A Methodology for High-Level Synthesis of Communication on Multicomputers*

Manish Gupta and Prithviraj Banerjee

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

Abstract

Freeing the user from the tedious task of generating explicit communication is one of the primary goals of numerous research projects on compilers for distributed memory machines. In the process of synthesis of communication, the effective use of collective communication routines offers a considerable scope for improving the program performance. This paper presents a methodology for determining the collective communication primitives that should be used for implementing the data movement at various points in the program. We introduce the notion of certain synchronous properties between array references in statements inside loops, and present tests to determine the presence of these properties. These tests enable the compiler to analyze quite precisely the communication requirements of those statements, and implement communication using appropriate primitives. These results not only lay down a framework for synthesis of communication on multicomputers, they also form the basis of our implementation of a system that statically estimates the communication costs of programs on multicomputers.

1 Introduction

Distributed memory machines (multicomputers) are increasingly being used to deliver very high levels of performance for a variety of applications. Over the last few years, considerable research effort has gone into developing compilers that make it easier to program such machines by relieving the programmer of the burden of managing communication explicitly [9, 22, 16, 12, 4, 17, 18]. Most of these compilers take a sequential or a shared memory parallel program, and based on the user-specified partitioning of data, generate the parallel program targeted to a multicomputer.

An obvious challenge before all of these efforts is to match the quality and efficiency of the code with explicit message-passing, written by an experienced programmer, with that of the compiler-generated code. This paper presents efforts in this direction, that help the compiler implement data movement between processors using efficient collective communication primitives.

The use of high-level collective communication routines (such as broadcast, reduction) is becoming increasingly popular with application programmers on multicomputers. Many ideas on these routines came from researchers involved in developing scientific application programs on such machines [5, 11]. A number of such communication libraries have been developed, in some cases, as part of a broader research effort [14, 5, 4, 19]. The use of high-level communication routines provides a number of benefits:

- The code generated is much more concise as numerous calls to the *send* and *receive* primitives get replaced by fewer calls to high-level primitives.
- The performance of programs can be improved by exploiting efficient implementations of such routines on the target machine. For instance, the broadcast routine can be implemented using $log_2(N)$ steps on an N-processor machine, whereas an equivalent piece of code using individual send and receive primitives would take up N steps.

Li and Chen [13] introduced compiler techniques based on matching source program references with known patterns to synthesize communication in terms of collective communication routines. Many other compilers perform similar analysis to detect opportunities for using collective communication primitives in simple cases [9, 17, 10]. However, there are numerous kinds of references on which existing techniques either fail to exploit the underlying regularity in communication (and use only the send and receive primitives), or suggest the use of excessively costly collective communication routines.

In this paper, we describe a methodology for synthesizing communication using high-level primitives, based on the analysis of data references in the given program. Our methodology is quite general, and overcomes many limitations of the existing techniques. Besides helping generate efficient communication, the analysis of data references proves to be extremely useful in estimating the performance of such programs. Through these techniques, the communication requirements of a program can be captured in a concise manner without actually generating communication, and hence can be used for statically estimating the communication costs. Most of the ideas presented in this paper arose out of, and have now been implemented in the context of such a tool for estimating communication costs of a program [7]. That

*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.

tool, in turn, is a part of the overall system being built for automatic partitioning of data on multicomputers [8].

The rest of this paper is organized as follow. Section 2 provides an overview of the machine model, the approach to compilation, the high-level communication support, and the categorization of subscripts used by us. Section 3 describes some limitations of the existing techniques for generating collective communication. Section 4 introduces terms that we use to characterize certain properties of the array references appearing in statements inside loops, and shows how the presence of these properties is inferred. Section 5 describes how the loop level at which the communication for a statement should be placed is determined. Section 6 presents our approach to determine the sequence of communication primitives to handle the data movement for a statement. Section 7 presents some examples. Section 8 discusses some issues regarding run-time support that would help the process of generation of communication. Finally, Section 9 presents conclusions, and ideas for future work.

2 Background and Concepts

The abstract target machine we assume is a D-dimensional mesh (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 be easily embedded on most distributed memory machines. We describe the distribution of an array by using a separate distribution function for each of its dimensions [8]. Each array dimension is mapped to a unique dimension of the processor mesh, and the associated distribution function $f(i)$ returns the processor number along that mesh dimension to which the ith element ($i \geq 0$) along the given array dimension gets mapped. The distribution function takes the following form:

$$f(i) = \lfloor \frac{i}{b} \rfloor [\mathrm{mod} N_d]$$

where b is the block size of distribution. The square parenthesis surrounding $\mathrm{mod} N_d$ indicates that it is an optional operation (included if the distribution is cyclic, N_d is the number of processors). This formulation of the distribution function captures both contiguous and cyclic distributions. An array having fewer than D (say, d) dimensions also has distribution functions corresponding to each of the $D-d$ "missing" dimensions. If all the elements varying along a given array dimension reside on the same processor, i.e., if the corresponding mesh dimension has only one processor, the array dimension is said to be *sequentialized*, otherwise we call it a *distributed* dimension. We refer to the kth dimension of an array A as A_k.

Given the data distribution, the basic rule of compilation used by virtually all multicomputer parallelization systems is the *owner computes* rule [9], according to which it is the processor owning a data item that has to perform all computations for that item. Any data values required for the computation that are not available locally have to be obtained via interprocessor communication. In this paper, we shall consider the communication requirements pertaining only to array references. The scalar variables are usually replicated on all processors. If they are not replicated, the techniques that we develop for array references can be extended to handle distributed scalar variables as well. A naive application of the *owner computes* rule without any further optimization leads to an extremely inefficient program with high masking and communication overhead. In such a program, there is a separate message generated for every non-local data item referenced in the program. Typically, the start-up cost of a message on any multicomputer is much higher than the per-element cost of sending messages. Hence, one of the most important optimization goals for a compiler is to combine messages between the same source and destination processors into a single message. When applied to a loop, this goal translates to one of taking communication out of the loop so that messages can be vectorized, i.e., messages corresponding to multiple iterations can be combined into a single message [2, 6].

