PTRAN II - A Compiler for High Performance Fortran

M. Gupta S. Midkiff E. Schonberg P. Sweeney K.Y. Wang M. Burke

{*mgupta,midkiff,schnbrg,pfs,kyw,burkem*}*@watson.ibm.com*

I.B.M. T.J Watson Research Center
T.J. Watson Research Center, Hawthorne
P O Box 704, Yorktown Heights, NY 10598
USA

Abstract

New languages for distributed memory compilation, including High Performance Fortran (HPF), provide user directives for partitioning data among processors to specify parallelism. We believe that it will be important for distributed memory compilers for these languages to provide not only efficiency close to a hand-written message passing program, but also flexibility, so that the user is not always required to provide directives nor specify the number of physical processors. We describe distributed-memory compilation techniques for both optimization and flexibility.

1 Introduction

The PTRAN II compiler is a prototype compiler for High Performance Fortran (HPF)[6] that we are developing as a testbed for experimenting with distributed memory compilation techniques, including automatic data partitioning and parallelization, cost modeling, and global communication optimizations. Our design builds on the existing work in distributed memory compilation, including [20], [13, 12], [18], [2], [16] and [7]. Our input language can be either Fortran 77 or Fortran 90, after all array and `forall` constructs have been scalarized into Fortran 77. HPF directives provided by the user guide the partitioning of data and computation. The compiler produces SPMD node programs, which contain explicit message-passing communication primitives.

Our aim is to provide both efficiency and flexibility. By flexibility we mean being able to compile when the number of processors, array bounds, or reaching data distributions is not known at compile-time. Flexibility and efficiency are generally contradictory goals, and may not be achievable for the same compilation instance. However, we believe that flexibility will become an issue for eventual usability of this technology, so that it important to learn how to provide flexibility without sacrificing too much efficiency. This flexibility is, in fact, required to support the full HPF language.

This paper describes the architecture of the PTRAN II compiler. Some of the features described (in the present tense) have already been implemented. Others described (in the future tense) are not yet implemented. The next two subsections give background on HPF, and an outline of the compiler architecture.

1.1 HPF Features

In HPF, the distribution and alignment of data objects is specified with a two level mapping. At the lowest level of this mapping, templates[1] and arrays can be distributed onto the processor grid. *Block*, *cyclic*, *block-cyclic* and *collapsed* distributions are supported. If the distribution for a dimension d of the template or array A is not collapsed, then the dimension is said to be *partitioned*.

Let A_k be the k'th dimension of A, and A_K be the dimension of A that is mapped onto the k'th dimension of a a processor grid or template. For simplicity, we assume that arrays, templates and processor grids are 0-origined. The function $f(A_K, i)$ returns the index into dimension k of the processor grid that element i of A_K is mapped onto. For block distributions, this function is:

$$f(A_K, i) = \left\lfloor \frac{i}{bs} \right\rfloor$$

[1] A template is an abstract index space, and can be thought of as an array without storage

```
DIM B(99,49), C(49,2:100), D(100,100)
PROCESSOR P(4,5)
TEMPLATE T(100,100)
DISTRIBUTE T(BLOCK,BLOCK) ONTO P
ALIGN B(I,J) WITH T(J+1,2*I+1)
ALIGN C(I,J) WITH B(J-1,I)
DISTRIBUTE D(CYCLIC,CYCLIC) ONTO P
```

Figure 1: An example of chains of alignment

where bs is the *blocksize* of A_K, i.e. the number of elements of A_K on each element of dimension k of the processor grid. In the example of Figure 1, the blocksize for T_1 is 25, and the blocksize for T_2 is 20. For block-cyclic distributions,

$$f(A_K, i) = \left\lfloor \frac{i}{bs} \right\rfloor \bmod N_K$$

where N_K is the extent of dimension k of the processor grid. In Figure 1, N_1 for P is 4 and N_2 is 5. Cylic distribution is a special case of the above, where $bsis1$.

At the next level of the two level mapping, arrays can be (optionally) aligned with arrays and templates that have been distributed. *Alignment chains* can be arbitrarily long, e.g. in Figure 1, C is aligned with B which is aligned with T which is distributed onto P.

The effects of alignment can also be specified as the alignment function $g(B_K, i) = c * i + Off$, where k is the dimension of the alignment target that B_K is aligned to. Thus, for B, $g(\mathtt{B}_2, i) = 2 * i + 1$; that is, every element i of the first dimension of B is mapped to element $2 * i + 1$ of the second dimension of T.

Thus, an array dimension is either collapsed or distributed over a processor grid dimension. The distribution may consist of

1. replication;
2. mapping to a constant processor position;
3. or partitioned.

If partitioned, the mapping can be specified with a mapping function of the form

$$f(A_K, i) = \left\lfloor \frac{s * i + Off}{bs} \right\rfloor [\bmod N_K]$$

The term mod N_K is needed only for the cyclic and block-cyclic distributions. The term $s*i+Off$ for array A is found by composing the alignment functions of the chain of alignments which map A onto its ultimate alignment target.

Depending on user-supplied directives, there are situations where precise information about the layout of data on processors is known, and therefore, the compiler should make full use of this information to produce efficient programs. There are also situations, however, where little or no information is available, and the compiler should not unacceptably degrade performance. Imprecise compile-time information arises from the following situations.

Alignment and distributions are dynamic – they can change during program execution. Because alignment and distribution may change, many operations (e.g. communication analysis and computation partitioning) that are desirable to perform during compilation may sometimes have to be postponed until runtime. Therefore, a run-time library to query alignment and distribution information, and act upon that information (perform communication, selectively execute a statement depending on whether the left hand side of the statement resides on the processor) is necessary.

Also, alignments and distributions need not be specified. When not specified the compiler must provide an alignment and distribution for the object. If alignment and distribution information is specified, it is advisory only and may be ignored by the compiler. This allows the compiler to improve upon the programmer's directives, and allows sophisticated optimizations of the data layout to be performed without violating the language standard.

Finally, the distribution of a data object may not be known at compile-time because the bounds of arrays or processor grids may not be known until run-time.

1.2 Architecture of the compiler

The major phases of the PTRAN II compiler are shown in Figure 2. The first phase builds distribution and alignment information. As mentioned above, the distribution of a data object may be defined via a long chain of alignments. This phase collapses this chain into a canonical form, and stores the resulting information in the HPF symbol table. The canonical form for partitioned data is the distribution function discussed in Section 1.1.

