Compile-Time Estimation of Communication Costs on Multicomputers*

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. 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 describe a methodology for statically estimating the communication times as certain functions of the array sizes and the numbers of processors over which various arrays are distributed. This work also lays down a framework for effective synthesis of communication on multicomputers from sequential program references.

1 Introduction

Recent years have seen a considerable research effort towards developing compilers for distributed memory multiprocessors (multicomputers) that automatically generate all interprocessor communication in the program [6, 10, 8, 13, 12]. Such compilers accept a program written in a sequential or shared-memory parallel language augmented with annotations specifying the distribution of data on various processors, and generate the target data-parallel program for the multicomputer. With such systems, the most challenging step in exploiting parallelism is to determine the best data partitioning scheme, which can be a tedious task. Recently several researchers [11, 9, 1, 4] have addressed the problem of automatically carrying out 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. Given a program without explicit communication, estimating the communication costs at compile time requires solving the following two problems: (1) determining the communication primitives that would efficiently realize the necessary data movement, and (2) determining the time taken to carry out the calls to communication primitives made by the program. To address the second problem, Balasundaram et al. have proposed a "training set" method [2], and Johnsson has presented a model for evaluating interprocessor communication [7]. In this paper, we focus our attention primarily on the first problem, and obtain cost estimates for programs in terms of those for collective communication routines. The techniques we develop can also be used to generate efficient communication using collective communication routines.

*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.

2 Background and Terminology

Distribution of arrays on target machine The abstract target machine we assume is a D-dimensional grid (D is the maximum dimensionality of any array used in the program), of $N_1 * N_2 * \ldots * N_D$ processors. Such a topology can easily be embedded on almost any distributed memory machine. We describe the distribution of an array by using a separate distribution function for each of its dimensions [4]. 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.

Communication Primitives Our system uses array reference patterns to determine which communication routines out of a given library of routines best realize the required data movement, a notion first introduced by Li and Chen [9] for synthesis of communication. Much of the ideas on these collective communication routines come from Fox et al. [3]. We assume that the following communication routines are supported by the run-time library:

- *Transfer* : send a message from a single source to a single destination processor.
- *OneToManyMulticast* : send a message to all processors along the specified dimension(s) of the processor grid.

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

Table 1: Costs of communication primitives on the hypercube architecture

- *ManyToManyMulticast* : replicate data from all processors on the given grid dimension(s) on to themselves.
- *Scatter* : send a different message to each processor lying on the specified grid dimension(s).
- *Gather* : receive a message from each processor lying on the specified grid dimension(s).
- *Shift* : circular shift of data among neighboring processors along the specified grid dimension.
- *Reduction* : reduce data using a simple associative and commutative operator, over all the processors lying on the specified grid dimension(s).

Table 1 shows the time 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 collective communication primitive is carried out. The function num applied to such a sequence simply returns the total number of processors involved.

Subscript Types For estimating communication overheads, we limit our attention to statements appearing in loops that involve assignment to arrays, since all scalar variables are replicated on all processors. Each subscript expression in an array reference 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.
- *variable*: this is the default case, and signifies that the compiler has no knowledge of how the subscript expression varies inside the loop.

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

3 Communication Costs for an Assignment Statement

For each statement in a loop in which the assignment to an array A (*lhs* array) uses values of an array B or A itself (*rhs* array), we express estimates of the communication costs incurred as the *rhs* array elements are sent to the processors owning the *lhs* array elements. This is in accordance with the *owner computes* rule used by almost all distributed-memory compilers, which makes the processor owning a data item responsible for all computations to that item, and requires interprocessor communication for any non-local values used in that computation. If the expression on the *rhs* of the assignment statement involves multiple arrays, the same analysis is repeated for each array.

