A Test Suite Approach for Fortran 90D Compilers on MIMD Distributed Memory Parallel Computers

Min-You Wu
Geoffrey C. Fox

CRPC-TR92254
1992

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

To appear in Proceedings of Scalable High Performance Computing Conference '92.

A Test Suite Approach for Fortran90D Compilers on MIMD Distributed Memory Parallel Computers

Min-You Wu

Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260

Geoffrey C. Fox

Syracuse Center for Computational Science
Syracuse University
Syracuse, NY 13244-4100

Abstract

This paper describes a test suite approach for a Fortran90D compiler, a source-to-source parallel compiler for distributed memory systems. Different from Fortran77 parallelizing compilers, a Fortran90D compiler does not parallelize *sequential constructs. Only parallelism expressed by Fortran90D parallel constructs is exploited. We discuss compiler directives and the methodology of parallelizing Fortran programs. An introductory example of Gaussian elimination is used, among other programs in our test suite, to explain the compilation techniques.*

1 Introduction

Current commercial parallel supercomputers are clearly the next generation of high performance machines. Although parallel computers have been commercially available for some time, their use has been mostly limited to academic and research institutions. This is mainly due to the lack of software tools to convert old sequential programs and to develop new parallel programs.

Fortran has been used as the language for developing most of the industrial (and practical) software in the past few decades. There has been significant research in developing parallelizing compilers. Most notable examples include Parafrase at the University of Illinois [1] and PFC at Rice university [2]. In this approach, the compiler takes a sequential Fortran77 program as input, applies a set of transformation rules, and produces a parallelized code for the target machine. However, it is not clear if this type of automatic parallelization will work in general, especially for large codes.

A sequential language, such as Fortran77, hides the parallelism of a problem in sequential loops and other sequential constructs. A program is written without any parallel constructs provided, even if the user is willing to express parallelism explicitly. Compiling a sequential program into a parallel program is not a natural approach. An alternative approach is to use a programming language that can naturally represent an application without losing the application's original parallelism. Fortran90 (with some extensions) is such a language. The extensions may include the *forall* statement and compiler directives for data partitioning, such as decomposition, alignment, and distribution. Fortran90 with these extensions is what we called "Fortran90D", a Fortran90 version of the FortranD language [3]. A Fortran90D parallel compiler exploits only the parallelism expressed in these parallel constructs. We do not attempt to *parallelize* other constructs, such as *do* loops and *while* loops, since they are naturally sequential. Developing a compiler under this assumption becomes much easier. Also users can reliably understand what parallelism will be exploited.

Different approaches to parallelizing Fortran programs are shown in Figure 1. First, the user who wants to write new programs can use Fortran90 with FortranD extensions. A parallel compiler then translates the Fortran90D program into Fortran plus Message-Passing (Fortran+MP) code. Secondly, the old Fortran77 codes can be rewritten into Fortran90D with the help of a migration tool. The migration step from Fortran77 to Fortran90 may be important for migrating existing codes to this portable standard. Note that Fortran+MP has been shown to work for a large set of applications on MIMD machines, but is not fully portable. Finally, parallelizing compilers can also convert some Fortran77 programs directly into Fortran+MP codes.

Tremendous effort in the last decade has been devoted to the goal of running existing Fortran programs on new parallel machines. Restructuring compilers for

