Compiling Fortran90 Programs for Distributed Memory MIMD Parallel Computers

Min-You Wu and Geoffrey Fox

CRPC-TR91126
January, 1991

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

Compiling Fortran90 Programs for Distributed Memory MIMD Parallel Computers

Min-You Wu and Geoffrey Fox

Syracuse Center for Computational Science

Syracuse University

111 College Place

Syracuse, NY 13244-4100

Abstract — This paper describes the design and motivation for a Fortran90 compiler, a source-to-source parallelizing compiler, for distributed memory systems. We discuss the methodology of parallelizing Fortran programs and the principle of compiler design. Then we describe compiler directives, data partitioning and sequentialization, communication insertion, and implementation of intrinsic functions. Some basic optimization techniques are also presented. We use an introductory example of Gaussian elimination to explain the compiling techniques. Other sample programs in our test suite, such as FFT and the N-body problem, are briefly discussed with their performance.

1. Introduction

Current commercial parallel supercomputers are clearly the next generation of high performance machines [1, 2]. However, 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. Writing programs for parallel machines is a complicated, time-consuming, and error-prone task [3]. Karp and Babb [4, 5] selected a simple program and rewrote it to run on nine commercially available parallel machines. They report being surprised to see how complicated some of these programs had become.

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 that take a sequential Fortran77 program as input and produce a parallelized version for the target machine. Most notable examples include Parafrase at the University of Illinois [6] and PFC at Rice university [7]. In this approach, the compiler takes a sequential program, applies a set of transformation rules, and produces a parallelized code for the target machine. New transformation rules are added to the compiler as they are learned. This approach has been successful in vectorizing loops. However, it is not clear if this type of automatic parallelization will work in general, especially for large codes, for several reasons:

- It can be very difficult in some cases to detect available parallelism because it is obscured by the way a sequential program is written.
- It is hard to know and incorporate rules for all peculiarities of sequential programs.
- It takes a long time to develop sophisticated compilers that provide reasonably good performance.

A sequential language, such as Fortran77, presents parallel parts of a problem as sequential loops and other sequential constructs. Compiling a sequential program into a parallel program is not a natural approach; people write a program even the parallel parts of a program are written sequentially. Usually, programmers also optimize a program to reduce memory usage and computation time. This makes the potentially parallel parts of the program more difficult to detect by a parallelizing compiler. Parallelization of a sequential program is limited by extracted parallelism. An alternative approach is to use a programming language that can naturally represent an application without losing the application's original parallelism. Fortran90 (possibly with some extensions) is such a language. From our point of view, Fortran90 is not regarded as the natural portable language for SIMD computers [8, 9], but as a natural language for parallelism of a class of what we have called synchronous problems [10]. In Fortran90, parallelism is represented with parallel constructs, such as array operations, forall loops (not a standard construct in Fortran90), and intrinsic functions. We do not attempt to *parallelize* other constructs, such as do loops and while loops, since they are sequential in natural. It becomes much easier if we develop a parallelizing compiler that deals only with parallel constructs.

Different approaches to parallelizing Fortran programs are shown in Figure 1. In our collaboration with Rice, we propose to combine all three steps shown here. Parallelizing compilers can convert some Fortran77 programs directly into Fortran+MP. This may include applications that cannot easely be written in Fortran90, but may exclude programs where Fortran77 coding practices have obscured the parallelism. Fortran90 is portable between SIMD and MIMD, and 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. Further, the Fortran+MP code varies for each MIMD machine depending on granularity, communication performance, and other system characteristics.

