IEEE Transactions on Parallel and Distributed Systems, March 1992

Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers*

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) 244-7166
Fax: (217) 244-1764
E-mail: manish@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, based on performance considerations, a compiler identifies constraints to be imposed on the distribution of various data structures. These constraints are then 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 present results of a study we performed on Fortran programs taken from the Linpack and Eispack libraries and the Perfect Benchmarks to determine the applicability of our approach to real programs. The results are very encouraging, and demonstrate the feasibility of automatic data partitioning for programs with regular computations that may be statically analyzed, which covers an extremely significant class of scientific application programs.

Keywords

automatic data partitioning, constraints, interprocessor communication, multicomputers, parallelizing compilers.

*This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096, in part by the National Science Foundation under Grant NSF MIP 86-57563 PYI, and in part by National Aeronautics and Space Administration under Contract NASA NAG 1-613.

1 Introduction

Distributed memory multiprocessors (multicomputers) are increasingly being used for providing 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 to relieve the programmer of this burden.

The area of parallelizing compilers for multicomputers has seen considerable research activity during the last few years. A number of researchers are developing compilers that take a program written in a sequential or shared-memory parallel language, and based on the user-specified partitioning of data, generate the target parallel program for a multicomputer. These research efforts include the Fortran D compiler project at Rice University [9], the SUPERB project at Bonn University [22], the Kali project at Purdue University [13], and the DINO project at Colorado University [20], all of them dealing with imperative languages that are extensions of Fortran or C. The Crystal project at Yale University [5] and the Id Nouveau compiler [19] are also based on the same idea, but are targeted for functional languages. The parallel program generated by most of these systems corresponds to the SPMD (single program, multiple-data) [12] model, in which each processor executes the same program but operates on distinct data items.

The current work on parallelizing compilers for multicomputers has, by and large, concentrated on automating the generation of messages for communication among processors. Our use of the term "parallelizing compiler" is somewhat misleading in this context, since all parallelization decisions are really left to the programmer who specifies data partitioning. It is the method of data partitioning that determines when interprocessor communication takes place, and which of the independent computations actually get executed on different processors.

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 the 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 remains incomplete as long as the programmer is forced to think about these issues and come up with the right data partitioning scheme for each program. The task of determining a good partitioning scheme manually can be extremely difficult and tedious. However, most of the existing projects on parallelization systems for multicomputers have so far chosen not to tackle this problem at the compiler level because it is known to be a difficult problem. Mace [16] 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 and Chen [15], and shown to be NP-complete.

Recently several researchers have addressed this problem of automatically determining a data partitioning scheme, or of providing help to the user in this task. Ramanujan and Sadayappan [18] have worked on deriving data partitions for a restricted class of programs. They, however, concentrate on individual loops and strongly connected components rather than considering the program as a whole. Hudak and Abraham [11], and Socha [21] present techniques for data partitioning for programs that may be modeled as sequentially iterated parallel loops. Balasundaram et al. [1] 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 is used to evaluate various alternatives regarding the distribution scheme. Li and Chen [15] address the issue of data movement between processors due to cross-references between multiple distributed arrays. They also describe how explicit communication can be synthesized and communication costs estimated by analyzing reference patterns in the source program [14]. These estimates are used to evaluate different partitioning schemes.

Most of these approaches have serious drawbacks associated with them. Some of them have a problem of restricted applicability, they apply only to programs that may be modeled as single, multiply nested loops. Some others require a fairly exhaustive enumeration of possible data partitioning schemes, which may render the method ineffective for reasonably large problems. Clearly, any strategy for automatic data partitioning can be expected to work well only for applications with a regular computational structure and static dependence patterns that can be determined at compile time. However, even though there exists a significant class of scientific applications with these properties, there is no data to show the effectiveness of any of these methods on real programs.

In this paper, we present a novel approach, which we call the *constraint-based approach* [7], to the problem of automatic data partitioning on multicomputers. In this approach, the compiler analyzes each loop in the program, and based on performance considerations, identifies some constraints on the distribution of various data structures being referenced in that loop. There is a quality measure associated with each constraint that captures its importance with respect to the performance of the program. Finally, the compiler tries to combine constraints for each data structure in a consistent manner so that the overall execution time of the parallel program is minimized. We restrict ourselves to the partitioning of arrays. The ideas underlying our approach can be applied to most distributed memory machines, such as the Intel iPSC/2, the NCUBE, and the WARP systolic machine. Our examples are all written in a Fortran-like language, and we present results on Fortran programs. However, the ideas developed on the partitioning of arrays are equally applicable to any similar programming language.

The rest of this paper is organized as follows. Section 2 describes our abstract machine and the kind of distributions that arrays may have in our scheme. Section 3 introduces the notion of constraints and describes the different kinds of constraints that may be imposed on array distributions. Section 4 describes how a compiler analyzes program references to record constraints and determine the quality measures associated with them. Section 5 presents our strategy for determining the data partitioning scheme. Section 6 presents the results of our study on Fortran programs performed to determine the applicability of our approach to real programs. Finally, conclusions are presented in Section 7.

2 Data Distribution

The abstract target machine we assume is a D-dimensional (D is the maximum dimensionality of any array used in the program) grid of $N_1 * N_2 * \ldots * N_D$ processors. Such a topology can easily be embedded on almost any distributed memory machine. A processor in such a topology is represented by the tuple $(p_1, p_2, \ldots, p_D), 0 \leq p_k \leq N_k - 1$ for $1 \leq k \leq D$. The correspondence between a tuple $(p_1, p_2, \ldots, p_D)$ and a processor number in the range 0 to $N - 1$ is established by the scheme which embeds the virtual processor 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. A processor tuple with an X appearing in the ith position denotes all processors along the ith grid dimension. 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 and small arrays used in the program are assumed to be replicated on all processors. For other arrays, we use a separate distribution function with each dimension to indicate how that array is distributed across processors. This turns out to be more convenient than having a single distribution function associated with a multidimensional array. We refer to the kth dimension of an array A as A_k. Each array dimension A_k gets mapped to a unique dimension $map(A_k)$, $1 \leq map(A_k) \leq D$, of the processor grid. If

$N_{map(A_k)}$, the number of processors along that grid dimension is one, we say that the array dimension A_k has been *sequentialized*. The sequentialization of an array dimension implies that all elements whose subscripts differ only in that dimension are allocated to the same processor. The distribution function for A_k takes as its argument an index i and returns the component $map(A_k)$ of the tuple representing the processor which *owns* the element $A[-,-,\ldots,i,\ldots-]$, where '$-$' denotes an arbitrary value, and i is the index appearing in the k^{th} dimension. The array dimension A_k may either be partitioned or replicated on the corresponding grid dimension. The distribution function is of the form

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

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

Examples of some data distribution schemes possible for a $16 * 16$ 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 an $N_1 * N_2$ mesh, and the processor number corresponding to the tuple (p_1, p_2) is given by $p_1 * N_2 + p_2$. The distribution functions corresponding to the different figures are given below. The array subscripts are assumed to start with the value 1, as in Fortran.