3.1 Placement of Communication

The first task is to determine where the communication implementing data movement for the given statement would be placed. Normally, the placement would be done immediately preceding that statement. However, in order to enable optimizations like vectorization of messages and the use of collective communication routines, the compiler attempts to take communication out of various loops that the given statement appears in. The algorithms to determine the placement of communication and to determine the need for program transformations – loop distribution and loop permutation (to move dependences to outer loops), are described in [5]. The loop structure surrounding a statement with array references (the loops need not be perfectly nested) can finally be transformed to the form shown in Figure 1. All loops at levels $l + 1$ to m are those from which communication can be taken outside (we shall refer to them as type-1 loops), while loops at levels 1 to l are those which must have communication taking place inside them due to dependence constraints (we shall refer to such loops as type-2 loops). In general, the compiler may choose not to combine communication corresponding to different iterations of a type-1 loop if that involves excessive overcommunication (communication of extraneous values in addition to the useful ones) or wastage of memory space. However, at the time of estimation of costs, we ignore these trade-offs, and assume that the combining of communication would be done whenever possible.

3.2 Identification of Communication Terms

Given a pair of array references, the *lhs* and the *rhs* reference, each pair of subscripts corresponding to aligned dimensions (array dimensions distributed on the same processor grid dimension) identifies a communication term representing data movement in that grid dimension. If the two arrays differ in the number of dimensions, the subscripts corresponding to the "extra" distributed dimensions of one array get paired up with "missing" subscripts of the other array, which are conceptually treated as being of the type *constant*. The larger of the values of *variation-level* of a subscript pair identifies the innermost loop in which data

do $i_1 = 1, n_1$
$\ddots$
do $i_l = 1, n_l$
$\langle$ communication for (lhs, rhs) $\rangle$
do $i_{l+1} = 1, n_{l+1}$
$\ddots$
do $i_m = 1, n_m$

lhs $= \mathcal{F}$(rhs)

Figure 1: Statement involving communication

LHS	*RHS*	*Communication Term*
i	i	IDM(n_i/N_I)
i	$i \pm c_1$	(i) IDM(n_i/N_I) (ii) ($N_I > 1$) Shift(c_1)
i	var_i	ManyToManyMulticast($n_i/N_I, \langle N_I\rangle$)
var_i	i, var_i	ManyToManyMulticast($n_i/N_I, \langle N_I\rangle$)
i, var_i	c_1	OneToManyMulticast($1, \langle N_I\rangle$)
c_1	i	Reduction/Gather($n_i/N_I, \langle N_I\rangle$)
c_1	var_i	Gather($n_i/N_I, \langle N_I\rangle$)
i, var_i	j, var_j	ManyToManyMulticast($n_j/N_I, \langle N_I\rangle$)

Table 2: Terms for Collective Communication

movement for the corresponding grid dimension takes place.

When the loop identified by a subscript pair is a type-1 loop, Table 2 lists the term(s) contributed by the pair, based on the type of subscripts involved. In the table, i and j denote different loop indices, c_1 denotes a constant, var_i and var_j represent subscripts of the type *variable* with their *variation-level* values corresponding to loops with indices i and j respectively. Among the entries for communication terms, IDM (internalized data movement) 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; n_i represents the loop range, and N_I represents the number of processors on which the aligned pair of array dimensions is distributed.

Table 3 describes the term contributed by a subscript pair corresponding to a type-2 loop for one iteration of such a loop. The factor $(1 - 1/N_I)$ appearing in many terms represents the probability of the source and the destination processors for the intended Transfer routine being different for an arbitrary loop iteration. For simplicity, the compiler regards any two elements along a given array dimension as belonging to the same processor with a probability of $1/N_I$, and therefore, as belonging to different processors with a probability of $1 - 1/N_I$. This elegantly takes care of the case $N_I = 1$, as the term for interprocessor communication vanishes when the array dimension is sequentialized.

LHS	*RHS*	*Communication Term*
i	i	IDM(1)
i	$i \pm c_1$	$c_1 * (N_I - 1)/n_i *$ Transfer(1)
i	var_i, c_1	$(1 - 1/N_I) *$ Transfer(1)
var_i	var_i, i, c_1	$(1 - 1/N_I) *$ Transfer(1)
c_1	var_i, i	$(1 - 1/N_I) *$ Transfer(1)

Table 3: Terms for Repeated Communication

LHS	*RHS*	*Communication Term*
c_1	c_2	$(1 - 1/N_I) *$ Transfer(1)
c_1	c_1	IDM(1)

Table 4: Terms for Loop-Independent Communication

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.