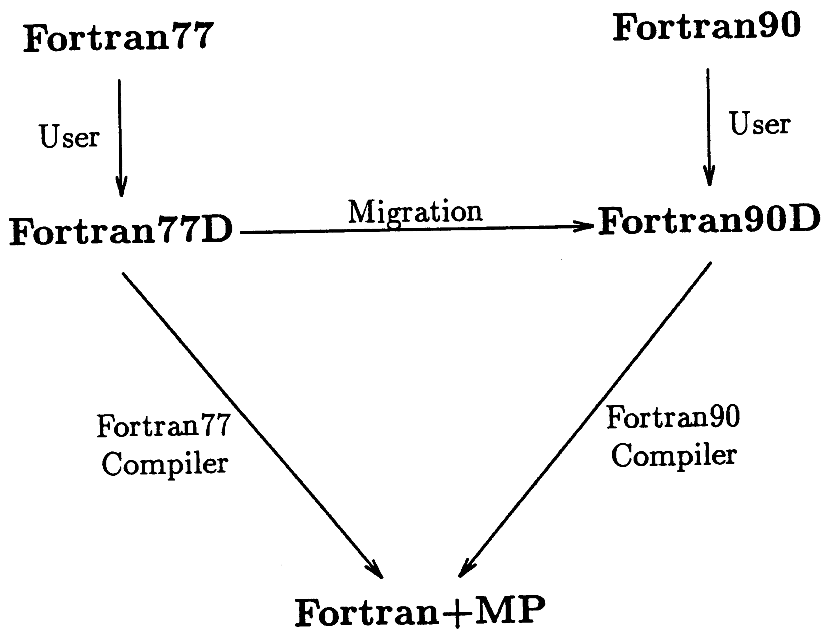

Figure 1: Approaches to parallelizing Fortran programs.

Intensive research has been done with shared memory systems [11, 12]. Compiling Fortran77 programs for distributed memory systems has been addressed by [13]. Sarkar compiled SISAL for multiprocessors with different partitioning and scheduling approaches [14]. 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 [15]. BLAZE is a high-level language designed for program portability. It can be compiled for different parallel systems [16, 17]. Koelbel extended the feature of BLAZE into a Kali language and compiled it for nonshared memory machines [18]. Crystal is another high-level language based on mathematical notations and lambda calculus. A Crystal compiler generates C code for hypercube multicomputers [19, 20]. There has been some work on data parallel program development by Hatcher and Quinn. 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 [21, 22]. Our approach uses many techniques that are similar to this work on C*.

In this paper, we will discuss the essential issues of a Fortran90 compiler. The Fortran90 compiler is a source-to-source parallelizing compiler, which compiles a Fortran90 program into a Fortran+MP program. The system diagram is described in Section 2. Data partioning, computation assignment, and sequentialization are discussed in Sections 3 and 4. In Section 4, we also discuss techniques to handle active areas. Different methods for data communication are discussed in Section 5. We focus largely on the source-side decision problem, since in a distributed memory system, the decision in the source side is more difficult than the decision in the destination side. In section 6, we present methodology to translate the intrinsic functions in Fortran90 into library routines in Fortran+MP. Some optimization techniques are given in Section 7. We use Gaussian elimination as an introductory example to illustrate the application of these compiling techniques in Section 8, and in Section 9, we present some experimental results.

2. Compiler System Diagram

The system diagram of the Fortran90 compiler is shown in Figure 2. Given a syntactically correct Fortran90 program, the first step of compilation is to parse the program and generate a parse tree. The partitioning module partitions data into tasks and allocates the tasks to processor elements (PEs) according to compiler directives — partitioning directives and alignment directives. There are three ways to generate these 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 partition programs with partitioning and alignment directives. A programming tool can generate useful analysis to help users decide partitioning styles, and give information to help users in improving their program partitioning interactively [23]. The directives can also be generated automatically by compilers. There has been promising work along these lines [24, 25]. However, these ideas have not yet been implemented in a practical general system, so we do not consider automatic partitioning in this paper.

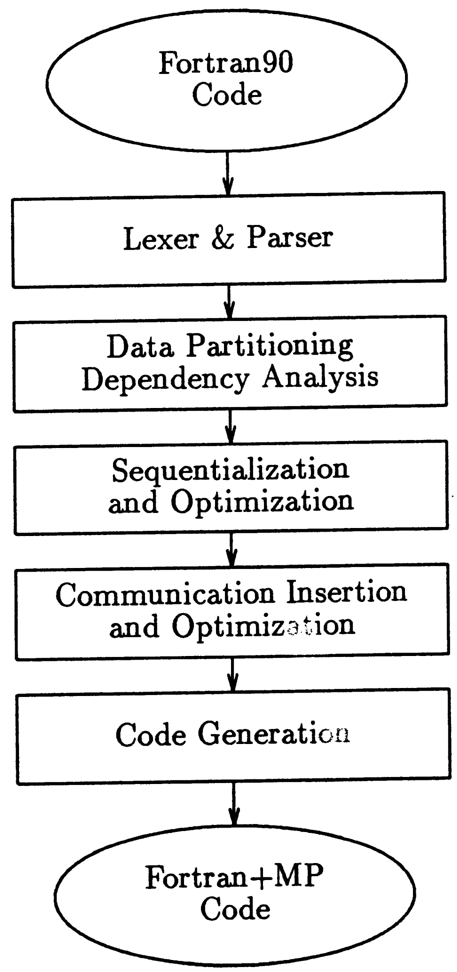

Figure 2: Diagram of the compiler.