Next loop transformations are performed. Loop transformations enable more parallelism to be uncovered and to allow more efficient communication and computation partitioning to be performed. This phase is discussed in Section 6.

The next phase performs automatic data partitioning. This must be performed for scalars and all arrays for which distribution information was not specified.

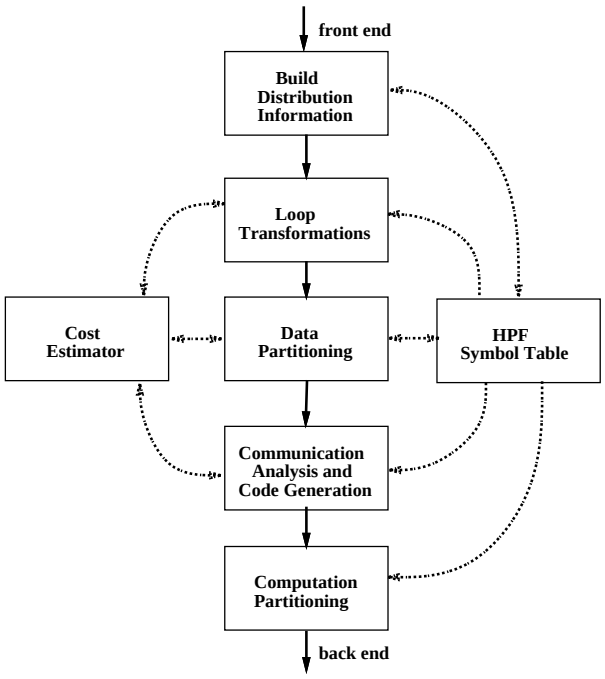

Figure 2: The architecture of the PTRAN II compiler

A cost estimator is used to guide the selection of various parameters of data partitioning. This phase is covered in more detail in Section 2.

The communication analysis phase selects the communication operation to be used (e.g. `many-to-many multicast` or `send/receive`), and determines at what level in the loop nest the communication operation can be performed. Numerous communication optimizations are also performed by this pass. This phase is discussed in Section 3.

Next, communication code generation inserts code for allocating buffers and performing communication operations into the intermediate form of the program maintained by the compiler. Also, this phase composes communication operations across a processor grid's dimensions. This phase is discussed in Section 4.

At this point, the program is converted into an SPMD program by the computation partitioner, which is guided by the owner computes rule. A combination of modifying the bounds of loops and the insertion of statement guards is used to restrict execution of statements to the appropriate processor. This pass is discussed in greater detail in Section 5.

During most of compilation, addressing and indexing is assumed to occur in the global data space. The program is translated into the local address space very late in the SPMDization process. This is to simplify analysis – for example dependence analysis must be performed in the global space.

2 Automatic Data Partitioning

While HPF assigns the primary responsibility of determining data distribution to the programmer, we believe a good HPF compiler should also perform automatic data partitioning for the following reasons:

1. It is the compiler which generates and optimizes communication, hiding those details from the programmer. It should warn the programmer if certain directives are expected to lead to bad performance. Our experience has shown that a compiler can often make intelligent decisions about data partitioning for programs with regular computations [10, 7].

2. The programmer may not specify partitioning for *all* arrays, since the directives are optional. The compiler must map those arrays which are not partitioned by the directives.

3. The compiler must always determine the distribution of array temporaries generated during the compilation process.

In addition, the compiler is required to assign ownership to each scalar variable.

2.1 Distribution of Arrays

For partitioning arrays, PTRAN II will use an approach similar to the PARADIGM compiler [10, 7], which we shall describe briefly. The data partitioning decisions are made in a number of distinct passes through the program, as shown in Figure 3. The `align dimension` pass determines the mutual alignment among array dimensions, and aligns each array dimension with a unique dimension of a template. The `align stride-offset` pass determines the stride and the offset in each alignment specification, i.e., for a specification of the form "`A(`i`)` ALIGNED WITH `T(`$c * i + Off$`)`", where `A` is an array and `T` is a template, it determines the values of c and *Off*. The `block-cyclic` pass determines for each template dimension, whether it should be distributed in a blocked or cyclic manner. The `num-procs` pass determines the number of processors across which each template dimension should be distributed.

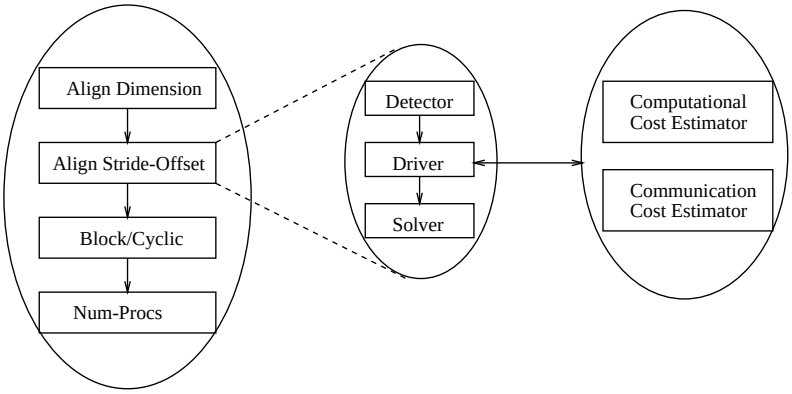

Figure 3: Overview of automatic data partitioning in PTRAN II

In each pass, the `detector` module first analyzes array references in each statement, and identifies desirable requirements on the relevant distribution parameter of each array. These desirable requirements are referred to as *constraints* [8]. In order to handle conflicting requirements on data distribution from different parts of the program, the `driver` module determines a *quality measure* for each constraint. The quality measure captures the importance of each constraint with respect to the performance of the program, and is obtained by invoking the communication cost estimator and/or the computational cost estimator. Once all the constraints and their quality measures have been recorded, the `solver` determines the value of the corresponding data distribution parameter. Essentially, the `solver` obtains an approximate solution to an optimization problem, where the objective is to minimize the execution time of the target data-parallel program.