3.3 Combining of Communication Terms

The communication terms obtained corresponding to all the distributed array dimensions are composed together in a definite order to determine the overall data movement taking place. First, different terms involving the same primitive are combined as shown in Table 5. In the entries for the resultant term, the operation "$\circ$" represents the concatenation of sequences. We distinguish between a IDM term occurring by itself and one appearing in conjunction with a Shift primitive by using the superscripts 1 and 2, respectively, with those terms. The order in which calls to different primitives are placed is significant in that it affects the message sizes (m) and the number of processors (p) involved in communications. The communication routines are invoked in the following order: Reduction (reduces m and p), Scatter (reduces m, increases p), Shift, Transfer (preserve m and p), OneToManyMulticast (preserves m, increases p), Gather (increases m, reduces p), and finally ManyToManyMulticast (increases m and p).

The costs due to all the communication routines determined in the previous step are multiplied by the product of the outer loop ranges, i.e., by $n_1 * n_2 * \ldots * n_l$ since the communications over the outer l type-2 loops are carried out sequentially.

Example 1

do $k = 1, n_1$
do $j = 1, n_2$
do $i = 1, n_3$
$A(i, j, k) = \mathcal{F}(B(i, 1, 1))$
$\vdots$

Term1	*Term2*	*Resultant Term*
$\text{IDM}^1(d_1)$	$\text{IDM}^1(d_2)$	$\text{IDM}^1(d_1 * d_2)$
$\text{Gather}(d_1, seq_1)$	$\text{Gather}(d_2, seq_2)$	$\text{Gather}(d_1 * d_2, seq_1 \circ seq_2)$
$\text{IDM}^2(d_1)$, $(N_1 > 1)$ Shift(c_1)	$\text{IDM}^2(d_2)$, $(N_2 > 1)$ Shift(c_2)	$\text{IDM}^2(d_1 * d_2)$, $(N_1 > 1$ & $N_2 > 1)$ Shift$(c_1 * c_2)$, $(N_1 > 1)$ Shift$(c_1 * d_2)$, $(N_2 > 1)$ Shift$(c_2 * d_1)$
$\text{Reduction}(d_1, seq_1)$	$\text{Reduction}(d_2, seq_2)$	$\text{Reduction}(d_1 * d_2, seq_1 \circ seq_2)$
$\text{ManyToManyMulticast}(d_1, seq_1)$	$\text{ManyToManyMulticast}(d_2, seq_2)$	$\text{ManyToManyMulticast}(d_1 * d_2, seq_1 \circ seq_2)$
$\text{OneToManyMulticast}(d_1, seq_1)$	$\text{OneToManyMulticast}(d_2, seq_2)$	$\text{OneToManyMulticast}(d_1 * d_2, seq_1 \circ seq_2)$
$(1 - \frac{1}{num(seq_1)})*$ Transfer(1)	$(1 - \frac{1}{num(seq_2)})*$ Transfer(1)	$(1 - \frac{1}{num(seq_1)*num(seq_2)})*$ Transfer(1)

Table 5: Combining of Terms with Identical Communication Primitives

Based on dependence constraints, let the combining of communication involving array B be possible only for the i and the j-loops, and let N_I, N_J and N_K denote the number of processors on which the three dimensions of the given arrays are distributed. The terms contributed by the reference to B are $\text{IDM}^1(n_3/N_I)$, $\text{OneToManyMulticast}(1, \langle N_J \rangle)$, and $(1 - 1/N_K)*$ Transfer(1). Hence, the overall expression for communication costs for this statement is:

$$\begin{aligned}\text{Cost} &= n_1 * (1 - 1/N_K) * \text{Transfer}(n_3/N_I) + \\ & \quad n_1 * \text{OneToManyMulticast}(n_3/N_I, \langle N_J \rangle)\end{aligned}$$

4 Multiple References to an Array

So far, we have seen how to estimate the communication costs corresponding to a single reference to an array on the *rhs* of a statement. When there are multiple references to the same array on the *rhs*, all such references are partitioned into equivalence classes of *isomorphic* references that are analyzed separately. Two array references are said to be isomorphic if they differ only in one subscript position, and the differing subscripts are of the same type and have the same value of *variation-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. Intuitively, if two *rhs* references are isomorphic, the required communication of those values can be handled either by (i) similar communication primitives, or by (ii) sending the value(s) corresponding to one reference to the processor(s) owning value(s) for the second reference, and then handling communication for the two references together. 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.

