2

May 1991

UILU-ENG-91-2226
CRHC-91-16

Center for Reliable and High-Performance Computing

COMPILE-TIME ESTIMATION OF COMMUNICATION COSTS IN MULTICOMPUTERS

Manish Gupta
Prithviraj Banerjee

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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-91-2226 CRHC-91-16	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL *(If applicable)* N/A	7a. NAME OF MONITORING ORGANIZATION ONR, NSF-PYI, NASA
6c. ADDRESS *(City, State, and ZIP Code)* 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS *(City, State, and ZIP Code)* Arlington VA 22217-5000 Washington DC 20550 Washington VA 20546
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 7a 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-91-J-1096 NSF MIP 86-57563 PYI NASA NAG 1-613
8c. ADDRESS *(City, State, and ZIP Code)* 7b.	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Compile-time Estimation of Communication Costs in Multicomputers

12. PERSONAL AUTHOR(S) Gupta, Manish and Banerjee, Prithviraj

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO ______	91-05-22	22

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	compilers, distributed memory, data partitioning, sequential program references, multicomputers

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

An important problem facing numerous research projects on parallelizing compilers for distributed memory machines is that of automatically determining a suitable data partitioning scheme for a program. Any strategy for automatic data partitioning needs a mechanism for estimating the performance of a program under a given partitioning scheme, the most crucial part of which involves determining the communication costs incurred by the program. In this paper, we decribe a methodology for estimating the communication costs at compile-time as functions of the numbers of processors over which various arrays are distributed. We also describe a strategy, along with its theoretical basis, for making program transformations that expose opportunities for combining of messages, leading to considerable savings in the communication costs. For certain loops with regular dependences, the compiler can detect the possibility of pipelining, and thus estimate communication costs more accurately than it could otherwise. These results are of great significance to any parallelization system supporting numeric applications on multicomputers. In particular, they lay down a framework for effective synthesis of communication on multicomputers from sequential program references.

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

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

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

Compile-Time Estimation of Communication Costs in 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. Any strategy for automatic data partitioning needs a mechanism for estimating the performance of a program under a given partitioning scheme, the most crucial part of which involves determining the communication costs incurred by the program. In this paper, we decribe a methodology for estimating the communication costs at compile-time as functions of the numbers of processors over which various arrays are distributed. We also describe a strategy for making program transformations that expose opportunities for combining of messages, leading to considerable savings in the communication costs. For certain loops with constant dependences, the compiler can detect the possibility of pipelining, and thus estimate communication costs more accurately than it could otherwise. These results are of great significance to any parallelization system supporting numeric applications on multicomputers. In particular, they lay down a framework for effective synthesis of communication on multicomputers from sequential program references.

*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. These machines offer significant advantages over the shared memory multiprocessors with regard to cost and scalabilty, however, they are also more difficult to program. Much of that difficulty is due to their lack of a single global address space. Hence, the last few years have seen considerable research effort [9, 10, 16, 3, 14] aimed at providing a shared name space to the programmer, with the task of generating messages relegated to the compiler. Most of these parallelization systems accept a program written in a sequential or shared-memory language augmented with annotations specifying distribution of data, and generate the target program for the multicomputer.

It is widely accepted that with such parallelization systems, the most challenging step in programming for multicomputers is determining the "right" data partitioning scheme. It is the data partitioning scheme which determines whether or not independent computations are executed on different processors, and when interprocessor communication is required to carry out a computation. Since the performance of a program depends critically on the data distribution scheme, most of the parallelization systems leave it to the programmer to decide. However, the task of determining a good data distribution scheme manually can be extremely difficult and tedious. Recently several researchers [13, 11, 2, 8] have addressed this problem of automatically determining a proper data partitioning scheme, or of providing help to the user in this task.

Any strategy for automatic data partitioning needs a mechanism for estimating the performance of a program to evaluate a data partitioning scheme (or some parameters related to the scheme). The most crucial part of that involves determining the communication costs incurred while executing the program. In this paper, we decribe a methodology for estimating the communication costs as functions of the numbers of processors over which various arrays are distributed.

Related Work Some of the ideas underlying our approach to estimating communication costs are closely related to those developed for automatically generating communication from sequential or shared memory language programs for multicomputers. The Kali system [10] and the Parascope system [2] perform analysis of source references to determine the communication induced by the data partitioning scheme for each loop. Li and Chen [11] introduce the notion of matching source program references with syntactic patterns associated with aggregate communication routines to generate communication in the Crystal system. They also define a metric based on those communication primitives to estimate communication costs. In these systems, the support for automatic data partitioning is based on trying out different possibilities regarding certain "standard" array partitionings, and using a performance estimation module to return estimates of the communication cost. The system uses these estimates to select the data partitioning scheme that leads to the best performance.

Motivation for this work We approach the problem of automatic data partitioning in a different manner, and correspondingly need a slightly more sophisticated mechanism for estimating communication costs. We deal with *constraints* [8], which represent only partial specifications of a data partitioning scheme. Hence, we need estimates of communication costs in a parameterized form, i.e., as functions of data distribution parameters not specified by the constraints. We do not attempt to explicitly evaluate different partitioning schemes because such a method would be prohibitively expensive if the number of candidate schemes is large. Instead, we record constraints indicated by different loops in the program, and then combine selected constraints in a consistent manner. Each constraint specifies only a few parameters pertaining to the distribution of an array, and the parameters left unspecified by one constraint may be selected by combining that constraint with others specifying those parameters. We have performed a study of real-life scientific application programs (Fortran programs taken from the Linpack and Eispack libraries and the Perfect Club Benchmark Suite [4]), and have found the idea of using constraints to be extremely useful [7].