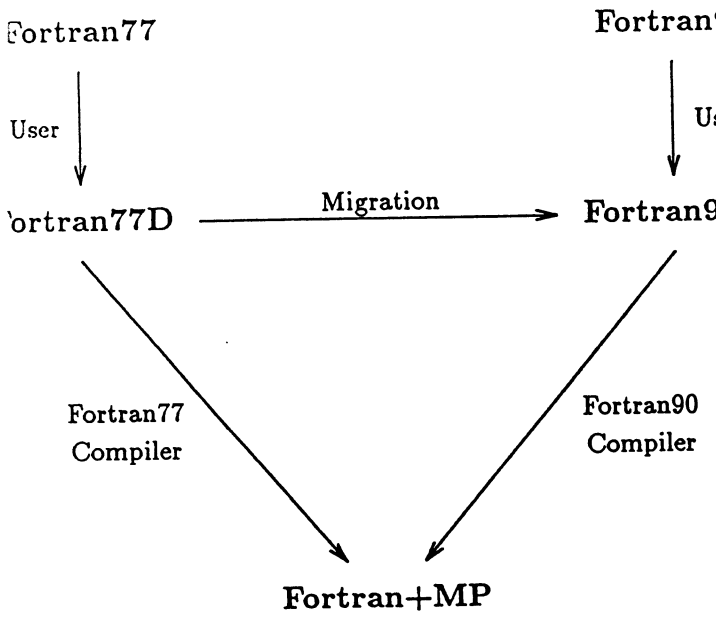

Figure 1: Approaches to parallelizing Fortran programs.

Fortran77 programs have been researched extensively for shared memory systems [4]. The compilation technique of Fortran77 for distributed memory systems has been addressed by Callahan and Kennedy [5]. Currently, a Fortran77D compiler is being developed at Rice [6]. Hatcher and Quinn provide a working version of a C* compiler. This work converts C* — an extension of C that incorporates features of a data parallel SIMD programming model — into C plus message passing for MIMD distributed memory parallel computers [7]. A Fortran90D compiler can share many techniques used in the C* compiler. The ADAPT system compiles Fortran90 for execution on MIMD distributed memory architectures [8]. It does not compile the *forall* statement but does translate sequential *do* loops into parallel loops. SUPERB is an interactive source-to-source parallelizer. It compiles a Fortran77 program into a semantically equivalent parallel SUPRENUM Fortran program for the SUPRENUM machine [9]. Koelbel extended the features of BLAZE into the Kali language and compiled it for nonshared memory machines [10].

2 Data Partitioning

A problem to be solved on a parallel computer must be partitioned into many parallel actions. Partitioning for data parallelism can be performed in two ways: *data partitioning* or *computation partitioning*. The former partitions data, and consequently, assigns related computations to PEs. The latter partitions computations and allocates the corresponding data to PEs [11]. Although computation partitioning could be better suited for problems with irregular structures, it may require more complex analysis and result in larger communication overhead. The data partitioning is suitable for many scientific computations in which the computation density associated with data is evenly distributed.

Data partitioning can be done in two steps which separate machine independent problem parallelism from machine dependent details. The first step is to determine the best alignment among different arrays. To reduce unnecessary data movement, distributed arrays should be aligned with each other in a fashion that is usually determined by the underlying computation structure. The alignment of arrays depends on the program itself and is usually machine-independent. The second step is to determine how arrays should be distributed to the underlying computer structures and is therefore machine dependent. The objective of array distribution is to balance the computation load for each PE and to minimize the communication between PEs. Array distribution is largely dependent on machine structures, such as the number of PEs, communication mechanisms, and interconnection topologies.

We provide users with some annotation facilities for data partitioning. The annotation takes the form of compiler directives.

Decomposition directives

A decomposition directive is used to declare a problem domain. It declares the name, dimensionality, and size of a decomposition. The decomposition directive defines arrays as data parallel and is machine independent. Examples of decomposition are shown below:

```
DECOMPOSITION A(N)
DECOMPOSITION B(N,N)
```

where A is declared as an one-dimensional decomposition of size N, and B is a two-dimensional N by N decomposition.

Alignment directives

An alignment directive aligns one array to another. Arrays aligned with each other will share a common "data parallelism". The alignment directive specifies which elements of two arrays are to be allocated to the same place by aligning each axis of a source array with a given target array. The following examples of the alignment directive specifies different alignment patterns:

1. Alignment offsets:
   ALIGN A(I,J) with X(I-1,J+1)
2. Alignment strides:
   ALIGN B(I,J) with X(I*2,J*2)
3. Embedding:
   ALIGN C(I) with X(I,2)
4. Permutation:
   ALIGN D(I,J) with X(J,I)

Alignment is usually machine independent. A complete specification of the alignment directive is described in [3].