For example, in the `block-cyclic` pass, the `detector` records a constraint favoring block distribution of a template dimension if an array dimension aligned to it is involved in a nearest-neighbor communication. It records a constraint favoring cyclic distribution if the computations across an aligned array dimension are better load balanced with cyclic distribution. In the above two cases, the communication and the computational cost estimators are respectively invoked to record the quality measure, which is the difference between the estimated costs when the array dimension is given a block (cyclic) distribution and when it is given a cyclic (block) distribution. Once these constraints are recorded, the `solver` compares the sum of quality measures favoring block distribution and those favoring cyclic distribution for each template dimension, and chooses the method of partitioning corresponding to the higher value.

We shall modify the above procedure for automatic data partitioning to incorporate the data distribution directives supplied by the user. Such directives may be regarded as constraints with very high values of quality measures, so that they are always honored. We shall also provide an option where the compiler might warn the programmer about expected bad performance resulting from a directive, if the program is amenable to static analysis.

2.2 Distribution of Scalars

Many distributed memory compilers simply replicate scalars on all processors. However, that can potentially lead to a great deal of unnecessary communication for a scalar assignment. PTRAN II chooses between replication and non-replicated alignment of a scalar variable. The ownership information for the second kind of scalar is specified in terms of an alignment relationship with a non-replicated variable. This subsumes privatization, and also includes the possibility of ownership by a single processor. PTRAN II uses the *static single assignment* (SSA) representation [4] to associate a separate ownership decision with each assignment to a scalar.

Initially, all scalars are assumed to be replicated. For each scalar definition, the compiler checks if the scalar can be privatized with respect to the innermost loop in which the assignment appears. If the analysis shows that the scalar can be privatized and if the given definition is not live outside the loop, the scalar variable is marked as a potential candidate for non-replicated alignment. If such a scalar is required to be available on all processors (for which the compiler checks if that variable is used as a loop bound or inside a subscript expression for an array reference), the variable is actually replicated. Otherwise, the compiler identifies a reference corresponding to a parti-

```
!HPF$ DISTRIBUTE (BLOCK) ::  A
DO I = 1, N
   ...
   T = A(K)
   A(K) = A(I)
   A(I) = T
ENDDO
```

Figure 4: Non-replicated alignment of a scalar

tioned variable in the given assignment statement to record the alignment information for the scalar. For example, in the program segment shown in Figure 4, the variable `T` is aligned with `A(K)`. Clearly, if all data items on the `rhs` of the assignment are themselves replicated, the scalar is effectively replicated too.

Any scalar computed in a reduction operation (such as sum) being carried out across a processor grid dimension is given special treatment. The definitions corresponding to its initialization and the reduction computation are handled together, and there is another copy created of that variable, that yields two versions. The first version is privatized, and represents the local reduction computation performed by each processor. The second version stores the result of global reduction, and may either be replicated or given a non-replicated alignment based on the procedure discussed above.

For example, consider the program shown in Figure 5. PTRAN II creates an additional version of `S`, referred to as `$S`. Due to reduction taking place across the second processor grid dimension, `$S` is privatized along that dimension, and aligned with `A(I,J)`. The scalar `S` is determined to be privatizable with respect to the I-loop, and is aligned with `B(I)`. This enables statements s_1 and s_3 to be executed without any communication, and the use of a reduction communication primitive in the second grid dimension.

3 Communication Analysis

PTRAN II uses analysis based on [9] to determine the communication required for each reference in the program. We have extended the analysis described in [9] to handle scalars, and to incorporate additional possibilities regarding the distribution of array dimensions, namely, replication and mapping to a constant processor position. We shall first describe how communication requirements are determined individually for each

```
!HPF$ PROCESSORS P(10,10)
!HPF$ ALIGN B(I) WITH A(I,1)
!HPF$ DISTRIBUTE (BLOCK, BLOCK) ::  A
DO I = 1, N
   S = B(I)                    :  s1
   DO J = 1, N
      S = S + A(I,J))          :  s2
   ENDDO
   B(I) = S                    :  s3
ENDDO
```

Figure 5: Scalar assignment in a reduction computation

reference. Later, we shall describe some global optimizations that can be used to reduce the overall cost of communication.

3.1 Analysis for Single Reference

For each read reference in the program, the communication analysis module first determines whether it requires interprocessor communication. If communication is needed, it determines (i) the placement of communication, and (ii) the communication primitive that will carry out the data movement in each grid dimension. The code generator then identifies the processors and data elements that need to participate in communication, and composes the primitives over different grid dimensions.

In order to facilitate communication analysis, PTRAN II classifies each array subscript into one of the following types: `constant`, `single-index` (affine function of a single loop induction variable), `multiple-index` (affine function of more than one loop induction variable), or `complex`. Information is also kept about its `variation-level`, i.e., the innermost loop in which the subscript changes values. We refer to a subscript associated with a distributed dimension in an array reference as a *sub-reference*.

3.1.1 Placement of Communication

Consider a read reference appearing in a statement inside one or more loops. If possible, the communication for that reference should be moved outside of loops to enable *message vectorization*, which combines a sequence of messages on individual array elements into a single message [13, 20]. The compiler examines all incoming flow dependences to the reference, and the innermost loop carrying any of those dependences identifies the loop from which communication

cannot legally be moved outside. Thus, using dependence analysis, the compiler determines the outermost loop level at which communication may be placed for a given reference.

The extent to which messages are actually combined can next be controlled by stripmining loops, whenever necessary. If any sub-reference is of the type `complex`, the compiler does not allow communication to be taken outside the loop corresponding to the `variation-level` of that sub-reference, to avoid excessive and potentially extraneous communication. Future versions of PTRAN II will use the runtime compilation techniques developed by Saltz et al. [19]. In the presence of a sub-reference of the type `multiple-index` as well, PTRAN II does not combine communication with respect to all the loops. Techniques like tiling can be used in such cases to allow greater combining of messages, without the overhead of extraneous communication [9].

3.1.2 Identification of Communication Primitive

We first describe how for each grid dimension, communication primitives are identified for a `rhs` reference in an assignment statement. By the *owner computes* rule, data is sent to the owner of the `lhs` reference. The first step is to identify the pairs of sub-references in the `lhs` and the `rhs` reference corresponding to aligned dimensions. For a pair of aligned sub-references, a comparison of `variation-levels` identifies the innermost loop in which data movement for that grid dimension takes place. If it is a communication-independent loop, i.e., if communication is placed outside that loop, then data movement over the grid dimension for the entire loop can be implemented using a collective communication primitive. Otherwise, the primitive used is a `send-receive`. The use of collective communication, such as broadcast and reduction, can significantly improve the performance on a massively parallel machine, and also leads to more concise and high-level code.