Dependency analysis is carried out to obtain dependency information among partitions. This information will be used for insertion of communication primitives. Standard techniques of data dependency analysis for Fortran programs can be applied here [26, 27].

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. Parallel constructs in the original program will be transferred into loops or nested loops. This module also performs optimization, such as extracting the common expression out of loops, integrating condition statements into loop boundaries, and reordering statements.

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. We need to apply optimizations to minimize synchronization, eliminate unnecessary or redundant data transfers, and to combine communication where possible. One important optimization is overlapping computation and communication to overcome communication latency. Analysis and 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 written to translate certain parallel constructs, such as reduction, broadcasting, etc. Finally, the code generator produces the Fortran+MP code for target message-passing systems.

3. Data Partitioning and Index Conversion

We provide users with some annotation facilities for data partitioning. The annotation takes the form of compiler directives, including partitioning directives and alignment directives. We term Fortran90 with these compiler directives as Fortran90D. This language has been developed in collaboration with our colleagues at Rice [28]. There is an analogous version of Fortran77 with user directives, namely Fortran77D.

A partitioning directive provides some control over the partitioning of an array with specification of block partitioning, scatter partitioning, block-scatter partitioning, or no partitioning. The relative partitioning weight along each axis indicates the partitioning ratio among axes.

A *partitioning-directive* is:

CDISTRIBUTE *partitioning-spec*-list

A *partitioning-spec* is:

array-*name* (*axis-descriptor*-list)

An *axis-descriptor* is one of:

- BLOCK[(*weight*)]
- CYCLIC[(*weight*)]
- BLOCK_CYCLIC(*size* [,*weight*])
- [NOP]

A *weight* is:

scalar-integer-*constant*

A *size* is:

scalar-integer-*constant*

The number of *axis-descriptors* in a *partitioning-spec* must equal the rank of the array specified by array-*name* in the *partitioning-spec*. Note that an *axis-descriptor* may be empty, but the commas separating each *axis-descriptor* must be present.

Each *partitioning-spec* specifies partitioning information for the array given by array-

name. The array is partitioned with the attributes specified by the *axis-descriptor*-list of that *partitioning-spec*. Each *axis-descriptor* defines the attributes of the corresponding dimension that is to be partitioned. The keywords BLOCK, CYCLIC, BLOCK_CYCLIC, and NOP control the partitioning style. For each *axis-descriptor* in the list:

- BLOCK indicates that the corresponding dimension is to be block-partitioned (contiguous).
- CYCLIC indicates that the corresponding dimension is to be scatter-partitioned (interleaving).
- BLOCK_CYCLIC(*size*) indicates that the corresponding dimension is to be block-scatter-partitioned; that is, blocks of size *size* are scattered.
- NOP indicates that the corresponding dimension will not be partitioned.

The *weight* specifies the partitioning weight for an axis. As an example, if the ratio of weights for two axes is 4, the partitioning ratio of the corresponding dimensions is roughly 4. When 64 PEs are used to run the program, the first dimension is partitioned into 16, and the second into 4. If 32 PEs are used, the first dimension is partitioned into 8, and the second into 4. The default weight is 1.

An alignment directive aligns an array to another array. 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.

An *alignment-directive* is:

CALIGN *source-spec* WITH *target-spec*

A *source-spec* is:

source-array-*name* (index-*name*-list)

A *target-spec* is:

target-array-*name* (*target-axis-spec*-list)

A *target-axis-spec* is one of:

- index-*name*
- index-*name* + *offset-value*
- *index-value*

An *offset-value* is:

integer-constant

An *index-value* is:

[–] *integer-constant*

The number of *index-names* in a *source-spec* must equal the rank of the array source-array-*name*. The number of *target-axis-specs* in a *target-spec* must equal the rank of the array target-array-*name*. Note that a given index-*name* must not be referenced by more than one *target-axis-spec*.