Distribution directives

A distribution directive provides some control over the distribution of an array. Specifications are block distribution, scattered distribution, block-scattered distribution, or no distribution. The relative weight of distribution along each axis indicates the distribution ratio among axes. The distribution ratio is defined as the ratio of the number of partitions along different axes. Examples of distribution are shown below:

DISTRIBUTION A(BLOCK,:)
DISTRIBUTION B(CYCLIC,BLOCK)

These compiler directives are inserted by the user to specify a partitioning pattern. Moreover, they could be generated by an automatic partitioner in future version of the compiler. According to distribution directives, data are either *distributed* or *replicated*. Data that are partitioned by directives will be distributed, and others will be replicated. A copy of replicated data resides in each PE. Some comments could be used to allow the user to print out the actual data distribution at runtime.

3 Compiler System Diàgram

The system diagram of the Fortran90D compiler is shown in Figure 2. Given a syntactically correct Fortran90D program, the first step of compilation is to generate a parse tree. The partitioning module divides the program into tasks and allocates the tasks to processor elements (PEs) according to compiler directives — decomposition directives, alignment directives, and distribution directives. There are three ways to generate the directives: 1) users can insert them, 2) programming tools can help users to insert them, or 3) automatic compilers can generate them. In the first approach, users write programs with explicit distribution and alignment directives. A programming tool can generate useful analysis to help users decide partitioning styles, and measure performance to help users improve program partitioning interactively.

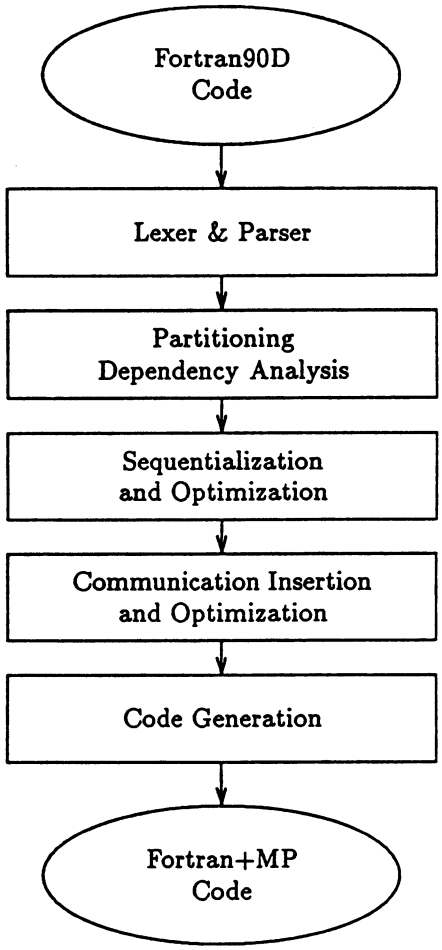

Figure 2: Diagram of the compiler.

Dependency analysis is carried out to obtain dependency information for use in sequentialization and insertion of communication primitives. Standard techniques of data dependency analysis for Fortran programs can be applied here [12]. Fortunately, the dependency analysis technique for Fortran90D is much simpler than the one for Fortran77 since we do not parallelize sequential constructs.

After partitioning, a program becomes a set of tasks. Each task must be sequentialized since it will be executed on a single processor. This is performed by the sequentialization module. Array operations and *forall* statements in the original program will be transferred into loops or nested loops. This module also performs computation optimization such as invariant expression floating and reordering.

The dependencies between tasks introduce interprocessor communication. Whenever the data required for executing a statement are not in the local memory, communication primitives are to be inserted. Optimization is applied to minimize synchronization, eliminate unnecessary or redundant data transfers, and to combine communication wherever possible. One important optimization is overlapping computation and communication to overcome communication latency. Optimization may be performed at compile time if the problem is statically defined, and all required information is available at that time. Otherwise, based on partial information, we do compile time analysis to generate runtime tests. At runtime, based on the test results, communication can be optimized. Library routines are used to translate certain parallel constructs, such as reduction, broadcasting, etc. Finally, the code generator produces the Fortran+MP code for target message-passing systems.

4 An Introductory Example: Gaussian Elimination

We use Gaussian elimination as an example for translating a Fortran90D program into a Fortran+MP program. The Fortran90D code is shown in Figure 3, and the hand-compiled Fortran+MP code is shown in Figure 4. We hand-compiled the code by applying rules stated in the previous sections. Note that the size of the Fortran90D code is much smaller than that of the Fortran+MP code. The former has 20 lines, and the latter has 66 lines.