The use of collective communication routines can be viewed as a step beyond mere vectorization of messages, in the following sense. A sequence of accesses corresponding to an array reference inside a loop may span a number of processors and a number of memory locations on each processor. (Throughout this paper, we shall use the term *array reference* to refer to the symbolic term appearing in the program, not the physical reference (read or write) to the element(s) represented by that term). For the given array reference inside the loop, we refer to the sequence of processors traversed as its *external traversal*, and the sequence of memory locations accessed on each processor as its *internal traversal* with respect to that processor. The vectorization of messages represents a high-level handling of the internal traversal of the reference for each processor. The use of collective communication primitives may be regarded as an extension of that step, one that exploits regularities in the patterns of external traversal as well.

Thus, the generation of high-level communication for programs requires an ability to group together processors corresponding to the external traversal of various references, and carry out collective communication operations over processors in those groups. While the compiler identifies the processors to be grouped together and the communication routines to be used, the support to do that has to come from the communication library on the target machine. For the moment, we ignore the problem of grouping, and assume that the communication routines described below are supported by the run-time library. All of these primitives, other than Transfer, are referred to as collective communication primitives, since they represent communication over a collection of processors.

- *Transfer* : send a message from a single source to a single destination processor.
- *OneToManyMulticast* : send a message to all the processors in the given group.
- *ManyToManyMulticast* : replicate data from all of the processors in the group on to themselves.
- *Scatter* : send a different message to each of the processors in the group.
- *Gather* : receive a message from each of the processors in the group.
- *Shift* : circular shift of data among adjacent processors in a group.

- *Reduction* : reduce data using a simple associative and commutative operator, over all of the processors in the group.

Much of the analysis of the communication requirements for array references is based on the kinds of subscripts in those references. Each subscript expression is assigned to one of the following categories:

- *constant*: if the subscript expression evaluates to a constant at compile time.
- *single-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.
- *multiple-index*: if the subscript expression reduces to the form $c_1 * i_1 + c_2 * i_2 + \ldots + c_k * i_k + c_{k+1}$, $k \geq 2$, where $c_1, \ldots, c_{k+1}$ are constants, and $i_1, \ldots, i_k$ are loop indices.
- *unknown*: 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 each subscript, the compiler records the value of a parameter called *variation-level*, which is the level of the innermost loop in which the subscript changes its value. For a subscript of the type *constant*, it is set to zero.

3 Limitations of Existing Techniques

Most existing techniques for synthesizing collective communication are limited in the extent of subscript analysis they use. Prominent among the patterns on which they fail, or suggest the use of excessively costly primitives are those with the following characteristics:

- *More than one procesor mesh dimension getting traversed in a single parallel loop.*
  Such references are extremely common in real programs, and may be the result of (i) the same loop index appearing in more than one subscript in an array reference, as in the reference $a(i, i)$, or (ii) array dimensions traversed in the same loop getting distributed on different mesh dimensions. Consider the following loop involving arrays A and B, that are distributed along both dimensions.

  do $i = 0, n-1$
  $\quad A(i, c_1) = \mathcal{F}(B(i, i)$
  enddo

  Depending on the relative block sizes of distributions of various dimensions, the data movement for this loop may be best realized using parallel Transfers, Gathers, or Scatters, or none of these (via a sequence of Transfers). Figure 1 illustrates one such case where the appropriate choice is the use of Gather primitives in parallel over mutually disjoint groups of processors. We shall present efficient tests that can be performed at compile time to determine the appropriate choice of communication primitive in such cases.

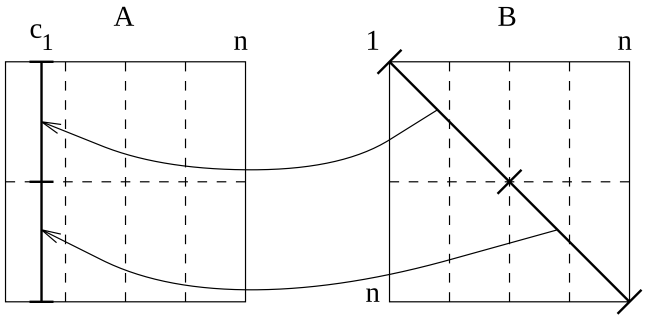

Figure 1: Parallel Gathers over different groups

- *More than one loop index appearing in the expression for a single subscript in an array.*
  Such references may appear by themselves in real programs, and perhaps more importantly, there are transformations like loop skewing and loop rotation [20] that often lead to the references taking such a form. Let us consider the following loop, described in [20]:

  do $i = 0, n-1$
  $\quad$ do $j = 0, n-1$
  $\quad\quad A(i) = \mathcal{F}(A(i), B(j))$
  $\quad$ enddo
  enddo

  On "wavefronting" this loop (loop skewing followed by loop interchange) to eliminate the need to multicast elements of B [20], we obtain the following loop:

  do $j = 0, 2*n-2$
  $\quad$ do $i = max(0, j-n+1), min(j, n-1)$
  $\quad\quad A(i) = \mathcal{F}(A(i), B(j-i))$
  $\quad$ enddo
  enddo

  The subscript in the reference to B involves multiple loop indices, and a straightforward approach to using collective communications would continue to suggest the use of the ManyToManyMulticast primitive for the reference to B. Our approach handles such references in a way that the intended objective of replacing the multicast with simple transfers between processors in different steps is met.