Different constraints may impose conflicting requirements on the distribution of various arrays. 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 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. The ideas presented in this paper were developed as part of our effort to come up with a methodology to determine the quality measures of various constraints at compile-time. However, the results we obtain are of great significance to any parallelization system for multicomputers, since our techniques can be applied to generate communication using appropriate primitives, and also to estimate the cost of communication.

Organization of the Paper The rest of this paper is organized as follows. Section 2 describes our abstract machine model, how distribution functions are specified for arrays, and introduces some terms that we shall use throughout the paper. Section 3 presents an algorithm to determine accurately when communication may be taken out of a loop. In that section, we also present a methodology for applying transformations on loops to expose opportunities for combining messages. Section 4 describes how, given an assignment statement appearing in an arbitrary number of loops (which need not be perfectly nested inside one another), the compiler estimates the costs of communications required to carry out that computation. The procedure for that as described in this section assumes that there is only a single reference to each array appearing on the right hand side of the assignment statement. Section 5 describes how multiple references to an array in a single assignment statement are handled. Section 6 describes how the compiler detects opportunities for pipelining a given computation and modifies the estimates of communication costs to take into account the speedup due to pipelining. Finally, conclusions along with a discussion on future work are presented in Section 7.

2 Background and Terminology

Distribution of arrays on target machine The abstract target machine we assume is a 2-D grid of $N_1 * N_2$ processors. Such a topology can easily be embedded on almost any distributed memory machine. To simply the notation describing distribution of arrays, we regard the topology conceptually as a D-dimensional grid (D is the maximum dimensionality of any array used in the program), and later set the values of $N_3, \ldots, N_D$ to 1.

We describe the distribution of an array by using a separate distribution function for each of its dimensions [7]. Each array dimension gets mapped to a unique dimension of the processor grid (if that grid dimension has only one processor, the array dimension is said to be *sequentialized*) and is distributed in a cyclic or contiguous manner, or it may be replicated. The distribution function chosen finally for an array dimension conveys all of this information.

The first aspect which gets determined regarding data distribution is the *alignment* of array dimensions. Two array dimensions are said to be aligned if they get distributed on the same processor grid dimension. We describe how communication costs incurred while executing a loop are estimated, given information about the alignment of array dimensions. Determining the quality measure of a constraint on alignment of two array dimensions then simply becomes a matter of evaluating communication costs for the cases when the particular array dimensions are aligned and when they are not, and taking the difference of the two. The communication costs are expressed as functions of the number of processors on which various aligned array dimensions are distributed, and sometimes, also of the method of partitioning, contiguous or cyclic.

Communication Primitives Our system uses array reference patterns to determine which communication routines out of a given library of routines best realize the required communication for various loops, a notion first introduced by Li and Chen [11] for synthesis of communication. We have made significant extensions to the ideas presented in their work, as we shall show later. 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).

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

- *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. An important aspect of the definition of primitives other than Transfer is that the processors over which the multicast (or reduction) operation is carried out include the processor from which multicast takes place (or the processor on which the reduced data is placed). This definition imposes the requirement that the cost functions Reduction, OneToManyMulticast, and ManyToManyMulticast return a zero when the number of processors involved is one. This is a convenient way of ensuring that any term for costs of communication among processors over which an array dimension is distributed evaluates to zero when that array dimension is sequentialized. Handling this boundary case correctly is very important since we always obtain communication-cost expressions in terms of the numbers of processors over which those dimensions are distributed, and eventually, all except for a maximum of two dimensions of each array are sequentialized. For communication-cost terms involving the Transfer function, we shall show later how this boundary case is handled correctly.

Subscript Types For establishing constraints and determining their quality measures, we limit our attention to statements appearing in loops that involve assignment to arrays, since all scalar variables are replicated on all processors. 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 *dependence-level*, which is the level of the innermost loop on which the subscript expression depends, i.e., 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.

Dependence Information The information regarding all data dependences in a program is available to the compiler in the form of a data dependence graph. Associated with each dependence edge representing a dependence from statement S1 to statement S2, both nested in n loops, is a direction vector $(d_1, d_2, \ldots, d_n)$, where $d_i \in \{<, =, >, \leq, \geq, \neq, *\}$ [15]. The direction vector describes the direction of dependence for each loop, with d_1 describing the direction of dependence for the outermost loop, and successive components of the vector describing directions for the inner loops. The forward direction "<" implies that the dependence is from an earlier to a later iteration of the corresponding loop, "=" implies that the dependence does not cross an iteration boundary, and ">" means that the dependence crosses an iteration boundary backwards. The other four directions are simply combinations of these three basic directions, and are associated with imprecision in data dependence tests. In Fortran *do* loops, a backward direction can occur only if there is a forward direction in an outer loop. Another way this fact is often expressed is that every legal direction vector has to be non-negative, where the directions "<","=", and ">" are expressed as $+1, 0$ and -1, respectively.

For simplicity, the only dependences we consider in this paper are the data dependences in a program. Control dependences can be handled by converting them to data dependences [1], or by using a different program representation, the *program dependence graph* [5] instead of a data dependence graph.

3 Combining of Communication

On current multicomputers, the start-up cost for sending a message is much greater than the per-element cost. Hence combining messages, particularly those being sent in different iterations of a loop, is an important optimization goal in most parallelization systems for multicomputers [9, 16, 3].

Our approach to combining messages builds up on the idea of using the loop distribution transformation [12], which has been used by other researchers [2, 6] on loops with cross-iteration dependences for achieving the same goal. We extend that work, and show how a combination of loop distribution for completely parallel (*doall*) loops and loop permutation making a parallel loop the innermost loop enables the use of aggregate communication primitives for as many loops as possible. First, let us see how loop distribution helps in combining messages. Consider a loop in which a given statement computes the values of data items that are used by another statement in the same loop, and this leads to communication during each iteration. If the loop can legally be distributed over these statements, the communication can now be done between the resulting loops.