Arrays a and *row* are partitioned by compiler directives. The second dimension of a is block-partitioned, while the first dimension is not partitioned. Array *row*

```
      integer, array(0:N-1) :: indx
      integer, array(1) :: iTmp
      real, array(0:N-1,0:NN-1) :: a
      real, array(0:N-1) :: fac
      real, array(0:NN-1) :: row
      real :: maxNum

C$  DECOMPOSITION a(N,NN)
C$  ALIGN row(J) WITH a(0,J)
C$  DISTRIBUTE a(:,BLOCK)

      indx = -1
      do k = 0, N-1
        iTmp = MAXLOC(ABS(a(:,k)), MASK=indx .EQ. -1)
        indxRow = iTmp(1)
        maxNum = a(indxRow,k)
        indx(indxRow) = k
        fac = a(:,k) / maxNum

        row = a(indxRow,:)
        forall(i=0:N-1, j=k:NN-1, indx(i) .EQ. -1)
     &    a(i,j) = a(i,j) - fac(i) * row(j)
      end do
```

Figure 3: Fortran90D code for Gaussian elimination.

is block-partitioned too. Each partition may include many array elements. Since they execute on a single PE, the parallel constructs must be sequentialized. An array operation in the Fortran90D program is sequentialized into a *do* loop. Loop boundaries are defined by the array declaration. When a replicated array is computed from replicated data, the operation is performed on each PE. For example, the array operation

```
indx = -1
```

is translated into

```
do i = 0, N-1
  indx(i) = -1
end do
```

This is executed on each PE. If the replicated array is computed from distributed data, the operation is performed on one PE, and the result may be broadcast to other PEs later. A test is inserted to determine which PE will execute the statement. For example, the statement

```
tmp = ABS(a(:,k))
```

is translated into

```
if (k/B .EQ. thisPE) then
  do i = 0, N-1
```

```
      thisPE = mynode()
      numNode = numnodes()
      B  = NN / numNode
      minCol  = thisPE * B
      logical mask(0:N-1)
      real tmp(0:N-1)

C     integer, array(0:N-1) :: indx
C     integer, array(1) :: iTmp
C     real, array(0:N-1,0:NN-1) :: a
C     real, array(0:N-1) :: fac
C     real, array(0:NN-1) :: row
C     real :: maxNum

C$  DECOMPOSITION a(N,NN)
C$  ALIGN row(J) WITH a(0,J)
C$  DISTRIBUTE a(:,BLOCK)
      integer indx(0:N-1)
      real aLoc(0:N-1,0:B-1)
      real fac(0:N-1)
      real rowLoc(0:B-1)
      real maxNum

C     indx = -1
      do i = 0, N-1
        indx(i) = -1
      end do

C     do k = 0, N-1
C       iTmp = MAXLOC(ABS(a(:,k)), MASK=indx .EQ. -1)
C       indxRow = iTmp(1)
      do k = 0, N-1
        if (k/B .EQ. thisPE) then
          do i = 0, N-1
            mask(i) = indx(i) .EQ. -1
          end do
          do i = 0, N-1
            tmp(i) = ABS(aLoc(i,k-minCol))
          end do
          indxRow = MaxLoc(tmp, N, mask)
        end if

C       maxNum = a(indxRow,k)
        if (k/B .EQ. thisPE)
     &    maxNum=aLoc(indxRow,k-minCol)

C       indx(indxRow) = k
        if (k/B .EQ. thisPE) then
          call csend(gtype+2*k+1,indxRow,intSize,
     &               allNode,npid)
        else
          call crecv(gtype+2*k+1,indxRow,intSize)
        endif
        indx(indxRow) = k

C       fac = a(:,k) / maxNum
        if (k/B .EQ. thisPE) then
          do i = 0, N-1
            fac(i) = aLoc(i,k-minCol) / maxNum
          end do
        end if
```