4 Properties of Array References

For the purpose of identifying the kind of communication primitives to be used, of special interest to us are the variations of subscripts corresponding to distributed dimensions of arrays. We refer to the array reference along a particular distributed dimension as a *sub-reference*. A sub-reference can be represented by a tuple $\langle r, d \rangle$, where r is an array reference, and d is the position ($d \geq 1$) of the distributed dimension. A sub-reference varying inside a loop can be seen as traversing a sequence of elements distributed on different processors along a mesh dimension. Every point in the loop (identified by the value of the loop index) at which the sub-reference crosses a processor boundary in that mesh

```
do i = 0, n − 1
    do j = 0, n − 1
        A(i, j) = F(B(j, j))
    enddo
enddo
```

Figure 2: Example program segment

dimension is called the *transition point* of the loop for the given sub-reference. We now define some properties describing the relationships between sub-references varying inside the same loop, that help characterize the data movement for that loop.

Definition 1 A sub-reference s_1 is said to be *k-synchronous* (k is a positive integer) with another sub-reference s_2 with respect to a loop L, if (i) every transition point of L for s_2 coincides with a transition point of L for s_1, and (ii) between every pair of consecutive transition points of L for s_2, there are exactly $k-1$ transition points of L for s_1.
Example: In Figure 2, if the dimensions A_2 and B_2 are distributed in a contiguous manner on $k * N_2$ and N_2 processors respectively, the sub-reference $\langle A(i,j), 2\rangle$ is k-synchronous with $\langle B(j,j), 2\rangle$ with respect to the j-loop.

Definition 2 A sub-references s_1 is said to be *strictly synchronous* with another sub-reference s_2 with respect to a loop L, if (i) s_1 is 1-synchronous with s_2 with respect to L (i.e., every transition point of L for s_1 is also a transition point of L for s_2, and vice versa), and (ii) the coinciding transition points represent the cross-over points between the same processor numbers in the respective mesh dimensions for those sub-references.
Example: In Figure 2, if arrays A and B have identical distributions, the sub-reference $\langle A(i,j), 2\rangle$ is strictly synchronous with $\langle B(j,j), 2\rangle$ with respect to the j-loop.

A useful convention we adopt regarding the k-synchronous property is that k is also allowed to be a reciprocal of positive integer. In that case, a statement that s_1 is k-synchronous with s_2 is really to be interpreted as conveying that s_2 is $1/k$-synchronous with s_1.

Recognition of properties

We now present results that allow the properties mentioned above to be recognized, given the subscript expressions associated with various sub-references. Intuitively, the presence of these properties suggests some regularity in the data movement that can be captured through high-level communication primitives over mutually disjoint groups of processors. The various requirements (which may sometimes be conservative) that are part of recognizing these properties are simply to be seen as tests that allow special handling of data movements for those references. Even when these tests are not satisfied (and even when the synchronous properties do not hold between sub-references), there still are alternate ways of generating the required communication, but usually those are less efficient. All of our results are presented for the case when the subscripts in array references start from 0. In our implementation which analyzes Fortran programs, we actually use trivial modifications of these tests, since the array subscripts in our Fortran programs start from 1 (the expressions involved in the other case are simpler and easier to understand, hence those are the ones that we report).

Strictly synchronous sub-references: Let s_1 and s_2 be two sub-references, and the associated subscript expressions be e_1 and e_2, both with a variation-level corresponding to loop L. Let b_1 and b_2 be the block sizes of distribution of the corresponding array dimensions. We derive conditions under which s_1 is strictly synchronous with s_2 with respect to the loop L, for $e_1 = c_1 * i + c_2$, and $e_2 = c_3 * i + c_4$. It is required that the array dimensions corresponding to both the sub-references be distributed in the same manner – contiguous or cyclic, and that in case of cyclic distribution, the dimensions be distributed on an equal number of processors. A sufficient condition for the strictly synchronous property is:

$$\lfloor \frac{c_1 * i + c_2}{b_1} \rfloor = \lfloor \frac{c_3 * i + c_4}{b_2} \rfloor \quad \text{(for all } i\text{)}$$

This leads to the following conditions, which put together are sufficient.

1. $c_1/b_1 = c_3/b_2$.
2. $c_2/b_1 = c_4/b_2$.
   When b_1 is a multiple of c_1, and b_2 is a multiple of c_3, the second condition can be relaxed to the following less restrictive condition: $\lfloor c_2/c_1 \rfloor = \lfloor c_4/c_3 \rfloor$

k-synchronous sub-references: The conditions we check to see whether s_1 is k-synchronous with s_2 are obtained in a similar manner, and are shown below:

1. $c_1/b_1 = k * (c_3/b_2)$.
2. $c_2/b_1 = k * (c_4/b_2) + l$,
   where l is an integer. Again, the second condition can be replaced by the following three conditions, (i) b_1 is a multiple of c_1, (ii) b_2 is a multiple of c_3, and (iii) $\lfloor c_2/c_1 \rfloor = \lfloor c_4/c_3 \rfloor + l'$, where l' is an integer, and a multiple of (b_1/c_1).

Boundary-check: Another specialized test is the "boundary-communication" test, that helps detect data movement taking place across boundaries of regions allocated to neighboring processors. This test checks the following conditions:

1. $c_1/b_1 = c_3/b_2$.
2. $|c_2/b_1 - c_4/b_2| \le 1$.

5 Placement of Communication

In the absence of any optimization, the communication required to implement any data movement for a statement is placed just before that statement. However, in order to enable optimizations like combining of messages, the compiler attempts to take communication out of as many loops (that the given statement appears in) as possible. Based on the examination of all incoming dependences into a statement, the compiler first determines the outermost loop level at which communication could be legally placed [7], and performs the necessary program transformations – loop distribution and loop permutation (like loop interchange) to

$$
\begin{array}{l}
\text{do } i_1 = 1, n_1 \\
\quad \ddots \\
\qquad \text{do } i_l = 1, n_l \\
\qquad\quad \langle \text{ communication for } (r_1, r_2) \rangle \\
\qquad\quad \text{do } i_{l+1} = 1, n_{l+1} \\
\qquad\qquad \ddots \\
\qquad\qquad\quad \text{do } i_m = 1, n_m \\
\qquad\qquad\qquad r_1 = \mathcal{F}(r_2)
\end{array}
$$

Figure 3: Statement involving communication

move dependences to outer loops. The extent to which messages are combined (the number of iterations of a particular loop for which combining is done) can now be controlled through loop tiling [21] to help exploit pipeline parallelism [9].