The choice of collective communication primitive depends on the type of sub-references and on the distribution parameters of the array dimensions. We shall only describe the most common case when the aligned array dimensions are distributed in a block, cyclic, or block-cyclic manner. Using similar analysis, PTRAN II is also able to handle other cases, where either of the array dimensions may be replicated or mapped to a constant processor position. Table 1 lists the selected communication primitive, given a pair of sub-references of type `constant` or `single-index`, varying in a communication-independent loop. When both the sub-references are of the type `single-index`, PTRAN II performs certain tests that are described below.

Synchronous Properties of Sub-references Consider two sub-references, $s_1 = \alpha_1 * i + \beta_1$, and $s_2 = \alpha_2 * i + \beta_2$. Let the distribution functions of the corresponding array dimensions be $f(d_1, i) = \lfloor \frac{c_1 * i + Off_1}{b_1} \rfloor [\mathrm{mod} N_K]$, and $f(d_2, i) = \lfloor \frac{c_2 * i + Off_2}{b_2} \rfloor [\mathrm{mod} N_K]$.

The sub-reference s_1 is said to be *strictly synchronous* with s_2, with respect to their common loop of variation, if both sub-references are mapped to the same processor position for all iterations of the loop. This property is satisfied if either of the following sets of conditions hold (the proof is given in [7]):

- (i) $(c_1 * \alpha_1)/b_1 = (c_2 * \alpha_2)/b_2$, and
  (ii) $(c_1 * \beta_1 - Off_1)/b_1 = (c_2 * \beta_2 - Off_2)/b_2$,
  or

- (i) $b_1 = m * c_1 * \alpha_1$, (ii) $b_2 = m * c_2 * \alpha_2$, and (iii) $\lfloor (c_1 * \beta_1 - Off_1)/(c_1 * \alpha_1) \rfloor = \lfloor (c_2 * \beta_2 - Off_2)/(c_2 * \alpha_2) \rfloor$,
  where m is a positive integer.

The sub-reference s_1 is said to be *k-synchronous* (k being an integer) with s_2, with respect to their common loop of variation, if at every iteration point where s_1 crosses a processor boundary, s_2 crosses the next kth processor boundary. The conditions to check for this property are obtained in a similar manner, as shown below:

- (i) $(c_1 * \alpha_1)/b_1 = k * ((c_2 * \alpha_2)/b_2)$, and
  (ii) $(c_1 * \beta_1 - Off_1)/b_1 = k * ((c_2 * \beta_2 - Off_2)/b_2) + l$,
  where l is an integer,
  or

- (i) $b_1 = m * c_1 * \alpha_1$, (ii) $b_2 = k * m * c_2 * \alpha_2$, and (iii) $\lfloor (c_1 * \beta_1 - Off_1)/(c_1 * \alpha_1) \rfloor = \lfloor (c_2 * \beta_2 - Off_2)/(c_2 * \alpha_2) \rfloor + m * l$,
  where m and l are integers, and $m > 0$.

The "boundary-communication" test helps detect nearest-neighbor communication. It checks for the following conditions:

1. $c_1 * \alpha_1/b_1 = c_2 * \alpha_2/b_2$.

2. $|(c_1 * \beta_1 - Off_1)/b_1 - (c_2 * \beta_2 - Off_2)/b_2| \leq 1$.

LHS	*RHS*	*Conditions Tested*	*Commn. Primitive*
single-index	constant	default	`broadcast`
constant	single-index	1. reduction op 2. default	`reduce` `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-comm check 4. default	no communication (parallel) `send-receives` `shift` sequence of `send-receives`

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

Coupled Sub-references In the presence of coupled sub-references (more than one sub-reference of an array, varying in a single loop), the analysis shown in Table 1 is not sufficient. In such cases, the compiler has to identify the relationship between simultaneous movement of sub-references in each of the grid dimensions. An extension of the analysis shown here helps detect when there are non-overlapping groups of processors (spanning multiple grid dimensions) over which collective communications may be carried out [9]. However, due to the complexity of forming processor groups, PTRAN II currently uses `send-receive` between processor pairs in those cases.

3.1.3 Conditional Statements

A naive distributed memory compiler would force the execution of a conditional statement on every processor. In such a case, data needed for evaluating the condition has to be broadcast to all processors, if it is not already replicated. As an optimization, PTRAN II examines all statements inside the scope of the conditional, and obtains a pseudo-`lhs` reference which "covers" all other `lhs` references in terms of mapping to processors. A test for checking whether $\texttt{lhs}_1$ "covers" $\texttt{lhs}_2$ is equivalent to checking whether an assignment statement of the form $\texttt{lhs}_1 = \texttt{lhs}_2$ requires any communication under the owner computes rule. In the worst case when the conditional statement has to be executed on all processors, the pseudo-`lhs` reference is to a dummy scalar variable replicated on all processors. Once the pseudo-`lhs` reference is obtained for a conditional statement, communication analysis is carried out using exactly the same procedure as that for an assignment statement.

For example, in Figure 6, PTRAN II obtains "`A(I)`" as the pseudo-`lhs` reference for the *if* statement, and further analysis shows that no communication is necessary for the reference to `C(I)`.

```
!HPF$ ALIGN B(I) WITH A(I)
!HPF$ ALIGN C(I) WITH A(I)
!HPF$ DISTRIBUTE (BLOCK) ::  A
DO I = 1, N
   IF (C(I) .NE. 0.0) THEN
      A(I) = A(I) / C(I)
      B(I) = 2 * B(I)
   END IF
ENDDO
```

Figure 6: Communication analysis for IF statement

3.2 Global Optimizations

So far we have described the communication analysis for a single reference, and various optimizations that are performed with regard to that communication. We now describe some global optimizations that help reduce the overall cost of communication for the program.