Figure 4: Hand-compiled Fortran77+MP code for Gaussian elimination.

```
C       row = a(indxRow,:)
        do j = 0, B-1
          rowLoc(j) = aLoc(indxRow,j)
        end do

C       forall(i=0:N-1, j=k:NN-1, indx(i) .EQ. -1)
C    &        a(i,j) = a(i,j) - fac(i) * row(j)
C     end do
        if (k/B .EQ. thisPE) then
          call csend(gtype+2*k,fac,realSize*N,
     &               allNode,npid)
        else
          call crecv(gtype+2*k,fac,realSize*N)
        endif
        lbound = MAX(0, k-minCol)
        do i = 0, N-1
          do j = lbound, B-1
            if (indx(i) .EQ. -1)
     &        aLoc(i,j) = aLoc(i,j)-fac(i)*rowLoc(j)
          end do
        end do
      end do

      integer function MaxLoc(x,n,mask)
      integer n
      real x(0:n-1)
      logical mask(0:n-1)
      real    t

      t = -MAXINT
      do i = 0, n-1
        if ((mask(i)) .AND. (t .LT. x(i))) then
          t = x(i)
          MaxLoc = i
        endif
      end do
      return
      end
```

Figure 4. Hand-compiled Fortran77+MP code for Gaussian elimination (cont.)

```
    tmp(i) = ABS(aLoc(i,k-minCol))
  end do
end if
```

where index k has been translated into $k - minCol$ by the local-to-global index conversion.

In the case of a distributed array, the operations are distributed to PEs. For example, the statement

```
row = a(indxRow,:)
```

is translated into

```
do j = 0, B-1
  rowLoc(j) = aLoc(indxRow,j)
end do
```

The following statement is to be duplicated:

```
indx(indxRow) = k
```

However, the value of *indxRow* is not available at every PE. Therefore, a pair of communication calls, *csend* and *crecv*, are inserted to broadcast *indxRow* to all the PEs as shown:

```
if (k/B .EQ. thisPE) then
  call csend(gtype+2*k+1,indxRow,intSize,
&            allNode,npid)
else
  call crecv(gtype+2*k+1,indxRow,intSize)
endif
indx(indxRow) = k
```

The *forall* statement

```
forall(i=0:N-1, j=k:NN-1, indx(i).EQ.-1)
&  a(i,j) = a(i,j) - fac(i) * row(j)
```

is to be translated into a nested loop. A pair of communication calls are inserted before the loop to broadcast *fac* as shown:

```
if (k/B .EQ. thisPE) then
  call csend(gtype+2*k,fac,realSize*N,
&            allNode,npid)
else
  call crecv(gtype+2*k,fac,realSize*N)
endif
lbound = MAX(0, k-minCol)
do i = 0, N-1
  do j = lbound, B-1
    if (indx(i) .EQ. -1) then
      aLoc(i,j)=aLoc(i,j)-fac(i)*rowLoc(j)
    endif
  end do
end do
```

where *lbound* is used to specify the active area, and the *mask* is translated into an *if* statement.

The code in Figure 4 has been translated directly from Fortran90D. We can optimize this code for better performance. The optimized code is shown in Figure 5. We have performed three kinds of optimizations:

1. Invariant expression floating

   The expressions that were executed many times have been floated. For example, we have floated $kLoc = k - minCol$. Also, an *if* statement has been pulled out of the inner loop.

2. Loop fusion

   We have put several loops and *if* statements together to reduce overhead.

```
      thisPE = mynode()
      numNode = numnodes()
      B  = NN / numNode
      minCol  = thisPE * B

      logical mask(0:N-1)
      real tmp(0:N-1)

C     integer, array(0:N-1) :: indx
C     integer, array(1) :: iTmp
C     real, array(0:N-1,0:NN-1) :: a
C     real, array(0:N-1) :: fac
C     real, array(0:NN-1) :: row
C     real :: maxNum

C$  DECOMPOSITION a(N,NN)
C$  ALIGN row(J) WITH a(0,J)
C$  DISTRIBUTE a(:,BLOCK)
      integer indx(0:N-1)
      real aLoc(0:N-1,0:B-1)
      real fac(0:N)
      real rowLoc(0:B-1)
      real maxNum

C     indx = -1
      do i = 0, N-1
        indx(i) = -1
      end do

C     do k = 0, N-1
C       iTmp = MAXLOC(ABS(a(:,k)), MASK=indx .EQ. -1)
C       indxRow = iTmp(1)
      do k = 0, N-1
        kLoc = k - minCol
        if (k/B .EQ. thisPE) then
          do i = 0, N-1
            mask(i) = indx(i) .EQ. -1
            tmp(i) = ABS(aLoc(i,kLoc))
          end do
          indxRow = MaxLoc(tmp, N, mask)

C         maxNum = a(indxRow,k)
          maxNum  = aLoc(indxRow,kLoc)

C         fac = a(:,k) / maxNum
          do i = 0, N-1
            fac(i) = aLoc(i,kLoc) / maxNum
          end do

C         indx(indxRow) = k
          fac(N) = REAL(indxRow)
          call csend(gtype+k,fac,realSize*(N+1),
     &               allNode,npid)
        else
          call crecv(gtype+k,fac,realSize*N)
          indxRow = INT(fac(N))
        endif
        indx(indxRow) = k
```