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 3. 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).

The characterization of a loop as type-1 or type-2 is always with respect to a particular *rhs* (right hand side) reference, since it depends on the outermost level at which communication for that reference can legally be placed. While a parallelizable loop is always type-1 (or can be transformed to become type-1) with respect to all *rhs* references in the statements inside it, an inherently sequential loop may also be type-1 with respect to a given *rhs* reference. Throughout the remainder of this paper, we shall refer to a loop simply as a type-1 or type-2 loop, where it is clear which *rhs* reference is being considered.

Any interprocessor data movement taking place over different iterations of a type-1 loop can always be implemented using a collective communication routine. However, when that data movement is not "regular enough" (or known at compile-time to be regular enough), the use of collective communication necessarily involves communication of extra values. In those cases, it may be better to carry out communication *inside* the type-1 loop, i.e., use repeated calls to the Transfer primitive inside that loop rather than a single call to collective communication primitive. For instance, consider the type-1 loop shown below:

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

The use of collective communication would involve each processor (on which B is distributed) sending the entire section of array B that it owns to all the processors on which A is distributed. This primitive is carried out only once, before entering the loop. However, it involves communication of larger amounts of data than necessary, and also requires each processor to allocate a greater amount of space to receive the incoming data. The other alternative that we mentioned is to carry out communication inside the loop. During each iteration, the owner of $B(D(i))$ is determined, and if it is different from the owner of $A(i)$, the value of $B(D(i))$ is communicated using a Transfer operation. Yet another alternative is to use the run-time compilation techniques developed by Saltz et al. [19] and Koelbel et al. [12]. The compiler generates an *inspector* that pre-processes the loop body at run-time to determine the communication requirements for each processor. The best method to use amongst these usually depends on the nature of the problem and the target machine characteristics. If the given loop itself appears inside another loop and the values of elements of D do not change inside that outer loop, using the inspector method is likely to be the best, since the overhead of inspector gets amortized over different iterations of the outer loop. Otherwise, if the target machine has a large set-up cost for sending messages, and has enough memory (given the data sizes used) on each node, it may be better to use collective communication. On a massively parallel machine tackling a large-sized problem, where the memory limitation on each node is more severe, the use of Transfer operations inside the loop may be the best, or the only choice.

Ideally, a compiler should choose amongst these alternatives only after evaluating these trade-offs based on the performance estimates [7] and taking into account the resource constraints. Further discussion of these issues is beyond the scope of this paper. However, we describe how either of these schemes may be used for synthesizing communication, once the choice has been made.

6 Identification of Communication Primitives

Given a statement of the form shown in Figure 3, the compiler determines the communication requirements for each *rhs* reference in that statement. For a pair of references, the *lhs* and the *rhs* references, potentially there is a need for interprocessor communication to handle data movement in each of their distributed dimensions. The nature of this data movement is identified by analyzing each pair of sub-references in the *lhs* and *rhs* references corresponding to the aligned dimensions (array dimensions distributed on the same mesh dimension). If the data movement takes place across processor boundaries in various mesh dimensions, the compiler generates separate primitives for those mesh dimensions, and composes these primitives in a definite order to implement the overall data movement.

If the *lhs* and the *rhs* arrays differ in the number of dimensions, the sub-references corresponding to the "extra" distributed dimensions of one array get paired up with sub-references corresponding to the "missing" dimensions of the other array. The "missing" subscript is regarded as being of the type *constant*. For each pair of sub-references, the larger of the values of *variation-level*, say, v, of the corresponding subscripts identifies the innermost loop in which data movement for that mesh dimension takes place. There are three possibilities regarding the value of v:

1. $l+1 \leq v \leq m$ – the loop identified is a type-1

loop. A detailed description is given below of how the data movement is analyzed, and an appropriate primitive chosen to implement the data movement.

2. $1 \leq v \leq l$ – the loop identified is a type-2 loop. A separate message (if any is required) is generated in every iteration of the loop. The primitive used is Transfer, and is carried out only if the elements corresponding to the two subscripts (for that iteration) get mapped to different positions, implying a need for interprocessor communication.

3. $v = 0$ – the subscripts corresponding to both the sub-references are of the type *constant*, and the data movement remains invariant inside every loop. As in the previous case, the mapping of elements corresponding to the two subscripts to the same position implies an internalization of data movement in that mesh dimension, i.e., no interprocessor communication. Otherwise, the primitive used is Transfer, to make up the processor difference along that dimension.

Data Movement in Type-1 Loops

The data movement taking place in different iterations of a type-1 loop can legally be combined. Therefore, the compiler attempts to recognize any regularity in that movement which makes it amenable to efficient implementation through collective communication primitive(s). The tests described below capture a wide range of patterns for which such special handling can be done. For those references on which these tests fail, the compiler can still use collective communication (involving some overcommunication), or a series of individual Transfers, identified by other techniques described in [9, 19, 12].

Given a type-1 loop, the extent of analysis required depends on the number of pair(s) of sub-references for which the value of variation-level v is equal to the loop level.

Single pair of varying sub-references Table 1 lists the collective communication primitive to be used if there is only one pair of sub-references varying in such a loop. This table enumerates the cases corresponding to only the "basic" categories of the subscripts for the *lhs* and *rhs* sub-references. The results for the other cases (when the two subscripts of the type single-index have different values of variation-level, or when one of the subscripts is of the type multiple-index) are derived in terms of these results, and are presented later in this section. The column marked "conditions tested" lists the tests performed by the compiler to obtain further information about the kind of data movement indicated by the given pair of sub-references. The entry *reduction op* represents the test to see if the *rhs* reference is involved in a reduction operation (such as addition, max, min) inside that loop. The case corresponding to both the sub-references being of the type single-index is perhaps the most commonly occurring pattern inside various loops in the scientific application programs. The choice of a primitive in that case is based on tests on the relationship between those sub-references. Figure 4 shows the different kinds of data movement taking place when different properties hold between the two sub-references. It may be noted that the *strictly synchronous* property is just a special case of the *1-synchronous* property, which is a special case of the *k-synchronous* property. Hence, the order in which these tests are performed, as indicated by their position in the table, is important.

