Automatic Array Alignment for Distributed Memory Multicomputers

Mitsuru Ikei

Hitachi Chemical Company Ltd.

Michael Wolfe *

Oregon Graduate Institute of Science & Technology

P.O. Box 91000

Portland OR 97291

503-690-1153

FAX: 503-690-1029

mwolfe@cse.ogi.edu

*Supported in part by grants from DARPA, Intel Supercomputer Systems Division and the Oregon Advanced Computing Institute

Abstract

Languages such as Fortran-D and High Performance Fortran use explicit virtual processor arrays (Templates or Decompositions) to which the programmer explicitly aligns the arrays of the program. We explore a technique to automate this process. In particular, we are interested in imperative languages, similar to Fortran, and scientific computations using indexed data structures (arrays). We have implemented automatic alignment in our imperative language using techniques similar to those used for the functional language Crystal. Here we show what changes we made to the procedure to take into account the imperative nature of our source language. In particular, a functional language has a single-assignment attribute; each array element is defined only once. An imperative language allows array elements to be redefined; this allows the imperative program to use arrays with fewer dimensions, as we show. Our compiler technique "resurrects" the lost dimensions of the data arrays to successfully apply automatic data alignment.

1 Introduction

Compilers for current supercomputers can achieve high performance for programs which are written using a "vector" model (using vectorizable loops). To achieve similar effectiveness on parallel computers using the data parallel model, several programming languages have been proposed. Typically these languages have global data which is processed by the entire processor ensemble and which can be accessed by global indices. They also have parallel loop syntax which programmers can use to explicitly express concurrency. To execute these programs, the global data needs to be decomposed into pieces and distributed among the processor ensemble. The output of the compiler often forms an SPMD (Single Program, Multiple Data) program, where each processor executes the same program text, but calculates its own share of the global work on the distributed data structure.

We can consider this as a two-phase process: selecting how to decompose the data, and distributing the work to match the data decomposition. Languages like Fortran D and High Performance Fortran have constructs with which programmers manually decompose the data [FHK+90, HPF92]. In this case, the main role of the compiler is to derive the SPMD node program with explicit communication from the global program and the given

data decomposition [HKT92]. Our main focus of this paper is to automate the data decomposition step.

Recent work on the functional language Crystal has produced a compiler which automatically converts Crystal programs into executable data-parallel SPMD programs for Distributed Memory Multicomputers [Li91]. The compiler uses three steps to transform the functional program to executable SPMD form: (1) Control Structure Synthesis, (2) Domain Alignment and (3) Communication Synthesis.

Our research examines how to apply these compiler steps to an *imperative* language; in particular, we have implemented analogs of Control Structure Synthesis and Domain Alignment in our Tiny program restructuring research tool [Wol91, Ike92]. In this paper we discuss mainly these two steps. The rest of this paper is organized as follows. In Section 2, we show a simple example of how a program can be converted for multicomputers. Sections 3 and 4 discuss Control Structure Synthesis and Domain Alignment respectively. In Section 5, we discuss other research topics and we conclude in Section 6.

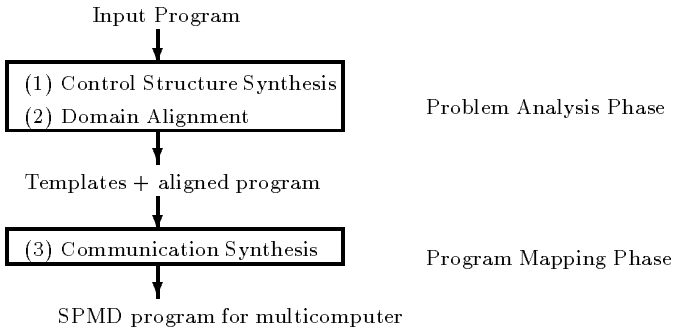

Figure 1: Compiling Steps

2 Compiler Phases

Our compiler for multicomputers will consist of two main phases: a Problem Analysis phase and Program Mapping phase, as shown Figure 1.