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 is done outside/inside the loop by referring to those terms as corresponding to type-1 or type-2. An entry "X" in the table denotes an arbitrary, "don't care" entry. While describing the modified communication term, we refer to the original term corresponding to the first *rhs* reference as $Term_1$. The last entry $Term_1(d \rightarrow 2 * d)$ in the Table represents $Term_1$ with its data size doubled over the original value. The modified terms are combined using the same procedure as that described in the previous section.

Example 2

```
do j = 2, n_2 - 1
    do i = 2, n_1 - 1
        A(i, j) = F(B(i - 1, j), B(i, j - 1), B(i + 1, j),
                    B(i, j + 1))
    enddo
enddo
```

We have two pairs of isomorphic references on the *rhs*, $B(i-1, j)\&B(i+1, j)$, and $B(i, j-1)\&B(i, j+1)$. The first pair yields the communication terms $\text{IDM}^2(n_1/N_I)$, $2 * (N_I > 1)$ Shift(1), and $\text{IDM}^1(n_2/N_J)$, which get combined to give the communication cost as $2 * (N_I > 1)$ Shift(n_2/N_J). From the second pair, we get the terms $\text{IDM}^1(n_1/N_I)$, $2 * (N_J > 1)$ Shift(1), and $\text{IDM}^2(n_2/N_J)$, combining which we get $2 * (N_J > 1)$ Shift(n_1/N_I) as the communication cost. Thus the overall expression for communication costs involving the given program segment is:

$$\begin{aligned}\text{Cost} &= 2 * (N_I > 1)\text{Shift}(n_2/N_J) + \\ & \quad 2 * (N_J > 1)\text{Shift}(n_1/N_I)\end{aligned}$$

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.

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

Table 6: Modification of terms for multiple references

5 Detection of Pipelining

So far, we have assumed strict sequentiality in the process of carrying out communication inside the type-2 loops. In practice, the generation of messages may get pipelined over different iterations of some type-2 loop(s). In certain special cases, our compiler detects this and generates more accurate cost estimates.

Consider an *rhs* array A for which the communication term corresponding to a loop L_k (at level k) is IDM, Shift and IDM, or Transfer (across boundaries of distributed regions of A). The following algorithm determines whether communication involving A taking place during different iterations of a type-2 loop L_{k-1} (at level $k-1$) can be pipelined.

1. Examine all dependence edges due to the array A coming into the node for the given statement from nodes corresponding to statements inside L_{k-1}. Let $(d_1, d_2, \ldots, d_n), (k \leq n)$ denote the direction vector associated with such an edge. If for every edge, either $d_k =$ "=", or all dependences corresponding to that edge have a constant distance vector, conclude that pipelining can be done.
2. If pipelining can be done, for all backward dependences (dependences in the ">" direction) at level k examined in Step 1, determine the maximum dependence distance at level k, and call it bd_k. If there is no backward dependence at level k, set the value of bd_k to 0.

If pipelining is detected, the precedence constraints indicate that successive iterations of the loop L_{k-1} may be started after waiting for $bd_k + 1$ iterations of L_k (corresponding to the previous iteration of L_{k-1}) to finish. The only resource constraint we assume is that a processor can participate in no more than one communication primitive at a time. Let $chunk_k$ be the number of iterations of loop L_k that sweep across the region of A lying on a single processor. The steps determining how the communication costs are obtained for loop L_{k-1} are modified as follows (this analysis can be extended if communications can also be pipelined for the loops outside L_{k-1}):