Multiple pairs of varying sub-references The tests for k-synchronous properties between various sub-references in such cases help characterize the relationship between simultaneous external traversals of those sub-references in different mesh dimensions. Table 2 lists the collective communication primitive to be used if there are two pairs of sub-references varying in a type-1 loop. These results can be readily extended to handle any arbitrary number of pairs of sub-references. As in the previous table, all the entries for a sub-reference of the type single-index correspond to those with the same value of variation-level. The unnumbered properties listed under the "conditions tested" column are those which must be satisfied by the given sub-references if a special primitive is to be used. When those properties are indeed satisfied, the compiler checks the numbered conditions to decide which primitive to use. The tests for determining whether the indicated properties are satisfied by the sub-references have already been described in Section 4. The primitives listed in these cases are carried out over groups possibly spanning both the mesh dimensions corresponding to the two pairs of sub-references. The term "parallel Scatters", for instance, implies that more than one such group of processors is formed, and the Scatter operation is carried out over those groups from different originating processors (which may not belong to the group over which the Scatter is done). The results summarized in this table highlight the need for analyzing all the pairs of sub-references varying in a loop *together*. For instance, a sub-reference pair in which the *lhs* subscript is of the type single-index and the *rhs* subscript is a constant, by itself, suggests using a OneToManyMulticast primitive. However, in conjunction with another pair of sub-references varying in the same loop, the appropriate choice may turn out to be a completely different kind of primitive, such as Transfer, Scatter, or even Gather.

While it is only the data movement in a distributed array dimension that leads to interprocessor communication, the variation of subscripts for the sequentialized dimensions also influences the choice of communication primitive in the following case: appearance of a subscript of the type *constant* in the *rhs* sub-reference, and of the type *single-index* in the *lhs* sub-reference. This pair normally suggests the use of OneToManyMulticast, from a source processor to a group of processors. However, if that loop index also appears in the subscript for a sequentialized dimension in the *rhs* reference, it implies that the source processor has to send different pieces of data to different processors in the group. Hence in that case, the compiler chooses the Scatter primitive instead of the OneToManyMulticast.

Cyclic distributions All the results we have described above are valid for both kinds of array distributions, contiguous and cyclic. For cyclic distributions, however, another condition is added to the tests for the regularity of data movement. For every sub-reference with a subscript of the form $e = c_1 * i + c_2$, it is checked if b is a multiple of $c_1 * i_i$, where b is the block size of distribution of the given dimension, and i_i is the stride of the i-loop. This ensures that the data elements involved in any collective communication corresponding to the given sub-reference can be accessed on the local memories of involved processors with a constant stride.

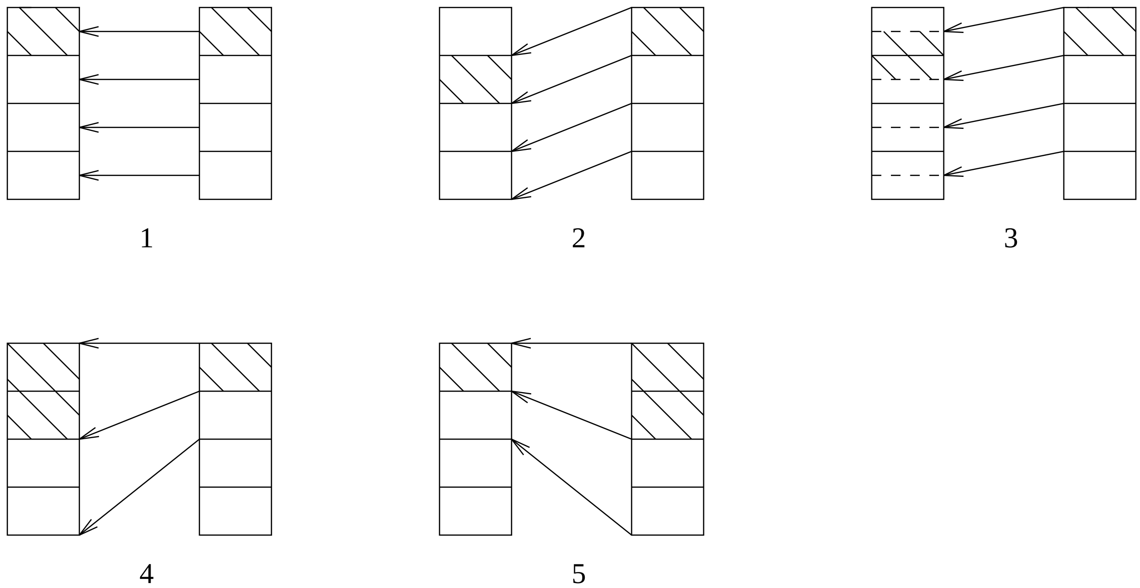

Figure 4: Data movements with single-index in both subscripts

LHS	*RHS*	*Conditions Tested*	*Commn. Primitive*
single-index	constant	default	OneToManyMulticast
constant	single-index	1. reduction op 2. default	Reduction Gather
single-index (s_1)	single-index (s_2) (same level)	1. s_1 strictly synch s_2 2. s_1 1-synch s_2 3. boundary-check 4. s_1 k-synch s_2, $k > 1$ 5. s_1 k-synch s_2, $k < 1$	internalized data movement (parallel) Transfers Shift (parallel) OneToManyMulticasts (parallel) Gathers