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

The last example illustrates how our notation allows us to specify *partial* replication of data, i.e., replication of an array dimension 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.

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

Most of the arrays used in real scientific programs, such as routines from LINPACK and EISPACK libraries and most of the Perfect Benchmark programs [6], 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 completely parallel loop nest shown below:

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

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 order to simplify our strategy, and with the above observation providing some justification, we shall assume for now that our underlying target topology is a *two-dimensional mesh*. For the sake of notation describing the distribution of an array dimension on a grid dimension, we shall continue to regard the target topology conceptually as a D-dimensional grid, with the restriction that the values of $N_3, \dots N_D$ are later set to one.

3 Constraints on Data Distribution

The data references associated with each loop in the program indicate some desirable properties that the final distribution for various arrays should have. We formulate these desirable characteristics as *constraints* on the data distribution functions. Our use of this term differs slightly from its common usage in the sense that constraints on data distribution represent requirements that *should* be met, and not requirements that necessarily *have* to be met.

Corresponding to each statement assigning values to an array in a parallelizable loop, there are two kinds of constraints, parallelization constraints and communication constraints. The former kind gives constraints on the distribution of the array appearing on the left hand side of the assignment statement. The distribution should be such that the array elements being assigned values in a parallelizable loop are distributed evenly on as many processors as possible, so that we get good performance due to exploitation of parallelism. The communication constraints try to ensure that the data elements being read in a statement reside on the same processor as the one that owns the data element being written into. The motivation for that is the *owner computes* rule [9] followed by almost all parallelization systems for multicomputers. According to that rule, the processor responsible for a computation is the one that owns the data item being assigned a value in that computation. Whenever that computation involves the use of a value not available locally on the processor, there is a need for interprocessor communication. The communication constraints try to eliminate this need for interprocessor communication, whenever possible.

In general, depending on the kind of loop (a single loop may correspond to more than one category), we have rules for imposing the following kinds of constraints:

1. Parallelizable loop in which array A gets assigned values – parallelization constraints on the distribution of A.
2. Loop in which assignments to array A use values of array B – communication constraints specifying the *relationship* between distributions of A and B.
3. Loop in which assignments to certain elements of A use values of different elements of A – communication constraints on the distribution of A.
4. Loop in which a single assignment statement uses values of multiple elements of array B – communication constraints on the distribution of B.

The constraints on the distribution of an array may specify any of the relevant parameters, such as the number of processors on which an array dimension is distributed, whether the distribution is contiguous or cyclic, and the block size of distribution. There are two kinds of constraints on the relationship between distribution of arrays. One kind specifies the *alignment* between dimensions of different arrays. Two array dimensions are said to be aligned if they get distributed on the same processor grid dimension. The other kind of constraint on relationships formulates one distribution function in terms of the other for aligned dimensions. For example, consider the loop shown below:

$$
\begin{array}{l}
\text{do } i = 1, n \\
\quad A(i, c_1) = \mathcal{F}(B(c_2 * i)) \\
\text{enddo}
\end{array}
$$

The data references in this loop suggest that A_1 should be aligned with B_1, and A_2 should be sequentialized. Secondly, they suggest the following distribution function for B_1, in terms of that for A_1.

$$\begin{aligned} f_B^1(c_2 * i) &= f_A^1(i) \\ f_B^1(i) &= f_A^1(\lfloor i/c_2 \rfloor) \end{aligned} \tag{1}$$

Thus, given parameters regarding the distribution of A, like the block size, the offset, and the number of processors, we can determine the corresponding parameters regarding the distribution of B by looking at the relationship between the two distributions.

Intuitively, the notion of constraints provides an abstraction of the significance of each loop with respect to data distribution. The distribution of each array involves taking decisions regarding a number of parameters described earlier, and each constraint specifies only the basic minimal requirements on distribution. Hence the parameters related to the distribution of an array left unspecified by a constraint may be selected by combining that constraint with others specifying those parameters. Each such combination leads to an improvement in the distribution scheme for the program as a whole.

However, different parts of the program may also impose conflicting requirements on the distribution of various arrays, in the form of constraints inconsistent with each other. In order to resolve those conflicts, we associate a measure of *quality* with each constraint. Depending on the kind of constraint, we use one of the following two quality measures – the *penalty* in execution time, or the *actual* execution time. For constraints which are finally either satisfied or not satisfied by the data distribution scheme (we refer to them as *boolean* constraints, an example of such a constraint is one specifying the alignment of two array dimensions), we use the first measure which estimates the penalty paid in execution time if that constraint is not honored. For constraints specifying the distribution of an array dimension over a number of processors, we use the second measure which expresses the execution time as a simple function of the number of processors. Depending on whether a constraint affects the amount of parallelism exploited or the interprocessor communication requirement, or both, the expression for its quality measure has terms for the computation time, the communication time, or both.

One problem with estimating the quality measures of constraints is that they may depend on certain parameters of the final distribution that are not known beforehand. We express those quality measures as functions of parameters not known at that stage. For instance, the quality measure of a constraint on alignment of two array dimensions depends on the numbers of processors on which the two dimensions are otherwise distributed, and is expressed as a function of those numbers.

4 Determining Constraints and their Quality Measures

The success of our strategy for data partitioning depends greatly on the compiler's ability to recognize data reference patterns in various loops of the program, and to record the constraints indicated by those references, along with their quality measures. We limit our attention to statements that involve assignment to arrays, since all scalar variables are replicated on all the processors. The computation time component of the quality measure of a constraint is determined by estimating the time for sequential execution based on a count of various operations, and by estimating the speedup. Determining the communication time component is a relatively tougher problem. We have developed a methodology for compile-time estimation of communication costs incurred by a program [8]. It is based on identifying the primitives needed to carry out interprocessor communication and determining the message sizes. The communication costs are obtained as functions of the numbers of processors over which various arrays are distributed, and of the method of partitioning, namely, contiguous or cyclic. The quality measures of various communication constraints are based on the estimates obtained by following this methodology. We shall briefly describe it here, further details can be found in [8].

Communication Primitives We use array reference patterns to determine which communication routines out of a given library best realize the required communication for various loops. This idea was first presented by Li and Chen [14] to show how explicit communication can be synthesized by analyzing data reference patterns. We have extended their work in several ways, and are able to handle a much more comprehensive set of patterns than those described in [14]. We assume that the following communication routines are supported by the operating system or by the run-time library:

- *Transfer* : sending a message from a single source to a single destination processor.
- *OneToManyMulticast* : multicasting a message to all processors along the specified dimension(s) of the processor grid.
- *Reduction* : reducing (in the sense of the APL *reduction* operator) data using a simple associative operator, over all processors lying on the specified grid dimension(s).
- *ManyToManyMulticast* : replicating data from all processors on the given grid dimension(s) on to themselves.