In the first phase, the compiler knows nothing about the physical machine structure; it concentrates only on the data spaces and the iteration spaces which are defined and used in the program. In other words, we assume a virtual processor array which has infinite dimensionality of infinite size and map the spaces of the program to a subset of the virtual processor array. In the following program,

```
real a(100),b(100)
for i=2,99 do
   b(i)=a(i-1)+a(i+1)
endfor
```

our compiler first finds concurrency and restructures it to parallelize as many loops as possible. Applying our array alignment algorithm, we get the following.

```
real a(100),b(100)
align(a(#1),b(#1))
doall i=2,99 do
   b(i)=a(i-1)+a(i+1)
enddoall
```

The output of the first phase assumes a single virtual processor array or *template* and all arrays in the program are aligned to this template. Every loop in the program can also be aligned to one dimension of the virtual processor array. In the example, one array (in this case `b()`), is chosen as the template; it is obvious that the doall loop is related to both the `a()` and `b()` arrays.

In the second phase, the compiler will actually decompose the template to fit in the physical nodes; it will organize a task for each node with the necessary data movement among processors. Assuming we have four processors connected in a line, and the template is divided into four blocks, the node program is:

```
real a(25),b(25)
real from_left,from_right
if(my_id()==1) then
   send(2,a(25))
   from_right=receive(2)
elseif(my_id()==4) then
   send(3,a(1))
   from_left=receive(3)
else
   send(my_id()+1,a(25))
   from_left=receive(my_id()-1)
   send(my_id()-1,a(1))
   from_right=receive(my_id()+1)
endif
```

```
for i=2,24 do
   b(i)=a(i-1)+a(i+1)
endfor
if(my_id()!=1) the
   b(1)=from_left+a(25)
endif
if(my_id()!=4) the
   b(25)=a(24)+from_right
endif
```

In this code, `my_id()` returns the processor id, `send(dist,data)` sends `data` to PE `dist` and `receive(src)` receives the data from PE `src`. In the beginning of the program, each processor sends its boundary data and receives the necessary data. In this example there is no more efficient way to divide the data; generally there are many ways to divide the data and the selection of the decomposition is the main aim of this phase.

Although we only consider block distribution here, we are also planning to use distributions with guard regions and some simple dynamic distributions, as discussed in Section 5. After the distribution strategy is set, we try to find necessary communication subroutines. We use aggregate communication routines, where applicable, to minimize communication cost. Evaluating communication overhead with several parameters under the target

```
dom T = [0..n]
dom D1 = [1..n]
dom D2 = D1 * D1
dom D3 = D2 * T
dfield a(i,j):  D2 = a0[i,j]
dfield b(i,j):  D2 = b0[i,j]
dfield c(i,j,k):  D3 =
   if( k = 0 )then 0.0
   else c(i,j,k-1) + a(i,k) * b(k,j) fi
```

Figure 2: Crystal mm program

machine constraints, we can generate the most appropriate SPMD program.

3 Control Structure Synthesis

In Control Structure Synthesis, a Crystal program is converted to an imperative data parallel program. This is a necessary step, since the source language is not imperative, but the target machine is. For an imperative source language like ours, the control structure already exists in the original program; however, we use an analog of this step to refine the program.

3.1 Crystal

In Figure 2, we show a Crystal matrix multiplication (mm) program. A Crystal program consists of two parts: domain definitions and data field

definitions. In the first half of the mm program, four domains `T`, `D1`, `D2` and `D3` are defined. Both `T` and `D1` are one dimensional and `D2` and `D3` are two and three dimensional domains respectively. In the second half, three data fields `a()` and `b()` (on `D2`) and `c()` (on `D3`) are defined. The two data fields `a()` and `b()` are used to hold the input matrices. Data field `c()` holds the result of the mm calculation.

The objective of the Control Structure Synthesis phase of the Crystal compiler is to synthesize imperative loops from data fields. The compiler takes the following four steps for this:

1. Compute the dependence graph
2. Decompose the graph into strongly connected regions (SCRs, or maximal cycles)
3. Sort the SCRs topologically to find an execution order
4. Break dependence cycles by finding a dependence carrying loop

First the compiler builds a dependence graph between all data fields as shown Figure 3. Each SCR in the graph can be an imperative loop which defines the data fields involved in the SCR. The compiler sorts the SCRs to find an execution order of the loops that preserves the dependence relations.

