dimension of the processor grid, a message of size m words to the p other processors.
- $OneToAllBroadcast(m, N)$: cost of broadcasting to all the N processors, a message of size m words.
- $UniReduction(m, p)$: cost of reducing (in the sense of the APL *reduction* operator) data of size m words, using a simple associative operator, over p processors lying on the same grid dimension.
- $MultiReduction(m, N)$: cost of reducing data of size m words, using a simple associative operator, over all the N processors,
- $AllToAllMulticast(m, p)$: cost of replicating m words of data each from all the p processors on a grid dimension on to themselves.
- $AllToAllBroadcast(m, p)$: cost of replicating m words of data each from all the N processors on to themselves.

The complexities of these functions on the architectures with the hypercube and mesh topologies are shown in Table 1. A parallelizing compiler written for a specific machine needs to know the actual timing figures for operations carrying out these primitives on that machine.

The estimates of communication costs generated by the compiler are based on certain simplifying assumptions. We ignore the effects of network contention. When parallel communications occur between different pairs of processors, we assume they can go on independently without any conflict. We also assume that the

Primitive	*Cost*	
	Hypercube	*Mesh*
Transfer(m)	O(m)	O(m)
OneToAllMulticast(m,p)	O($m \log p$)	O($m\ p$)
OneToAllBroadcast(m,N)	O($m \log N$)	O($m \sqrt{N}$)
UniReduction(m,p)	O($m \log p$)	O($m\ p$)
MultiReduction(m,N)	O($m \log N$)	O($m \sqrt{N}$)
AllToAllMulticast(m,p)	O($m\ p$)	O($m\ p$)
AllToAllBroadcast(m,N)	O($m\ N$)	O($m\ N$)

Table 1: Cost of Communication Primitives for Different Architectures

message transmission time is independent of the number of hops the message has to travel. This assumption is somewhat justifiable for the current generation of distributed memory machines, which use wormhole or cut-through routing and special purpose router chips [18]. However, for a more accurate analysis, we do need to take the traffic patterns into consideration. Another simplification we make while estimating communication costs is to ignore the overlap possible between computation and communication.

We now illustrate how various patterns in the source code for loops can be analyzed to determine what constraints they imply and the appropriate time or goodness measures. We ignore statements not inside loops since they are not expected to contribute significantly to the program execution time. Most of the time we restrict ourselves to loops in which the subscripts in the array references are linear functions of the loop indices. Moreover, each subscript should be a function of no more than one loop index. For example, each subscript expression for a two-dimensional array should be of the form $c_1 * i + c_2$, or $c_3 * j + c_4$, where $c_1 \ldots c_4$ are constants and i, j are loop indices. These restrictions cover most of the instances of array references seen in real programs.

We present the patterns in the following manner. First we mention the general properties on the basis of which the pattern is identified in the source code. Each pattern is then illustrated with a typical example for which we show the constraints implied. The data reference patterns in the examples presented here are not contrived ones, most of them have been taken from real scientific applications programs, such as the Perfect Benchmark programs. For each example loop, we first indicate the range of the loop index (indices). The increment for each index is assumed to be 1, unless otherwise stated. The number of loop indices mentioned indicates the number of levels of nesting, with the indices appearing on the right corresponding to the inner loops. Following that, we indicate the forms of array reference on the LHS and of the pertinent array references on the RHS of an assignment statement. Finally we indicate the constraints that can be inferred. As mentioned earlier, for constraints that can be modelled as being satisfied or not satisfied, we indicate their goodness measures. For other constraints, namely the ones suggesting that an array dimension

be partitioned over processors so that speedups may be obtained, we indicate the time measure as a function of the number of processors.

In cases where the pattern to be identified itself appears inside another loop in the source code, the goodness measures for the constraints need to be modified appropriately using the rules given in the next section. For the sake of clarity and ease of presentation, our examples in the following discussion involve only one-dimensional and two-dimensional arrays. Unless otherwise stated, each one-dimensional array in our example is assumed to have n_1 elements, and each two-dimensional array assumed to have $n_1 * n_2$ elements. In all cases, it is easy to see how these results extend to patterns involving higher-dimensional arrays. The processor topology considered is a two-dimensional grid with $N_1 * N_2$ processors. Since the decision regarding the grid dimension to which a particular array dimension gets mapped on is taken at a later stage, we refer to the two grid dimensions simply as I and J. To estimate the time taken to execute a loop sequentially, the compiler counts the number of operations executed in that loop. This time could be parameterized in terms of n_1, n_2, in case their values are not known at compile time. We shall use the function C_p to denote for each pattern, the estimate for time taken to execute it sequentially. The patterns presented here have been categorized according to the kind of constraints they represent, the parallelization constraints and the three kinds of communication constraints discussed earlier.

4.1 Parallelization

1. Completely parallel loop nested at D levels, each loop index used to reference all elements along a different array dimension.

   $1 \leq i \leq n_1, 1 \leq j \leq n_2 : A(i,j) = \ldots$

   Constraints : None, distribute A on N processors in ***any*** manner.

   Regardless of whether the distribution is contiguous or cyclic, and how many processors each array dimension gets mapped on to (as long as the product of those numbers is N), we get the same speedup of N over the sequential time.

2. Completely parallel loop with fewer levels of nesting than the number of dimensions, each index used to reference all elements along a different array dimension.

   $1 \leq i \leq n_1 : A(i,j) = \ldots$

   Constraints : Distribute A_1 on N_I processors.

   Each of the N_I processors can execute its part of the loop in parallel. Since this constraint cannot be modelled as being satisfied or not satisfied, we simply record the expected execution time for the loop.

$$Time = \frac{C_p(n_1)}{N_I}$$

3. Multiply nested parallel loop in which the extent of variation of the index in an inner loop varies with the value of the index in an outer loop.

   $1 \leq i \leq n_1, i+1 \leq j \leq n_2 : A(i,j) = \ldots$

   Constraints : Distribute A_1 and A_2 in a cyclic manner.

   Here we assume, based on a simplistic analysis, that if A_1 and A_2 are distributed in a cyclic manner, we would obtain a speedup of nearly N, otherwise the imbalance caused by contiguous distribution would lead to the effective speedup decreasing by a factor of two. The goodness measure for the constraint is the execution time penalty for having contiguous rather than cyclic distribution.

$$\begin{aligned} Goodness &= \frac{C_p(n_1,n_2)}{N/2} - \frac{C_p(n_1,n_2)}{N} \\ &= \frac{C_p(n_1,n_2)}{N} \end{aligned}$$

   A similar constraint applies to all other patterns for parallel loops in which the bounds for the inner loop themselves vary in the outer loop. From now on, we shall not present those cases explicitly as separate patterns.

4. Reduction operation across a loop.

   $1 \leq i \leq n_1 : s = s \oplus D(i)$

   $\oplus$ represents a simple associative operator.

   Constraints : Distribute D_1 on N_I processors.

   The N_I processors perform the operation on their individual data in parallel, then the result is combined across processors.

$$Time = C_p(n_1)/N_I + UniReduction(1, N_I)$$

   In place of a one-dimensional array, we could have a particular dimension of a multi-dimensional array. For example, $D(i)$ could be replaced by $A(i,k)$ in the reference pattern, the timing estimate would remain the same.