Table 1 shows the cost complexities of functions corresponding to these primitives on the hypercube architecture. The parameter m denotes the message size in words, seq is a sequence of numbers representing the numbers of processors in various dimensions over which the aggregate communication primitive is carried out. The function *num* applied to a sequence simply returns the total number of processors represented by that sequence, namely, the product of all the numbers in that sequence. In general, a parallelization system written for a given machine must have a knowledge of the actual timing figures associated with these primitives on that machine. One possible approach to obtaining such timing figures is the "training set method" that has recently been proposed by Balasundaram et al. [2].

Subscript Types An array reference pattern is characterized by the loops in which the statement appears, and the kind of subscript expressions used to index various dimensions of the array. Each subscript expression is assigned to one of the following categories:

- *constant*: if the subscript expression evaluates to a constant at compile time.
- *index*: if the subscript expression reduces to the form $c_1 * i + c_2$, where c_1, c_2 are constants and i is a loop index. Note that induction variables corresponding to a single loop index also fall in this category.
- *variable*: this is the default case, and signifies that the compiler has no knowledge of how the subscript expression varies with different iterations of the loop.

For subscripts of the type *index* or *variable*, we define a parameter called *change-level*, which is the level of the innermost loop in which the subscript expression changes its value. For a subscript of the type *index*, that is simply the level of the loop that corresponds to the index appearing in the expression.

Method For each statement in a loop in which the assignment to an array uses values from the same or a different array (we shall refer to the arrays appearing on the left hand side and the right hand side of the assignment statement as *lhs* and *rhs* arrays), we express estimates of the communication costs as functions of the numbers of processors on which various dimensions of those arrays are distributed. If the assignment statement has references to multiple arrays, the same procedure is repeated for each *rhs* array. For the sake of brevity, here we shall give only a brief outline of the steps of the procedure. The details of the algorithm associated with each step are given in [8].

1. For each loop enclosing the statement (the loops need not be perfectly nested inside one another), determine whether the communication required (if any) can be taken out of that loop. This step ensures that whenever different messages being sent in different iterations of a loop can be combined, we recognize that opportunity and use the cost functions associated with aggregate communication primitives rather than those associated with repeated Transfer operations. Our algorithm for taking

this decision also identifies program transformations, such as loop distribution and loop permutations, that expose opportunities for combining of messages.

2. For each *rhs* reference, identify the pairs of dimensions from the arrays on *rhs* and *lhs* that should be aligned. The communication costs are determined assuming such an alignment of array dimensions. To determine the quality measures of alignment constraints, we simply have to obtain the difference in costs between the cases when the given dimensions are aligned and when they are not.

3. For each pair of subscripts in the *lhs* and *rhs* references corresponding to aligned dimensions, identify the communication term(s) representing the "contribution" of that pair to the overall communication costs. Whenever at least one subscript in that pair is of the type *index* or *variable*, the term represents a contribution from an enclosing loop identified by the value of *change-level*. The kind of contribution from a loop depends on whether or not the loop has been identified in step 1 as one from which communication can be taken outside. If communication can be taken outside, the term contributed by that loop corresponds to an aggregate communication primitive, otherwise it corresponds to a repeated Transfer.

4. If there are multiple references in the statement to an *rhs* array, identify the *isomorphic* references, namely, the references in which the subscripts corresponding to each dimension are of the same type. The communication costs pertaining to all isomorphic references are obtained by looking at the costs corresponding to one of those references, as in step 3, and determining how they get modified by "adjustment" terms from the remaining references.

5. Once all the communication terms representing the contributions of various loops and of various loop-independent subscript pairs have been obtained, compose them together using an appropriate ordering, and determine the overall communication costs involved in executing the given assignment statement in the program.

Examples We now present some example program segments to show the kind of constraints inferred from data references and the associated quality measures obtained by applying our methodology. Along with each example, we provide an explanation to justify the quality measure derived for each constraint. The expressions for quality measures are, however, obtained automatically by following our methodology.

Example 1:

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

```
        enddo
    enddo
```

Parallelization Constraints: Distribute A_1 and A_2 in a cyclic manner.

Our example shows a 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. A simplified analysis indicates 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. If C_p is the estimated time for sequential execution of the given program segment, the quality measure is:

$$\begin{aligned} Penalty &= \frac{C_p}{N/2} - \frac{C_p}{N} \\ &= \frac{C_p}{N} \end{aligned}$$

Example 2:

$$\begin{array}{l} \text{do } i = 1, n_1 \\ \quad \text{do } j = 1, n_2 \\ \quad\quad A(i,j) = \mathcal{F}(B(j, c_1 * i)) \\ \quad \text{enddo} \\ \text{enddo} \end{array}$$

Communication Constraints: Align A_1 with B_2, A_2 with B_1, and ensure that their distributions are related in the following manner:

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

$$f_B^2(c_1 * i) = f_A^1(i)$$

$$f_B^2(i) = f_A^1(\lfloor \frac{i}{c_1} \rfloor) \tag{3}$$

If the dimension pairs we mentioned are not aligned or if the above relationships do not hold, the elements of B residing on a processor may be needed by any other processor. Hence all the $n_1 * n_2/(N_I * N_J)$ elements held by each processor are replicated on all the processors.

$$Penalty = ManyToManyMulticast(n_1 * n_2/(N_I * N_J), \langle N_I, N_J \rangle)$$

Example 3 :

do $j = 2, n_2 - 1$

do $i = 2, n_1 - 1$

$$A(i,j) = \mathcal{F}(B(i-1,j), B(i,j-1), B(i+1,j), B(i,j+1))$$

enddo

enddo

Communication Constraints :

- Align A_1 with B_1, A_2 with B_2.

  As seen in the previous example, the values of B held by each of the $N_I * N_J$ processors have to be replicated if the indicated dimensions are not aligned.

$$Penalty = \text{ManyToManyMulticast}(n_1 * n_2/(N_I * N_J), \langle N_I, N_J\rangle)$$

- Sequentialize B_1.

  If B_1 is distributed on $N_I > 1$ processors, each processor needs to get elements on the "boundary" rows of the two "neighboring" processors.

$$CommunicationTime = 2 * (N_I > 1)\text{Transfer}(n_2/N_J)$$

  The given term indicates that a Transfer operation takes place only if the condition $(N_I > 1)$ is satisfied.

- Sequentialize B_2.

  The analysis is similar to that for the previous case.

$$CommunicationTime = 2 * (N_J > 1)\text{Transfer}(n_1/N_I)$$

- Distribute B_1 in a contiguous manner.

  If B_1 is distributed cyclically, each processor needs to communicate all of its B elements to its two neighboring processors.

$$Penalty = 2 * (N_I > 1)\text{Transfer}(n_1 * n_2/(N_I * N_J)) - 2 * (N_I > 1)\text{Transfer}(n_2/N_J)$$

- Distribute B_2 in a contiguous manner.

