An Overview of the Fortran D Programming System

Seema Hiranandani
Ken Kennedy
Charles Koelbel
Ulrich Kremer
Chau-Wen Tseng

CRPC-TR91121
March, 1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Revised August, 1991. In the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA, August 1991.

An Overview of the Fortran D Programming System *

Seema Hiranandani
Ken Kennedy
Charles Koelbel
Ulrich Kremer
Chau-Wen Tseng

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

The success of large-scale parallel architectures is limited by the difficulty of developing machine-independent parallel programs. We have developed Fortran D, a version of Fortran extended with data decomposition specifications, to provide a portable data-parallel programming model. This paper presents the design of two key components of the Fortran D programming system: a prototype compiler and an environment to assist automatic data decomposition. The Fortran D compiler addresses program partitioning, communication generation and optimization, data decomposition analysis, run-time support for unstructured computations, and storage management. The Fortran D programming environment provides a static performance estimator and an automatic data partitioner. We believe that the Fortran D programming system will significantly ease the task of writing machine-independent data-parallel programs.

1 Introduction

It is widely recognized that parallel computing represents the only plausible way to continue to increase the computational power available to computational scientists and engineers. However, it is not likely to be widely successful until parallel computers are as easy to use as today's vector supercomputers. A major component of the success of vector supercomputers is the ability to write machine-independent vectorizable programs. Automatic vectorization and other compiler technologies have made it possible for the scientist to structure Fortran loops according the well-understood rules of "vectorizable style" and expect the resulting program to be compiled to efficient code on any vector machine [6, 32].

Compare this with the current situation for parallel machines. Scientists wishing to use such a machine must rewrite their programs in an extension of Fortran that explicitly reflects the architecture of the underlying machine, such as a message-passing dialect for MIMD distributed-memory machines, vector syntax for SIMD machines, or an explicitly parallel dialect with synchronization for MIMD shared-memory machines. This conversion is difficult, and the resulting parallel programs are machine-specific. Scientists are thus discouraged from porting programs to parallel machines because they risk losing their investment whenever the program changes or a new architecture arrives.

One way to overcome this problem would be to identify a "data-parallel programming style" that allows the efficient compilation of Fortran programs on a variety of parallel machines. Researchers working in the area, including ourselves, have concluded that such a programming style is useful but not sufficient in general. The reason for this is that not enough information can be included in the program text for the compiler to accurately evaluate alternative translations. Similar reasoning argues against cross-compilations between the

*This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center.

current parallel extensions of Fortran.

For these reasons, we have chosen a different approach. We believe that selecting a data decomposition is one of the most important intellectual step in developing data-parallel scientific codes. However, current parallel programming languages provide little support for data decomposition [26]. We have therefore developed an enhanced version of Fortran that introduces data decomposition specifications. We call the extended language Fortran D, where "D" suggests data, decomposition, or distribution. When reasonable data decompositions are provided for a Fortran D program written in a data-parallel programming style, we believe that advanced compiler technology can implement it efficiently on a variety of parallel architectures.

We are developing a prototype Fortran D compiler to generate node programs for the iPSC/860, a MIMD distributed-memory machine. If successful, the result of this project will go far towards establishing the feasibility of machine-independent parallel programming, since a MIMD shared-memory compiler could be based directly on the MIMD distributed-memory implementation. The only additional step would be the construction of an effective Fortran D compiler for SIMD distributed-memory machines. We have initiated at Rice a project to build such a compiler based on existing vectorization technology.

The Fortran D compiler automates the time consuming task of deriving node programs based on the data decomposition. The remaining components of the Fortran D programming system, the static performance estimator and automatic data partitioner, support another important step in developing a data-parallel program—selecting a data decomposition. The rest of this paper presents the data decomposition specifications in Fortran D, the structure of a prototype Fortran D compiler, and the design of the Fortran D programming environment. We conclude with a discussion of our validation strategy.

2 Fortran D

The data decomposition problem can be approached by considering the two levels of parallelism in data-parallel applications. First, there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. We call this the *problem mapping* induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program, and is largely independent of any machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays.

Second, there is the question of how arrays should be *distributed* onto the actual parallel machine. We call this the *machine mapping* caused by translating the problem onto the finite resources of the machine. It is affected by the topology, communication mechanisms, size of local memory, and number of processors in the underlying machine. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

Fortran D is a version of Fortran that provides data decomposition specifications for these two levels of parallelism using DECOMPOSITION, ALIGN, and DISTRIBUTE statements. A decomposition is an abstract problem or index domain; it does not require any storage. Each element of a decomposition represents a unit of computation. The DECOMPOSITION statement declares the name, dimensionality, and size of a decomposition for later use.

The ALIGN statement is used to map arrays onto decompositions. Arrays mapped to the same decomposition are automatically aligned with each other. Alignment can take place either within or across dimensions. The alignment of arrays to decompositions is specified by placeholders in the subscript expressions of both the array and decomposition. In the example below,

```
REAL X(N,N)
DECOMPOSITION A(N,N)
ALIGN X(I,J) with A(J-2,I+3)
```

A is declared to be a two dimensional decomposition of size $N \times N$. Array X is then aligned with respect to A with the dimensions permuted and offsets within each dimension.