4.2 Data transfer between different arrays

1. Parallel loop in which assignments to one array need the values of another array, and the subscript expressions for referencing one array are *linear functions of* simple permutations of those for the other.

   $1 \leq j \leq n_2, 1 \leq i \leq n_1 : A(i,j) = \mathcal{F}(B(3*i-1, j+1))$

   Constraints : Align A_1 with B_1, A_2 with B_2, and ensure that the distributions of the aligned dimensions are related in the following manner :

$$f_B^1(3*i-1) = f_A^1(i)$$

$$f_B^1(i) = f_A^1(\lfloor \frac{i+1}{3} \rfloor) \quad (1)$$

$$f_B^2(j+1) = f_A^2(j)$$

$$f_B^2(j) = f_A^2(j-1) \quad (2)$$

If the dimensions mentioned above are not aligned, or the given relationships not satisfied by the distributions, we assume that the value of an array element of B residing on a processor may be needed by any other processor. Hence all the $n_1 * n_2/N$ elements held by each processor are broadcast to all other processors.

$$Goodness = AllToAllBroadcast(n_1 * n_2/N, N)$$

2. Same as the previous pattern, except that the two arrays have different number of dimensions.
   $1 \leq i \leq n_1 : A(i,j) = \mathcal{F}(D(i))$
   Constraints :

   - Align A_1 with D_1.
     If the dimensions indicated above are not aligned, the elements of D held by each of the N_I processors have to be sent to all the processors in the grid dimension, say J, on which A_1 is distributed.

     $$Goodness = N_I * OneToAllMulticast(n_1/N_I, N_J)$$

   - Sequentialize A_2.
     If A_2 is distributed on $N_J > 1$ processors, unless we specifically make sure that we have $f_D^2(i) = f_A^2(j)$ (a constant), the given equality will hold only with a probability of $1/N_J$, since $f_D^2(i)$ could take any value from 0 to $N_J - 1$. Hence with a probability of $1 - 1/N_J$, each D element held by a processor has to be sent to a different processor.

     $$Goodness = \frac{N_J - 1}{N_J} * Transfer(n_1/N_I)$$

4.3 Data transfer within the same array

1. Non-parallelizable loop with regular dependence along the dimension referenced with the loop index.
   $1 \leq i \leq n_1 : D(i) = \mathcal{F}(D(i-1))$
   Constraints :

   - Sequentialize D_1.
     If D_1 is distributed on $N_I > 1$ processors, communication takes place between those processors.

The best possible scenario now is that D_1 is distributed in a contiguous manner, in which case the communications occur only across the boundaries of blocks allocated to each processor.

$$Goodness \quad = \quad (N_I - 1) * Transfer(1)$$

- Distribute D_1 in a contiguous manner.

  If D_1 is distributed in a cyclic manner so that successive elements lie on different processors, each iteration would require a communication. If the block size chosen with cyclic distribution is greater than one, the number of communications would be proportionately smaller. However, to keep the expression for goodness measure simple, we always assume a block size of one initially for cyclic distributions.

$$Goodness \quad = \quad (n - 1) * Transfer(1)$$

2. Loop with irregular dependence patterns along one of the dimensions referenced with the loop index.

   $1 \leq j \leq n_2, 1 \leq i \leq n_1 : A(i,j) = \mathcal{F}(A(D(i),j))$

   Constraints :

   - Sequentialize A_1.

     If A_1 is distributed on $N_I > 1$ processors, any array A element held by a processor may need to be sent to any of the processors along the I^{th} dimension of the grid.

$$Goodness = AllToAllMulticast(n/N_I, N_I)$$

   - Partition A_2 on N_J processors.

     The speedup obtained by executing the j-loop in parallel varies linearly with the number of processors on the J^{th} grid dimension.

$$Time = C_p(n_1, n_2)/N_J$$

3. Parallel loop with data transfer within the array along the dimension(s) not being referenced with the loop index (indices).

   $1 \leq i \leq n_1 : A(i, c_1) = \mathcal{F}(A(i, c_2))$, c_1, c_2 are constants.

   Constraints : Sequentialize A_2.

   Sequentializing A_2 is desirable so that communication between processors holding the values of columns c_1 and c_2 gets internalized. For simplicity we assume that distribution of A_2 on N_J processors results in columns c_1 and c_2 getting assigned to different processors with a probability of $(1 - 1/N_J)$.

$$Goodness \quad = \quad \frac{N_J - 1}{N_J} * Transfer(n_1/N_I)$$

4.4 Multiple references to the same array

1. Stencil based computation in a parallel loop.

   We present here an example of a five-point stencil. Since such computations are widely used in scientific applications, such as those involving solutions of partial differential equations, we may treat them as special patterns, and formulate special constraints associated with each different stencil. The performance analysis for some of the stencils is presented in [22], and some researchers [24, 9] have also addressed the problem of automatic data partitioning for such computations.

   $2 \leq j \leq n_2 - 1, 2 \leq i \leq n_1 - 1 : A(i,j) = \mathcal{F}(B(i-1,j), B(i,j-1), B(i+1,j), B(i,j+1))$

   Constraints :

   - Align A_1 with B_1, A_2 with B_2.

     As seen earlier, values of B held by each processor need to be broadcast if the indicated dimensions are not aligned.

     $$Goodness = AllToAllBroadcast(n_1 * n_2/N, N)$$

   - Sequentialize B_1.

     Each processor needs to get elements on the "boundary" columns of the two "adjacent" processors. Given that the above constraint is not satisfied, the best case assumption we make is that B_1 is distributed in a contiguous manner, B_2 is sequentialized, and the proper alignment of array dimensions has been done.

     $$Goodness = 2 * Transfer(n_2/N_J)$$

   - Sequentialize B_2.

     This case is analogous to the previous case, except that here we assume that B_1 is sequentialized and B_2 is distributed in a contiguous manner.

     $$Goodness = 2 * Transfer(n_1/N_I)$$

   - Distribute B_1 in a contiguous manner.

     If B_1 is distributed cyclically, assuming the best case now that B_2 is sequentialized, each processor needs to communicate all of its B elements to the two neighboring processors.

     $$Goodness = 2 * Transfer(n_1 * n_2/N)$$

   - Distribute B_2 in a contiguous manner.

     The analysis is similar to that done for the previous case, this time we assume that B_1 has been sequentialized.

     $$Goodness = 2 * Transfer(n_1 * n_2/N)$$

```
do i = 2, n
    A(i) = A(i) + c * B(i)
enddo
```

Figure 6: Different code patterns

2. Parallel loop referencing multiple, contiguous array elements, such that each array element gets accessed only in one iteration.

   $1 \leq i \leq n_1$, k has a value of 1 initially : $D(i) = \mathcal{F}(E(k), E(k+1), E(k+2))$; $k = k + 3$

   Constraints :

   - Align D_1 with E_1.

     Given that E_1 has been distributed on the I^{th} grid dimension, and D_1 on the J^{th} one, all the E values held by a processor may need to be multicast to processors on the J^{th} grid dimension if the above array dimensions are not aligned.

     $$Goodness = N_I * OneToAllMulticast(3 * n_1/N_I, N_J)$$

   - Block size for $E_1 = 3*$ Block size for D_1. More formally, we have :

     $$f_D^1(i-1) = f_E^1(\lfloor \frac{i-1}{3} \rfloor)$$

     If the above ratio of block sizes is not maintained, in the most favorable case that D_1 and E_1 are aligned, have the same block size, and D_1 has been distributed in a contiguous fashion, some of the processors computing the D values would need values of E from three other processors.

     $$Goodness = 3 * Transfer(n_1/N_I)$$

3. Parallel loop with the subscripts differing in multiple references to an array being independent of the loop index (indices).