Table 1: Collective communication for single pair of sub-references

Different/multiple loop indices When the subscripts corresponding to a matched pair of sub-references involve different loop indices, or when one of the subscripts is of the type multiple-index, one simple way to analyze the data movement in terms of our earlier results would be to "freeze" all relevant loops except for the innermost one, for the purpose of analysis. This would involve *not* combining communication with respect to the frozen loops, and treating the corresponding loop indices as constants. Our compiler uses an extension of this idea, with *tiling* [21] instead of freezing of outer loops, so that the communication may be combined with respect to at least the inner tiles. Thus, all relevant loops except for the innermost one are tiled, and the tile loops (the outer ones from each loop split by tiling) are moved outside the level at which communication is placed. Now the indices corresponding to the loops split by tiling are regarded as constants for the purpose of identifying the communication primitives, and the results we have described above in Tables 1 and 2 can be used. Of course, the compiler has to determine the index ranges for these tiles.

Consider a pair of sub-references s_1 and s_2 with subscripts of the form $e_1 = c_1 * i + c_2$ (lhs), $e_2 = c_3 * j + c_4$ (rhs). The j-loop is tiled such that the starting points of the tiles are precisely the transition points of the j-loop for the sub-reference s_2. Now, within each tile, the value of j can be regarded as a constant for the purpose of using Table 1. This approach continues to work even when there is another pair of sub-references s_3 and s_4 that vary inside the j-loop. Similar to the kind of analysis shown in Table 2, the compiler checks if s_2 and s_4 are k-synchronous. If they are not, then communication is done entirely within the j-loop (implying a tile size of one, i.e., no tiling). Otherwise, the starting points of the tiles are set to the transition points of the j-loop for the sub-reference that leads to a greater number of transition points (s_4 if $k > 1$, s_2 otherwise). This process is illustrated through Example 3 in the next section.

The same idea of tiling the loops is used again for handling sub-references with subscripts of the type multiple-index. Consider a pair of sub-references s_1 and s_2 with subscripts of the form $e_1 = c_1 * i + c_2$ (lhs), $e_2 = c_3 * i + c_4 * j + c_5$ (rhs), distributed with block sizes b_1 and b_2 respectively. Let the original j loop vary from j_s to j_e. Using our results from Section 4, if the two sub-references satisfy the condition $c_1/b_1 = k * c_3/b_2$, the lowest admissible value of j ($j \geq j_s$) at which the two sub-references become k-synchronous with respect to the i-loop is determined. Given this value, say j_0, the j-loop is tiled such that the loop ranges for the successive tiles are: $j_s : j_0 - 1, j_0 : j_0 + b_2/c_4 - 1, \ldots, j_0 +$

LHS	*RHS*	*Conditions Tested*	*Commn. Primitive*
single-index (s_1) single-index (s_3)	single-index (s_2) constant (s_4)	s_1 k_1-synch s_2, s_3 k_2-synch s_2, s_1 k_3-synch s_3. 1. $max(k_1, k_2) > 1$ 2. $max(k_1, k_2) = 1$ 3. $max(k_1, k_2) < 1$	 (parallel) Scatters (parallel) Transfers (parallel) Gathers
single-index (s_1) single-index (s_3)	single-index (s_2) single-index (s_4)	s_1 k_1-synch s_2, s_3 k_2-synch s_4, s_1 k_3-synch s_3, s_2 k_4-synch s_4. 1. s_1 strictly synch s_2, s_3 strictly synch s_4. 2. $max(k_1, k_2) > 1$ 3. $max(k_1, k_2) = 1$ 4. $max(k_1, k_2) < 1$	 internalized data movement (parallel) Scatters (parallel) Transfers (parallel) Gathers
single-index (s_1) constant (s_3)	single-index (s_2) single-index (s_4)	s_2 k_1-synch s_1, s_4 k_1-synch s_1, s_2 k_3-synch s_4. 1. $max(k_1, k_2) > 1$ 2. $max(k_1, k_2) = 1$ 3. $max(k_1, k_2) < 1$	 (parallel) Gathers (parallel) Transfers (parallel) Scatters
single-index (s_1) single-index (s_3)	constant (s_2) constant (s_4)	s_1 k-synch s_3	OneToManyMulticast
single-index (s_1) constant (s_3)	constant (s_2) single-index (s_4)	s_1 k-synch s_4. 1. $k > 1$ 2. $k = 1$ 3. $k < 1$	 (parallel) Scatters (parallel) Transfers (parallel) Gathers
constant (s_1) constant (s_3)	single-index (s_2) single-index (s_4)	s_2 k-synch s_4 1. reduction op 2. default	 Reduction Gather

Table 2: Collective communication for multiple pairs of sub-references

$(m-1)*b_2/c_4 : j_0 + m*b_2/c_4 - 1, j_0 + m*b_2/c_4 : j_e$. The sizes of all the tiles, except for possibly the first and the last tile are the same. The tile loop is moved outside the loop level at which the communication is taking place. For the purpose of identifying the communication requirements (for the i-loop), now j is regarded as a constant.

In the future, we plan to extend our techniques to allow detection of properties analogous to the k-synchronous properties, between sub-references varying in different loops. That would enable the compiler to recognize data movements corresponding to operations like *transpose* in their generalized form, and use such primitives whenever available on the target machine.

Using collective primitives for unknown patterns We now describe how collective communication can be used for implementing data movement in a type-1 loop, when the compiler in unable to infer the precise pattern of data movement in that loop. The call to such a primitive is placed outside the type-1 loop, resulting in fewer calls to the primitive. However, the potential disadvantage of using a collective primitive in this case is the extra amount of data that needs to be communicated, because of the precise communication requirements being unknown at compile time. Table 3 shows the primitive used for a pair of sub-references in a loop when at least one of the subscripts is of the type *unknown*. In this case, there is no special treatment needed for multiple pairs of sub-references varying in a loop. Those pairs are analyzed independently, and the resulting primitives are composed in the normal manner.