After arrays have been aligned with a decomposition, the DISTRIBUTE statement maps the decomposition to the finite resources of the physical machine. Distributions are specified by assigning an independent *attribute* to each dimension of a decomposition. Predefined attributes are BLOCK,

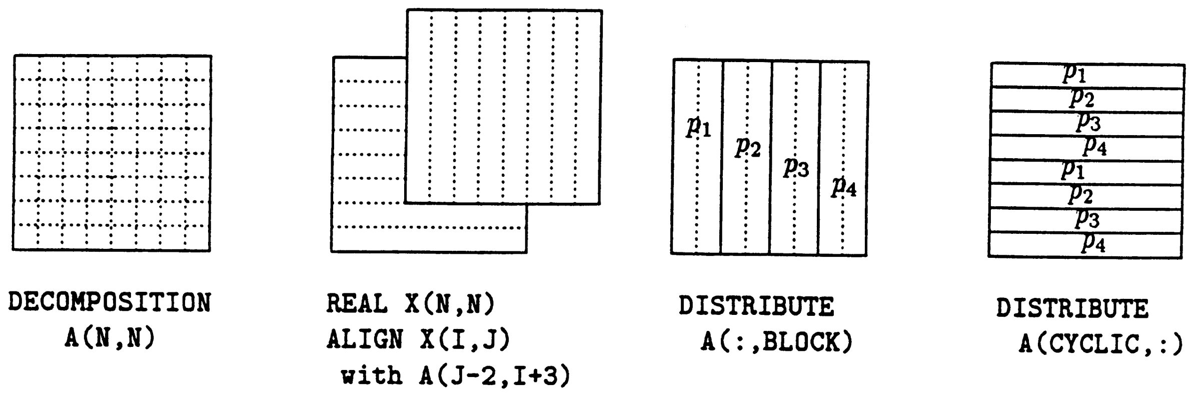


Figure 1: Fortran D Data Decomposition Specifications

CYCLIC, and BLOCK_CYCLIC. The symbol ":" marks dimensions that are not distributed. Choosing the distribution for a decomposition maps all arrays aligned with the decomposition to the machine. In the following example,

```
        DECOMPOSITION A(N,N)
        DISTRIBUTE A(:, BLOCK)
        DISTRIBUTE A(CYCLIC,:)
```

distributing decomposition A by (:,BLOCK) results in a column partition of arrays aligned with A. Distributing A by (CYCLIC,:) partitions the rows of A in a round-robin fashion among processors. These sample data alignment and distributions are shown in Figure 1.

Predefined regular data distributions can effectively exploit regular data-parallelism. However, irregular distributions and run-time processing is required to manage the irregular data parallelism found in many unstructured computations. In Fortran D, irregular distributions may be specified through an explicit user-defined function or data array. In the example below,

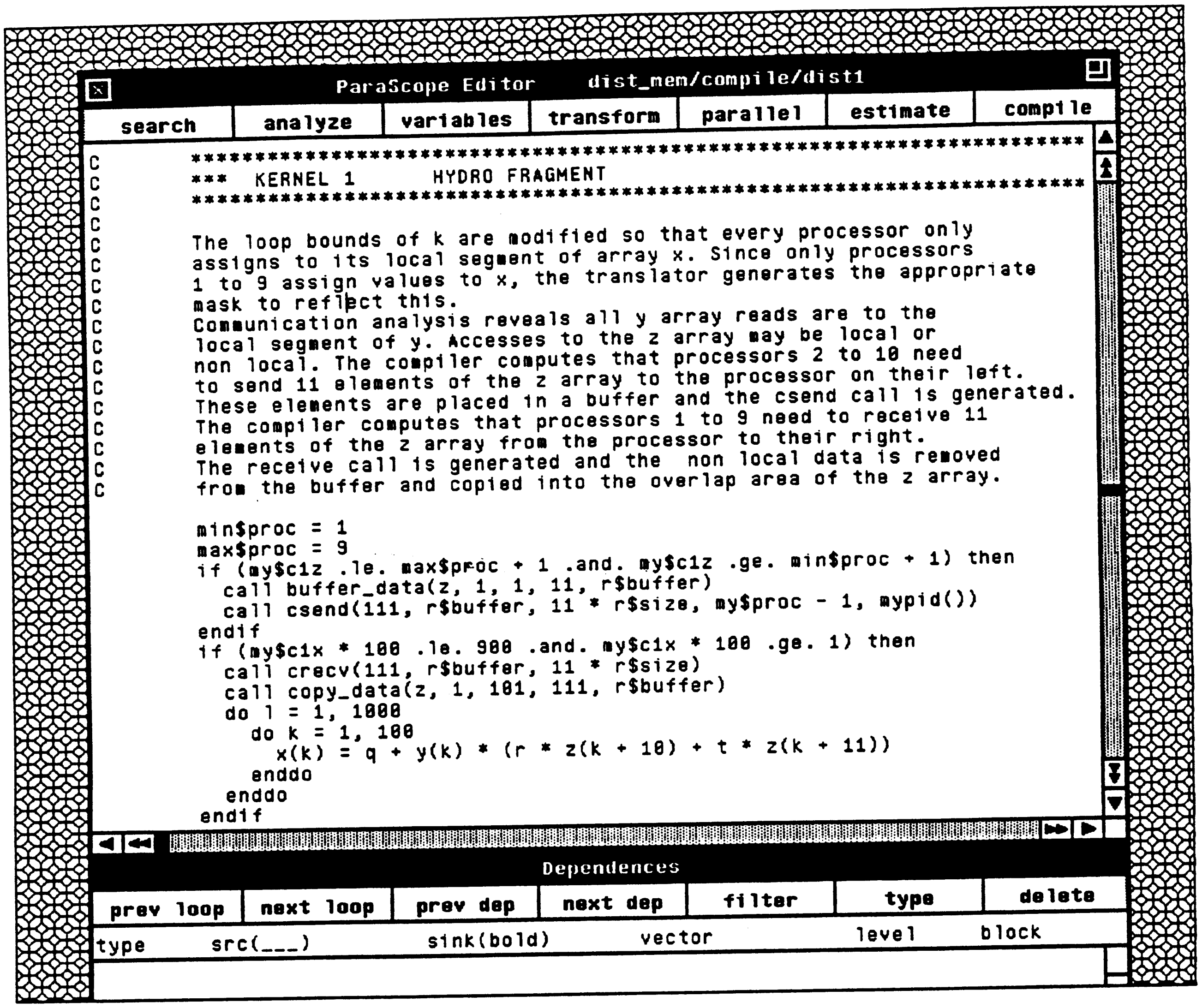
```
        INTEGER MAP(N)
        DECOMPOSITION IRREG(N)
        DISTRIBUTE IRREG(MAP)
```