   $1 \leq i \leq n_1 : \ldots = \mathcal{F}(A(i, c_1), A(i, c_2))$, c_1, c_2 are constants.

   Constraints : Same as those for Pattern 3 of Section 4.3.

Each pattern captures the significance of a loop with respect to the distribution of a single array or the relationship between distributions of two different arrays. A single assignment statement referencing multiple arrays in a loop would therefore correspond to more than one source pattern. For example, the loop shown in Figure 6 matches the form of two patterns, the Pattern 2 of Section 4.1 and the Pattern 1 of Section 4.2.

It can be readily seen that all the patterns described in this section have a simple enough form, so that they can be detected using already known techniques [14, 1, 19, 25, 20] that are available to the state-of-the-art parallelizing compilers. Even though almost all the patterns presented here have just a single assignment statement inside their loop bodies, effectively they do cover cases where loops have multiple statements. In many cases, simple transformations such as loop distribution and forward substitution [19] are needed so that the pattern(s) underlying the loop structure may be revealed. An important case when these transformations do not help, or are not applicable, is that of an inherently sequential loop, with statements having the same form as that of statements in our parallel loop patterns. All the constraints that attempt to internalize communication in case of parallel loops are equally applicable to this case. The only change is that the goodness measures of these constraints are different. They are evaluated under the assumption that all communications involved in executing that statement (because of some constraints not getting satisfied) get *repeated* as separate communications for each loop iteration, as opposed to getting *combined* when the loop is parallel. We shall illustrate all these cases for loops appearing in *real* programs, in Section 7, where we present results on TRED2, a routine from the EISPACK library.

Admittedly, the code patterns discussed here, along with their obvious generalizations, do not take into account all possible ways in which programs are written, but do cover most of the patterns occurring in real scientific programs. For parts of the programs where not much information is available about the data references at compile-time, or no match can be found to a known pattern, the compiler can either ignore those segments of the program, or generate estimates based on some worst case scenario. For instance, in the example given for Pattern 2 of Section 4.4, the compiler assumes that the references to $A(D(i), j)$ lead to irregular dependence patterns among the iterations over index i of the loop.

5 Determining Goodness Measures for the Entire Program

The basic idea in estimating the goodness measures of constraints for the entire program is to weigh each measure indicated by a specific code pattern by the number of times the program segment corresponding to that pattern is expected to execute. Considering each loop corresponding to a pattern as a single entity, we try to identify constraints for the entire program by looking at the program as having been composed of those entities. This can be done by looking at the control flow graph of the program and treating loops corresponding to our basic entities as the basic nodes. The composition of these entities may have been done through one of the following ways – sequencing, conditional execution, or looping. In this paper, we do not deal directly with the problem caused by procedure calls. We assume that each called procedure has already been expanded in-line when the program is analyzed.

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

Figure 7: Change of communication primitive

When a loop follows another one in sequence, we add together the constraints implied by each loop. When a loop appears inside a branch of a conditional statement, the goodness and time measures for various constraints are multiplied by the probability of executing that branch. Exploring the techniques a compiler may use for estimating these probabilities is beyond the scope of this paper. When the loop pattern itself appears inside another loop, the time estimates associated with that pattern can simply be multiplied by the number of iterations of the outer loop the processor doing that computation is expected to execute. While the above mentioned rules can be applied for the computation time portion of the goodness or time measures of various constraints, in case of a sequence of loops or a loop appearing within another loop, the costs associated with communication might get modified in a number of ways, as explained below. We present three cases of a (loop) pattern occurring inside another loop, one in which the communication primitive used changes from *transfer* to *multicast* (or from *multicast* to *broadcast*), another in which the non-local data received from a message can be reused, and finally one in which separate messages are required for each loop iteration.

The first case is illustrated by the program segment shown in Figure 7. The inner loop matches the form of Pattern 2 of Section 4.2. The goodness measure for the constraint that A_2 be sequentialized is $\frac{N_J-1}{N_J} * Transfer(n/N_I)$, as explained earlier. When the value of j itself varies in the outer loop, the D values held by each processor now need to be sent to all the N_J processors along the grid dimension J on which A_2 is distributed. Hence the goodness measure changes from $\frac{N_J-1}{N_J} * Transfer(n/N_I)$ for the inner loop to $OneToAllMulticast(n/N_I, N_J)$ for the outer loop.

The example shown in Figure 8 illustrates the second and the third cases. The inner loop again matches the form of Pattern 2 of Section 4.2, though the form of the example given for that pattern is different. The constraint that the only dimension of array D be aligned with the first dimension of A has a goodness measure which is the cost of multicasting $A(i, k)$ values to the processors owning the $D(i)$ values. First let us consider the case in which the statements in the outer loop following the end of the inner loop do not modify any element in the k^{th} column of the array A. Once the $A(i, k)$ values have been received, they can get reused in different iterations of the outer loop. Thus the communication costs get amortized over the

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

Figure 8: Reuse of non-local data/Repeated communications

```
do j = 1, n
    do i = 1, n
        A(i, j) = F(D(i))
    enddo
enddo
do i = 1, n
    E(i) = F(D(i))
enddo
```

Figure 9: Identical communication patterns in different loops

larger program segment, and the goodness measure remains the same for the outer loop as it was for the inner loop.

Considering the program segment shown in Figure 8 again, let us now consider the case in which the statements preceeding the second *enddo* do modify the k^{th} column of the array A. In that case the cost of communicating the $A(i, k)$ values (if the relevant dimensions are not aligned) has to be incurred for each iteration of the outer loop. Hence that particular goodness measure determined for the inner loop gets multiplied by n when applied to the complete program segment.

Combining communication costs contributed by a sequence of different loops can be quite complicated. Sometimes, different constraints which are not satisfied may lead to identical communication patterns. For instance, consider the program segment shown in Figure 9. If the array D is broadcast to all processors because of D_1 and A_1 not being aligned, in the second loop no further communication occurs even if E_1 and D_1 are not aligned. In our strategy, we take such potential savings into account only at the stage of estimating communication costs once certain basic decisions related to data distribution, such as alignment of dimensions, have been taken. We do not make the goodness measures for constraints dependent on whether some other constraints would be finally satisfied, since that would blow up the complexity of our problem.

6 Strategy for Data Partitioning

The basic idea in our strategy is to consider all constraints on distribution of various arrays suggested by the important segments of the program, and finally obtain a complete and consistent picture of the desirable data distribution. Our objective is to minimize execution time of the program. Each constraint leads to some savings in execution time because of parallelization or internalization of communication. However, given a set of constraints, not all of them may be consistent with each other. Hence we need to identify the maximal set of non-conflicting constraints such that the sum total of goodness measures of those constraints gets maximized. Since this problem is known to be computationally intractable, we look for approximate solutions.