  The analysis is similar to that for the previous case.

$$Penalty = 2 * (N_J > 1)\text{Transfer}(n_1 * n_2/(N_I * N_J)) - 2 * (N_J > 1)\text{Transfer}(n_1/N_I)$$

Note : The above loop also has parallelization constraints associated with it. If C_p indicates the estimated sequential execution time of the loop, by combining the computation time estimate given by the parallelization constraint with the communication time estimates given above, we get the following expression for execution time:

$$Time = \frac{C_p}{N_I * N_J} + 2 * (N_I > 1) Transfer(n_2/N_J) + 2 * (N_J > 1) Transfer(n_1/N_I)$$

It is interesting to see that the above expression captures the relative advantages of distribution of arrays A and B by rows, columns, or blocks for different cases corresponding to the different values of n_1 and n_2.

5 Strategy for Data Partitioning

The basic idea in our strategy is to consider all constraints on distribution of various arrays indicated by the important segments of the program, and combine them in a consistent manner to obtain the overall data distribution. We resolve conflicts between mutually inconsistent constraints on the basis of their quality measures.

The quality measures of constraints are often expressed in terms of n_i (the number of elements along an arry dimension), and N_I (the number of processors on which that dimension is distributed). To compare them numerically, we need to estimate the values of n_i and N_I. The value of n_i may be supplied by the user through an assertion, or specified in an interactive environment, or it may be estimated by the compiler on the basis of the array declarations seen in the program. The need for values of variables of the form N_I poses a circular problem – these values become known only after the final distribution scheme has been determined, and are needed at a stage when decisions about data distribution are being taken. We break this circularity by assuming initially that all array dimensions are distributed on an equal number of processors. Once enough decisions on data distribution have been taken so that for each boolean constraint we know whether it is satisfied or not, we start using expressions for execution time as functions of various N_I, and determine their actual values so that the execution time is minimized.

Our strategy for determining the data distribution scheme, given information about all the constraints, consists of the steps given below. Each step 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 a partial specification of the data distribution. All the steps presented here are simple enough to be automated. Hence the "we" in our discussion really refers to the parallelizing compiler.

1. *Determine the alignment of dimensions of various arrays*: This problem has been referred to as the *component alignment* problem by Li and Chen in [15]. They prove the problem NP-complete and give an efficient heuristic algorithm for it. We adapt their approach to our problem and use their algorithm to determine the alignment of array dimensions. 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 quality measure of the constraint is generated between the corresponding two nodes. 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, with the restriction that no two nodes corresponding to the same array are in the same subset. Thus the (approximate) solution to the component alignment problem indicates which dimensions of various arrays should be aligned. We can now establish a one-to-one correspondence between each class of aligned array dimensions and a virtual dimension of the processor grid topology. Thus, the mapping of each array dimension to a virtual grid dimension becomes known at the end of this step.

2. *Sequentialize array dimensions that need not be partitioned*: If in a given class of aligned array dimensions, there is no dimension which necessarily has to be distributed across more than one processor to get any speedup (this is determined by looking at all the parallelization constraints), we sequentialize all dimensions in that class. This can lead to significant savings in communication costs without any loss of effective parallelism.

3. *Determine the following parameters for distribution along each dimension – contiguous/cyclic and relative block sizes*: For each class of dimensions that is not sequentialized, all array dimensions with the same number of elements are given the same kind of distribution, contiguous or cyclic. For all such array dimensions, we compare the sum total of quality measures of the constraints advocating contiguous distribution and those favoring cyclic distribution, and choose the one with the higher total quality measure. Thus a collective decision is taken on all dimensions in that class to maximize overall gains.

   If an array dimension is distributed over a certain number of processors in a contiguous manner, the block size is determined by the number of elements along that dimension. However, if the distribution is cyclic, we have some flexibility in choosing the size of blocks that get cyclically distributed. Hence, if cyclic distribution is chosen for a class of aligned dimensions, we look at constraints on the relative block sizes pertaining to the distribution of various dimensions in that class. All such constraints may not be mutually consistent. Hence, the strategy we adopt is to partition the given class of aligned dimensions into equivalence sub-classes, where each member in a sub-class has the same block size. The assignment of 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 quality measures. 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 constraint with a higher quality measure. Once all the relative block sizes have been determined using this heuristic, the smallest block size is fixed at one, and the related block sizes determined accordingly.

4. *Determine the number of processors along each dimension*: At this point, for each boolean constraint we know whether it has been satisfied or not. By adding together the terms for computation time and communication time with the quality measures of constraints that have not been satisfied, we obtain an expression for the estimated execution time. 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 numbers of processors along the corresponding grid dimensions. For most real programs, we expect the value of D' to be two or one. If $D' > 2$, we first sequentialize all except for two of the given dimensions based on the following heuristic. 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 (N is the total number of processors in the system). 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 now evaluate the execution time expression for different values of N_1, various factors of N ranging from 1 to N, and select the one which leads to the smallest 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 when 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. We assume that there is enough memory on each processor to support replication of any array deemed necessary. (If this assumption does not hold, the strategy simply has to be modified to become more selective about choosing arrays or array dimensions for replication).

The second distribution function of a one-dimensional array may be an integer 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 quality measure of some constraint not being satisfied shows that the array has to be multicast over that dimension. An example of such an array is the array B in the example loop shown in Section 3, if A_2 is not sequentialized. A decision favoring replication is taken only if each time the array is 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 the cost for performing a computation on all processors can turn out to be less only if all the values needed for that computation are themselves replicated. For every one-dimensional array that is not replicated, the second distribution function is given the constant value of zero.

A decision to override the distribution function of an array dimension from partitioning to replication on a grid dimension is taken very sparingly. Replication is done only if no array element is written more than once in the program, and there are loops that involve sending values of elements from that array to processors along that grid dimension.

A simple example illustrating how our strategy combines constraints across loops is shown below:

```
do i = 1, n
    A(i, c1) = A(i, c1) + m * B(c2, i)
enddo
do i = 1, n
    do j = 1, i
        s = s + A(i, j)
    enddo
enddo
```

The first loop imposes constraints on the alignment of A_1 with B_2, since the same variable is being used as a subscript in those dimensions. It also suggests sequentialization of A_2 and B_1, so that regardless of the values of c_1 and c_2, the elements $A(i, c_1)$ and $B(c_2, i)$ may reside on the same processor. The second loop imposes a requirement that the distribution of A be cyclic. The compiler recognizes that the range of the inner loop is fixed directly by the value of the outer loop index, hence there would be a serious imbalance of load on processors carrying out the partial summation unless the array is distributed cyclically. These constraints are all consistent with each other and get accepted in steps 1, 4 and 3 respectively, of our strategy. Hence finally, the combination of these constraints leads to the following distributions – row-wise cyclic for A, and column-wise cyclic for B.

In general, there can be conflicts at each step of our strategy because of different constraints implied by various loops not being consistent with each other. Such conflicts get resolved on the basis of quality measures.