Message Coalescing

Message coalescing [13] involves combining messages between the same pair of processors, that correspond to different references. `Ptran` II can detect the possibility of coalescing messages corresponding to references even in different statements by symbolic analysis *before* the actual sets of processors and the array sections involved are identified. Consider two sets of communications that are placed at the same program point (for instance, in the preheader of a loop). Let the communications correspond to `rhs` references r_1 and r_2, appearing in statements with `lhs` references l_1 and l_2 respectively. Clearly, all messages corresponding to the two references may be combined if r_1 and r_2, and similarly l_1 and l_2, are mapped to the same processors,

for all instances of those references corresponding to different loop iterations. The compiler detects this in constant time by carrying out an extended version of the test for strictly-synchronous property between all pairs of sub-references corresponding to aligned dimensions.

Eliminating Redundant Communication

Most distributed memory compilers generate communication for data that is not owned by the processor which needs it. If the data is already available locally as a result of prior communication, and if it has not been modified by its owner, the compiler can eliminate that communication, and make the processor use its local copy. PTRAN II will use a data-flow analysis framework that has been developed to keep track of the availability of data at processors, at different points in the program [11]. Such a framework can also be used to relax the "owner sends" rule, so that any processor having a valid copy of data, not just its owner, can be the sender.

4 Communication Code Generation

Because a program's data is distributed among processors, data movement may be required to perform computation. The communication code generation phase of the PTRAN II compiler performs data movement by generating calls to communication library routines. The input to the communication code generator is the communication analysis information and an HPF program with data distribution annotations. The output is the HPF program augmented with communication code.

If possible, the communication code generator attempts to determine, at compile-time, the actual data items that need to be communicated between processors, and for each data item the processor that owns it and the processors that need it. This can be accomplished by computing the following information for each dimension of an array reference. These computations are a simple extension of Koelbel's work [16].

- *Processor Set*: $PS(sr, L)$ - set of processor positions along a grid dimension traversed by the sub-reference sr in loop L.
- *Local Iteration Set*: $LIS(sr, L, p)$ – set of iterations of loop L in which the sub-reference sr is mapped to processor position p.

The above sets are used to determine the communication required between any arbitrary pair of processors, with the *owner computes* rule. The first step is to determine the iterations in which communication takes place.

- *Send Iteration Set*: $SIS(l, r, L, p, q)$ – set of iterations of L for which the processor at position p has to send data to the processor at position q, corresponding to the sub-references l (`lhs`) and r (`rhs`). It is computed using the following equation:

$$SIS(l, r, L, p, q) = LIS(r, L, p) \cap LIS(l, L, q)$$

- *Receive Iteration Set*: $RIS(l, r, L, p, q)$ – set of iterations of L for which the processor at position p has to receive data from processor at position q, corresponding to the sub-references l (`lhs`) and r (`rhs`). It is computed using the following equation:

$$RIS(l, r, L, p, q) = LIS(l, L, p) \cap LIS(r, L, q)$$

The array subscript positions corresponding to the data being sent/received, referred to as data sets, can now be determined by applying the subscripting function of the `rhs` sub-reference to the above iteration sets.

When the data and processor sets can be computed at compile-time for a reference to a partitioned array, the communication code generator composes the communication primitives across the corresponding processor grid's dimensions. Let the tuple, $(P_s, d(p_s), P_r, d(p_r))$, denote the information that we will use to compose communication where P_s and P_r are the send and receive processor sets and $d(p_s)$ and $d(p_r)$ are the send and receive data sets for that dimension. For a two dimensional array reference, such as B in Figure 7 the composition would be as follows if communication in the first dimension occurs first:

$$1.(\langle P_s^1, P_s^2\rangle, d(\langle p_s^1, p_s^2\rangle), \langle P_r^1, P_s^2\rangle, d(\langle p_r^1, p_s^2\rangle))$$

$$2.(\langle P_r^1, P_s^2\rangle, d(\langle p_r^1, p_s^2\rangle), \langle P_r^1, P_r^2\rangle, d(\langle p_r^1, p_r^2\rangle))$$

where P^n refers to the n^{th} dimension of P. This approach works for any multi-dimensional communication.

Although executing the communication operations in any order provides correct results, the ordering may have significant impact on the cost of communication. The communication code generator orders communication primitives to first reduce message length and

```
REAL B(3,18)
!HPF$ PROCESSORS P(3,3)
!HPF$ DISTRIBUTE B(BLOCK, BLOCK)
DO I = 1, N
   B(1,1) = B(1,1) + B(N,I)
END DO
```

Figure 7: An example to illustrate communication composition across a processor grid's dimensions.

then to reduce the number of processors involved in the communication [7, 18]. We now illustrate the composition procedure for an example that is presented in Figure 7 to demonstrate how the order of communication primitives effects the cost of communication. The array reference $B(N, I)$ requires communication along both dimensions of the processor grid P. The processor and data sets for each dimension are:

$$(2, B_1(N), 0, B_1(1))$$

for the send-receive communication in the first dimension, and

$$(0:2, B_2(p_2 * BS_{B_2} + 1 : (p_2 + 1) * BS_{B_2}), 0, B_2(1))$$

for the reduction communication in the second dimension, where p_n is a processor's identification number in the n^{th} dimension of the processor grid, and BS_{B_2} is the blocksize of the 2^{nd} dimension of B. The composition when communication in the second dimension occurs first is:

$$(\langle 2, 0:2\rangle, B(N, p_2 * BS_{B_2} + 1 : (p_2 + 1) * BS_{B_2}),$$

$$\langle 2, 0\rangle, buf(B(N, 1)))$$

for the reduction, and

$$(\langle 2, 0\rangle, buf(B(N, 1))\rangle, \langle 0, 0\rangle, B(1, 1))$$

for the send-receive where $buf(B(N, 1))$ denotes the buffer that contains the value generated by the reduction. The communication code generator allocates and manages communication buffers. Figure 8 illustrates the data movement along the dimensions of the processor grid P.