elements of the decomposition IRREG(i) will be mapped to the processor indicated by the array MAP(i). Fortran D also supports dynamic data decomposition; *i.e.*, changing the alignment or distribution of a decomposition at any point in the program.

We should note that our goal in designing Fortran D is not to support the most general data decompositions possible: Instead, our intent is to provide decompositions that are both powerful enough to express data parallelism in scientific programs, and simple enough to permit the compiler to produce efficient programs. Fortran D is a language with semantics very similar to sequential Fortran. As a result, it should be quite usable by computational scientists. In addition, we believe that our two-phase strategy for specifying data decomposition is natural and conducive to writing modular and portable code. Fortran D bears similarities to both CM Fortran [31] and KALI [22]. The complete language is described in detail elsewhere [8].

3 Fortran D Compiler

As we have stated previously, two major steps in writing a data-parallel program are selecting a data decomposition, and then using it to derive node programs with explicit communications to access nonlocal data. Manually inserting communications is unquestionably the most time-consuming, tedious, non-portable, and error-prone step in parallel programming. Significant increases in source code size are not only common but expected. A major advantage of programming in Fortran D will be the ability to utilize advanced compiler techniques to automatically generate node programs with explicit communication, based on the data decompositions specified in the program. The prototype compiler is being developed in the context of the ParaScope parallel programming environment [4], and will take advantage of the analysis and



```
C       ************************************************************************
C       ***  KERNEL 1       HYDRO FRAGMENT
C       ************************************************************************
C
C       The loop bounds of k are modified so that every processor only
C       assigns to its local segment of array x. Since only processors
C       1 to 9 assign values to x, the translator generates the appropriate
C       mask to reflect this.
C       Communication analysis reveals all y array reads are to the
C       local segment of y. Accesses to the z array may be local or
C       non local. The compiler computes that processors 2 to 10 need
C       to send 11 elements of the z array to the processor on their left.
C       These elements are placed in a buffer and the csend call is generated.
C       The compiler computes that processors 1 to 9 need to receive 11
C       elements of the z array from the processor to their right.
C       The receive call is generated and the  non local data is removed
C       from the buffer and copied into the overlap area of the z array.

        min$proc = 1
        max$proc = 9
        if (my$c1z .le. max$proc + 1 .and. my$c1z .ge. min$proc + 1) then
          call buffer_data(z, 1, 1, 11, r$buffer)
          call csend(111, r$buffer, 11 * r$size, my$proc - 1, mypid())
        endif
        if (my$c1x * 100 .le. 900 .and. my$c1x * 100 .ge. 1) then
          call crecv(111, r$buffer, 11 * r$size)
          call copy_data(z, 1, 101, 111, r$buffer)
          do l = 1, 1000
            do k = 1, 100
              x(k) = q + y(k) * (r * z(k + 10) + t * z(k + 11))
            enddo
          enddo
        endif
```

```
type     src(___)          sink(bold)          vector          level     block
```

Figure 2: Fortran D Compiler Output

transformation capabilities of the ParaScope Editor [19, 20].

The main goal of the Fortran D compiler is to derive from the data decomposition a parallel node program that minimizes load imbalance and communication costs. Our approach is to convert Fortran D programs into *single-program, multiple-data* (SPMD) form with explicit message-passing that executes directly on the nodes of the distributed-memory machine. Our basic strategy is to partition the program using the *owner computes* rule, where every processor only performs computation on data it owns [5, 29, 34]. However, we will relax the rule where it prevents the compiler from achieving good load balance or reducing communication costs.

The Fortran D compiler bears similarities to ARF [33], ASPAR [18], ID NOUVEAU [29], KALI [22], MIMDIZER [13], and SUPERB [34]. The current prototype generates code for a subset of the decompositions allowed in Fortran D, namely those with BLOCK distributions. Figure 2 depicts the output of a Livermore loop kernel generated by the Fortran D compiler.

3.1 Program Partitioning

The first phase of the compiler partitions the program onto processors based on the data decomposition. We define the *iteration set* of a reference R on the local processor t_p to be the set of loop iterations that cause R to access data owned by t_p. The iteration set is calculated based on the alignment and distribution specified in the Fortran D program. According to the *owner computes rule*, the set of loop iterations that t_p must execute is the union of the iteration sets for the left-hand sides (*lhs*) of all the individual assignment statements within the loop.