The first step taken by the compiler after building the data dependence graph is to determine for each loop whether it can be concurrentized or not. A loop at level k is marked sequential if there is at least one dependence edge between nodes representing statements in that loop that has a direction vector of $(d_1, d_2, \ldots, d_n)$ satisfying the following properties: (i) $d_k \in \{<, \leq, \neq, *\}$ (ii) $\forall i$ in $[1, k-1], d_i \notin \{<, >, \neq\}$. These conditions check for the existence of a cross-iteration dependence that it is not satisfied by the sequentiality of an outer loop. The significance of ignoring a dependence in the "$>$" direction at level k lies in the fact that a "$>$" direction can occur only when there is a "$<$" direction occuring at an outer level [15]. In the rest of the paper, we shall use the term "parallel loop" to mean a loop marked concurrent in this step.

For each loop (say, at level k) that is marked sequential, we conceptually define a graph G_k to represent the relevant part of the complete dependence graph for that loop. The nodes included in G_k are those corresponding to statements in the body of that loop. An edge between these nodes present in the original dependence graph is included in G_k if and only if the associated direction vector $(d_1, d_2, \ldots, d_n)$ satisfies the following properties: (i) $d_k \neq$ "$>$" (ii) $\forall i$ in $[1, k-1], d_i \notin \{<, >, \neq\}$. These conditions once again ensure that any dependence that is satisfied by the sequentiality of an outer loop is ignored. The compiler identifies the strongly connected components in each such graph G_k "built" for a loop at level k.

Let us now consider a statement S1 of the form array_element = rhs_expression appearing inside a loop at level k. Each array appearing in the rhs_expression could potentially be involved in the interprocessor communication required to carry out this computation. In this section, we only describe how to determine for each array whether the communication required (if any) can be taken out of the given loop. How that knowledge helps us estimate the actual cost of communication shall be explained in the following sections. The algorithm to determine whether communication involving an array B appearing in the rhs_expression can be combined (taken out of the loop) consists of the following steps:

1. If the loop is marked parallel, then return with the answer that communication can be combined.

2. Check all the dependence edges in the graph G_k incident on the node corresponding to S1, and coming from a node in the same strongly connected component as the given node. If any of these dependence edges is due to the array B or any variable being used in its subscripts (when the subscript is of the type *variable*), then the communication cannot be combined, otherwise it can be combined.

The existence of a dependence edge due to array B incident on the node for S1 implies that the values of some elements of B that may be involved in communication are computed within the loop. Further, if that edge comes from a node belonging to the same strongly connected component, the communication has to be

```
do $i = 1, n$
    do $j = 2, n$
        $A(i,j) = \mathcal{F}(A(i,j-1), B(i,j-1))$        (S1)
        $B(i,j) = \mathcal{F}(A(i,j), A(i,j-1))$          (S2)
    enddo
    $D(i) = \mathcal{F}(A(i,n), D(i-1))$                  (S3)
enddo
```

Figure 1: Original program segment

done within that loop to honor the data dependence. However, if the edge comes from a node belonging to a different strongly connected component, by distributing the loop over these components, we can take the communication out of the loop. If there is no dependence edge in G_k due to B incident on the node, there is no other statement in the loop that writes into an element of B before it gets used at S1. In that case, communication involving the elements of B can be done before executing the loop. Before taking a decision on combining communication, another condition we check for is that the value of any variable that governs which element of B gets used in S1 should also be available outside the loop. That is the reason for our checking the dependence edges due to such variables also in step 2 of the algorithm.

The need for step 1 arises when a parallel loop appears outside another loop that has communication taking place inside it, or when there are intra-iteration dependences within the parallel loop. Unless the program is transformed, each iteration of the parallel loop has communication taking place inside it. Step 1 is based on the observation that in such cases, the program can always be transformed so that no communication needs to be done inside the parallel loop. The two transformations that ensure such a behavior are:

1. Distribution of a parallel loop over statements constituting its body.

2. Loop permutation that makes a parallel loop the innermost loop in a perfectly nested loop structure.

It is well known that these transformations are always valid [15]. After the distribution of a parallel loop over various statements in its body (that may themselves be loops), whenever that loop surrounds another loop that happens to have repeated communications taking place inside it, we transform the loop structure so that the parallel loop becomes the innermost loop in that nest. Once a parallel loop has been made the innermost loop, by distributing it whenever necessary over statements inside it, communication can be taken out of that loop.

The usefulness of these transformations is illustrated by the program segment shown in Figure 1. The dependence graph for this program segment is shown in Figure 2. Without any transformation, none of the loops can be parallelized. However, in the graph G_1 that we build corresponding to the i-loop, the j-loop

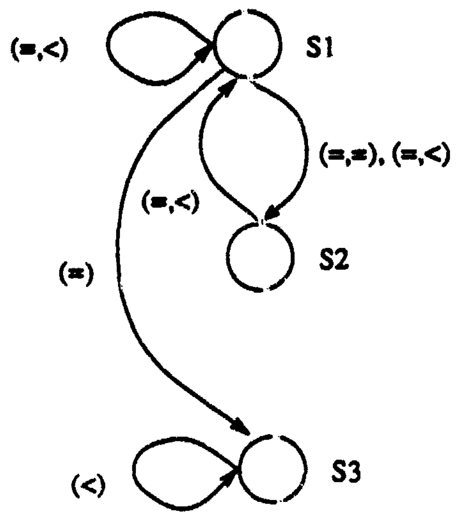

Figure 2: Dependence graph for the example program segment

```
do $j = 2, n$
    do $i = 1, n$
        $A(i,j) = \mathcal{F}(A(i, j-1), B(i, j-1))$        (S1)
    enddo
    do $i = 1, n$
        $B(i,j) = \mathcal{F}(A(i,j), A(i, j-1))$        (S2)
    enddo
enddo
do $i = 1, n$
    $D(i) = \mathcal{F}(A(i,n), D(i-1))$        (S3)
enddo
```

Figure 3: Transformed program segment

statements and the statement S3 belong to different strongly connected components, and hence the i-loop can be distributed over them. In the resultant perfectly nested loop structure with statements S1 and S2, the i-loop is marked parallel and the j-loop is marked sequential. Hence, we transform this loop structure to make the i-loop the innermost loop. The communication involving the A values in statement S2 still occurs inside the i-loop. By distributing the i-loop further over statements S1 and S2, that communication can now be taken out of the inner loop. The final program segment corresponding to which the parallel version incurs much lower communication overhead is shown in Figure 3.