Choosing to execute the reduction before the send-receive communication in the program segment presented in Figure 7 results in unit message length for the send/receive communication and requires only one send-receive to occur. If, instead, the send-receive

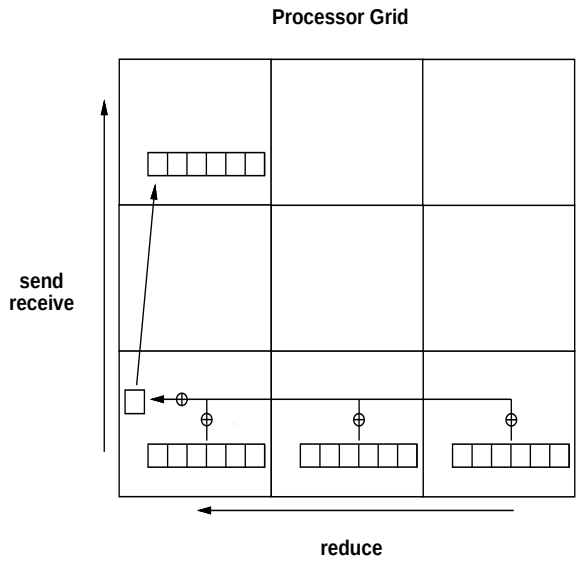

Figure 8: Communication composition

were to be executed before the reduction, the message length for the send-receive communication would increase to BS_{B_2}, and the number of (parallel) send-receives would increase to the extent of a row in the processor grid.

This approach of composing communication operations across a processor grid's dimensions is conceptually clean, and helps to simplify communication code generation. Furthermore, our table-driven implementation will allow this approach to be easily modified to optimize for other characteristics, such as reducing network traffic.

When the actual data items that need to be communicated or the processors that are involved in the communication are not known at compile-time due to missing information, such as an unknown distribution of an array reference or a complex subscript, this information must be computed at run-time. In this case, the communication code generator generates calls to a different set of communication library routines that perform the computations at run-time and then execute the appropriate communication [5]. Currently, these library routines do not provide inspector/executor functionality; however, in such cases, tests for the ownership of data are used to ensure the correct communication of data. In addition, PTRAN II relies on run-time support to handle block-cyclic distributions.

5 Computation partitioning

When compiling any language for a parallel machine, a problem that arises is what processors should perform what operations. Solving this problem for a particular program is the task of the computation partitioner.

In solving the problem, the computation partitioner is guided by the *owner computes* rule: the processor that owns the modified data item (e.g. the owner of the left hand side of an assignment statement) involved in an operation performs the operation. The input to the computation partitioner is an HPF program with data distribution annotations, the output is a Single Program Multiple Data (SPMD) program, where each node processor executes only those operations in the original HPF program that it should following the owner computes rule. In converting the HPF program to an SPMD program, a major goal is to allow innermost loops to run efficiently on a node processor.

A brute force method of performing computation partitioning is surrounding every statement in the program with a test to determine if that processor owns the left hand side reference, i.e. an *IOWN(α)* test. This strategy has the advantage of needing to know nothing about the distribution and alignment of the α, but every processor must execute the entire loop, and every iteration of the loop contains an additional branch. Intelligent computation partitioning can be thought of as an attempt to replace IOWN tests with less expensive techniques.

One way to do this is by *shrinking* the bounds of the loop[16] using the LIS components $i_{\texttt{low}}$ and $i_{\texttt{high}}$ (Section 4). Loop bound shrinking allows the IOWN test to be implicitly enforced by executing on each processor only those iterations of the loop whose statements would be executed with using the IOWN test.

5.1 Coupled Subscripts

Coupled subscripts are subscripts for different dimensions of an array that reference the same loop index variable. The effects of all subscripts in the reference must be taken into account when shrinking the loop. Consider for example:

```
PROCESSOR P(4,4)
DISTRIBUTE A(BLOCK,BLOCK)
DO I = 2, 100
   A(I+1,I-1) = ...
END DO
```

A processor $\texttt{P}(u, v)$ owns an element $\texttt{A}(r, c)$ if and only if owns the part of column c that contains row r. This implies that the values of i that access elements of A that $\texttt{P}(u, v)$ owns must map onto both a row and column owned by $\texttt{P}(u, v)$. Stated succinctly, the values of i that accesses these elements of A are the intersection of the sets of values that access a column and a row. Thus, the LIS of the I loop for processor $\texttt{P}(u, v)$ is

$$\mathcal{L} = \texttt{LIS}(\texttt{A}_1, \texttt{I}, \texttt{u}) \cap \texttt{LIS}(\texttt{A}_2, \texttt{I}, \texttt{v}) \quad (1)$$

5.2 Multiple Statements in the Body of the Loop

If multiple statements are present in a loop, then the loop must execute all iterations needed by every statement. This is the union of the iteration sets of each statement in the loop intersected with the original iteration set of the loop. Thus, if the statement:

```
B(3*I - 2, I) = ...
```

is added to the above loop, where B is aligned with A, the LIS for the I loop would be

$$\mathcal{L} \cup (\texttt{LIS}(\texttt{B}_1, \texttt{I}, \texttt{u}) \cap \texttt{LIS}(\texttt{B}_2, \texttt{I}, \texttt{v}))$$

using $\mathcal{L}$ from Equation 1.

5.3 References that have an identical effect on the loop bounds

Consider the loop:

```
PROCESSOR P(4)
DISTRIBUTE A(BLOCK)
ALIGN B(I) with A(I-1)
DO I = 2, 100
   A(I) = ...
   B(I+1) = ...
END DO
```

Naively applying the strategy above would require the iteration set for the I loop to be the union of the LIS's for both references. Because the subscripts for A and B are strictly synchronous (see Section 3.1.2), the LIS's for A and B are equal. Therefore the LIS for either A or B can be used to specify the iteration set for the I loop.

5.4 The effects of communication on computation partitioning

In general, computation partitioning only looks at the left hand sides of statements, since these contain the modified operand. However, it is necessary that statements to send data also be executed. For example, in the loop nest:

```
DISTRIBUTE A(BLOCK) ON P(1024)
DISTRIBUTE B(BLOCK) ON P(1024)
DO I = 1, N
   A(I) = A(I-1) + ...
END DO
```

elements of the A array must be communicated. Exactly what iterations are needed is determined by the communication code generation phase and is expressed as an LIS. This LIS is unioned with the LIS of the left hand side reference to A formed using the owner computes rule. The union of these two LIS's is the the iteration set of the I loop.

5.5 When guard statements are needed

As mentioned previously, loop bounds shrinking is an optimization to reduce the number of guard statements that must be used within a loop. If the bounds of a shrunk loop do not precisely specify the iteration set of a reference, then it will still be necessary to add guard statements. This occurs whenever:

- The LIS for the loop nest is not equal to the LIS for the reference;
- The LIS for the statement cannot be accurately computed because of a complicated subscript or distribution.