6 Study of Numeric Programs

Our approach to automatic data partitioning presupposes the compiler's ability to identify various dependences in a program. We are currently in the process of implementing our approach using Parafrase-2 [17], a source-to-source restructurer being developed at the University of Illinois, as the underlying tool for analyzing programs. Prior to that, we performed an extensive study using some well known scientific application programs to determine the applicability of our proposed ideas to real programs. One of our objectives was to determine to what extent a state-of-the-art parallelizing compiler can provide information about data references in a program so that our system may infer appropriate constraints on the distribution of arrays. However, even when complete information about a program's computation structure is available, the problem of determining an optimal data decomposition scheme is NP-hard. Hence, our second objective was to find out if our strategy leads to good decisions on data partitioning, given enough information about data references in a program.

Application Programs Five different Fortran programs of varying complexity are used in this study. The simplest program chosen uses the routine *dgefa* from the Linpack library. This routine factors a real matrix using gaussian elimination with partial pivoting. The next program uses the Eispack library routine, *tred2*, which reduces a real symmetric matrix to a symmetric tridiagonal matrix, using and accumulating orthogonal similarity transformations. The remaining three programs are from the Perfect Club Benchmark Suite [6]. The program *trfd* simulates the computational aspects of a two-electron integral transformation. The code for *mdg* provides a molecular dynamics model for water molecules in the liquid state at room temperature and pressure. *Flo52* is a two-dimensional code providing an analysis of the transonic inviscid flow past an airfoil by solving the unsteady Euler equations.

Methodology The testbed for implementation and evaluation of our scheme is the Intel iPSC/2 hypercube system. Our objective is to obtain good data partitionings for programs running on a 16-processor configuration. Obtaining the actual values of quality measures for various constraints requires us to have a knowledge of the costs of various communication primitives and of arithmetic operations on the machine. We use the following approximate function [10] to estimate the time taken, in microseconds, to complete a $Transfer$ operation on l bytes :

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

In our parallel code, we implement the ManyToManyMulticast primitive as repeated calls to the OneToManyMulticast primitive, hence in our estimates of the quality measures, the former functions is expressed in terms of the latter one. Each OneToManyMulticast operation sending a message to p processors is assumed to be $\lceil log_2 p \rceil$ times as expensive as a Transfer operation for a message of the same size. The time taken to execute a double precision floating point add or multiply operation is taken to be approximately 5 microseconds. The floating point division is assumed to be twice as expensive, a simple assignment (load and store) about one-tenth as expensive, and the overhead of making an arithmetic function call about five times as much. The timing overhead associated with various control instructions is ignored.

In this study, apart from the use of Parafrase-2 to indicate which loops were parallelizable, all the steps of our approach were simulated by hand. We used this study more as an opportunity to gain further insight into the data decomposition problem and examine the feasibility of our approach.

Results For a large majority of loops, Parafrase-2 is able to generate enough information to enable appropriate formulation of constraints and determination of their quality measures by our approach. There are some loops for which the information about parallelization is not adequate. Based on an examination of all the programs, we have identified the following techniques with which the underlying parallelizing compiler used in implementing our approach needs to be augmented.

- More sophisticated interprocedural analysis – there is a need for constant propagation across procedures [3], and in some cases, we need additional reference information about variables in the procedure or in-line expansion of the procedure.
- An extension of the idea of scalar expansion, namely, the expansion of small arrays. This is essential to get the benefits of our approach in which, like scalar variables, we also treat small arrays as being replicated on all processors. This helps in the removal of anti-dependence and output-dependence from loops in a number of cases, and often saves the compiler from getting "fooled" into parallelizing the smaller loops involving those arrays, at the expense of leaving the bigger parallelizable loops sequential.
- Recognition of reduction operators, so that a loop with such an operator may get parallelized appropriately. Examples of these are the addition, the min and the max operators.

Since none of these features are beyond the capabilities of Parafrase-2 when it gets fully developed (or for that matter, any state-of-the-art parallelizing compiler), we assume in the remaining part of our study that these capabilities are present in the parallelizing compiler supporting our implementation.

We now present the distribution schemes for various arrays we arrive at by applying our strategy to the programs after various constraints and the associated quality measures have been recorded. Table 2 summarizes the final distribution of significant arrays for all the programs. We use the following informal notation in this description. For each array, we indicate the number of elements along each dimension and specify how each dimension is distributed (cyclic/contiguous/replicated). We also indicate the number of processors on which that dimension is distributed. For the special case in which that number is one, we indicate that the dimension has been sequentialized. For example, the first entry in the table shows that the 2-D arrays A and Z consisting of 512 x 512 elements each are to be distributed cyclically by rows on 16 processors. We choose the *tred2* routine to illustrate the steps of our strategy in greater detail, since it is a small yet reasonably complex program which defies easy determination of "the right" data partitioning scheme by simple inspection. For the remaining programs, we explain on what basis certain important decisions related to the formulation of constraints on array distributions in them are taken, sometimes with the help of sample program segments. For the *tred2* program, we show the effectiveness of our strategy through actual results on the performance of different versions of the parallel program implemented on the iPSC/2 using different data partitioning strategies.

TRED2 The source code of *tred2* is listed in Figure 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 be supplied to the compiler. Also, corresponding to each statement in a loop that imposes some constraints, we have indicated which of the four categories (described in Section 3) the constraint belongs to.

Based on the alignment constraints, a component affinity graph (CAG), shown in Figure 3, is constructed for the program . Each node of a CAG [15] represents an array dimension, the weight on an edge denotes the communication cost incurred if the array dimensions represented by the two nodes corresponding to that edge are not aligned. The edge weights for our CAG are as follows:

$$
\begin{aligned}
c_1 &= N * \mathrm{OneToManyMulticast}(n^2/N, \langle N_I, N_J\rangle) \text{ (line 3)} \\
c_2 &= N * \mathrm{OneToManyMulticast}(n^2/N, \langle N_I, N_J\rangle) \text{ (line 3)} \\
c_3 &= N_I * \mathrm{OneToManyMulticast}(n/N_I, \langle N_J\rangle) \text{ (line 4)} \\
c_4 &= (n-1) * N_I * \mathrm{OneToManyMulticast}(n/2N_I, \langle N_J\rangle) \text{ (line 71)} + \\
&\quad (n-1) * N_I * \mathrm{OneToManyMulticast}(n/2N_I, \langle N_J\rangle) \text{ (line 77)}
\end{aligned}
$$

$$
\begin{aligned}
c_5 &= N_I * \mathrm{OneToManyMulticast}(n/2N_I, \langle N_J \rangle) \text{ (line 18)} + (n-1) * (n/2) * \mathrm{Transfer}(1) \text{ (line 59)} + \\
&\quad N_I * \mathrm{OneToManyMulticast}(n/N_I, \langle N_J \rangle) \text{ (line 83)} \\
c_6 &= (n-2) * N_I * \mathrm{OneToManyMulticast}(n/2N_I, \langle N_J \rangle) \text{ (line 53)} \\
c_7 &= (n-2) * (n/2) * N_I * \mathrm{OneToManyMulticast}(n/4N_I, \langle N_J \rangle) \text{ (line 42)}
\end{aligned}
$$

Along with each term, we have indicated the line number in the program to which the constraint corresponding to the quality measure may be traced. The total number of processors is denoted by N, while N_I and N_J refer to the number of processors along which various array dimensions are initially assumed to be distributed. Applying the algorithm for component alignment [15] 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.