The goodness measures for various constraints are often expressed in terms of n_i (the number of elements along that dimension), and N_I (the number of processors along the corresponding grid dimension). In order to be able to compare them numerically, we need estimates on the values of n_i and N_I. The value of n_i may either be supplied by the user in an interactive session or through an assertion, or it may be estimated by the compiler on the basis of the array declaration seen in the program. Regarding the value of N_I, we face a familiar problem. The value is needed to help determine the distribution scheme, and becomes known only after the distribution scheme is determined. However, we make a simplifying assumption for the initial few steps that each N_I has a value equal to $\sqrt{N}$ (N being the total number of processors). Eventually we want to distribute all data on a two-dimensional grid, and if the I^{th} dimension does not get sequentialized, the number of processors along that dimension, in the absence of any preference given to a particular grid dimension would be $\sqrt{N}$. However, once enough decisions on data distribution have been taken so that it is clear which constraints are satisfied and which are not, we obtain expressions for execution time in terms of various N_I, and determine their actual values so that the execution time is minimized.

We assume that apart from scalar variables, the small arrays, such as those with less elements than the number of processors available, are replicated on all the processors. Typically we find almost all elements of these arrays being used by different processors, and the communication overhead saved by replicating them is much greater than the savings made by exploiting any parallelism in computing their values.

Our strategy for determining the data distribution scheme consists of the steps given below. The first step collects information about the program. Each of the remaining steps involves taking decisions about some aspect of the data distribution. In this manner, we keep building upon the partial information describing the data partitioning scheme until the complete picture emerges. Such an approach fits in naturally with our idea of using constraints on distributions, since each constraint can itself be looked upon as giving a partial description of what the data distribution should look like. All the steps presented here are simple enough to

be automated. Hence the "we" in our discussions really refers to the parallelizing compiler.

1. *Record the constraints and their goodness measures* : We examine each loop in the program and match it to some code pattern discussed in Section 4. The implied constraints along with the goodness measures are recorded for each array that is involved. For every array used in the program, we maintain a data structure that records constraints associated with each dimension in the form of a list. When different program segments lead to identical constraints, the goodness measures for those constraints are simply added up in the data structure carrying that information for the relevant array(s). Whenever the source code patterns appear within loops or conditional statements, the goodness measures of the constraints are modified appropriately, again as discussed in Section 5.

2. *Determine the alignment of dimensions of various arrays* : This problem has been referred to as the *component alignment* problem by Li et al. in [15]. They prove the problem NP-complete and give an efficient heuristic algorithm for it. We use their solution and discuss it briefly here. The interested user is referred to [15] for more details. In this approach an undirected, weighted graph called a *component affinity graph* (CAG) is constructed from the source program. The nodes of the graph represent dimensions of arrays. For every constraint on the alignment of two dimensions, an edge having a weight equal to the goodness measure of the constraint is generated between the corresponding two nodes. The use of goodness measure as the edge weight represents a slight modification in our approach, even though they also use something similar as the edge weight. The component alignment problem is defined as partitioning the node set of the CAG into D (D being the maximum dimension of arrays) disjoint subsets so that the total weight of edges across nodes in different subsets is minimized. There is the obvious restriction that no two nodes corresponding to the same array can be in the same subset. Thus the (approximate) solution to the component alignment problem indicates which dimensions of various arrays should be aligned. We obtain D classes of aligned dimensions, corresponding to the D subsets into which the CAG is partitioned. At this point, we can establish a one-to-one correspondence between each class of aligned dimensions and a virtual dimension of the processor grid topology, since each array dimension in that class will get distributed on the same virtual grid dimension.

3. *Determine the following parameters for distribution along each dimension – contiguous/cyclic, relative block sizes, and offsets* : As a result of the previous step, we now know which virtual grid dimensions, the variables N_I, N_J etc. appearing in the expressions for goodness or time measures of various constraints refer to. We consider each class of aligned dimensions one at a time. If for a class i, there is no array dimension which necessarily has to be distributed across more than one processor to get any speedup (indicated by the absence of any term in the expression for execution time with N_i in the denominator), we sequentialize all dimensions in that class. This can lead to great savings in communication costs,

without any loss of effective parallelism.

For each class that has to be distributed, all array dimensions with the same number of elements need to have the same kind of distribution, contiguous or cyclic. For all such array dimensions, we compare the sum total of goodness measures of the constraints advocating contiguous distribution and those favoring cyclic distribution, and choose the one with the higher total goodness measure. Thus a collective decision is taken on all dimensions in that class to maximize overall gains.

Next we determine block sizes for all the array dimensions in the class, in case those dimensions have a cyclic distribution. As mentioned earlier, the desired ratio of block sizes of different array dimensions can be inferred from the mathematical formula expressing the desired relationship between their distributions. Thus the given class of dimensions can be partitioned into equivalence sub-classes, where all the members of a sub-class have the same block size. The assignment of array dimensions to these sub-classes is done by following a greedy approach. The constraints implying such relationships between two distributions are considered in the non-increasing order of their goodness values. If any of the two concerned array dimensions has not yet been assigned to a sub-class, the assignment is done on the basis of their relative block sizes implied by that constraint. If both dimensions have already been assigned to their respective sub-classes, the present constraint is ignored, since the assignment must have been done using some previous constraint with a higher goodness measure. Once the relative block sizes have been determined using this heuristic, the smallest block size can be fixed at one, and the related block sizes determined accordingly.

We now determine the offset values for all distributions in the class. A group of distributions requiring the same offset are given an offset equal to the subscript value of the first array element (1 in Fortran, 0 in C). Then we determine appropriate offsets for other related distributions. For example, for the loop having a reference of the form $D(i) = \mathcal{F}(E(i-1))$, if the offset value for distribution of D_1 has been set to 1, that for E_1 is set to 2. Conflicts among constraints requiring different offset values are resolved again by following a greedy approach in which constraints get considered in the non-increasing order of their goodness measures.

4. *Determine the number of processors along each dimension* : In this step, all virtual grid dimensions, barring two at most, are sequentialized, and we determine the number of processors along the real grid dimensions. The expression for execution time of the program can be obtained in this step as a function of various N_i, since for each constraint on distribution of various array dimensions, we know at this point whether it is satisfied or not. Based on that knowledge we determine for each loop what interprocessor communications are needed and what speedups, if any, can be obtained. Both of these are functions of various N_i, and we add the contributions of the interprocessor communication

part and the computation part to the expression for execution time of the program. We ignore the constant terms (those independent of any N_i), that add to the execution time, since we are only interested in determining values of various N_i. The expressions for time obtained for individual loops are modified and combined to give those for the entire program, using the method discussed in Section 5. As pointed out earlier, at this step we can identify cases, such as the one shown in Figure 9, where repeated communication of the same values caused by different constraints not getting satisfied can be eliminated.

Let D' denote the number of virtual grid dimensions not yet sequentialized at this point. The expression obtained for execution time is a function of variables $N_1, N_2, \ldots N_{D'}$, representing the number of processors along the corresponding grid dimensions. The problem at hand is to minimize the execution time subject to the constraint

$$N_1 * N_2 * \ldots * N_{D'} = N$$

As remarked earlier in Section 2, we rarely expect to find a program in which more than two dimensions of some of its arrays are required to be distributed. Hence for most real programs, D' would have the value two or one. In case the value is one, we simply set N_1 equal to N, and are done with this step. However, if the value does exceed two, we follow this somewhat ad hoc approach to sequentialize $D' - 2$ dimensions. We evaluate the execution time expression of the program for $C_2^{D'}$ cases, each case corresponding to 2 different N_i variables set to $\sqrt{N}$, and the other $D' - 2$ variables set to 1. The case which gives the smallest value for execution time is chosen, and the corresponding $D' - 2$ dimensions are sequentialized.