With alignment directives, arrays aligned to a partitioned target array simply follow the partitioning patterns of the target array. If the alignment directives appear ahead of the partitioning directive, the compound array (by the alignment directives) will be partitioned by the partitioning directive. For example, the following alignment directives align arrays *b* and *c* to array *a*:

CALIGN b(i) WITH a(1,i)

CALIGN c(i) WITH a(i+4,1)

and the following partitioning directive partitions the compound array of *a*, *b*, and *c*:

CDISTRIBUTE a(,BLOCK).

Note that array *b* and the first dimension of array *a* are block-partitioned, but array *c* is not partitioned. The combination of partitioning and alignment directives can specify various data partitioning patterns.

According to partitioning 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.

Consider a one-dimensional array $a(0 : N-1)$, which is partitioned into P tasks of equal size. The size of a task is:

$$B = N/P$$

where N is the array size and $MOD(N,P) = 0$. Array a can be block-partitioned by

CDISTRIBUTE a(BLOCK),

scatter-partitioned by

CDISTRIBUTE a(CYCLIC),

or block-scatter-partitioned by

CDISTRIBUTE a(BLOCK_CYCLIC($size$)).

Data distribution and index conversion (global-to-local and local-to-global) for different partitioning are shown in Table 1. In the row of "data distribution," the array sections in task k are listed. In the row of "location of data," the ID of the task that holds the data $a(ginx)$ is given. The next two rows list the rules of index conversion. The method for one-dimensional partitioning can be generalized to multiple dimensions.

Table 1: Data Distribution and Index Conversion

	Block-partitioning	Scatter-partitioning	Block-scatter-partitioning
data distribution in task k	$a(minLoc : maxLoc)$ where, $minLoc = k * B$ $maxLoc = minLoc+B-1$	$a(k : N-1 : P)$	$a((k+i*P)*size :$ $(k+i*P)*size+size-1)$, $i = 0, N/size-1$
location of data $a(ginx)$	$k = ginx/B$	$k = MOD(ginx, P)$	$k = MOD(ginx/size, P)$
global to local index conversion	$linx = ginx - minLoc$	$linx = ginx/P$	$linx = MOD(ginx, size)$ $+ginx/(size*P)*size$
local to global index conversion	$ginx = linx + minLoc$	$ginx = linx * P + k$	$ginx = MOD(linx, size)$ $+(linx/size*P+k)*size$

Currently, we do not encourage complicated partitioning patterns, since their index calculation may lead to large overhead.

4. Computation Assignment and Sequentialization

Computation is assigned to each PE based on data partitioning. There are two different principles to assign computation: *majority principle* and *owner principle.* In the majority principle, a computation is assigned to the PE that holds the most data. Here, we count both data read and write. In the owner principle, a computation is assigned to the PE on which the data to be written reside. The former minimizes data communication but requires more analysis, possibly resulting in more overhead. The latter is simple to implement and optimal in most cases, so currently we apply this owner principle.

There are two frequently used parallel constructs in Fortran90, array operations and forall loops. We apply the owner principle to these constructs.

Block-partitioning

For the array operation

```
a = b + d(c:N-1+c)
```

PE k is assigned the computation

```
a(minLoc:maxLoc) = b(minLoc:maxLoc) + d(minLoc+c:maxLoc+c)
```

It is sequentialized into

```
do i = minLoc, maxLoc
  a(i) = b(i) + d(i+c)
```

and with the global-to-local index conversion, we have

```
do i = 0, B-1
  aLoc(i) = bLoc(i) + dLoc(i+c)
```

For the forall loop

```
forall (i=0:N-1)
  a(i) = b(i+c) + func(i)
```

PE k is assigned the computation

```
forall (i=minLoc:maxLoc)
  a(i) = b(i+c) + func(i)
```

where $func$ is a function. It is sequentialized into

```
do i = minLoc, maxLoc
  a(i) = b(i+c) + func(i)
```

and with the global-to-local index conversion, we have

```
do i = 0, B-1
  aLoc(i) = bLoc(i+c) + func(i+minLoc)
```

Scatter-partitioning

For the array operation

```
a = b + d(c:N-1+c)
```

PE k is assigned the computation

```
a(k:N-1:P) = b(k:N-1:P) + d(k+c:N-1+c:P)
```

It is sequentialized into

```
do i = k, N-1, P
  a(i) = b(i) + d(i+c)
```

and with the global-to-local index conversion, we have

```
do i = 0, B-1
  aLoc(i) = bLoc(i) + dLoc(i+c)
```