In Step 2 of our strategy, none of the array dimensions is sequentialized because there are parallelization constraints favoring the distribution of both dimensions Z_1 and Z_2. In Step 3, the distribution functions for all array dimensions in each of the two classes are determined to be cyclic, because of numerous constraints on each dimension of arrays Z, D and E favoring cyclic distribution. The block size for the distribution for each of the aligned array dimensions is set to one. Hence, at the end of this step, the distributions for various array dimensions are:

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

Moving on to Step 4, we now determine the value of N_1, the value of N_2 simply gets fixed as N/N_1. By adding together the actual time measures given for various constraints, and the penalty measures of various constraints not getting satisfied, we get the following expression for execution time of the program (only the part dependent on N_1).

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

For $n = 512$, and for $N = 16$ (in fact, for all values of N ranging from 4 to 16), 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 quality 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:

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

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 is chosen for replication in Step 5. As specified above by the formal definitions of distribution functions, the data distribution scheme that finally emerges is – distribute arrays A and Z by *rows* in a *cyclic* fashion, distribute arrays D and E also cyclically, on all the N processors.

DGEFA The *dgefa* routine operates on a single n x n array A, which is factorized using gaussian elimination with partial pivoting. Let N_1 and N_2 denote, respectively, the number of processors over which the rows and the columns of the array are distributed. The loop computing the maximum of all elements in a column (pivot element) and the one scaling all elements in a column both yield execution time terms that show the communication time part increasing and the computation time part decreasing due to increased parallelism, with increase in N_1. The loop involving exchange of rows (due to pivoting) suggests a constraint to sequentialize A_1, i.e., setting N_1 to 1, to internalize the communication corresponding to the exchange of rows. The doubly-nested loop involving update of array elements corresponding to the triangularization of a column shows parallelism and potential communication (if parallelization is done) at each level of nesting. All these parallelizable loops are nested inside a single inherently sequential loop in the program, and the number of iterations they execute varies directly with the value of the outer loop index. Hence these loops impose constraints on the distribution of A along both dimensions to be cyclic, to have a better load balance. The compiler needs to know the value of n to evaluate the expression for execution time with N_1 (or N_2) being the only unknown. We present results for two cases, $n = 128$ and $n = 256$. The analysis shows that for the first case, the compiler would come up with $N_1 = 1, N_2 = 16$, where as for the second case, it would obtain $N_1 = 2, N_2 = 8$. Thus given information about the value of n, the compiler would favor column-cyclic distribution of A for smaller values of n, and grid-cyclic distribution for larger values of n.

TRFD The *trfd* benchmark program goes through a series of passes, each pass essentially involves setting up some data arrays and making repeated calls in a loop to a particular subroutine. We apply our approach to get the distribution for arrays used in that subroutine. There are nine arrays that get used, as shown in Table 3 (some of them are actually aliases of each other). To give a flavor for how these distributions get arrived at, we show some program segments below:

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

The first loop leads to the following constraints – alignment of $xrsiq_1$ with v_2, and sequentialization of $xrsiq_2$ and v_1. The second loop advocates alignment of xij_1 with v_2, sequentialization of v_1, and cyclic distribution of xij_1 (since the number of iterations of the inner loop modifying xij varies with the value of the outer loop index). The constraint on cyclic distribution gets passed on from xij_1 to v_2, and from v_2 to $xrsiq_1$. All these constraints together imply a column-cyclic distribution for v, a row-cyclic distribution for $xrsiq$, and a cyclic distribution for xij. Similarly, appropriate distribution schemes are determined for the arrays $xijks$ and xkl. The references involving arrays $xrspq, xijrs, xrsij, xijkl$ are somewhat complex, the variation of subscripts in many of these references cannot be analyzed by the compiler. The distribution of these arrays is specified as being contiguous to reduce certain communication costs involving broadcast of values of these arrays.

MDG This program uses two important arrays, var and vm, and a number of small arrays which are all replicated according to our strategy. The array vm is divided into three parts, which get used as different arrays named xm, ym, zm in various subroutines. Similarly, the array var gets partitioned into twelve parts which appear in different subroutines with different names. In Table 3 we show the distributions of these individual, smaller arrays. The arrays fx, fy, fz all correspond to two different parts of var each, in different invocations of the subroutine *interf*.

In the program there are numerous parallelizable loops in each of which three distinct contiguous elements of an array corresponding to var get accessed together in each iteration. They lead to constraints that the distributions of those arrays use a block size that is a multiple of three. There are doubly nested loops in subroutines *interf* and *poteng* operating over some of these arrays with the number of iterations of the inner loop varying directly with the value of the outer loop index. As seen for the earlier programs, this leads to constraints on those arrays to be cyclically distributed. Combined with the previous constraints, we get a distribution scheme in which those arrays are partitioned into blocks of three elements each, distributed cyclically. We show parts of a program segment, using a slightly changed syntax (to make the code more concise), that illustrates how the relationship between distributions of parts of var (x, y, z) and parts of vm (xm, ym, zm) gets established.

$$
\begin{array}{ll}
 & iw1 = 1;\ iwo = 2;\ iw2 = 3 \\
 & \text{do } 1000\ i = 1, nmol \\
 & \quad xm(i) = c1 * x(iwo) + c2 * (x(iw1) + x(iw2)) \\
 & \quad ym(i) = c1 * y(iwo) + c2 * (y(iw1) + y(iw2)) \\
 & \quad zm(i) = c1 * z(iwo) + c2 * (z(iw1) + z(iw2)) \\
 & \quad \vdots \\
 & \quad iw1 = iw1 + 3;\ iwo = iwo + 3;\ iw2 = iw2 + 3 \\
1000 & \text{continue}
\end{array}
$$

In this loop, the variables $iwo, iw1, iw2$ get recognized as induction variables and are expressed in terms of the loop index i. The references in the loop establish the correspondence of each element of xm with a three-element block of x, and yield similar relationships involving arrays ym and zm. Hence the arrays xm, ym, zm are given a cyclic distribution (completely cyclic distribution with a block size of one).

FLO52 This program has computations involving a number of significant arrays shown in Table 3. Many arrays declared in the main program really represent a collection of smaller arrays of different sizes, referenced in different steps of the program by the same name. For instance, the array w is declared as a big 1-D array in the main program, different parts of which get supplied to subroutines such as *euler* as a parameter (the formal parameter is a 3-D array w) in different steps of the program. In such cases, we always refer to these smaller arrays passed to various subroutines, when describing the distribution of arrays. Also, when the size of an array such as w varies in different steps of the program, the entry for size of the array in the table indicates that of the largest array.