Figure 5: Optimized Fortran77+MP code for Gaussian elimination.

```
C         row = a(indxRow,:)
          do j = 0, B-1
            rowLoc(j) = aLoc(indxRow,j)
          end do

C         forall(i=0:N-1, j=k:NN-1, indx(i) .EQ. -1)
C     &         a(i,j) = a(i,j) - fac(i) * row(j)
C       end do
          lbound = MAX(0, kLoc)
          do i = 0, N-1
            if (indx(i) .EQ. -1) then
              do j = lbound, B-1
                aLoc(i,j)=aLoc(i,j)-fac(i)*rowLoc(j)
              end do
            end if
          end do
        end do

        integer function MaxLoc(x,n,mask)
        integer n
        real x(0:n-1)
        logical mask(0:n-1)
        real    t
        t = -MAXINT
        do i = 0, n-1
          if ((mask(i)) .AND. (t .LT. x(i))) then
            t = x(i)
            MaxLoc = i
          endif
        end do
        return
        end
```

Figure 5. Optimized Fortran77+MP code for Gaussian elimination (cont.)

3. Reordering

We have reordered statements without changing the results of the program. More loop and *if* statement fusions can be performed with reordering.

5 Experimental Results

We are building a test suite including a set of test programs. For each of the programs, we have the following versions:

- original Fortran77 code
- CMFortran code
- Fortran90D code
- hand-written Fortran77+MP code
- hand-written iPSC Fortran code
- hand-compiled Fortran77+MP code from Fortran90D code
- hand-compiled iPSC Fortran code from Fortran90D code

The Fortran77+MP codes were written in EXPRESS.

Here, we will describe three small test programs in our test suite: Gaussian elimination, FFT, and the N-body problem. Performance of iPSC Fortran codes (on the iPSC/2 hypercube) are shown in Tables 1, 2, and 3, respectively. The "Hand" programs are hand-written codes and the "Comp" programs are hand-compiled codes.

Table 1: Performance for Gaussian Elimination 255*256 (time in sec.)

	Number of PEs				
	1	2	4	8	16
Hand1	85.4	58.1	31.1	16.0	8.42
Hand2	73.4	50.1	26.9	13.8	7.53
Comp1	80.0	50.2	26.6	13.8	7.72

For the Gaussian elimination with partial pivoting shown in Table 1, the program has been block-partitioned in columns. Essentially, the Fortran90D code produced a code with performance equal to that of direct Fortran+MP code. Moreover, we found that "Comp1" had better performance than "Hand1." By comparing the two codes, we discovered that the difference was the index calculation. We optimized "Hand1" into "Hand2," changing the following code segment:

```
from
 do i = 0, N-1
   do j = start, numCol-1
     a(i,j) = a(i,j) - fac(i) * y(maxRow,j)
   end do
 end do
to:
 do j = 0, B-1
   row(j) = y(maxRow,j)
 end do
 do i = 0, N-1
   do j = start, numCol-1
     a(i,j) = a(i,j) - fac(i) * row(j)
   end do
 end do
```

This reduced the duplicated index calculation in the inner loop. Indeed, the "automatic" Fortran90D code revealed a possible improvement that we could apply to our hand-written code.

