Automatic Data Partitioning on Distributed Memory Multiprocessors*

Manish Gupta and Prithviraj Banerjee

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
Urbana, IL 61801

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.

1 Introduction

The area of parallelizing compilers for distributed-memory multiprocessors (multicomputers) has seen considerable research activity during the last few years [2, 5, 13, 11]. In this approach, the compiler accepts a program written in a sequential or a shared-memory parallel language, and based on the programmer-specified partitioning of data, generates a parallel program to be executed on that machine. Thus the programmer is freed from the burden of managing communication among tasks explicitly.

Our use of the term "parallelizing compiler", however, is somewhat misleading, because all parallelization decisions in these systems 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 task of determining a good data partitioning scheme can be extremely difficult and tedious, and hence there is a need to automate this task.

*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.

Finding the optimal data partitioning scheme automatically has been known to be a difficult problem. Various simple versions of it have been proved to be NP-hard [8, 6].

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 [10] 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. Socha [12] presents techniques for data partitioning for single-point iterative programs. 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 [6] 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 [7]. 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*, to the problem of automatic data partitioning. 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.

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 is represented by the tuple $(p_1, p_2, \ldots, p_D), 0 \leq p_k \leq N_k - 1$ for $1 \leq k \leq D$. 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. 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

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

where the square parentheses surrounding 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.

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.

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 can be explained by looking at the basic compilation rule [2] followed by all parallelization systems for multicomputers – the processor responsible for a computation is the one that owns the data item being assigned a value in that computation. Whenever a computation involves the use of a value not available locally on that processor, there is a need for interprocessor communication. Thus, 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 different kinds of constraints as shown below:

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

```
do i = 1, n
      A(i, c1) = F(B(c2 * i))
enddo
```

Figure 1: Example illustrating the relationship between distributions

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, the reference pattern shown in Figure 1 suggests that A_1 should be aligned with B_1, and A_2 should be sequentialized. Secondly, it suggests the following distribution function for B_1, in terms of that for A_1.

$$\begin{aligned} f^1_B(c_2 * i) &= f^1_A(i) \\ f^1_B(i) &= f^1_A(\lfloor i/c_2 \rfloor) \end{aligned} \quad (1)$$

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, and each constraint specifies only the basic minimal requirements on distribution. Hence the parameters regarding the distribution of an array left unspecified by a constraint may be selected by combining that constraint with other constraints specifying those parameters. Each such combination leads to an improvement in the distribution scheme.

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.

3.1 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 patterns 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. The communication time component is determined by identifying the primitives needed to carry out the inter-processor communication and determining the message sizes. We have developed a methodology for compile-time estimation of communication costs incurred by the program [4]. These 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. We shall briefly describe our methodology here, further details can be found in [4].

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 introduced by Li and Chen [7]; they 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 [7]. We assume that the following communication routines are supported by the operating system or by the runtime 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

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

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.

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 the values of 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 [4].

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 cost functions associated with aggregate communication primitives rather than those associated with repeated Transfer operations. Apart from developing an algorithm to take this decision, we have also identified program transformations 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 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. Determine the "adjustment" terms due to the remaining references that modify the communication terms obtained by considering one of the isomorphic RHS references together with the LHS reference in step 3.

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 the data references and the associated quality measures obtained by applying our methodology.

Example 1 :

do $i = 1, n_1$
  do $j = 1, n_2$
    $A(i,j) = \mathcal{F}(B(j, c_1 * i))$
  enddo
enddo

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

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

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 2 :

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

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.
  $Penalty = ManyToManyMcast(n_1 * n_2/(N_I * N_J), \langle N_I, N_J \rangle)$
- 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.
  $$Penalty = 2 * Transfer(n_1 * n_2/(N_I * N_J))$$
- Distribute B_2 in a contiguous manner.
  The analysis is similar to that for the previous case.
  $$Penalty = 2 * Transfer(n_1 * n_2/(N_I * N_J))$$
- 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.
  $$Commn.\ Time = 2 * (N_I > 1) Transfer(n_2/N_J)$$
  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.
  $$Commn.\ Time = 2 * (N_J > 1) Transfer(n_1/N_I)$$

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.

4 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 – the 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.

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 [6]. 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 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 dimensions, there is no array 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 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 when the finally chosen grid topology has two real dimensions. The other kind

```
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
```

Figure 2: Example illustrating the combining of constraints

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 loop shown in Figure 1, 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 in Figure 2. 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.

5 Results

We are currently in the process of implementing our approach using Parafrase-2 [9] (a source-to-source restructurer being developed at the University of Illinois) as our basic tool for identifying the dependencies in a program. Earlier, to determine the applicability of our proposed ideas to real programs, we performed a study using five Fortran programs taken from the Linpack and Eispack libraries and the Perfect Benchmarks. We hand-simulated our approach and found that reasonable constraints on data distribution could be identified for a large majority of the loops. Our strategy lead to the selection of good data partitioning schemes for these programs. These results are described in [3].

6 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 reference patterns 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 :

- *The notion of constraints on data distribution*: Each constraint specifies only the basic minimal requirements on data distribution indicated by a loop. By avoiding over-specification of requirements, we are often able to combine different constraints that affect 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.

- *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. This is one of the most important advantages that our approach offers over others that have so far been proposed. The quality measures associated with constraints capture the relative importance to be given to different program segments depending on the estimated amount of time spent by the program in those segments.

- *Balance between parallelization and communication considerations*: We take into account both communication costs *and* parallelization considerations so that the overall execution time is reduced.

- *Methodology for compile-time estimation of communication costs*: The methodology developed by us for compile-time determination of quality measures of various communication constraints is general enough to be used by any parallelization system for its performance estimation module.

- *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 propagate to other arrays related to the original array.

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 estimation of communication costs is based on a number of simplifying assumptions which may not always hold, leading to inaccuracies in the estimates generated. In spite of these limitations, our approach does seem to work well for numerous real programs that we have examined.

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

[3] M. Gupta and P. Banerjee. Demonstration of data decomposition techniques in parallelizing compilers for multicomputers. To appear in IEEE Transactions on Parallel and Distributed Systems.

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

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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