Once we get down to two dimensions, the execution time expression is a function of just one variable, N_1, since N_2 is given by N/N_1. We can now evaluate the expression for different values of N_1, various factors of N ranging from 1 to N, and select the one which leads to the lowest execution time.

5. *Take decisions on replication of arrays or array dimensions* : We take two kinds of decisions in this step. The first kind consists of determining the additional distribution function for each one-dimensional array, in case the finally chosen grid topology has two real dimensions. The other kind involves deciding whether to override the given distribution function for an array dimension to ensure that it is replicated rather than partitioned over processors in the corresponding grid dimension. For the following discussion, we assume there is enough memory on each processor to support replication of any array deemed necessary. In case this assumption does not hold, it should be straightforward to develop an extension of our strategy, in which we are more selective about which of the arrays or array dimensions approved earlier for replication actually get replicated. We take decisions on both kinds of replication rather conservatively. It is only if certain conditions more stringent than the minimal ones

implying better performance with replication are met, that we take a decision in favor of replication.

The second distribution function of a one-dimensional array may be a positive constant, in which case each array element gets mapped to a unique processor, or may take the value X, signifying that the elements get replicated along that dimension. For each array, we look at the constraints corresponding to the loops where that array is being used. The array is a candidate for replication along the second grid dimension if the goodness measure of some constraint not being satisfied shows that the array has to be multicast over that dimension. An example for that is the array D in the program segment shown in Figure 7, if A_2 is not sequentialized. A decision favoring replication is taken only if each time the array is being written into, the cost of all processors in the second grid dimension carrying out that computation is less than the sum of costs of performing that computation on a single processor and multicasting the result. Note that for performing that computation on all processors in that dimension, it is desirable that all the values needed for that computation be already replicated. Otherwise the communication costs involved in multicasting the operands to these processors would be prohibitive.

For all the one-dimensional arrays on which a decision is taken not to replicate them, we fix the same constant value for the second distribution function. For each array we look for constraints, such as the one seen for Pattern 2 of Section 4.2, which might require the distribution function to take a specific constant value. In case there are such patterns, we choose an appropriate constant value based on the goodness measure, otherwise we arbitrarily set the value of that function for all those arrays to 0.

A decision to override the distribution function of an array dimension from partitioning over a grid dimension to replication on that dimension is taken even more conservatively than for the first case. It is done only when the array dimension is not written into more than once in the program, and it is used by the different processors in the corresponding grid dimension.

At the end of the steps given above, we have a complete specification of the data distribution functions for all arrays. It can be seen that a number of steps in which decisions are taken on certain data distribution parameters need a lot of information, some of which becomes available only after a later step in the strategy. For instance, the step which determines the alignment of array dimensions needs information about the number of processors in each virtual grid dimension, which becomes known only later. We break these cycles of dependences between various steps of the strategy by having the initial steps operate on whatever partial information is available at that point and make use of simplifying assumptions regarding information not available yet. Thus the initial steps yield some approximate results, based on which the later steps can proceed. This strategy seems to work well for a number of real programs we have examined. Work is in progress for evaluating certain iterative variations of this strategy, in which some of the decisions taken earlier may get reversed on the basis of information becoming available during the later steps.

7 Experimental Results

In this section, we present the application of our strategy to a sample routine, TRED2, from the EISPACK library. This routine reduces a real symmetric matrix to a symmetric tridiagonal matrix using and accumulating orthogonal similarity transformations. We show the results obtained at the end of each individual step of our strategy. Then we present actual performance results obtained by implementing different versions of the parallel program, corresponding to different ways of partitioning the arrays. The testbed for our implementation and evaluation is the Intel iPSC/2 hypercube system.

The source code of the subroutine is listed in Figure 10. We shall refer to a statement on line l as S(l), and a loop extending from lines l_1 to l_2 as L(l_1-l_2). Along with the code listing, we have shown the probabilities of taking the branch on various conditional *go to* statements. These probabilities are assumed to have been determined correctly by the compiler. Also, corresponding to each statement in a loop that gets matched to a known pattern, we have indicated the pattern number, expressed as section number – pattern number. The idea expressed in Pattern 3 of Section 4.1, suggesting cyclic distribution for load balancing, gets used in practically every loop in the routine. We have omitted referring to that pattern explicitly in the figure to avoid repetitiveness. It can be seen that we have known patterns corresponding to all the statements that should provide constraints on data distribution. There are numerous loops in which transformations like loop distribution and global forward substitution need to be applied to reveal those patterns.

As an example of loop distribution, consider the loop L(23-26). After loop distribution, it gets transformed to :

```
      do 24 K = 1, L
24          D(K) = D(K)/SCALE
      do 25 K = 1, L
25          H = H + D(K) * D(K)
```

Now the two loops can be identified as corresponding to appropriate patterns, as shown on lines 24 and 25 in Figure 10. In many cases, as in lines 1-5, we may not actually do loop distribution. However, since the compiler recognizes that this transformation can be legally done, the statements S(3) and S(4) get recognized as individually belonging to different loops, and get matched to appropriate patterns.

By applying forward substitution, we may delete statements on lines 55-56, and transform S(58) to the following statement :