4 Communication Costs for an Assignment Statement

For each statement in a loop in which the assignment to an array A uses values of an array B or A itself, we express estimates of the communication costs as functions of the numbers of processors on which various array dimensions are distributed. If the expression on the right hand side (*rhs*) of the assignment statement involves multiple arrays, the same analysis is repeated for each array. In this section, we describe how communication costs corresponding to a single reference to a *rhs* array are estimated. Section 5 describes how we handle the case when the *rhs* has multiple references to the same array.

If the *rhs* array is different from the *lhs* array, the first task at hand is to determine which dimensions of the two arrays should get aligned. The algorithm for that consists of the following steps. First, we match all pairs of dimensions for which the subscripts are of the type *index* and both subscripts correspond to the same loop index (indicated by identical values of *dependence-level* recorded with each subscript). In that step, we also match dimensions corresponding to subscripts with the same constant value. Next, we match dimensions for which one or both the subscripts are of the type *variable*, provided they have the same value of *dependence-level*. The remaining subscripts are now simply paired up on the basis of the order in which they are seen in a left-to-right scan of subscripts. If the number of dimensions in the two arrays are different, some dimensions in the array with greater dimensionality get matched to a "missing" dimension.

4.1 Identification of Communication Terms

To estimate the communication costs associated with an assignment statement, we first identify terms representing a "contribution" from each pair of subscripts for the aligned dimensions. Whenever at least one subscript in that pair is of the type *index* or *variable*, the term represents a contribution from one of the enclosing loops identified by the value of *dependence-level*. A loop enclosing a statement makes a contribution to its costs even if no subscript in any array reference in that statement identifies it as the innermost loop in which it changes its value. Once all the contribution terms have been obtained, we compose them together using definite rules to determine the overall communication costs involved in executing that statement in the program.

Determining these contributions to communication costs is based on the idea, proposed by Li and Chen [11], of matching reference patterns in the program to those associated with various communication primitives. We are able to handle a much more comprehensive set of patterns than those described in [11]. For contributions from a loop, we first apply the algorithm described in the previous section to determine whether communication can be taken out of the loop. For cases when that can be done, Table 2 lists the terms contributed by the pair of subscripts corresponding to aligned dimensions. We assume that all loops have been normalized, i.e., they have a stride of one, and that each loop index used as a subscript in an array

LHS	*RHS*	*Commn. Term Type*	*Parameters*
i	i	SizeChange	$d = n_i/N_I$
i	$i \pm c_1$	(i) SizeChange (ii) ($N_I > 1$) Transfer	$d = n_i/N_I$ $d = c_1$
i	var_i	ManyToManyMulticast	$d = n_i/N_I, p = N_I$
var_i	i	ManyToManyMulticast	$d = n_i/N_I, p = N_I$
var_i	var_i	ManyToManyMulticast	$d = n_i/N_I, p = N_I$
i or var_i	c_1 or missing	OneToManyMulticast	$d = 1, p = N_I$
c_1 or missing	i	Reduction, or Transfer	$d = n_i/N_I, p = N_I$ $product = N_I - 1, d = n_i/N_I$
c_1 or missing	var_i	Reduction, or Transfer	$d = n_i/N_I, p = N_I$ $product = N_I - 1, d = n_i/N_I$
i or var_i	j or var_j	ManyToManyMulticast	$d = n_i/N_I, p = N_I$

Table 2: Terms for Aggregate Communication

reference varies so as to sweep over the entire range of elements along that dimension. The selection of the communication primitive used in a communication term is based on the kind of subscripts appearing in the left hand side (*lhs*) and the right hand side (*rhs*) references. We use i and j to denote different loop indices, c_1 and c_2 to denote different constants; var_i and var_j represent subscripts of the type *variable* with their *dependence-level* values corresponding to loops with indices i and j respectively. In the entries for parameters of various communication terms, n_i represents the loop range, and N_I represents the number of processors on which the aligned pair of array dimensions is distributed. The parameter d denotes the message size, and p denotes the number of processors involved. Most of the entries for communication terms indicate the primitive to be used. *SizeChange*, however, is not a communication primitive, it is merely a term that modifies the data size parameter of other primitives when the overall communication cost is determined from various contributing terms.

For a subscript in the *rhs* reference varying in a loop, if there is no matching array dimension in the *lhs* reference, or if the corresponding subscript on *lhs* is a constant, the choice of the communication primitive depends on whether the *rhs* operands are involved in a reduction operation. In case of a reduction operation, the necessary communication may be carried out using the Reduction routine rather than multiple Transfers, which would be more expensive.