In Table 2, we used the FFT algorithm in [13] with modification. We applied vector communication and reduced repeated computation. There was

Table 2: Performance for FFT 16384 Points (time in sec.)

	Number of PEs				
	1	2	4	8	16
Hand1	13.0	6.67	3.42	1.75	0.91
Comp1	18.8	10.1	5.36	2.84	1.50

a 50% degradation in performance for the "Comp1" code, since it tested for possible communication patterns and involved larger overhead. For example, to test if there was communication for a shift operation in FFT, a compiler must generate 20-lines of code to test if there were any data in the shift range needed to be transferred to other PEs. On the other hand, the user knew they were not necessary. In the hand-written code, a line of code was used to test if the loop variable k was less than a given constant for determination of no communication.

Table 3: Performance for N-body 1024 Particles (time in sec.)

	Number of PEs				
	1	2	4	8	16
Hand1	71.7	35.9	17.9	8.98	4.83
Hand2	66.5	33.3	16.7	8.38	4.26
Comp1	139.6	69.1	35.5	18.1	9.40
Comp2	66.6	33.5	16.8	8.45	4.32

Table 3 is for the N-body problem using the algorithm in [13]. Note that the example is the simple $O(N^2)$ algorithm and not the more challenging $O(N(\log N))$ approach [14]. "Comp1" was not optimized, and communication was inserted in each iteration. "Comp2" grouped possible communications together. It reduced the number of communications and increased granularity. The performance of "Comp2" was better than "Hand1," since "Hand1" exchanged the order of array indices to avoid copying for communication. However, index calculation in this order consumed even more time than copying. Therefore, in "Hand2," we did not exchange the index order.

Our initial experiments are sufficiently encouraging. We believe that a language like Fortran90D will become an efficient vehicle for applications with regular structures. We also hope that it can be extended with higher level data structures to accommodate the more complex problem architectures.

6 Conclusion

Fortran90D is a language that can naturally represent the parallelism of many applications, especially those with static and regular array structures. This language can be extended to represent applications with irregular and dynamic structures.

Each stage of the Fortran90D project will be motivated and tested using carefully selected applications. The test suite is developed for testing our Fortran90D compiler. We will add more applications, including a number of sparse matrix problems, such as linear systems, linear programming, and irregular finite elements.

Acknowledgments

The authors thank Wei Shu for her contribution in building the test suite. The generous support of the Center for Research on Parallel Computation is gratefully acknowledged. This work was supported by the National Science Foundation under Cooperative Agreement No. CCR-8809165 – the Government has certain rights in this material.

References

[1] C. D. Polychronopoulos et al. Parafrase-2 : An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proc. Int'l Conf. on Parallel Processing*, pages II.39–48, August 1989.

[2] J.R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In *Supercomputers: Design and Applications*, pages 186–205. IEEE Computer Society Press, 1984.

[3] G.C. Fox, S. Hiranadani, K. Kennedy, C. Koelbel, U. Kremer, C.W. Tseng, and M.Y. Wu. Fortran D language specifications. Technical Report COMP TR90-141, Rice University, December 1990.

[4] D. A. Padua, D. J. Kuck, and D. L. Lawrie. High speed multiprocessor and compilation techniques. *IEEE Trans. Computers*, C–29(9):763–776, September 1980.

[5] D. Callahan and K. Kennedy. Compiling programs for distributed memory multiprocessors. *The Journal of Supercomputing*, 2(2):171–207, 1988.

[6] S. Hiranandani, K. Kennedy, and C.W. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, 1991.

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

[8] J.H. Merlin. ADAPTing Fortran 90 array programs for distributed memory architectures. In *Proc. of the 1st Int'l Conf. of the ACPC, Saltzberg, Austria*, October 1991.

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

[10] C. Koelbel. Compiling programs for nonshared memory machines. Technical Report CSD-TR-1037, Purdue University, November 1990.

[11] M. Y. Wu and D. D. Gajski. Computer-aided programming for message-passing systems: Problems and a solution. *IEEE Proceedings*, 77(12):1983–1991, December 1989.

[12] U. Banerjee. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, 2(2), 1988.

[13] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice-Hall, 1988.

[14] J. Barnes and P. Hut. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324:446, 1986.