A number of parallelizable loops referencing the 3-D arrays w and x access all elements varying along the third dimension of the array together in a single assignment statement. Hence the third dimension of each of these arrays is sequentialized. There are numerous parallelizable loops that establish constraints on all of the 2-D arrays listed in the table to have identical distributions. Moreover, the two dimensions of these array are aligned with the first two dimensions of all listed 3-D arrays, as dictated by the reference patterns in several other loops. Some other interesting issues are illustrated by the following program segments.

```
      do 20 j = 2, jl
          ⋮
          do 18 i = 1, il
              fs(i, j, 4) = fs(i, j, 4) + dis(i, j) * (p(i + 1, j) − p(i, j))
18        continue
          ⋮
20    continue
      ⋮
      do 38 j = 1, jl
          do 38 i = 2, il
              fs(i, j, 4) = fs(i, j, 4) + dis(i, j) * (p(i, j + 1) − p(i, j))
38    continue
```

These loops impose constraints on the (first) two dimensions of all of these arrays to have contiguous rather than cyclic distributions, so that the communications involving p values occur only "across boundaries" of regions of the arrays allocated to various processors. Let N_1 and N_2 denote the number of processors on which the first and second dimensions are distributed. The first part of the program segment specifies a constraint to sequentialize p_1, while the second one gives a constraint to sequentialize p_2. The quality measures for these constraints give terms for communication costs that vanish when N_1 and N_2 respectively are set to one. In order to choose the actual values of N_1 and N_2 (given that $N_1 * N_2 = 16$), the compiler has to evaluate the expression for execution time for different values of N_1 and N_2. This requires it to know the values of array bounds, specified in the above program segment by variables il and jl. Since these values actually depend on user input, the compiler would assume the real array bounds to be the same as those given in the array declarations. Based on our simplified analysis, we expect the compiler to finally come up with $N_1 = 16, N_2 = 1$. Given more accurate information or under different assumptions about the array bounds, the values chosen for N_1 and N_2 may be different. The distributions for other 1-D arrays indicated in the table get determined appropriately – in some cases, based on considerations of alignment of array dimensions, and in others, due to contiguous distribution on processors being the default mode of distribution.

Experimental Results on TRED2 Program Implementation We now show results on the performance of different versions of the parallel *tred2* program implemented on the iPSC/2 using different data partitioning strategies. The data distribution scheme selected by our strategy, as shown in Table 2 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.

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 [4] and [22], and hand-optimized the code. We first implemented the version that uses the data distribution scheme suggested by our strategy, i.e, row cyclic. 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 made in performance by sequentializing a class of dimensions, i.e., by not distributing A and Z in a blocked (grid-like) manner, and also gains made 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 4. 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 as shown in Figure 7 (b). For the smaller data size (Figure 7 (a)), the scheme using column distribution of arrays works slightly better when fewer processors are being used. Our approach does identify a number of constraints that favor the column distribution scheme, they just 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 for this program that our approach is able to take the right decisions regarding certain key parameters of data distribution, and does suggest a data partitioning scheme that leads to good performance.

7 Conclusions

We have presented a new approach, the constraint-based approach, to the problem of determining suitable data partitions for a program. Our approach is quite general, and can be applied to a large class of programs having references that can be analyzed at compile time. We have demonstrated the effectiveness of our approach for real-life scientific application programs. We feel that our major contributions to the problem of automatic data partitioning are:

- *Analysis of the entire program*: We look at data distribution from the perspective of performance of the entire program, not just that of some individual program segments. The notion of constraints makes it easier to capture the requirements imposed by different parts of the program on the overall data distribution. Since constraints associated with a loop specify only the basic minimal requirements on data distribution, we are often able to combine constraints affecting different parameters relating to the distribution of the same array. Our studies on numeric programs confirm that situations where such a combining is possible arise frequently in real programs.
- *Balance between parallelization and communication considerations*: We take into account both communication costs *and* parallelization considerations so that the overall execution time is reduced.
- *Variety of distribution functions and relationships between distributions* : Our formulation of the distribution functions allows for a rich variety of possible distribution schemes for each array. The idea of relationship between array distributions allows the constraints formulated on one array to influence the distribution of other arrays in a desirable manner.

Our approach to data partitioning has its limitations too. There is no guarantee about the optimality of results obtained by following our strategy (the given problem is NP-hard). The procedure for compile-time determination of quality measures of constraints is based on a number of simplifying assumptions. For instance, we assume that all the loop bounds and the probabilities of executing various branches of a conditional statement are known to the compiler. For now, we expect the user to supply this information interactively or in the form of assertions. In the future, we plan to use profiling to supply the compiler with information regarding how frequently various basic blocks of the code are executed.

As mentioned earlier, we are in the process of implementing our approach for the Intel iPSC/2 hypercube using the Parafrase-2 restructurer as the underlying system. We are also exploring a number of possible extensions to our approach. An important issue being looked 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 also plan to look at the problem of interprocedural analysis, so that the formulation of constraints may be done across procedure calls. Finally, we are examining how better estimates could be obtained for quality measures of various constraints in the presence of compiler optimizations like overlap of communication and computation, and elimination of redundant messages via liveness analysis of array variables [9].

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. There are a number of issues that need to be resolved through further research before a truly automated, high quality system can be built for data partitioning on multicomputers. However, we believe that the ideas presented in this paper do lay down an effective framework for solving this problem.

Acknowledgements

We wish to express our sincere thanks to Prof. Constantine Polychronopoulos for giving us access to the source code of the Parafrase-2 system, and for allowing us to build our system on top of it. We also wish to thank the referees for their valuable suggestions.

References

[1] 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.

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

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

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

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

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

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

[8] M. Gupta and P. Banerjee. Compile-time estimation of communication costs in multicomputers. Technical Report CRHC-91-16, University of Illinois, May 1991.

[9] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report TR90-149, Rice University, February 1991. To appear in J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*, Elsevier, 1991.

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

[11] 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.

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

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

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

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

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

[17] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors. In *Proc. 1989 International Conference on Parallel Processing*, August 1989.

[18] 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.

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

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

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

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

Primitive	*Cost on Hypercube*
Transfer(m)	O(m)
Reduction(m, seq)	O($m*$ log $num(seq)$)
OneToManyMulticast(m, seq)	O($m*$ log $num(seq)$)
ManyToManyMulticast(m, seq)	O($m*$ $num(seq)$)