$$Z(K,J) = Z(K,J) - D(J) * E(K) - E(J) * D(K)$$

This enables the compiler to recognize the need to broadcast the arrays D and E to execute that loop in

```
1   DO 5 I = 1, N
2       DO 3 J = I, N
3         Z(J,I) = A(J,I)                    4.2-1, 4.1-1
4       D(I) = A(N,I)                        4.2-2, 4.1-2
5   CONTINUE
6   IF (N .EQ. 1) GO TO 82                   prob = 0
7   DO 63 II = 2, N
8       I = N + 2 - II
9       L = I - 1
10      H = 0.0D0
11      SCALE = 0.0D0
12      IF (L .LT. 2) GO TO 16               prob = 1/(N-1)
13      DO 14 K = 1, L
14        SCALE = SCALE + DABS(D(K))         4.1-4
15      IF (SCALE .NE. 0.0D0) GO TO 23       prob = 1
16      E(I) = D(L)
17      DO 21 J = 1, L
18          D(J) = Z(L,J)                    4.2-2, 4.1-2
19          Z(I,J) = 0.0D0                   4.1-2
20          Z(J,I) = 0.0D0                   4.1-2
21      CONTINUE
22      GO TO 62
23      DO 25 K = 1, L
24          D(K) = D(K) / SCALE              4.1-2
25          H = H + D(K) * D(K)              4.1-4
26      CONTINUE
27      F = D(L)
28      G = -DSIGN(DSQRT(H),F)
29      E(I) = SCALE * G
30      H = H - F * G
31      D(L) = F - G
32      DO 33 J = 1, L
33        E(J) = 0.0D0                       4.1-2
34      DO 45 J = 1, L
35          F = D(J)
36          Z(J,I) = F
37          G = E(J) + Z(J,J) * F
38          JP1 = J + 1
39          IF (L .LT. JP1) GO TO 43
40          DO 43 K = JP1, L
41              G = G + Z(K,J) * D(K)        4.1-4
42              E(K) = E(K) + Z(K,J) * F     4.2-2, 4.1-2
43          CONTINUE
44          E(J) = G
45      CONTINUE
46      F = 0.0D0
47      DO 50 J = 1, L
48          E(J) = E(J) / H                  4.1-2
49          F = F + E(J) * D(J)              4.1-4
50      CONTINUE
51      HH = F / (H + H)
52      DO 53 J = 1, L
53        E(J) = E(J) - HH * D(J)            4.2-1, 4.1-2
54      DO 61 J = 1, L
55          F = D(J)
56          G = E(J)
57          DO 58 K = J, L
58              Z(K,J) = Z(K,J) - F * E(K) - G * D(K)  4.1-1
59          D(J) = Z(L,J)                    4.2-2
60          Z(I,J) = 0.0D0                   4.1-2
61      CONTINUE
62      D(I) = H
63  CONTINUE
64  DO 81 I = 2, N
65      L = I - 1
66      Z(N,L) = Z(L,L)                      4.3-3
67      Z(L,L) = 1.0D0
68      H = D(I)
69      IF (H .EQ. 0.0D0) GO TO 78           prob = 0
70      DO 71 K = 1, L
71        D(K) = Z(K,I) / H                  4.2-2, 4.1-2
72      DO 78 J = 1, L
73          G = 0.0D0
74          DO 75 K = 1, L
75            G = G + Z(K,I) * Z(K,J)        4.4-3, 4.1-4
76          DO 78 K = 1, L
77            Z(K,J) = Z(K,J) - G * D(K)     4.2-2
78      CONTINUE
79      DO 80 K = 1, L
80        Z(K,I) = 0.0D0                     4.1-2
81  CONTINUE
82  DO 85 I = 1, N
83      D(I) = Z(N,I)                        4.2-2, 4.1-2
84      Z(N,I) = 0.0D0                       4.1-2
85  CONTINUE
86  Z(N,N) = 1.0D0
87  E(1) = 0.0D0
88  END
```

Figure 10: Fortran code for TRED2 routine

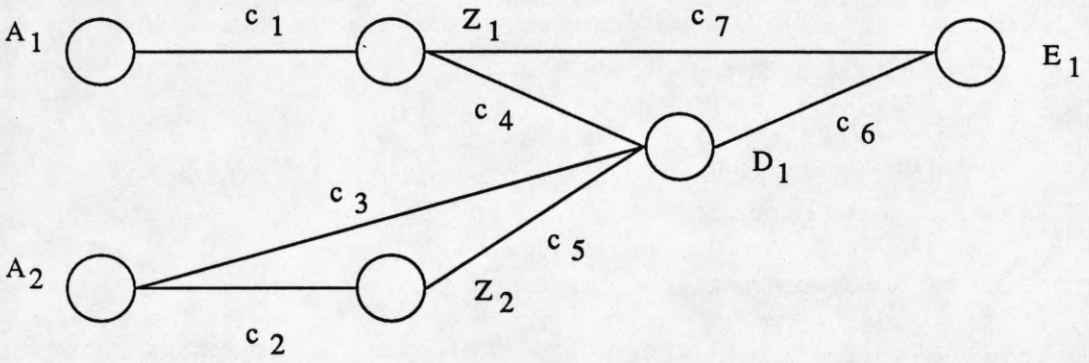


Figure 11: Component affinity graph for TRED2

parallel, and not formulate any constraint on alignment between dimensions of Z and of D or E for this loop.

The loops L(7-63), L(34-45) and L(64-81) are inherently sequential, and the statements S(59) and S(66) provide examples of statements within such loops that provide communication constraints.

Regarding the estimation of goodness values of various constraints, the following points deserve special mention. In our parallel code, we implement the primitives *AllToAllMulticast / AllToAllBroadcast* as repeated *OneToAllMulticast / OneToAllBroadcast* primitives, hence in our goodness estimates, we express the latter two functions in terms of the former two. Secondly, a number of loops in the program use the variable L as the upper bound, where L itself varies in an outer, sequential loop. We simplify the analysis by assuming that L takes its average value of $n/2$ during all iterations of the outer loop.

Based on the constraints on alignments, a component affinity graph is constructed for the program, as shown in Figure 11. The weights associated with various edges are as follows :

$$
\begin{aligned}
c_1 &= N * OneToAllBroadcast(n^2/N, N)^{(3)} \\
c_2 &= N * OneToAllBroadcast(n^2/N, N)^{(3)} \\
c_3 &= N_I * OneToAllMulticast(n/N_I, N_J)^{(4)} \\
c_4 &= (n-1) * N_I * OneToAllMulticast(n/2N_I, N_J)^{(71)} + \\
&\quad (n-1) * N_I * OneToAllMulticast(n/2N_I, N_J)^{(77)} \\
c_5 &= N_I * OneToAllMulticast(n/2N_I, N_J)^{(18)} + (n-1) * (n/2) * Transfer(1)^{(59)} + \\
&\quad N_I * OneToAllMulticast(n/N_I, N_J)^{(83)} \\
c_6 &= (n-2) * N_I * OneToAllMulticast(n/2N_I, N_J)^{(53)}
\end{aligned}
$$

$$c_7 \quad = \quad (n-2) * (n/2) * N_I * OneToAllMulticast(n/4N_I, N_J)^{(42)}$$

The numbers in parentheses along with each term indicate the line number in the program, which the constraint corresponding to the goodness measure may be traced to. Each *OneToAllMulticast / OneToAllBroadcast* operation sending messages to p processors is $\lceil log_2 p \rceil$ times as expensive as a $Transfer$ operation, for messages of the same size. We use the following approximate function [8] to estimate the time taken, in microseconds, to complete a $Transfer$ operation on l bytes :

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

The value of n, which depends on the size of the arrays, is fixed at 512, and the number of processors, N, takes the value 16. Hence for this step, the values of both N_I and N_J are taken to be 4. Applying the algorithm for component alignment on this graph, we get the following classes of dimensions – class 1 consisting of A_1, Z_1, D_1, E_1, and class 2 consisting of A_2, Z_2. These classes get mapped to the dimensions 1 and 2 respectively of the processor grid.