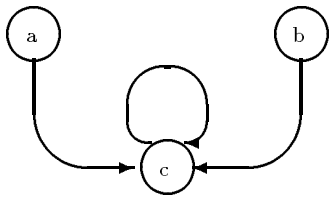

Figure 3: Crystal mm dependence graph

SCRs which have no dependence cycles, such as `a()` and `b()` in Figure 3, can be executed in parallel. For SCRs with dependence cycles, the compiler finds a dependence-carrying loop and makes this be an outermost sequential loop, thus allowing the others loops to execute in parallel. Figure 4 shows the mm program after Control Structure Synthesis. Forall loops are used to designate that the `a()` and `b()` definitions can be executed on parallel. The for-loop along the third dimension of the `c()` definition is found as the dependence carrying loop and the inner forall-loop on `D2` is used to describe the parallelism. Thus the Crystal compiler derives an imperative data parallel program from the functional specification.

3.2 Tiny

Tiny is a program restructuring tool for a small imperative language; it supports sequential and parallel loop constructs, such as used in many scientific

```
dom T = [0..n]
dom D1 = [1..n]
dom D2 = D1 * D1
dom D3 = D2 * T
forall((i,j):D2){ a(i,j) = a0[i,j] }
forall((i,j):D2){ b(i,j) = b0[i,j] }
for(k:T){
   forall((i,j):D2){
      c(i,j,k) = if( k = 0 )then 0.0
         else c(i,j,k-1) + a(i,k) * b(k,j) fi
   }
}
```

Figure 4: Crystal mm data parallel program after Control Structure Synthesis

```
real a(n,n),b(n,n),c(n,n)
for i = 1, n do
   for j = 1, n do
      c(i,j) = 0.0
      for k = 1, n do
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
      endfor
   endfor
endfor
```

Figure 5: Tiny mm program

programs. We select this simple language to demonstrate our research. (We use Tiny to mean both the tool and its language.) Since Tiny is an imperative language, the compiler doesn't need to synthesize loops as with Crystal; the original program already has a legal loop structure. As an analog to the Control Structure Synthesis phase of the Crystal compiler, our compiler finds implicit parallelism in the imperative Tiny program, and converts the program to a normalized form. In some sense, the Crystal compiler sequentializes and normalizes a declarative program, whereas the Tiny compiler parallelizes and normalizes an imperative program. The output of the first phase of both compilers is similar. We show a Tiny mm program in Figure 5. We take the following four steps to convert a program to normalized form.

1. Compute a data dependence graph
2. Decompose the loops (loop distribution)
3. Interchange sequential loops outwards
4. Parallelize inner loops (doall-loops)

First we construct a data dependence graph to preserve the semantics of the program throughout the restructuring phase. Next we try to decompose all non-tightly nested loops to tightly nested ones, comparable to data fields

```
real a(n,n),b(n,n),c(n,n)
doall i = 1, n do
   doall j = 1, n do
      c(i,j) = 0.0
   enddoall
enddoall
for k = 1, n do
   doall i = 1, n do
      doall j = 1, n do
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
      enddoall
   enddoall
endfor
```

Figure 6: Tiny mm program, normalized by the analog of Control Structure Synthesis

in Crystal programs. We then try to interchange sequential dependence-carrying loops outwards, so that inner loops can be parallelized. In Figure 6, we show the result of this conversion. Here, the non-tightly nested loop can be decomposed into two tightly nested loops. The dependence carrying (`k`) loop in the second nested loop is interchanged to the outermost position and the two inner loops have been successfully parallelized. This normalized data parallel program is intentionally organized to resemble the Crystal data parallel program. It may not have an efficient loop structure for direct mapping to a parallel computer, however this representation shows the relations between data spaces (arrays) and iteration spaces (loops) clearly,

which we use in the next step for array alignment.

4 Domain Alignment

After an imperative data parallel program is derived from the original functional definitions, the Crystal compiler tries to find the optimal spatial alignment of data fields to minimize communications among virtual processors. We apply essentially the same method to arrays in Tiny programs.

4.1 Crystal

Since an imperative data parallel program is derived, we need to distribute the data fields among the processors to run the program on a multicomputer. The Crystal compiler takes a two level distribution strategy. First, it aligns all data fields to a global Template (Domain Alignment), where a Template is a Cartesian mesh of virtual processors. The Template (with all data fields aligned to it) will subsequently be distributed to physical processors (Distribution). Domain Alignment proceeds in the following three steps:

1. Find the reference patterns
2. Build a Component Affinity Graph (CAG)
3. Partition the CAG (using a bipartite partitioning heuristic)

```
c(i,j,k) ← c(i,j,k-1)
c(i,j,k) ← a(i,k)
c(i,j,k) ← b(k,j)
```