To partition the computation among processors, we first reduce the loop bounds so that each processor only executes iterations in its own set. With multiple statements in the loop, the iteration set of an individual statement may be a subset of the iteration set for that loop. For these statements we also add guards based on membership tests for the iteration set of the *lhs* to ensure that all assignments are to local array elements.

3.2 Communication Introduction

Once the computation has been partitioned, the Fortran D compiler must introduce communications for nonlocal data accesses to preserve the semantics of the original program. This requires calculating the data that must be sent or received by each processor. We can calculate the *send iteration set* for each right-hand side (*rhs*) reference as its iteration set minus the iteration set of its *lhs*. Similarly, the *receive iteration set* for each *rhs* is the iteration set of its *lhs* minus its own iteration set. These sets represent the iterations for which data must be sent or received by t_p. The Fortran D compiler summarizes the array locations accessed on the send or receive iterations using rectangular or triangular regions known as *regular sections* [12]; they are used to generate calls to communication primitives.

3.3 Communication Optimization

A naive approach for introducing communication is to insert send and receive operations directly preceding each reference causing a nonlocal data access. This generates many small messages that may prove inefficient due to communication overhead. The Fortran D compiler will use *data dependence* information to determine whether communication may be inserted at some outer loop, *vectorizing* messages by combining many small messages. The algorithm to calculate the appropriate loop level for each message is described by Balasundaram *et al.* and Gerndt [2, 10].

A major goal of the Fortran D compiler is to aggressively optimize communications. We intend to apply techniques proposed by Li and Chen to recognize regular computation patterns that can utilize collective communications primitives [24]. It will be especially important to recognize reduction operations. For regular communication patterns, we plan to employ the collective communications routines found in EXPRESS [27]. For unstructured computations with irregular communications, we will incorporate the PARTI primitives of Saltz *et al.* [33].

The Fortran D compiler may utilize data decomposition and dependence information to guide program transformations that improve communication patterns. We are considering the usefulness of several transformations, particularly loop interchanging, strip mining, loop distribution, and loop alignment. Replicating computations and processor-specific dead code elimination will also be applied to eliminate communication.

Communications may be further optimized by considering interactions between all the loop nests in the program. Intra- and interprocedural dataflow analysis of array sections can show that an assignment to a variable is *live* at a point in the program if there are no intervening assignments to that variable. This information may be used to eliminate redundant messages. For instance, assume that messages in previous loop nests have already retrieved nonlocal elements for a given array. If those values are *live*, messages to fetch those values in succeeding loop nests may be eliminated. Data from different arrays being sent to the same processor may also be buffered together in one message to reduce communication overhead.

The *owner computes* rule provides the basic strategy of the Fortran D compiler. We may also relax this rule, allowing processors to compute values for data they do not own. For instance, suppose that multiple *rhs* of an assignment statement are owned by a processor that is not the owner of the *lhs*. Computing the result on the processor owning the *rhs* and then sending the result to the owner of the *lhs* could reduce the amount of data communicated. This optimization is a simple case of the *owner stores* rule proposed by Balasundaram [1].

In particular, it may be desirable for the Fortran D compiler to partition loops amongst processors so that each loop iteration is executed on a single processor, such as in KALI [22] and PARTI [33]. This technique may improve communication and provide greater control over load balance, especially for irregular computations. It also eliminates the need for individual statement guards and simplifies handling of control flow within the loop body.

3.4 Data Decomposition Analysis

Fortran D provides dynamic data decomposition by permitting `ALIGN` and `DISTRIBUTE` statements to be inserted at any point in a program. This complicates the job of the Fortran D compiler, since it must know the decomposition of each array in order to generate the proper guards and communication. We define *reaching decompositions* to be the set of decomposition specifications that may reach an array reference aligned with the decomposition; it may be calculated in a manner similar to *reaching definitions*. The Fortran D compiler will apply both intra- and interprocedural analysis to calculate reaching decompositions for each reference to a distributed array. If multiple decompositions reach a procedure, node splitting or run-time techniques may be required to generate the proper code for the program.

To permit a modular programming style, the effects of data decomposition specifications are limited to the scope of the enclosing procedure. However, procedures do inherit the decompositions of their callers. These semantics require the compiler to insert calls to run-time data decomposition routines to restore the original data decomposition upon every procedure return. Since changing the data decomposition may be expensive, these calls should be eliminated where possible.

We define *live decompositions* to be the set of decomposition specifications that may reach some array reference aligned with the decomposition; it may be calculated in a manner similar to *live variables*. As with reaching decompositions, the Fortran D compiler needs both intra- and interprocedural analysis to calculate live decompositions for each decomposition specification. Any data decompositions determined not to be *live* may be safely eliminated. Similar analysis may also hoist dynamic data decompositions out of loops.

3.5 Run-time Support for Irregular Computations

Many advanced algorithms for scientific applications are not amenable to the techniques described in the previous section. Adaptive meshes, for example, often have poor load balance or high communication cost if static regular data distributions are used. These algorithms require dynamic irregular data distributions. Other algorithms, such as fast multipole algorithms, make heavy use of index arrays that the compiler cannot analyze. In these cases, the communications analysis must be performed at run-time.