Whenever the subscript is of the type *variable* (i.e., unpredictable at compile time), and yet communication can be taken out of the loop, we assume that communication is carried out based on the following philosophy: each processor, before executing the loop, sends *all* the values that might be needed by the destination processor, even though at that time it is not known *which* values would actually get used. For instance, consider the following loop:

LHS	*RHS*	*Commn. Term*	*Parameters*
i	i	None	not applicable
i	$i \pm c_1$	Transfer	$product = c_1 * (N_I - 1), d = 1$
i	var_i or c_1 or missing	Transfer	$product = n_i, probability = (1 - 1/N_I), d = 1$
var_i	var_i or i or c_1 or missing	Transfer	$product = n_i, probability = (1 - 1/N_I), d = 1$
c_1 or missing	var_i or i	Transfer	$product = n_i, probability = (1 - 1/N_I), d = 1$

Table 3: Terms for Repeated Communication

LHS	*RHS*	*Commn. Term Type*	*Parameters*
c_1 or missing	c_2 or missing	Transfer	$probability = 1 - 1/N_I, d = 1$
c_1	c_1	None	not applicable

Table 4: Terms for Loop-Independent Communication

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

If each processor on which array B is distributed sends the entire section it owns to all the processors on which A is distributed, repeated communication inside the loop can be avoided even though the $D(i)$ values are unknown at compile time. Such an approach has an obvious disadvantage that it may lead to excessive wastage of memory on processors. In case memory space is at a premium, we may instead choose to carry out communication inside the loop, so that each processor receives only the values that it actually needs.

Table 3 describes the terms contributed by loops when communication has to be done repeatedly inside them. Many entries corresponding to the term for Transfer routine have a parameter *probability* associated with them. This parameter estimates the probability of the source and the destination processors for the intended Transfer routine being different. For simplicity, the compiler assumes that any two arbitrary elements along a given array dimension belong to the same processor with a probability of $1/N_I$, where N_I is the number of processors on which that array dimension is distributed. Hence, if the value of one of those elements is needed by the processor holding the other element, the probability that interprocessor communication is required is $1 - 1/N_I$. This elegantly takes care of the case $N_I = 1$, since the term for interprocessor communication vanishes when the array dimension is sequentialized.

Table 4 describes the terms contributed by a pair of subscripts corresponding to aligned dimensions, when both the subscripts are constants, and hence loop-independent.

```
do $i_1 = 1, n_1$
  $\ddots$
    do $i_l = 1, n_l$
      do $i_{l+1} = 1, n_{l+1}$
        $\ddots$
          do $i_m = 1, n_m$
            $S_1$
```

Figure 4: Statement involving communication

4.2 Combining of Communication Terms

We have shown in the previous section that by moving the parallel loops to inner levels, we can always ensure for any statement that each loop from which communication can be taken outside is relatively inner to a loop which has communication taking place inside it. Consider the statement nested inside loops shown in Figure 4. All loops at levels $l+1$ to m are those from which communication can be taken outside, while loops at levels 1 to l are those which have communication taking place inside them. We shall now describe how the communication costs involved in executing that statement are determined from the communication terms obtained for various loops and for the loop-independent subscript pairs.

Aggregate Communication We first determine the costs associated with calls to various aggregate communication primitives contributed by the loops at levels $l+1$ to m. These calls get placed inside the loop at level l, just before entering the loop at level $l+1$. Any terms coming from different loops that use the same primitive are combined as shown in Table 5. In the entries for the resultant term, the operation "o" represents the concatenation of sequences. We distinguish between a SizeChange term occurring by itself and one appearing in conjunction with a (conditional) Transfer primitive by using the superscripts 1 and 2, respectively, with those terms. Following the combination shown in Table 5, let the remaining terms be: SizeChange$^1(d_1)$, SizeChange$^2(d_2)$, Transfer(c_1), $\ldots$, Transfer(c_i), Reduction(d_3, seq_3), OneToManyMulticast(d_4, seq_4), ManyToManyMulticast(d_5, seq_5). When these terms are combined, the data size associated with each primitive is modified to take into account the number of elements along each array dimension held by a processor participating in communication. The order in which calls to these primitives are placed is significant [11] in that it affects the data sizes involved in communications. The first routine invoked is Reduction, because that leads to a reduction in the size of data for the remaining communication routines. Following that we have calls to all the Transfer routines. The ManyToManyMulticast operation gets performed after that; it leads to an increase in the amount of data held by each processor due to replication. Finally, the data is multicast using the OneToManyMulticast routine. Hence, the sequence of calls to various

Term1	*Term2*	*Resultant Term*
SizeChange$^1(d_1)$	SizeChange$^1(d_2)$	SizeChange$^1(d_1 * d_2)$
$(N_1 - 1)*$ Transfer(d_1)	$(N_2 - 1)*$ Transfer(d_2)	$(N_1 * N_2 - 1)*$ Transfer$(d_1 * d_2)$
SizeChange$^2(d_1)$, $(N_1 > 1)$ Transfer(c_1)	SizeChange$^2(d_2)$, $(N_2 > 1)$ Transfer(c_2)	SizeCh$^2(d_1 * d_2)$, $(N_1 > 1$ & $N_2 > 1)$ Transfer$(c_1 * c_2)$, $(N_1 > 1)$ Transfer$(c_1 * d_2)$, $(N_2 > 1)$ Transfer$(c_2 * d_1)$
$(N_1 - 1)*$ Transfer(d_1)	SizeChange$^2(d_2)$, $(N_2 > 1)$ Transfer(c_2)	$(N_1 - 1) *$ Transfer$(d_1 * d_2)$, SizeChange$^2(d_2)$ $(N_2 > 1)$Transfer$(c_2 * d_1)$
Reduction(d_1, seq_1)	Reduction(d_2, seq_2)	Reduction$(d_1 * d_2, seq_1 \circ seq_2)$
ManyToManyMulticast(d_1, seq_1)	ManyToManyMulticast(d_2, seq_2)	ManyToManyMulticast$(d_1 * d_2, seq_1 \circ seq_2)$
OneToManyMulticast(d_1, seq_1)	OneToManyMulticast(d_2, seq_2)	OneToManyMulticast$(d_1 * d_2, seq_1 \circ seq_2)$

Table 5: Combining of Terms with Identical Communication Primitives

communication routines assumed to be inserted for the purpose of estimating costs is: Reduction$(d_1 * d_2 * d_3 * d_4 * d5, seq_3)$, Transfer$(d_1 * c_1 * d_4 * d_5), \ldots,$ Transfer$(d_1 * c_i * d_4 * d_5)$, ManyToManyMulticast$(d_1 * d_2 * d_4 * d_5, seq_5)$, OneToManyMulticast$(d_1 * d_2 * d_4 * d_5 * num(seq_5), seq_4)$. While we have shown how to handle a completely general case when terms corresponding to all the primitives may be present, actual reference patterns in loops are expected to involve composition of much fewer primitives at a time.