Figure 7: Crystal mm reference patterns

The Crystal compiler selects a data field with maximum dimensionality to represent the Template and tries to relate or align all other data fields to this Template. First it collects all reference patterns in the program. For instance, the reference patterns for the Crystal mm data parallel program in Figure 2 are shown in Figure 7. These three patterns are derived from the line where `c(i,j,k)` is defined. Each reference pattern relates a right hand side data field reference to the left hand side element where it is used. From these reference patterns, the compiler builds a Component Affinity Graph. The CAG is a weighted undirected graph. Nodes of CAG are domain components (one for each dimension) and edges are relations derived from the reference patterns. We denote a domain component using notation like `a.1` which refers to 1^{st} domain component of the data field `a`. In Figure 8, the domain components of the three data fields in the program are in the three columns of the CAG. A domain component which is defined using a sequential (dependence-carrying) index (`k` in this case) like `c.3` is called a

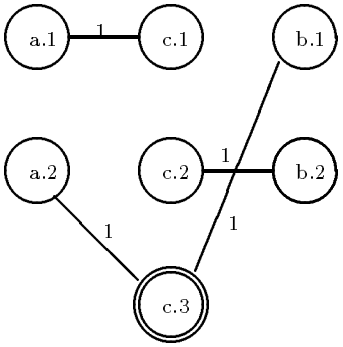

Figure 8: Crystal mm CAG

temporal domain component and is represented by a double circle. Given a reference pattern, we define the distance between a domain component in the LHS and that in the RHS as the difference of an index expression in the RHS domain from the corresponding LHS index expression. If the distance between two different domain components is constant, we say that there is an affinity relation between the domain components and draw an edge between the components. For instance, in the reference pattern:

$$\texttt{c(i,j,k)} \leftarrow \texttt{a(i,k)}$$

the distance of the index expression of `c.1` and the index expression of `a.1` is $(i - i) = 0$. Since it is constant, there is an edge between the two component domains in Figure 8. There are three kinds of weights that the edges can have: epsilon, one and infinity. If one node has two edges to two different

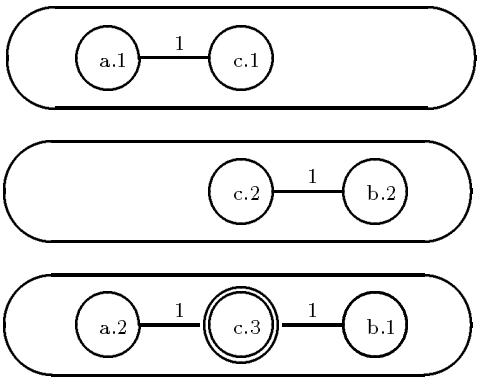

Figure 9: Crystal mm partitioned CAG

components of the same domain, both edges are weighted as epsilon. An edge between two temporal components has a weight of infinity. All others are given a weight of one, such as the edge between `a.1` and `c.1`.

After the CAG is build, the compiler partitions it. Assuming n is the dimensionality of the Template, we want to partition the CAG into n groups such that the sum of the edges cut by partitioning is minimized. Components which are derived from the same data field must belong to different partitions. In Figure 9, we show the partitioned CAG of the example mm program. In this case, partitioning is easy, but this problem is in general NP-complete. The Crystal compiler uses simple but effective heuristics to solve this problem. From Figure 9, we can get the chosen relative array alignment. Using `c` as the Template, we denote the alignment as follows.

```
align(a(#1,#2),c(#1,1,#2))
align(b(#1,#2),c(1,#2,#1))
```

4.2 Tiny

Applying the same method to Tiny looks straightforward. Data fields are similar to arrays. Data field definitions inside loops are essentially the same as assignment statements. However there are important differences. Crystal is a single assignment language, with a declarative style; there is no way to redefine elements of a data field. But a Tiny program is quite different. Reassignment and reuse of array elements in imperative languages is quite common. Not all loop index variables will appear in array references, even on the left hand side. In the Crystal mm program, the `c` data field has three dimensions, one of which is indexed by `k`; the Tiny mm program has only two dimensions for `c`, and the `k` index is not used for `c`. As we will explain shortly, these "missing indices" are important, and our compiler adds implicit dimensions to arrays such as `c`. The Tiny compiler uses the following four steps in its Domain Alignment phase:

1. Find the reference patterns
2. Add implicit dimensions
3. Build a Component Affinity Graph

```
c(i,j) ← c(i,j)
c(i,j) ← a(i,k)
c(i,j) ← b(k,j)
```

Figure 10: Tiny mm reference patterns

4. Partition the CAG

First we derive reference patterns from a Tiny program, just as the Crystal compiler does. From the program in Figure 6, we get reference patterns as shown in Figure 10. Comparing this to Figure 7, we realize that the Tiny `c(i,j)` array has only two domain components; in the Tiny mm program, array element `c(i,j)` is reassigned values along the sequential `k` loop. To use the Crystal data alignment scheme, we need to add this temporal domain as an implicit dimension; the implicit dimension is used only for the Domain Alignment phase, and is ignored thereafter. The Tiny compiler currently uses the following simple algorithm.