Composition of primitives

Once the communication primitives corresponding to all the mesh dimensions have been obtained, they are composed together in a definite order, and inserted at the place determined by the compiler as shown in the previous section. Since these primitives implement the data movement in distinct mesh dimensions, they can legally be composed in any order. However, the order in which they are invoked is important because the position of each primitive affects the message sizes and the number of processors involved in subsequent primitives. It is desirable to obtain an ordering that leads to fewer processors getting involved and smaller messages handled by each processor, but sometimes, there is a trade-off between the two. Consider the following loop, where the arrays A and B are distributed in an identical manner on $N_1 * N_2$ processors.

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

The primitives required are: Gather in the first dimension, and Transfer in the second dimension. If Gather is

LHS	*RHS*	*Commn. Primitive*
unknown	constant	OneToManyMulticast
constant	unknown	Gather
unknown	single-index multiple-index unknown	ManyToManyMulticast
single-index multiple-index	unknown	ManyToManyMulticast

Table 3: Collective communication for sub-references involving unknowns

invoked first, it is carried out with a data size of n/N_1 words, over N_1 processors, followed by a single Transfer of n words of data. If this ordering is reversed, there are N_1 parallel Transfers that take place, each involving n/N_1 words, followed by a Gather operation, also with a data size of n/N_1 words, over N_1 processors.

The second ordering in the above example leads to the use of parallelism in implementing communication, and would yield better performance if this were the only communication being carried out on the mesh of processors. This suggests resolving the trade-off in favor of reducing the message sizes handled by processors. When there is no trade-off involved, the compiler must use an ordering that reduces the message sizes (m) and/or the number of processors involved (p). These considerations suggest the following ordering: 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).

While the internalized data movement does not represent any interprocessor communication, it does affect the message sizes and the number of processors involved, in the following manner. The external traversal of the sub-reference pair corresponding to such internalization along a mesh dimension determines the number of instances of communication over other dimensions that are carried out along the given dimension. The internal traversal of the sub-references on each processor affects the data size involved in each such instance of the communication. Consider another program segment involving the same arrays, A and B:

do $j = 1, n$
  do $i = 1, n$
    $A(i, j) = \mathcal{F}(B(i, c_1))$

The internalized data movement along the first dimension spans all the N_1 processors. Thus, there are N_1 instances of the OneToManyMulticast primitive in the second dimension that are carried out in parallel (corresponding to different processor positions along the first dimension). Also, the internal traversal of the sub-references along the first dimension leads to the data size for the OneToManyMulticast being set to n/N_1 elements.

Prior to determining the order in which various primitives are invoked, the compiler may combine primitives of the same type, operating over different dimensions, into a single primitive over each of those dimensions. The different Transfer primitives covering processor-number differences in various mesh dimensions may be combined into a single Transfer primitive covering the overall processor-difference (between the source and the destination processors) in each of those dimensions. Similarly, a collective communication operation such as a OneToManyMulticast from one processor to all processors along mesh dimensions d_1 and d_2 can be combined into a single OneToManyMulticast operation over a bigger group covering both the mesh dimensions.

7 Examples

We now present some examples illustrating the choice of communication primitives for various data references inside type-1 loops. In Examples 2 and 3, the arrays A and B are assumed to be distributed identically on $N_1 * N_2$ processors.

Example 1 In the following program segment, the array A is distributed by rows. Both the array dimensions A_1 and B_1 have a contiguous distribution, and their block sizes of distribution are $n/16$ and $3 * n/16$ respectively.

do $i = 1, n$
  $A(i, 1) = \mathcal{F}(B(3 * i), B(3 * i - 1), B(3 * i - 2))$
enddo

All of the sub-references $\langle B(3 * i), 1\rangle, \langle B(3 * i - 1), 1\rangle$, and $\langle B(3 * i - 2), 1\rangle$ are strictly synchronous with the sub-reference $\langle A(i, 1), 1\rangle$ for the i-loop. Hence, the data movement is internalized, and no communication is required.

Example 2

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

Both pairs of sub-references are varying inside the same loop. As shown in Table 2, the choice of primitive in this case is governed by the test for the k-synchronous property between sub-references $\langle A(i, 1), 1\rangle$, and $\langle B(1, i), 2\rangle$. For the cases $N_1 = 4, N_2 = 2$, and $N_1 = 2, N_2 = 4$, Figures 5 and 6 show the use of Scatter and Gather primitives respectively, over groups of 2 processors each. This follows from the sub-reference $\langle A(i, 1), 1\rangle$ being 2-synchronous and 1/2-synchronous, respectively, with the sub-reference $\langle B(1, i), 2\rangle$, in those two cases.

Example 3

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

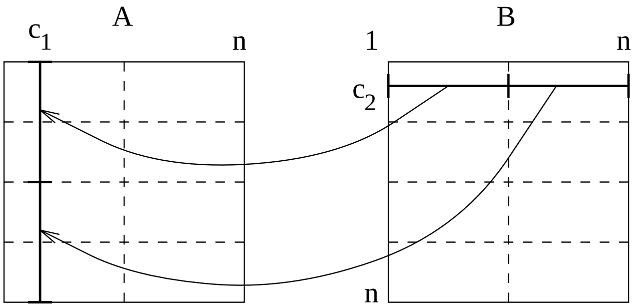

Figure 5: Parallel Scatters in Example 2

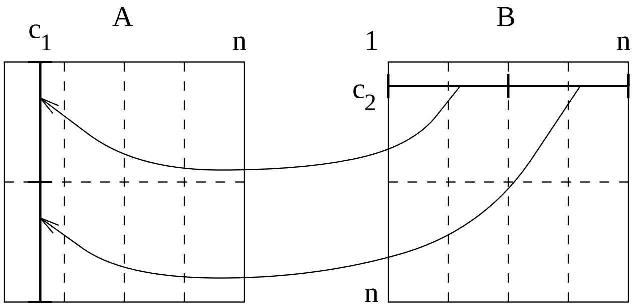

Figure 6: Parallel Gathers in Example 2

Figure 7 shows the data movement when $N_1 = 2, N_2 = 4$. The first pair of aligned dimensions have subscripts i and j varying in different loops. The j-loop is tiled, and j is regarded as a constant for the purpose of identifying the communication primitive, OneToManyMulticast. The sub-reference $\langle B(j,j), 2\rangle$ is 2-synchronous with $\langle B(j,j), 1\rangle$ with respect to the j-loop. Hence, tiling is done such that the starting point of the mth ($m \geq 1$) tile is set to the mth transition point of the j-loop for $\langle B(j,j), 2\rangle$, which is determined to be $(m-1) * n/N_2 + 1$.