Repeated Communication As shown in Table 3, the terms contributed by the loops at levels 1 to l consist of multiple Transfers of one of the following two kinds – regular Transfers involving data elements lying on the "boundaries" of regions allocated to processors, or *probabilistic* Transfers, namely those with a certain probability associated with them. The overall contribution to communication costs made by all the loops is determined as follows:

1. The costs due to all the aggregate communication routines determined in the previous step are multiplied by the product of the outer loop ranges, i.e., by the quantity $n_1 * n_2 * \ldots * n_l$. This is done because the calls to all those routines are placed inside the loop at level l, and the outer l loops are executed sequentially.

2. All the probabilistic Transfer terms contributed by the loops at levels 1 to l are combined into a single such term multiplied by the product $n_1 * n_2 * \ldots * n_l$. The probability associated with the overall term is computed as the union of probabilities attached to the individual terms. Let there be k such terms originally ($k \leq l$) with probabilities $1 - 1/N_{i_1}, \ldots, 1 - 1/N_{i_k}$. The probability associated with the overall Transfer term is set to $1 - \frac{1}{N_{i_1} * \ldots * N_{i_k}}$. The data size for the Transfer routine is set to the data size for the last aggregate communication primitive called (if any) just before entering the loop at level $l + 1$. If there is no call to any aggregate communication primitive, the data size is set to the value given by the SizeChange^1 term (if any), otherwise it is set to the default value of 1. Each regular Transfer term contributed by a loop remains separate from the probabilistic Transfer term. The product of such a term is multiplied by the range of all the other $l - 1$ loops, and its data size is

also set according to the rule described above.

Loop-Independent Communication As shown in Table 4, each term contributed by a pair of different constants appearing in matching subscripts consists of a probabilistic Transfer. These terms are combined among themselves and with the probabilistic Transfer terms (if any) contributed by loops at levels 1 to l, in the same manner as described above, to yield finally a single Transfer term.

Example We now present an example to show how we obtain an estimate of the overall communication costs by combining individual communication terms. Consider the program segment shown below:

```
do $j = 1, n$
    do $i = 1, n$
        $A(i, j, 4) = \mathcal{F}(B(i, 1, 4), C(i, j), D(i, j, 4))$
    enddo
enddo
```

Let N_I, N_J and N_K denote the number of processors on which the three dimensions of the array A are distributed. The terms contributed by the reference to B are SizeChange$^1(n/N_I)$ and OneToManyMulticast$(1, \langle N_J \rangle)$, which get combined to give the communication cost as OneToManyMulticast$(n/N_I, \langle N_J \rangle)$. The reference to the array C leads to the terms SizeChange$^1(n/N_I)$, SizeChange$^1(n/N_J)$, and $(1 - 1/N_K) *$ Transfer(1). These terms combine to give the cost as $(1 - 1/N_K) *$ Transfer$(n^2/(N_I * N_J))$. The reference to the array D leads to the terms SizeChange$^1(n/N_I)$ and SizeChange$^1(n/N_J)$, which give no contribution to the communication cost. Hence, the overall expression for the communication costs for this program segment is

$$\text{Communication cost} = \text{OneToManyMulticast}(n/N_I, \langle N_J \rangle) + (1 - 1/N_K) * \text{Transfer}(n^2/(N_I * N_J))$$

It is worth noting that the expressions for communication costs are determined assuming that proper alignment of array dimensions has been done. To determine the quality measures for the constraints on alignment, we simply determine the communication costs when alignment is not done that way, and subtract the original communication costs from these costs. For instance, if the first and second dimensions of D are aligned respectively with the second and the first dimension of A, the communication terms contributed are ManyToManyMulticast$(n/N_I, \langle N_I \rangle)$, and ManyToManyMulticast$(n/N_J, \langle N_J \rangle)$. These terms when combined indicate a cost of ManyToManyMulticast$(n^2/(N_I * N_J), \langle N_I, N_J \rangle)$, which is also the quality measure for the constraint on the proper alignment of those dimensions, since the communication cost for the case when those dimensions are properly aligned is zero.

As mentioned earlier, the results obtained in this section are based on the assumption that each loop index used as a subscript in an array reference varies so as to sweep over all the elements along that dimension. Our approach does not break down when this assumption does not hold, only the expressions involved become more complex. The parameters for data size and the number of processors get modified, the values of these parameters now become functions of the method of partitioning, whether it is contiguous or cyclic. We have omitted these details for simplicity.

5 Multiple References to an Array

So far, we have seen how to estimate the communication costs corresponding to a single reference to an array in the *rhs* expression of the assignment statement. We now describe how multiple references to the same array in the *rhs* expression are handled. All such references are partitioned into equivalence classes of *isomorphic* references. Two array references are said to be isomorphic if the subscripts corresponding to each array dimension in those references are of the same type, and have the same value of *dependence-level* associated with them. For instance, $B(i,1)$ and $B(i,2)$ are isomorphic references, while $B(i,j), B(j,i)$ and $B(1,j)$ are all mutually non-isomorphic. The communication costs pertaining to all isomorphic references in a given class are obtained by looking at the costs corresponding to one reference and determining how they get modified by the adjustment terms from the remaining references. The steps described in the previous section are applied for only one reference in each class after determining the adjustment terms given by the remaining references.