1. If there is an assignment statement to a scalar inside loops, the scalar is a candidate and all the loops that include the assignment statement are possible implicit domains.

2. If there is an assignment to an array inside loops and one or more loop indices do not appear in the LHS array subscript expressions,

```
c(i,j)[k] ← c(i,j)[k-1]
c(i,j)[k] ← a(i,k)
c(i,j)[k] ← b(k,j)
```

Figure 11: Tiny mm reference patterns with implicit dimensions

the array is a candidate. The loops with the missing indices are the possible implicit domains.

3. If an LHS is a candidate and if in the RHS there is a subscript expression which contains a loop index of the possible implicit domain such that the distance between the subscript expression and the loop index is constant, an implicit dimension is added to the candidate.

According to rule 2, the `k` loop is a possible implicit domain for array `c`, and according to rule 3, the distance between the `k` subscript for array `a` (and `b`) and the implicit loop index is constant. Using this algorithm, the compiler revises the reference patterns as shown in Figure 11. Denoting the implicit dimension of array `c` as `c.k`, the CAG of the Tiny mm program is as shown in Figure 12. We select the `c` array as our Template since it has the highest dimensionality (after adding the implicit dimension). The Tiny compiler uses exactly the same algorithm for partitioning that was developed for the Crystal compiler. The input program shown in Figure 5

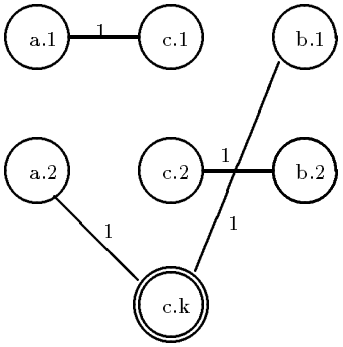

Figure 12: Tiny mm CAG

is automatically converted to the program with `align` statements as shown in Figure 13.

5 Data Distribution

Before generating communication constructs to interchange data efficiently among physical processors, the compiler must decide how to decompose the Template (and hence the data arrays) among the physical processors. This takes place in the Distribution phase. The Crystal compiler takes a simple static distribution strategy. We are investigating this strategy and evaluating other potentially more effective strategies; a strategy will be selected after evaluating and comparing the cost of data movement. In this paper, we discuss one dimensional physical processor arrays.

```
real a(n,n),b(n,n),c(n,n)
align(a(#1,#2),c(#1,1,#2))
align(b(#1,#2),c(1,#2,#1))
doall i = 1, n do
   doall j = 1, n do
      c(i,j) = 0.0
   enddoall
enddoall
for k = 1, n do
   doall i = 1, n do
      doall j = 1, n do
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
   enddoall
   enddoall
endfor
```

Figure 13: Tiny aligned mm program

5.1 Simple Distribution Strategy

After Domain Alignment, we have a Template to which all arrays are aligned in a program. Each domain of the Template is either *temporal*, indexed by a sequential for-loop, or *spatial*, indexed by a parallel doall-loop. Each domain can be either decomposed (distributed among processors), or kept entirely within a single processor's local memory. Decomposing a temporal domain cannot effectively use parallelism; thus one obvious way to decompose the data is to select one spatial domain to be distributed. For instance, the calculation part of the program in Figure 13 can be converted to the SPMD

```
real c(mylower:myupper,n),a(mylower:myupper,n),b(n,n)
for k = 1, n do
   for i = mylower, myupper do
      for j = 1, n do
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
      endfor
   endfor
endfor
```

Figure 14: Simple Local Tiny mm program

program in Figure 14. Here, we decomposed along `c.1`. Since array `b()` has no domain aligned to `c.1`, each processor has a copy of the entire array `b()`. If `b()` is too large to fit in the memory of each processor, there are several options. One way is to decompose also along `c.2`; another is a secondary decomposition of `b.1` without decomposing `c.k`. We don't discuss this further in this paper, but the decision can be made after the evaluation of communication overhead.