Table 1: Costs of communication primitives on the hypercube architecture

Program	*Arrays*	*Size*	*Distribution*	*No. of Processors*
tred2	A, Z	512 x 512	cyclic x seq	16 x 1
	D, E	512	cyclic	16
dgefa	A	256 x 256 or 128 x 128	cyclic x cyclic	2 x 8 or 1 x 16
	v	40 x 40	seq x cyclic	1 x 16
trfd	$xrsiq, xijks$	40 x 40	cyclic x seq	16 x 1
	xij, xkl	40	cyclic	16
	$xrspq, xijrs, xrsij, xijkl$	672400	contiguous	16
	x, y, z	1029	cyclic blocks of 3	16
mdg	vx, vy, vz	1029	cyclic blocks of 3	16
	fx, fy, fz	1029	cyclic blocks of 3	16
	xm, ym, zm	343	cyclic	16
	$w, w1$	194 x 34 x 4	contig x seq x seq	16 x 1 x 1
	x	194 x 34 x 2	contig x seq x seq	16 x 1 x 1
	wr, ww	98 x 18 x 4	contig x seq x seq	16 x 1 x 1
	fw, dw, fs	193 x 33	contig x seq x seq	16 x 1 x 1
	vol, dtl, p	194 x 34	contiguous x seq	16 x 1
	$radi, radj$	194 x 34	contiguous x seq	16 x 1
flo52	$dp, ep, dis2, dis4$	193 x 33	contiguous x seq	16 x 1
	$count1, count2, res, fsup$	501	contiguous	16
	$a0, s0$	193	contiguous	16
	$b0$	33	replicated	16
	xp, yp, cp	385	contiguous	16
	$d1, d2, d3, xs, ys$	385	contiguous	16
	$qs1, cs1, qsj, csj$	193	contiguous	16
	xx, yx, sx, xxx, yxx	194	contiguous	16

Table 2: Distribution of arrays for various programs

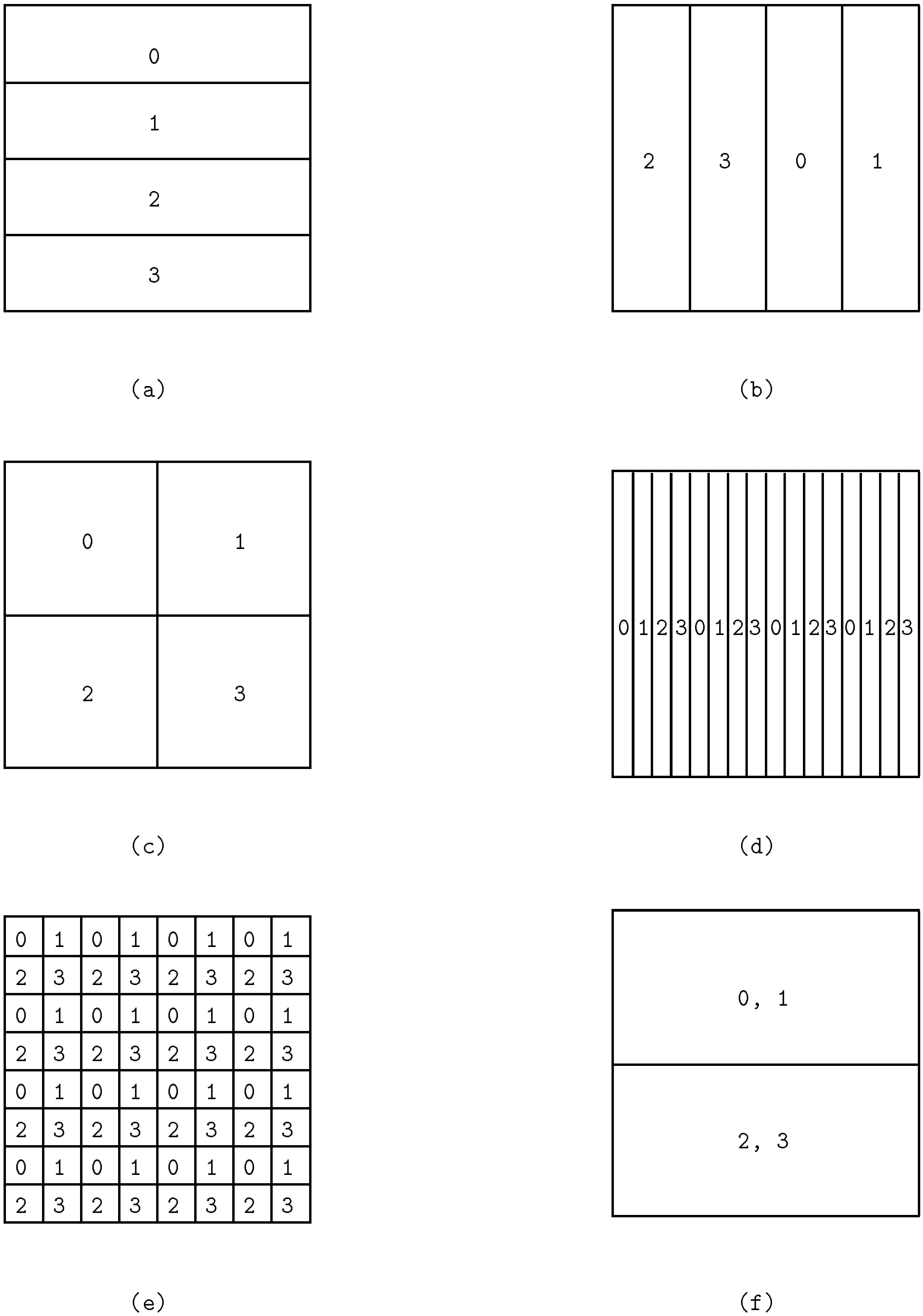

Figure 1: Different data partitions for a $16 * 16$ array

```
1   DO 5 I = 1, N
2       DO 3 J = I, N
3         Z(J,I) = A(J,I)                  2, 1
4       D(I) = A(N,I)                      2, 1
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)
13      DO 14 K = 1, L
14        SCALE = SCALE + DABS(D(K))       1
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)                  2, 1
19          Z(I,J) = 0.0D0                 1
20          Z(J,I) = 0.0D0                 1
21      CONTINUE
22      GO TO 62
23      DO 25 K = 1, L
24          D(K) = D(K) / SCALE            1
25          H = H + D(K) * D(K)            1
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                     1
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)      1
42              E(K) = E(K) + Z(K,J) * F   2, 1
43          CONTINUE
44          E(J) = G
45      CONTINUE
46      F = 0.0D0
47      DO 50 J = 1, L
48          E(J) = E(J) / H                1
49          F = F + E(J) * D(J)            1
50      CONTINUE
51      HH = F / (H + H)
52      DO 53 J = 1, L
53        E(J) = E(J) - HH * D(J)          2, 1
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)  1
59          D(J) = Z(L,J)                  2
60          Z(I,J) = 0.0D0                 1
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)                    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                2, 1
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, 1
76          DO 78 K = 1, L
77            Z(K,J) = Z(K,J) - G * D(K)   2
78      CONTINUE
79      DO 80 K = 1, L
80        Z(K,I) = 0.0D0                   1
81  CONTINUE
82  DO 85 I = 1, N
83      D(I) = Z(N,I)                      2, 1
84      Z(N,I) = 0.0D0                     1
85  CONTINUE
86  Z(N,N) = 1.0D0
87  E(1) = 0.0D0
88  END
```

Figure 2: Fortran code for TRED2 routine

Figure 2: Fortran code for TRED2 routine

Figure 3: Component affinity graph for TRED2

Figure 4: Performance of different versions of TRED2 on the iPSC/2 system