We now move on to the Step 3 of our strategy. None of the classes of array dimensions gets sequentialized in *this* step, since there are constraints with terms for time having N_1 and also those having N_2 in the denominator. The distribution functions for all array dimensions in each of the two classes is determined to be cyclic, because of constraints on each dimension of arrays Z, D and E favoring cyclic distribution. The block sizes for all the aligned array dimensions determined to be the same, and the distributions for all array dimensions are made to use the same offset value of 1. Hence, at the end of this step, the distributions for various array dimensions are :

$$\begin{aligned} f_A^1(i) \quad &= \quad f_Z^1(i) \quad = \quad f_D^1(i) \quad = \quad f_E^1(i) \quad = \quad (i-1) \bmod N_1 \\ f_A^2(j) \quad &= \quad f_Z^2(j) \quad = \quad (j-1) \bmod N_2 \end{aligned}$$

Now we determine the value of N_1. The value of N_2 gets fixed as simply N/N_1. By adding together the expressions for time for various parallelization constraints, and the goodness measures of various communication constraints not getting satisfied, we get the following expression for execution time of the program (only the part dependent on N_1).

$$\begin{aligned} Time \quad = \quad & (1 - \frac{\mathbf{N_1}}{\mathbf{N}}) * (\mathbf{n-2}) * [\frac{\mathbf{n}}{\mathbf{2}} * \mathbf{Transfer(n/4N)} + \mathbf{2} * \mathbf{Transfer(nN_1/2N)}] + \\ & \frac{2N}{N_1} * OneToAllMulticast(nN_1/N, N_1) + \frac{N}{N_1} * OneToAllMulticast(nN_1/2N, N_1) + \\ & (1 - \frac{1}{N_1}) * (n-2) * Transfer(1) + 5 * (n-2) * UniReduction(1, N_1) + \\ & \frac{c}{N_1} * [7.6 * n * (n-2) + .1 * n] + \frac{n * (n+1) * c * N_1}{20 * N} \end{aligned}$$

The following assumptions regarding computation costs are used by the compiler in determining this expression. Let c denote the time taken to execute a double precision floating point add operation. We assume that multiplication takes time c, division takes time $2c$, and each simple assignment (load and store) takes time $0.1c$. The timing overhead associated with various control instructions is ignored. The value of c is taken to be approximately 5 microseconds. For all values of N ranging from 4 to 16, and using $n = 512$, we see that the above expression for execution time gets minimized when the value of N_1 is set to N. This is easy to see since the first term (appearing in boldface), which dominates the expression, vanishes when $N_1 = N$. Incidentally, that term comes from the goodness measures of various constraints to sequentialize Z_2. The real processor grid, therefore, has only one dimension, all array dimensions in the second class get sequentialized. Hence the distribution functions for the array dimensions at the end of this step are :

$$\begin{aligned} f_A^1(i) &= f_Z^1(i) = f_D^1(i) = f_E^1(i) = (i-1) \bmod N \\ f_A^2(j) &= f_Z^2(j) = 0 \end{aligned}$$

Since we are using the processor grid as one with a single dimension, we do not need the second distribution function for the arrays D and E to uniquely specify which processors own various elements of these arrays. None of the array dimensions meet the conditions required for replication. We do expect the array D to be broadcast to all processors immediately after executing the loop L(23-25), and the array E to be broadcast before loop L(54-61). Thus all the processors needing values of various elements of array D may use their local copy while they are executing their part of the code between lines 27 and 61. This does not count as replication by our definition.

The data distribution scheme that finally emerges is – distribute arrays A and Z by *rows* in a *cyclic* fashion, distribute array D and E also in a cyclic manner, on all the N processors. The formal definitions of distribution functions have already been given above.

Starting from the sequential program, we wrote the target host and node programs for the iPSC/2 by hand, using the scheme suggested for a parallelizing compiler in [3] and [26], and hand-optimized the code. We first implemented the version that uses the data distribution scheme suggested by our strategy, i.e, row cyclic. The reader can appreciate that just by looking at the sequential TRED2 routine, it is not obvious what data partitioning scheme would be the best. An alternate scheme that also looks reasonable, by looking at various constraints, is one which distributes the arrays A and Z by *columns* instead of rows. To get an idea of the gains in performance made by sequentializing a class of dimensions, i.e., by not distributing A and Z in a blocked manner, and also by choosing a cyclic rather than contiguous distribution for all arrays, we implemented two other versions of the program. These versions correspond to the "bad" choices on data distribution that a user might make if he is not careful enough. 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 Figure 12. 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 suggested by our strategy performs much better than other schemes for that data size. For the smaller data size, the scheme using column distribution of arrays works slightly better when fewer processors are being used. Our approach identifies a number of constraints that do favor the column distribution scheme, however they get outweighed by the constraints that favor row-wise distribution of arrays. Regarding other issues, our strategy clearly advocates the use of cyclic distribution rather than contiguous, and also the sequentialization of a class of dimensions, as suggested by numerous constraints to sequentialize various array dimensions. The fact that both these observations are indeed crucial can be seen from the poor performance of the program corresponding to contiguous (row-wise, for A and Z) distribution of all arrays, and also of the one corresponding to blocked (grid-like) distribution of arrays A and Z. These results show that for this program, our approach is able to take the right decisions regarding certain key issues in data distribution, and does suggest a data partitioning scheme that leads to good performance.

8 Related Work

A number of researchers are developing compilers that take a sequential program augmented with annotations specifying data distribution, and generate the target program with explicit communication primitives, meant to be executed on distributed memory machines. These include the Superb project at Bonn University [26], Callahan and Kennedy's work at Rice University [3], the Kali project at Purdue University [12, 13], and the DINO project at Colarado University [23]. The Crystal project at Yale University [4] is also based on the same idea, but is targeted for the functional language Crystal, as opposed to the other projects which have concentrated on imperative languages, mainly extensions of Fortran. By and large, the task of determining suitable data partitions, which may be regarded as the most crucial and challenging of all tasks in the whole process has been left completely to the user, at least in the initial stages of these projects.

Recently several researchers have addressed this problem of determining proper data partitioning schemes automatically, or of providing help to the user in this task. Li and Chen [15] address the issue of data movement between processors due to cross-references between multiple distributed arrays. They refer to it as the index domain alignment problem. We use their algorithm for this problem to determine the alignment of various array dimensions in one of the key steps in our strategy. The way the problem instance is constructed is slightly different in our approach. The problem we address in this paper is much broader than