The Fortran D project supports dynamic irregular distributions. The *inspector/executor* strategy to generate efficient communications has been adapted from KALI [22] and PARTI [25]. The inspector is a transformation of the original Fortran D loop that builds a list of nonlocal elements, known as the IN set, that will be received during the execution of the loop. A global transpose operation is performed using collective communications to calculate the set of data elements that must be sent by a processor, known as the OUT set. The executor uses the computed sets to control the actual communication. Performance results using the PARTI primitives indicate that the inspector can be implemented with acceptable overhead, particularly if the results are saved for future executions of the original loop [33].

3.6 Storage Management

Once guards and communication have been calculated, the Fortran D compiler must select and manage storage for all nonlocal array references received from other processors. There are several different storage schemes, described below:

- *Overlaps*, developed by Gerndt, are expansions of local array sections to accommodate neighboring nonlocal elements [10]. They are useful for programs with high locality of reference, but may waste storage when nonlocal accesses are distant.

- *Buffers* are designed to overcome the contiguous nature of overlaps. They are useful when the nonlocal area is bounded in size, but not near the local array section.

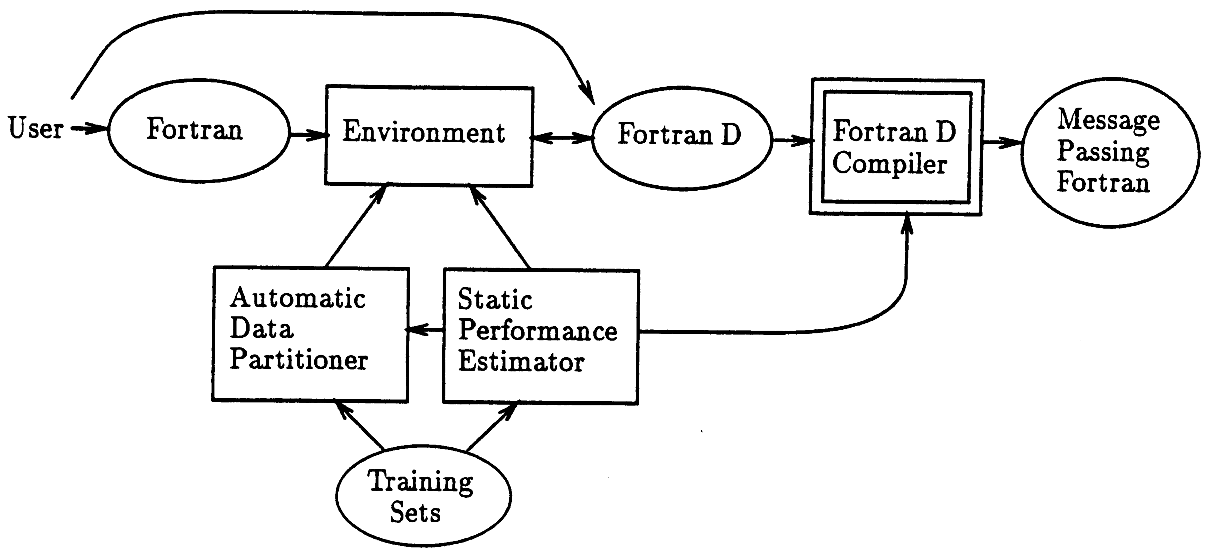


Figure 3: Fortran D Parallel Programming System

- *Hash tables* are used when the set of accessed nonlocal elements is sparse. This is the case in many irregular computations. Hash tables provide a quick lookup mechanism for arbitrary sets of nonlocal values [16].

Once the storage type for all nonlocal data is determined, the compiler needs to analyze the space required by the various storage structures and generate code so that nonlocal data is accessed from its correct location. Storage management and other parts of the Fortran D compiler are described in more detail elsewhere [14, 15].

4 Fortran D Programming Environment

Choosing a decomposition for the fundamental data structures used in the program is a pivotal step in developing data-parallel applications. Once selected, the data decomposition usually completely determines the parallelism and data movement in the resulting program. Unfortunately, there are no existing tools to advise the programmer in making this important decision. To evaluate a decomposition, the programmer must first insert the decomposition in the program text, then compile and run the resulting program to determine its effectiveness. Comparing two data decompositions thus requires implementing and running both versions of the program, a tedious task at best. The process is prohibitively difficult without the assistance of a compiler to automatically generate node programs based on the data decomposition.

Several researchers have proposed techniques to automatically derive data decompositions based on simple machine models [17, 28, 30]. However, these techniques are insufficient because the efficiency of a given data decomposition is highly dependent on both the actual node program generated by the compiler and its performance on the parallel machine. "Optimal" data decompositions may prove inferior because the compiler generates node programs with suboptimal communications or poor load balance. Similarly, marginal data decompositions may perform well because the compiler is able to utilize collective communication primitives to exploit special hardware on the parallel machine.

What we need is a programming environment that helps the user to understand the effect of a given data decomposition and program structure on the efficiency of the *compiler-generated* code running on a given target machine. The Fortran D programming system, shown in Figure 3, provides such an environment. The main components of the environment are a static performance estimator and an automatic data partitioner [2, 3].

Since the Fortran D programming system is built on top of ParaScope, it also provides program analysis, transformation, and editing capabilities that allow users to restructure their programs according to a data-parallel programming style. Zima and others at Vienna are working on a similar tool to support data decomposition decisions using automatic techniques [7]. Gupta and Banerjee propose automatic data decomposition techniques based on assumptions about a proposed Parafrase-2 distributed-memory compiler [11].