1. The costs due to any collective communication routines involving A over the other grid dimensions (representing contributions from the loops nested inside L_k) are multiplied by $n_k + max(chunk_k, bd_k + 1) * (n_{k-1} - 1)$ rather than $n_k * n_{k-1}$.
   Explanation Let the time for a collective communication carried out during one iteration of the loop L_k be one unit. The communications associated with the inner loop L_k for the first iteration of L_{k-1} see a full latency of n_k time units. Due to precedence constraints, each successive iteration of L_{k-1} contributes to a further latency of at least $bd_k + 1$ units. The resource constraints further dictate that a processor complete its participation in all the $chunk_k$ collective communications associated with an iteration of L_{k-1} before staring one for the next iteration of L_{k-1}. Thus, each of the remaining $n_{k-1} - 1$ iterations of that loop contributes to a latency of $\max(bd_k + 1, chunk_k)$ time units.
2. If there is a Shift or Transfer term for A associated with loop L_k, the term as seen for loop L_k is not multiplied by the loop range n_{k-1}, rather we *add* $\max(1, \lfloor \frac{bd_k+1}{chunk_k} \rfloor) * (n_{k-1} - 1)$ to its product to get the final term for the loop L_{k-1}.
   Explanation For the first iteration of L_{k-1}, the full latency corresponding to all the Shifts/Transfers taking place during loop L_k is seen. For the remaining $n_{k-1} - 1$ iterations of L_{k-1}, before starting a new iteration, each processor has to wait for $bd_k + 1$ iterations of the loop L_k to finish, i.e., for $(\lfloor \frac{bd_k+1}{chunk_k} \rfloor)$ Shifts/Transfers to be over. Also, since a processor cannot take part in more than one communication primitive at a time, it has to wait for at least one Shift/Transfer (representing its participation in the previous iteration of L_{k-1}) before initiating a new one.

Example 3

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

Let N_1 denote the number of processors over which the array D is distributed. The communication term associated with statement S1 contributed by the i-loop is $(N_1 - 1) * \text{Transfer}(1)$. Without taking pipelining into account, the overall communication cost estimate for executing S1 would be $n_1 * (N_1 - 1) * \text{Transfer}(1)$.

Let us now see how pipelining gets detected and the cost estimates modified. Consider the communications involving D during different iterations of the j-loop. All the dependence edges involving array D do satisfy all the conditions of Step 1, and we conclude that pipelining can be done. The only backward dependence at level 2 is the anti-dependence from S1 to itself with the distance vector $(1, -1)$, hence c_2 is set to 1. Our modified procedure now estimates the communication costs associated with S1 to be $(n_1 - 1 + N_1 - 1) * \text{Transfer}(1)$, rather than a conservative $n_1 * (N_1 - 1) * \text{Transfer}(1)$.

6 Experimental Results

Much of the ideas presented in this paper have been implemented. In order to see how well the parameterized cost estimates generated by our compiler work in actual practice, we conducted some experiments to study the communication overheads incurred by a parallel program running on the Intel iPSC/2 hypercube system. After having generated the cost estimates automatically for a program carrying out relaxation computations shown in Example 2, we parallelized it by hand. Two different versions of the parallel program were implemented on the hypercube, one corresponding to column-partitioning of arrays A and B, and the other corresponding to block-partitioning of the arrays. The programs were run on the 16-processor system for different data sizes. Through measurements of the actual times spent in communication by the program under the two partitioning schemes, we found the estimates to be quite accurate, usually with an error margin of less than 10%. The cross-over point (in terms of matrix size) at which block-partitioning starts outperforming column-partitioning was also predicted accurately. The details regarding those measurements are given in [5].

7 Conclusions and Future Work

This work has been done as part of the development of a system to perform automatic data partitioning on multicomputers [4]. The analysis we perform for estimating communication costs can as well be used by the compiler to recognize opportunities for using collective communication routines in a program to implement data movement. We are currently exploring how synthesis of communication can be guided by the techniques we have developed.

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 this task. Various loop bounds in the program and the probabilities of executing different branches of a conditional statement are assumed to be known to the compiler. Currently, we make the user specify those values interactively or through assertions. In the future, we plan to use profiling to supply the compiler with information regarding how frequently various basic blocks of the code are executed. We expect such a system to be an extremely useful tool for parallelizing compilers for distributed memory 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*, April 1990.

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

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

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

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

[6] 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.

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

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

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

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

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

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

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