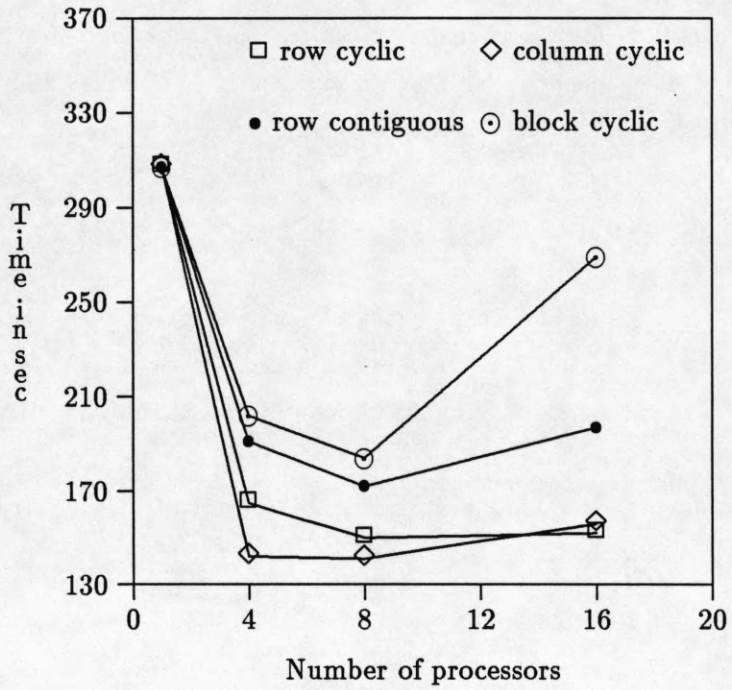


Figure 12: Performance of TRED2 on Intel iPSC/2 for data size $n = 256$

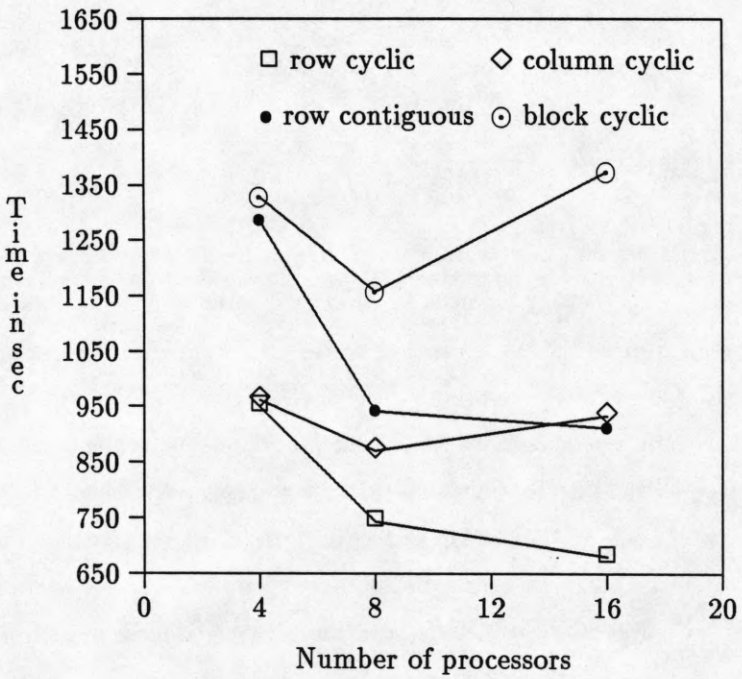


Figure 12: Performance of TRED2 on Intel iPSC/2 for data size $n = 512$

simply determining the alignment between various array dimensions. Ramanujan and Sadayappan [21] have worked on deriving data partitions for a restricted class of programs, they concentrate more on individual loops and strongly connected components, rather than the entire programs. Hudak and Abraham [9] present techniques for data partitioning for the class of programs which may be modeled as sequentially iterated parallel loops. Snyder and Socha [24] also present a partitioning algorithm that is useful for a similar class. Balasundaram et al. [2] discuss an interactive tool that provides assistance to the user for data distribution. The key element in their tool is a performance estimation module, which can be extremely useful. In the context of the Crystal project again, Li and Chen [16] describe how synthesis of explicit communication can be done by analyzing reference patterns in the source program. That is very closely related to our notion of recognizing definite patterns in the source program and formulating constraints on data distribution based on them. King et al. [11] present a methodology for grouping together related iterations on distributed memory machines, which has implications for how data partitioning should be done. They restrict themselves to the case where the algorithm can be expressed as a nested loop with constant loop-carried dependences.

9 Conclusions

We have presented a new approach, the constraint-based approach, to the problem of determining suitable data partitions for a program, that can easily be implemented as a back end of a parallelizing compiler for a distributed memory machine. Our approach is quite general, and can be applied to a large class of programs having reference patterns that can be analyzed at compile time. We used the routine TRED2 as an example to demonstrate how our strategy works on real programs. We feel that our major contributions to the problem of automatic data partitioning are :

- *The notion of constraints on data distribution implied by individual loops* : These constraints provide the right abstraction of what the significance of each loop is with respect to data distribution, and the weights attached to them are adjusted according to their relative effect on program execution time. The distribution of each array involves taking decisions regarding a number of parameters discussed earlier, and each constraint specifies only the basic minimum requirements on distribution. Hence the parameters related to the distribution of a particular array, left unspecified by a constraint, can be further selected by combining that constraint with more constraints, each successful combination leading to an increase in the goodness of the distribution scheme. Wherever there is a conflict between constraints, it gets resolved in favor of the one with the higher goodness measure.

- *Attempted optimization of the right quantity* : We try to take into account both communication costs *and* parallelization considerations, so that the overall execution time may get minimized. Also, we look

at data distribution from the perspective of performance of the entire program, not just that of some individual program segments. This makes our approach applicable to a much bigger class of programs, not just those that can be modelled as being a single, nested loop.

- *Distribution functions for the array dimensions* : Our formulation of the distribution functions allows for a rich variety of distribution schemes possible for each array. We may have cyclic distributions that can be very useful for ensuring proper load balance for certain applications. In fact, we allow for a complete range of possibilities from completely interleaved distibutions to completely contiguous distributions, through variations in the block size for cyclic distributions. We also allow for a number of meaningful possibilities with regard to replication. An array may be replicated completely on all processors, or only partially on processors along a grid dimension. In many cases this helps reduce communication costs.

- *Variety of relationships possible between array distributions* : We have seen how our definition of distribution functions allows us to capture any relationship between distributions of different array dimensions implied by a loop, when the subscripts being used to reference arrays are linear functions of loop indices. Thus we support a wide variety of relationships between array distributions, not just alignment between array dimensions.

We are currently examining a number of directions in which our approach can be extended. As mentioned earlier, we are investigating the expected benefits and costs of an iterative version of our strategy in which a certain sequence of steps could be applied repeatedly to get further refinements in the data distribution scheme. Another important issue that we are looking at is data reorganization. For some programs it might be desirable to partition the data one way for a particular program segment, and then repartition it before moving on to the next program segment. We are also working on developing a more comprehensive categorization of loop patterns occurring in programs. In future, we plan to look at the problem of inter-procedural analysis, so that the formulation of constraints and their goodness estimation could be done across procedure calls.

The importance of the problem of data partitioning is bound to continue growing as more and more machines with larger number of processors keep getting built. We expect that the ideas presented in this paper shall prove to be quite useful for the efforts to develop parallelizing compilers for such machines.

References

[1] J. R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans-*

actions on Programming Languages and Systems, 9(4):491–542, October 1987.

[2] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proc. Fifth Distributed Memory Computing Conference*, Charleston, S. Carolina, April 1990. (to appear).

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

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

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

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

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

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

[9] 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.

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

[11] C. T. King, W. H. Chou, and L. M. Ni. Pipelined data-parallel algorithms: Part ii - design. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):486–499, October 1990.

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

[13] C. Koelbel, P. Mehrotra, and J. van Rosendale. Semi-automatic process partitioning for parallel computation. *International Journal of Parallel Programming*, 16(5):365–382, 1987.

[14] 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.

[15] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical Report YALEU/DCS/TR-725, Yale University, November 1989.

[16] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Yale University, May 1990.

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

[18] S. Nugent. The ipsc/2 direct-connect technology. In *Proc. 3rd Conference on Hypercube Concurrent Computers and Applications*, 1988.

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

[20] C. D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, MA, 1988.

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

[22] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers*, C-36(7):845–858, July 1987.

[23] M. Rosing, R. B. Schnabel, and R. P. Weaver. The dino parallel programming language. Technical Report CU-CS-457-90, University of Colarado at Boulder, April 1990.

[24] L. Snyder and D. Socha. A partitioning algorithm to allocate arrays to processor memories. In *Proc. Fifth Distributed Memory Computing Conference*, Charleston, S. Carolina, April 1990. (to appear).

[25] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.

[26] H. Zima, H. Bast, and H. Gerndt. Superb: A tool for semi-automatic mimd/simd parallelization. *Parallel Computing*, 6:1–18, 1988.