Table 6 shows how the terms contributed by matching subscript pairs from a given *rhs* array reference (and the *lhs* array reference) get modified in the presence of another isomorphic reference on the *rhs*. Terms corresponding to the subscripts that are exactly identical in the isomorphic references do not undergo any change. We distinguish between the cases when communication corresponding to the references can, and cannot be taken out of the loop by referring to the terms as corresponding to aggregate or to repeated communication. An entry "X" in the table denotes an arbitrary entry, implying that it does not matter what that entry is. While describing the modified communication term, we refer to the original term corresponding to the first *rhs* reference as $Term_1$. The modified terms are combined using the same procedure as that described in the previous section, thus we obtain estimates of communication costs involving all the isomorphic references. The procedure is repeated for each class of isomorphic references.

As an example, consider the loop shown in Figure 5. We have two pairs of isomorphic references on the *rhs*, $B(i-1,j)$ and $B(i+1,j)$, and $B(i,j-1)$ and $B(i,j+1)$. The first pair yields the communication terms SizeChange$^2(n_1/N_I)$, $2*(N_I > 1)$ Transfer(1), and SizeChange$^1(n_2/N_J)$, which get combined to give the communication cost as $2*(N_I > 1)$ Transfer(n_2/N_J). From the second pair, we get the terms SizeChange$^1(n_1/N_I)$, $2*(N_J > 1)$ Transfer(1), and SizeChange$^2(n_2/N_J)$, combining which we get $2*(N_J > 1)$ Transfer(n_1/N_I) as the communication cost. Thus the overall expression for communication costs involving

LHS	RHS_1	RHS_2	Original $Term_1$	Modified $Term(s)$
i	$i + c_1$	$i - c_2$	Agg: $(N_I > 1)$ Transfer(c_1) Rep: $c_1 * (N_I - 1)*$ Transfer(1)	Agg: $Term_1, (N_I > 1)$ Transfer(c_2) Rep: $(c_1 + c_2) * (N_I - 1)*$ Transfer(1)
i	$i + c_1$ $(i - c_1)$	$i + c_2$ $(i - c_2)$	Agg: $(N_I > 1)$ Transfer(c_1) Rep: $c_1 * (N_I - 1)*$ Transfer(1)	Agg: $(N_I > 1)$ Transfer(max(c_1, c_2)) Rep: max(c_1, c_2) $* (N_I - 1)*$ Transfer(1)
X	$var1_i$	$var2_i$	Agg: X Rep: X	Agg: $Term_1$ Rep: $2 * Term_1$
X	c_1	c_2	X	$Term_1, (1 - 1/N_I)*$ Transfer(1)

Table 6: Modification of terms for multiple references

```
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
```

Figure 5: Multiple references to an array

the given program segment is:

$$\text{Communication cost} = 2 * (N_I > 1)\text{Transfer}(n_2/N_J) + 2 * (N_J > 1)\text{Transfer}(n_1/N_I)$$

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

6 Detection of Pipelining

The procedure described in the preceding sections generates reasonably good estimates of communication costs for loops from which communication can be taken outside. However, when communication takes place inside loops due to dependences, the estimates are often overly conservative. The primary reason for that is the assumption that all iterations of a loop with cross-iteration dependences must be completed before any further computation can be done. In practice, it may well happen that a processor executing the earlier iterations of the loop is able to proceed with other computation while the remaining iterations of that loop are being executed on other processors. An important such case is the pipelining of an outer loop when successive iterations of the outer loop can be started without waiting for all the iterations of the inner loop to finish. We detect such opportunities (for pipelining) at compile-time in order to generate more accurate cost estimates.

```
do $j = 2, n_1$
    do $i = 2, n_2$
        $D(i) = \mathcal{F}(D(i-1))$                (S1)
        $A(i,j) = \mathcal{F}(B(i,j), D(i))$        (S2)
    enddo
enddo
```

Figure 6: Example to illustrate pipelining

We restrict our attention to the simple but frequently occurring case of loops having *constant dependences* between statements. A constant dependence is one for which the associated dependence distance vector is a constant. Consider the program segment shown in Figure 6. There are dependence edges from statement S1 to itself with distance vectors $(0,1)$ (flow dependence), $(1,-1)$ (anti dependence), and $(1,0)$ (output dependence). There is an edge from S1 to S2 with distance vector $(0,0)$ due to flow dependence involving array D, and another edge with vector $(1,0)$ due to anti-dependence from S2 to S1. Due to the dependence from S1 to itself, communication cannot be taken out of either of the loops. However, the given code segment lends itself easily to pipelining; we need not wait for all the iterations of the inner loop to finish before the next iteration of the outer loop is started. Hence, it would not be accurate to multiply the communication costs involved in executing the inner loop by the range of the outer loop. We first describe our simple approach for detecting the possibilities of pipelining and modifying the communication cost estimates, and then illustrate it for this example.

The algorithm to decide whether successive instances of an inner loop (L_k) at level k corresponding to different iterations of an outer loop (L_{k-1}) at level $k-1$ can be pipelined, consists of the following steps:

1. Examine the graph G_{k-1} constructed earlier as described in Section 3. Identify the set of nodes that belong to the same strongly connected component as the one containing nodes corresponding to all the statements inside loop L_k.

2. Examine all edges between the nodes identified in step 1. Let $(d_1, d_2, \ldots, d_n), (k \leq n)$ denote the direction vector associated with an edge. If for every edge, either d_k = "=", or all dependences corresponding to that edge have a constant distance vector, conclude that pipelining can be done. Further, if pipelining can be done, for all backward dependences (dependences in the ">" direction) at level k, determine the maximum dependence distance at that level, and call it c_k. If there is no backward dependence at level k, set the value of c_k to 0.