4.1 Static Performance Estimator

It is clearly impractical to use dynamic performance information to choose between data decompositions in our programming environment. Instead, a *static* performance estimator is needed that can accurately predict the performance of a Fortran D program on the target machine. Also required is a scheme that allows the compiler to assess the costs of communication routines and computations. The static performance estimator in the Fortran D programming system caters to both needs.

The performance estimator is not based on a general theoretical model of distributed-memory computers. Instead, it employs the notion of a *training set* of kernel routines that measures the cost of various computation and communication patterns on the target machine. The results of executing the training set on a parallel machine are summarized and used to train the performance estimator for that machine. By utilizing training sets, the performance estimator achieves both accuracy and portability across different machine architectures. The resulting information may also be used by the Fortran D compiler to guide communication optimizations.

The static performance estimator is divided into two parts, a machine module and a compiler module. The *machine module* predicts the performance of a node program containing explicit communications. It uses a *machine-level* training set written in message-passing Fortran. The training set contains individual computation and communication patterns that are timed on the target machine for different numbers of processors and data sizes. To estimate the performance of a node program, the machine module can simply look up results for each computation and communication pattern encountered.

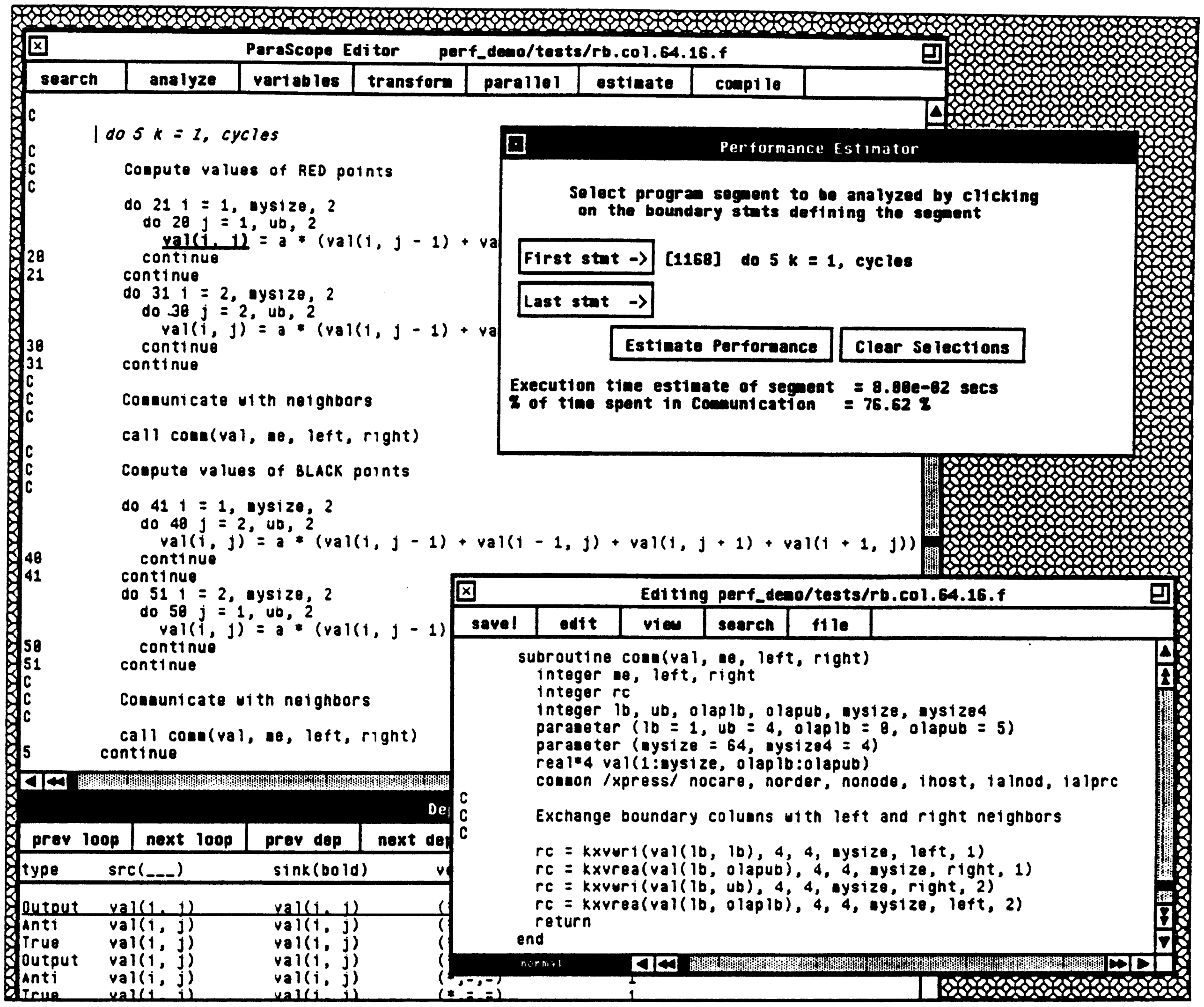
The *compiler module* forms the second part of the static performance estimator. It assists the user in selecting data decompositions by statically predicting the performance of a program for a set of data decompositions. The compiler module employs a *compiler-level* training set written in Fortran D that consists of program kernels such as stencil computations and matrix multiplication. The training set is converted into message-passing Fortran using the Fortran D compiler and executed on the target machine for different data decompositions, numbers of processors, and array sizes. Estimating the performance of a Fortran D program then requires matching computations in the program with kernels from the training set.