If the iteration set for the statement can be determined exactly, it is only necessary to determine if the current loop iteration is a member of the iteration set for the statement. If the bounds for the statement cannot be determined exactly (e.g. for non-linear subscripts) an IOWN test is necessary.

We attempt to move guards statements to the outermost loop where the guard is invariant. For example, consider the loop nest:

```
DISTRIBUTE A(BLOCK,BLOCK) ONTO P(1024,1024)
DO I = 1, N
   A(5,B(I)) = ...  :  s1
END DO
```

Statement s_1 should only be executed on processor $P(u,v)$ if the mapping functions (Section 1.1) $f(\mathtt{A}_1, 5) = u$ and $f(\mathtt{A}_2, \mathtt{B(I)}) = v$. The first test is invariant to the loop, and is hoisted out of the loop, whereas the second test, being dependent on the value of I, must remain in the loop. This results in the code:

```
IF (f(A1, 5) = u) THEN
   DO I = 1, N
      IF (f(A2, B(I)) = v) THEN
         A(5,B(I)) = ...  :  s1
      END IF
   END DO
END IF
```

5.6 Fusing loops and guard conditions

One condition (in the PTRAN II compiler) for fusing a pair of adjacent loops is that the bounds of the loops be identical. The bounds that result from computation partitioning can be very complicated even if the original loop bounds were constant expressions. It is therefore desirable that the computation partitioner identify the bounds of loops (formed by computation partitioning) that are identical. To do this requires the same analysis of the synchronous property used in Section 3.1.2 to determine if two statements have the same iteration sets. If the bounds of two or more loops are identical, the upper and lower bounds will be found and saved to compiler temporaries, and these compiler temporaries will then be used as the upper/lower bounds expression for each of the loops.

For example, the loop:

```
PROCESSOR P(4)
DISTRIBUTE A(BLOCK)
ALIGN B(I) WITH A(I-1)
DIMENSION A(100)
DO I = 2, 100
   A(I) = ...
   B(I+1) = ...
END DO
```

after loop distribution and computation partitioning will become:

```
lb = max(pid*25+1, 2)
ub = min((pid+1)*25, 100)
DO I = lb, ub
   A(I) = ...
END DO
DO I = lb, ub
   B(I+1) = ...
END DO
```

and the bounds will cease to be a concern as far as loop fusion is concerned.

Similarly, if adjacent statements have identical iteration sets, the guard conditions for the statements are identical, and the guard IF statements surrounding each of the statements can be replaced by a single guard IF surrounding all of the statements. This transformation will enable better uniprocessor optimizations to take place.

6 Loop Transformations

We believe that good HPF compilers need loop transformations to enhance performance. Some of the loop

transformations will be performed before automatic data partitioning and communication analysis, while others will be performed just before code generation, as explained below. In contrast to the SUIF[1] compiler, we have made the following design decisions:

- We assume that uniprocessor optimizations to improve cache locality are performed by the back-end, after the SPMD program is produced. Such uniprocessor optimizations may include additional loop blocking, interchange, and loop fusion. This division of functionality leads to a simpler, more modular design than if both uniprocessor and multiprocessor optimizations are considered together. However, it is important that the SPMDizer does not inhibit subsequent uniprocessor optimization. This requires some care in code generation, so that opportunities for such transformations as loop fusion are not inhibited.

- We do not use any loop skewing transformations in the SPMDizer. Skewing is useful for the following purposes:

  1. Wavefronting.
  2. To achieve outer-loop parallelism.

  However, for distributed-memory SPMD programs, wavefronting is achieved through data communication, so that loop skewing is not necessary. Furthermore, outer-loop parallelism achieved through loop skewing results in non-rectilinear data partitions, which are not currently in HPF. Therefore, we do not consider loop skewing to be particularly useful in our current system design.

Our SPMDizer will include loop distribution, interchange, and blocking to both increase parallelism and reduce communication costs.

6.1 Loop Distribution

Full loop distribution will be performed before automatic data partitioning. Loop distribution maximizes the number and nesting depth of perfect loop nests, thereby exposing more opportunities for optimization. For distributed memory compilation, loop distribution is especially useful for optimizations like message vectorization and eliminating statement guards. For example, consider the following program segment, where `A` and `B` are aligned with each other:

```
DO I = 1, N
   A(I) = ...
   B(I+1) = A(I) ...
END DO
```

```
DO J = 1, N
   DO I = 1, N
      A(I, J) = A(I-1, J) + D(I)
   END DO
END DO
```

Better Loop Ordering:

```
DO I = 1, N
   DO J = 1, N
      A(I, J) = A(I-1, J) + D(I)
   END DO
END DO
```

Figure 9: Example Program For Message Vectorization

Each assignment statement requires a separate guard, and there is a communication inserted between the two statements. Loop distribution results in the code:

```
DO I = 1,N
   A(I) = ...
END DO
<communication>
DO I = 1, N
   B(I+1) = A(I) ...
END DO
```

In the resulting program, each loop is completely parallel, with a single communication between the two loops. Further, the guard needed for each of the statements is easily eliminated by the computation partitioning module.

Even though the SPMDizer performs full loop distribution, no loop fusion is performed. We rely on subsequent uniprocessor optimizations to reassemble the program loop structure, according to locality considerations.

6.2 Loop Permutations and Stripmining

In addition to loop distribution, loop permutations can also be used to enable vectorization at the outermost legal loop levels. For example, in the code of Figure 9, if `A` is partitioned (BLOCK, *), different iterations of the `I` loop require communication, which cannot be moved out of the loop. However, if `I` and `J` are interchanged, the `J` loop is communication-free, and each communication operation can consist of an entire row of `A`. This reduces the number of messages.

Preliminary steps for permuting loops to vectorize messages include:

1. Identify loops which carry communication-inducing dependences.

2. Move parallelizable loops inwards to enable message vectorization.

When the distribution of arrays is not specified by user directives, we assume that each array dimension has been distributed for this step by the compiler.

In general, there may be more than a single legal loop configuration with inner parallelizable loops, and there should be an algorithm for selecting the best configuration.