Step 1 narrows down the set of statements examined in the outer loop body to those over which loop distribution cannot be carried out any further. Step 2 checks for the regularity of communications due to dependences inside loop L_k to determine whether computations can be pipelined. In the event of pipelining, if there is a dependence from a later iteration (say, i_2) of the inner loop to an earlier iteration (say, i_1) of the same loop corresponding to a later iteration of the outer loop, the difference between i_2 and i_1 must be bounded by a constant. That constant is determined in Step 2 as c_k; it signifies that successive iterations of the outer loop can be started after waiting for only $c_k + 1$ iterations of the inner loop to finish, thus permitting an overlapping of computation.

Once the possibility of pipelining the execution of instances of loop L_k inside loop L_{k-1} has been recognized, the steps determining how the overall costs for loop L_{k-1} are computed from the communication terms are modified as follows:

1. The costs due to the aggregate communication routines are multiplied by $n_k + (c_k + 1) * (n_{k-1} - 1)$ rather than $n_k * n_{k-1}$. Let the time for aggregate communications taking place during one iteration of the loop L_k denote one unit of time. The communications associated with the first instance of the inner loop L_k see a full latency of n_k time units, whereas those associated with the remaining $n_{k-1} - 1$ instances of that loop see a latency of $c_k + 1$ time units each.
2. The product of the Transfer term contributed by loop L_k is not multiplied by n_{k-1}, rather we *add* $(\lfloor \frac{c_k}{block_k} \rfloor + 1) * (n_{k-1} - 1)$ to it to get the product for the final Transfer term. Each Transfer corresponding to the dependence carried by loop L_k takes place across the "boundary" of block of elements assigned to a processor, and hence takes place after every $block_k$ iterations of L_k, where $block_k$ is the block size of distribution of the array dimension being indexed in that loop. Before starting a new iteration of the loop L_{k-1}, since each processor has to wait for $c_k + 1$ iterations of the loop L_k to get over, it is able to initiate a new Transfer only after waiting for $(\lfloor \frac{c_k}{block_k} \rfloor + 1)$ Transfers corresponding to the previous iteration of L_{k-1} to be over.

It may be noted that when the dependence distance c_k is large, the communication costs are higher because we need to wait for a larger number of iterations of the inner loop to finish before starting the next iteration of the outer loop. In the limiting case when c_k takes the value $n_k - 1$, we need to wait for all iterations of the inner loop to finish, and our expression reduces to the original expression (without pipelining) for communication costs.

Let us now return to the example shown in Figure 6. The i-loop gets distributed over S1 and S2, but the j-loop cannot be distributed. Let N_1 and N_2 denote respectively the number of processors over which the first and the second dimension of the array A are distributed. The communication term associated with statement S1 contributed by the i-loop is $(N_1 - 1) *$ Transfer(1). Without taking pipelining into account, the overall communication cost estimate for executing S1 would be $n_1 * (N_1 - 1) *$Transfer(1). For statement S2, the i-loop contributes the term SizeChange[1](n_2/N_1), while the j-loop contributes the term $n_1 * (1 - 1/N_2) *$ Transfer(1).

These terms when combined according to our rules yield an estimate of $n_1 * (1 - 1/N_2) *$ Transfer(n_2/N_1). Now we explain how the detection of pipelining changes these estimates. First consider the i-loop over S1 which appears inside the j-loop. The only node we see in Step 1 of our algorithm is the node corresponding to S1. The dependences associated with the edge from S1 to itself do satisfy all the conditions of Step 2, and we conclude that pipelining can be done. The only backward dependence is the anti-dependence with the distance vector $(1, -1)$, hence we set c_2 to the value 1. Our modified procedure now estimates the communication costs associated with S1 to be $(n_1 - 1 + N_1 - 1) *$ Transfer(1), rather than a conservative $n_1 * (N_1 - 1) *$ Transfer(1). The i-loop over S2 does not involve any communication, so there is no need to check for pipelining.

7 Conclusions and Future Work

We have presented a methodology for estimating the communication costs at compile-time as functions of the numbers of processors over which various arrays are distributed. We have also developed a strategy for making program transformations to create more opportunities for taking communication out of loops, which can lead to considerable savings in communication costs. Finally, we have described a procedure for detecting when a loop with regular dependences may be pipelined, and how the estimates of communication costs are modified to take into account the speedup due to pipelining.

This work has been done as part of the development of a system to perform automatic data partitioning on multicomputers [7]. Our system shall generate annotations specifying how arrays being used in a program should be distributed on the processors of a specific machine. There are a number of compilers being developed [9, 16] that accept a sequential program augmented with such annotations, and generate the target parallel program for the distributed memory machine. Clearly, our estimates of communication costs guiding the automatic data partitioning system would be useful only if the actual compiler goes through a similar analysis for generating communication as the analysis we go through for estimating it. On the other hand, our results have interesting implications for *how* the generation of communication should be done by such compilers to reduce the communication overheads. We are currently exploring how synthesis of communication can be guided by the techniques we have developed for identifying the primitives that best realize the communications involved in carrying out a given computation.

We have used a number of simplifying assumptions in our work that need to be relaxed in order to build a truly automated system for handling this task. So far, we have assumed that the loop bounds and the probabilities for executing various branches of a conditional statement become "known" eventually (the values of loop bounds may be determined by making the user specify the values of certain critical parameters interactively or through assertions, and the compiler may simply guess the probabilities associated with various conditional statements). 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. Our system also needs to recognize opportunities for further optimizations regarding generation of communication [9], such as eliminating redundant messages via *liveness* analysis of array variables, and combining messages consisting of elements of different arrays into a single message whenever possible.

These issues do not undermine the applicability of our approach. They simply indicate the need for further research in this area so that the ideas we have presented can be augmented with other techniques to finally obtain a system performing automatic parallelization of programs on multicomputers.

References

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

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

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

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

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

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

[7] 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.

[8] 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).

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

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

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

[13] 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.

[14] 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.

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

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