5.2 Dynamic Distribution

Previous work has assumed that the owner of an array would not change throughout the execution of program. But sometimes it is better to move chunks of data among processors along with their ownerships. Using this strategy, the mm program can be converted as shown in Figure 15, us-

```
real c(mylower1:myupper1,n),a(mylower1:myupper1,n),b(n,mylower2:myupper2)
dorotate(b,2)
   for i = mylower1, myupper1 do
      for j = mylower2, myupper2 do
         for k = 1, n do
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
         endfor
      endfor
   endfor
enddorotate
```

Figure 15: Local Tiny mm program with rotate

ing a data rotation strategy [Wol90]. Arrays `c()` and `a()` are decomposed row-wise and distributed statically among all processors. Although array `b()` is decomposed column-wise and distributed among all processors similarly, we redistribute it by the dorotate construct. In each iteration of the `dorotate(b,2)` loop, each processor rotates its share of array `b()` to the next processor synchronously; eventually, each processor sees the entire array `b()` piece by piece, and the array ends up in the same place it started. Note this distribution also uses the previously calculated alignment. Domain `c.1` and `a.1` are both decomposed together. Dimension `c.2` is not decomposed, even though `b.2` is divided column-wise; however, we notice that only part of `c.2` is used in each iteration of the dorotate calculation.

We are investigating automation of this transformation. In particular, it changes the order of the dot product accumulation for each element of the result array in this program.

5.3 Communication Constructs

After the distribution strategy is set, the compiler must insert communication constructs into the SPMD program. Right now, the Tiny compiler does not generate executable SPMD output. As with previous work, we will focus on global communication patterns, which can be implemented efficiently and more effective utilize the limited interprocessor bandwidth. The reference patterns and data dependence relations, which have already been collected, will be used to detect global communication points and insert the necessary communication primitives.

6 Conclusion

We have implemented automatic array alignment in the Tiny program restructuring tool. The methods we used to find the optimal alignment are analogous to the methods used in the compiler for the functional language Crystal. Even though the imperative language we used is very different

from the declarative style used in Crystal, we found that a similar alignment strategy worked well. Specific differences are:

- The Crystal compiler needs to deduce a legal imperative loop structure from the source program. A Tiny program already has a legal loop structure, but it may need to be restructured or optimized. Both compilers use dependence information to generate and/or optimize the loop structure.

- Tiny assignment statements are similar to data field definitions in Crystal, but not exactly the same. In particular, two statements may assign to the same array (as in the mm program), but a Crystal program has only one definition for each data field. Thus, Tiny loop restructuring may separate the assignments to an array into separate loops, while Crystal will always keep the data field definition together.

- Due to the single-assignment nature of the Crystal language, each data field element can be defined only once; thus, a data field defined in a 3-dimensional domain will have three dimensions. In a Tiny program, this is not true; to apply the same alignment strategy, we modified the Tiny compiler to introduce implicit dimensions. We found that implicit dimensions, which capture the iteration space of a program,

are important for proper alignment.

We are now developing the algorithms to convert the aligned program to an efficient SPMD node program.

The Tiny compiler accepts a small, "toy" language; we properly view this effort as technology demonstration only. Our results so far are encouraging, and we are also investigating how to use these methods in a full industrial-strength parallel language.

References

[FHK+90] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report TR90-141, Rice Univ., December 1990. Revised April,1991.

[HKT92] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communication of the ACM*, 35(8):66–88, August 1992.

[HPF92] HPFF. *High Performance Fortran Specification*, draft ver. 0.4 edition, November 1992.

[Ike92] Mitsuru Ikei. Automatic program restructuring for distributed memory multicomputers. M.S. thesis, Oregon Graduate Institute, Dept. of Computer Science and Engineering, April 1992.

[Li91] Jingke Li. Compiling Crystal for distributed-memory machines. PhD dissertation, Yale Univ., Dept. Computer Science, October 1991.

[Wol90] Michael Wolfe. Loop rotation. In David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 531–553. MIT Press, Boston, 1990.

[Wol91] Michael Wolfe. The Tiny loop restructuring research tool. In *Proc. 1991 International Conf. on Parallel Processing*, volume II, pages 46–53, St. Charles, IL, August 1991. Penn State Press.