To illustrate, consider the program:

```
DO J = 1, N
   DO I = 1,N
      A(I, J) = f(A(I-1, J-1))
   END DO
END DO
```

If `A` is partitioned `(BLOCK, *)`, there is a communication-inducing dependence from `A(I, J)` to `A(I-1, J-1)`, which has dependence vector $(1, 1)$ carried by the `J` loop. Communication can be moved outside the `I` loop, but there is no significant vectorization. Because of the way that `A` is partitioned, there will be only a single element of `A` communicated at each iteration of the `J` loop. If the `J` loop and `I` loop are interchanged, the `J` loop is parallelizable, and an entire row of `A` can be communicated at a time. The algorithm for determining the best loop ordering should therefore compare legal loop permutations, where the ability to perform message vectorization is the criterion for choosing the best.

Message vectorization is not unconditionally good, since it sometimes can prohibit parallelism, when a processor executing a later iteration must wait for an earlier iteration to finish. Therefore, after the above loop permutation transformation has been performed, loop blocking should be performed so that the computation can be pipelined. For this transformation, the inner parallelizable loops, which enable the message vectorization, are stripmined, and the resulting sectioning loops are permuted outside the loop carrying the communication-inducing dependence.

Consider again the interchanged form of the loop nest in Figure 9. Any processor that depends on receiving values of `A(I-1, J)` must wait until all iterations of the `J` loop complete, so that there is no parallelism. The parallel `J` loop is stripmined, and the resulting sectioning J' loop is interchanged with the `I` loop:

```
DO J' = 1, N, b
   DO I' = mylo_A, myhi_A, b'
      (Vectorized Communication Level)
      DO I = I', min(myhi_A, I' + b')
         DO J = J', min(N, J' + b)
         A(I, J) = A(I-1, J) + D(J)
      END DO
   END DO
END DO
```

Figure 10: The Canonical Loop Form

```
DO J' = 1, N, b
   DO I = mylo_A, myhi_A
   (Vectorized Communication Level)
   DO J = J', min(N, J' + b)
      A(I, J) = A(I-1, J) + D(I)
   END DO
END DO
END DO
```

The blocksize b must be chosen so that the cost of the computation for the inner loop is balanced with the (vectorized) communication overhead. Note that the the message length is proportional to b.

So far, we have not considered the interaction of both cache locality and message vectorization transformations on the loop nest of Figure 9. In this loop nest, the `I` loop carries locality, so that it is important for uniprocessor performance for the `I` loop to be in the innermost position. Therefore, for the canonical loop form (Figure 10), the `I` loop is also stripmined, so that subsequent uniprocess cache optimizations are not inhibited by the generated communication code.

7 Related work

There are several groups which have looked at the problem of compiling for distributed memory. A summary of this work can be found in [21].

The SUPERB Compiler [20] and Fortran D[13] share some of the goals of our project. All accept distribution and alignment information from the users, all target distributed memory machines, and all automatically generate communication.

SUPERB is the earliest of these and differs from PTRAN II in that only block distributions are supported. Message vectorization, and overlay areas are supported, as in PTRAN II. Elimination of identical redundant communication is also supported. The PTRAN II project is investigating more general optimizations for the elimination of redundant communication. The second generation SUPERB compiler will use expert system techniques to do automatic data partitioning.

Fortran D allows cyclic and block cyclic distributions. Currently, partitioning of only a single dimension of an array is supported, although plans call for supporting the partitioning of more than one dimension. As with SUPERB, message vectorization, overlay areas and elimination of some redundant communication is supported. Fortran D converts addressing into local space very early in the compilation process, and all analysis is performed using local space. PTRAN II stays in global space as long as possible – doing so allows symbolic information to be maintained that is true across the processor grid, information that can be resolved at run-time. PTRAN II allows arrays to be partitioned on all dimensions.

Our use of the iteration sets for generating communication and partitioning the computation draws on the work of Koebel on the Kali project[16].

Li and Chen have introduced the *component affinity graph* framework [17] for aligning array dimensions, which we use with improved cost estimation. They have also described techniques for automatically generating communication. We have extended that work by incorporating dependence information, and by introducing the notion of synchronous properties between sub-references, that allows our compiler to perform a more accurate communication analysis.

Knobe, Lukas and Steele's [14, 15] *preferences* are similar to the constraints of Section 2, but our use of sophisticated cost measures should allow these constraints to be used more accurately in determining a data partitioning.

Chatterjee et al. [3] describe a framework for automatic alignment of arrays in data-parallel languages. They do not provide experimental results, and do not yet deal with the problem of determining the method of partitioning and the number of processors in different mesh dimensions.

The SUIF compiler, [1] performs loop transformations both for increasing parallelism and reducing communication costs, and to enhance uniprocesor performance. PTRAN II differs from SUIF by performing loop transformations related to parallel program performance and uniprocessor performance in different phases. SUIF also performs loop skewing optimizations, which are not included in the current PTRAN II design.

8 Conclusion

The design of a SPMDizer for HPF has been described. The current implementation status is that programs with block distributions are being successfully compiled and executed. The compilation does not depend on the number of processors. However, if the number of processors is known at compile-time, better code will be generated. With this base system in place, we plan to incorporate the automatic data partitioning and communication optimizations that have been presented.

References

[1] S. Amarasinghe, J. Anderson, M. Lam, and A. Lim. An overview of a compiler for scalable parallel machines. In *Proc. Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers. In *Proc. Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[3] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proc. Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.

[4] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control depoendence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] K. Ekanadham and Y.A. Baransky. *PDM: Partitioned Data Manager*. IBM T.J. Watson Research Center, 1993.

[6] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0.

Technical Report CRPC-TR92225, Rice University, January 1993.

[7] M. Gupta. *Automatic data partitioning on distributed memory multicomputers.* PhD thesis, University of Illinois, September 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] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication on multicomputers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[10] M. Gupta and P. Banerjee. Paradigm: A compiler for automatic data distribution on multicomputers. In *Proc. 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[11] Manish Gupta and Edith Schonberg. A framework for exploiting data availability to optimize communication. In *Proc. 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[12] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[13] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[14] K. Knobe, J. Lukas, and G. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[15] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Proc. Third Symposium on the Frontiers of Massively Parallel Computation*, October 1990.

[16] C. Koelbel. *Compiling programs for nonshared memory machines.* PhD thesis, Purdue University, August 1990.

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

[18] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[19] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

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

[21] H. Zima and B. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993.