The compiler-level training set also provides a natural way to respond to changes in the Fortran D compiler as well as the machine. We simply recompile the training set with the new compiler and execute the resulting programs to reinitialize the compiler module for the performance estimator.

Since it is not possible to incorporate all possible computation patterns in the compiler-level training set, the performance estimator will encounter code fragments that cannot be matched with existing kernels. To estimate the performance of these codes, the compiler module must rely on the machine-level training set. We plan to incorporate elements of the Fortran D compiler in the performance estimator so that it can mimic the compilation process. The compiler module can thus convert any unrecognized Fortran D program fragment into an equivalent node program, and invoke the machine module to estimate its performance.

Note that even though it is desirable, to assist automatic data decomposition the static performance estimator does not need to predict the *absolute* performance of a given data decomposition. Instead, the it only needs to accurately predict the performance *relative* to other data decompositions. A prototype of the machine module has been implemented for a common class of *loosely synchronous* scientific problems[9]. It predicts the performance of a node program using EXPRESS communication routines for different numbers of processors and data sizes [27]. The prototype performance estimator has proved quite precise, especially in predicting the relative performances of different data decompositions [3].

A screen snapshot during a typical performance



```
C
      do 5 k = 1, cycles
C
C        Compute values of RED points
C
         do 21 i = 1, mysize, 2
           do 20 j = 1, ub, 2
             val(i, j) = a * (val(i, j - 1) + va
20         continue
21       continue
         do 31 i = 2, mysize, 2
           do 30 j = 2, ub, 2
             val(i, j) = a * (val(i, j - 1) + va
30         continue
31       continue
C
C        Communicate with neighbors
C
         call comm(val, me, left, right)
C
C        Compute values of BLACK points
C
         do 41 i = 1, mysize, 2
           do 40 j = 2, ub, 2
             val(i, j) = a * (val(i, j - 1) + val(i - 1, j) + val(i, j + 1) + val(i + 1, j))
40         continue
41       continue
         do 51 i = 2, mysize, 2
           do 50 j = 1, ub, 2
             val(i, j) = a * (val(i, j - 1)
50         continue
51       continue
C
C        Communicate with neighbors
C
         call comm(val, me, left, right)
5     continue
```

```
Select program segment to be analyzed by clicking
 on the boundary stmts defining the segment

First stmt -> [1168]  do 5 k = 1, cycles
Last stmt  ->

Execution time estimate of segment  = 8.00e-02 secs
% of time spent in Communication    = 76.62 %
```

```
      subroutine comm(val, me, left, right)
        integer me, left, right
        integer rc
        integer lb, ub, olaplb, olapub, mysize, mysize4
        parameter (lb = 1, ub = 4, olaplb = 0, olapub = 5)
        parameter (mysize = 64, mysize4 = 4)
        real*4 val(1:mysize, olaplb:olapub)
        common /xpress/ nocare, norder, nonode, ihost, ialnod, ialprc
C
C       Exchange boundary columns with left and right neighbors
C
        rc = kxvwri(val(lb, lb), 4, 4, mysize, left, 1)
        rc = kxvrea(val(lb, olapub), 4, 4, mysize, right, 1)
        rc = kxvwri(val(lb, ub), 4, 4, mysize, right, 2)
        rc = kxvrea(val(lb, olaplb), 4, 4, mysize, left, 2)
        return
      end
```

```
type      src(___)       sink(bold)
Output    val(i, j)      val(i, j)
Anti      val(i, j)      val(i, j)
True      val(i, j)      val(i, j)
Output    val(i, j)      val(i, j)
Anti      val(i, j)      val(i, j)
True      val(i, j)      val(i, j)
```

Figure 4: Static Performance Estimator

estimation session is shown in Figure 4. The user can select a program segment such as a do loop and invoke the performance estimator by clicking on the $\boxed{\texttt{Estimate Performance}}$ button. The prototype responds with an execution time estimate of the selected segment on the target machine, as well as an estimate of the communication time represented as a percentage of the total execution time. This allows the effectiveness of a data partitioning strategy to be evaluated on any part of the node program.

4.2 Automatic Data Partitioner

The goal of the automatic data partitioner is to assist the user in choosing a good data decomposition. It utilizes training sets and the static performance estimator to select data partitions that are efficient for both the compiler and parallel machine.

The automatic data partitioner may be applied to an entire program or on specific program fragments. When invoked on an entire program, it automatically selects data decompositions without further user interaction. We believe that for regular loosely synchronous problems written in a data-parallel programming style, the automatic data partitioner can determine an efficient partitioning scheme without user interaction.

Alternatively, the automatic data partitioner may be used as a starting point for choosing a good data decomposition. When invoked interactively for specific program segments, it responds with a list of the best decomposition schemes, together with their static performance estimates. If the user is not satisfied with the predicted overall performance, he or she can use the performance estimator to locate communication and computation intensive program segments. The Fortran D environment can then advise the user about the effects of program changes on the choice of a good data decomposition.

The analysis performed by the automatic data partitioner divides the program into separate *computation phases*. The *intra-phase* decomposition problem consists of determining a set of good data decompositions and their performance for each individual phase. The data partitioner first tries to match the phase or parts of the phase with computation patterns in the compiler training set. If a match is found, it returns the set of decompositions with the best measured performance as recorded in the compiler training set. If no match is found, the data partitioner must perform alignment and distribution analysis on the phase. The resulting solution may be less accurate since the effects of the Fortran D compiler and target machine can only be estimated.