8 Run-Time Support

The ideas we have presented so far on identifying communication primitives have been implemented as part of a system that, given a sequential program and parameters for data partitioning, statically estimates the communication costs of the program when parallelized on a multicomputer [7]. Our system has been built on top of the Parafrase-2 system [15], which is used to set up data structures describing the syntax of the given

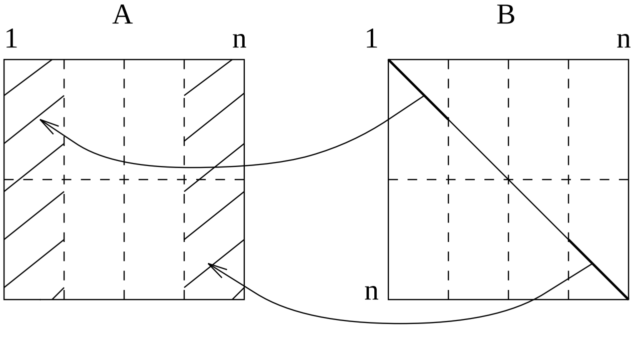

Figure 7: Parallel OneToManyMulticasts in Example 3

program and the dependences in it. In order to generate code implementing the required communication, the compiler has to further specify the actual processors and the data elements involved in each communication. This section identifies certain enhancements needed in high-level communication libraries, which we believe would go a long way in supporting the effective use of collective communication routines by the compiler, and also by the user.

An essential step in carrying out collective communication is to first group together (at least logically) the relevant processors after identifying the external traversal of the given sub-references for the given loop. Often, the loop range is such that the external traversal does not cover all the processors along the mesh dimension. Further, whenever there are multiple pairs of sub-references varying in a single loop, the groups over which communication is to be carried out may span more than one mesh dimension. For instance, in a 2-D mesh of processors, there may be a need to group together a certain number of processors along the diagonal. In order to handle these cases which do arise often enough, the high-level communication libraries should be designed to allow such groups to be formed, and support efficient collective communication over the restricted set of processors in each such group. This would also allow the processors not involved in the communication to proceed with their respective computations while the processors in each group that is created carry out communication within that group. We suggest the following way of defining the scope of this kind of grouping – just like representations of groups of data, such as *regular section descriptor* (RSD) [3] and *data access descriptor* (DAD) [1], the libraries should also support similar representations of groups of processors.

The techniques presented in this paper allow the compiler to identify the sections of an array that are involved in a collective communication. These sections can be represented using the RSD representation, and in more general cases, with the DAD representation. However, in the calls to collective communication routines, most current libraries require the message to be specified as a contiguous chunk of data. Thus, there is a distinct need for support for packing and unpacking of messages, i.e., for conversion between the contiguous buffer representation for a message and the representation of an array section convenient to a compiler (and the user) such as a DAD.

9 Conclusions

We have presented a methodology for determining the collective communication primitives to use at various points in the program, for implementing the data movement. Our approach is quite comprehensive in terms of its ability to handle different kinds of subscripts, and in terms of the variety of communication primitives used. Much of the analysis presented in this paper has been implemented, we are currently working on the problem of actual code generation. Towards that end, we have identified some key areas of improvements required in the design of high-level communication libraries, which should help not only the compiler, but also the user who explicitly uses collective communication routines in programs. Another related code generation issue that we are working on is the management of space for non-local data received through messages.

There are a number of possible extensions to our approach that we plan to examine. Currently, we analyze the communication requirements of only one *rhs* reference at a time. We plan to extend those techniques so that multiple *rhs* references may be analyzed together. That would allow the compiler to perform optimizations like combining messages corresponding to different array references, or to different arrays. We are also trying to integrate the processes of synthesis of communication and the estimation of communication costs. Amongst other benefits, that would allow the compiler to be guided by these estimates. Hence, decisions at various steps in the synthesis process, such as determining whether to carry out communication repeatedly inside a loop, or use fewer, but potentially wasteful collective communication routines, could be taken based on such estimates.

Acknowledgements

The authors wish to thank Vas Balasundaram and Jeanne Ferrante for the numerous helpful discussions, and the anonymous referees for their useful comments.

References

[1] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: the data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[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*, April 1990.

[3] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proc. First International Conference on Supercomputing*, Athens, Greece, 1987.

[4] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Yale University, December 1989.

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

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

[7] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proc. 6th International Parallel Processing Symposium*, Beverly Hills, California, March 1992.

[8] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[9] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proc. Supercomputing '91*, Albuquerque, NM, November 1991.

[10] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proc. Fifth Distributed Memory Computing Conference*, April 1990.

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

[12] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

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

[14] Parasoft Corporation. *Express User's Manual*, 1989.

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

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

[17] 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.

[18] 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.

[19] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. Technical Report ICASE 90-59, Institute for Computer Applications in Science and Engineering, Hampton, VA, September 1990.

[20] M. J. Wolfe. Loop rotation. In *Proc. 2nd Workshop on Languages and Compilers for Parallel Processing*, Urbana, IL, August 1989.

[21] M. J. Wolfe. More iteration space tiling. In *Proc. Supercomputing 89*, Reno, Nevada, November 1989.

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