Alignment analysis is used to prune the search space of possible arrays alignments by selecting only those alignments that minimize data movement. Alignment analysis is largely machine-independent; it is performed by analyzing the array access patterns of computations in the phase. We intend to build on the inter-dimensional and intra-dimensional alignment techniques of Li and Chen [23] and Knobe *et al.* [21].

Distribution analysis follows alignment analysis. It applies heuristics to prune unprofitable choices in the search space of possible distributions. The efficiency of a data distribution is determined by machine-dependent aspects such as topology, number of processors, and communication costs. The automatic data partitioner uses the final set of alignments and distributions to generate a set of reasonable data decomposition schemes. In the worst case, the set of decompositions is the cross product of the alignment and distribution sets. Finally, the static performance estimator is invoked to select the set of data decompositions with the best predicted performance.

After computing data decompositions for each phase, the automatic data partitioner must solve the *inter-phase* decomposition problem of merging individual data decompositions. It also determines the profitability of realigning or redistributing arrays between computational phases. Interprocedural analysis will be used to merge the decomposition schemes of computation phases across procedure boundaries. The resulting decompositions for the entire program and their performance are then presented to the user.

5 Validation Strategy

We plan to establish whether our compilation and automatic data partitioning schemes for Fortran D can achieve acceptable performance on a variety of parallel architectures. We will use a benchmark suite being developed by Geoffrey Fox at Syracuse that consists of a collection of Fortran programs. Each program in the suite will have five versions:

(v1) the original Fortran 77 program,

(v2) the best hand-coded message-passing version of the Fortran program,

(v3) a "nearby" Fortran 77 program,

(v4) a Fortran D version of the nearby program, and

(v5) a Fortran 90 version of the program.

The "nearby" version of the program will utilize the same basic algorithm as the message-passing program, except that all explicit message-passing and blocking of loops in the program are removed. The Fortran D version of the program consists of the nearby version plus appropriate data decomposition specifications.

To validate the Fortran D compiler, we will compare the running time of the best hand-coded message-passing version of the program (v2) with the output of the Fortran D compiler for the Fortran D version of the nearby program (v4). To validate the automatic data partitioner, we will use it to generate a Fortran D program from the nearby Fortran program (v3). The result will be compiled by the Fortran D compiler and its running time compared with that of the compiled version of the hand-generated Fortran D program (v4).

The purpose of the validation program suite is to provide a fair test of the prototype compiler and data partitioner. We do not expect these tools to perform high-level algorithm changes. However, we will test their ability to analyze and optimize whole programs based on both machine-independent issues such as the structure of the computation, as well as machine-dependent issues such as the number and interconnection of processors in the parallel machine. Our validation strategy will test three key parts of the Fortran D programming system: the limits of our machine-independent Fortran D programming model, the efficiency and ability of our compiler technology, and the effectiveness of our automatic data partitioning and performance estimation techniques.

6 Conclusions

Scientific programmers need a simple, machine-independent programming model that can be efficiently mapped to large-scale parallel machines. We believe that Fortran D, a version of Fortran enhanced with data decompositions, provides such a portable data-parallel programming model. Its success will depend on the compiler and environment support provided by the Fortran D programming system.

The Fortran D compiler includes sophisticated intraprocedural and interprocedural analyses, dynamic data decomposition, program transformation, communication optimization, and support for both regular and irregular problems. Though significant work remains to implement the optimizations presented in this paper, based on preliminary experiments we expect the Fortran D compiler to generate efficient code for a large class of data-parallel programs with only minimal user effort.

The Fortran D environment is distinguished by its ability to accurately estimate the performance of programs using collective communication on real parallel machines, as well automatically choose data partitions that account for the characteristics of both the compiler-generated code and underlying machine. It will assist the user in developing efficient Fortran D programs. Overall, we believe hat the Fortran D programming system is a powerful and useful tool that will significantly ease the task of writing portable data-parallel programs.

7 Acknowledgements

The authors wish to thank Vasanth Bala, Geoffrey Fox, and Marina Kalem for inspiring many of the ideas in this work. We are also grateful to the ParaScope research group for providing the underlying software infrastructure for the Fortran D programming system.

References

[1] V. Balasundaram. Translating control parallelism to data parallelism. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Houston, TX, March 1991.

[2] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

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

[4] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

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

[6] D. Callahan, K. Kennedy, and U. Kremer. A dynamic study of vectorization in PFC. Technical Report TR89-97, Dept. of Computer Science, Rice University, July 1989.

[7] B. Chapman, H. Herbeck, and H. Zima. Automatic support for data distribution. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[8] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[9] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[10] M. Gerndt. Updating distributed variables in local computations. *Concurrency—Practice & Experi-*

ence, 2(3):171–193, September 1990.

[11] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[12] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[13] R. Hill. MIMDizer: A new tool for parallelization. *Supercomputing Review*, 3(4):26–28, April 1990.

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

[15] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report TR90-149, Dept. of Computer Science, Rice University, January 1991. To appear in J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*, Elsevier, 1991.

[16] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12(4), August 1991.

[17] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[18] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[19] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[20] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[21] 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(2):102–118, February 1990.

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

[23] 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.

[24] 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.

[25] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[26] C. Pancake and D. Bergmark. Do parallel languages respond to the needs of scientific programmers? *IEEE Computer*, 23(12):13–23, December 1990.

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

[28] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[29] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[30] L. Snyder and D. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[31] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 5.2-0.6 edition, September 1989.

[32] M. J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[33] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

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