For the forall loop

```
forall (i=0:N-1)
  a(i) = b(i+c) + func(i)
```

PE k is assigned the computation

```
forall (i=k:N-1:P)
  a(i) = b(i+c) + func(i)
```

It is sequentialized into

```
do i = k, N-1, P
  a(i) = b(i+c) + func(i)
```

and with the global-to-local index conversion, we have

```
do i = 0, B-1
  aLoc(i) = bLoc(i+c) + func(i*P+k)
```

Block-scatter-partitioning

For the array operation

```
a = b + d(c:N-1+c)
```

PE k is assigned the computation

```
forall (i = k*size:N-1:P*size)
  a(i:i+size-1) = b(i:i+size-1) + d(i+c:i+size-1+c)
```

It is sequentialized into

```
do i = k*size, N-1, P*size
  do j = i, i+size-1
    a(j) = b(j) + d(j+c)
```

and with the global-to-local index conversion, we have

```
do i = 0, B-1
  aLoc(i) = bLoc(i) + dLoc(i+c)
```

For the forall loop

```
forall (i=0:N-1)
  a(i) = b(i+c) + func(i)
```

PE k is assigned the computation

```
forall (i = k*size:N-1:P*size)
  forall (j = i, i+size-1)
    a(j) = b(j+c) + func(j)
```

It is sequentialized into

```
do i = k*size, N-1, P*size
  do j = i, i+size-1
    a(j) = b(j+c) + func(j)
```

and with the global-to-local index conversion, we have

```
do i = 0, B-1
  aLoc(i) = bLoc(i+c) + func(MOD(i,size)+(i/size*P+k)*size)
```

We have assumed so far that all data items in an array are active. In many cases, not every array element is active. Here we only consider block partitioning for simplicity, but the technique can be applied to other partitioning styles.

There are different ways to specify the active area. The first is to use array operations with *where* statements or *forall* loops with masks. They can be translated directly into *if* statements. In many cases, the *if* statements can then be transferred into loop boundaries to reduce overhead.

The *where* statement:

```
where (mask)
  a(0:N-1) = b(1:N)
elsewhere
  a = d
```

can be translated into:

```
do i = 0, B-1
  if (mask(i)) then
    aLoc(i) = bLoc(i+1)
  else
    aLoc(i) = dLoc(i)
```

The *forall* loop statement:

```
forall (i=0:N-1, mask(i))
  a(i) = b(i) + func(i)
```

can be translated into:

```
do i = 0, B-1
  if (mask(i))
    aLoc(i) = bLoc(i) + func(i+minLoc)
```

Array sections or loop boundaries can be also used to specify the active area. The following statement

```
a(c:N-1) = b(0:N-1-c) + d(0:N-1-c)
```

can be translated into:

```
do i = 0, B-1
  if (i+minLoc .GE. c)
    aLoc(i) = bLoc(i-c) + dLoc(i-c)
```

and the *if* statement can be transferred into loop boundary:

```
lbound= MAX(0, c-minLoc)
do i = lbound, B-1
  aLoc(i) = bLoc(i-c) + dLoc(i-c)
```

The following *forall* loop

```
forall (i = l:u)
  a(i) = b(i-c) + d(i-c)
```

can be translated into:

```
lbound= MAX(0, l-minLoc)
ubound= MIN(B-1, u-minLoc)
do i =lbound, ubound
  aLoc(i) = bLoc(i-c) + dLoc(i-c)
```

The active area can also be specified by a linear combination of indices, such as the following statement:

```
forall (i=l1:u1, j=l2:u2)
  a(i+j*r+c) = func(i,j)
```

where, $r > u1 - l1$, otherwise a will be multiple-assigned; *func* is a function.

It can be transferred into:

```
forall (s=l1+l2*r:u1+u2*r)
  if ((MOD(s,r) .GE l1) .AND. (MOD(s,r) .LE. u1))
    a(s+c) = func(MOD(s,r), s/r)
```

and

```
forall (s=l1+l2*r+c:u1+u2*r+c)
  if ((MOD(s-c,r) .GE. l1) .AND. (MOD(s-c,r) .LE. u1))
    a(s) = func(MOD(s-c,r), (s-c)/r)
```

When $l1 = l2 = 0$, it is simplified as:

```
forall (s=c:u1+u2*r+c)
  if (MOD(s-c,r) .LE. u1)
    a(s) = func(MOD(s-c,r), (s-c)/r)
```

It can be sequentialized into:

```
lbound= MAX(0, c-minLoc)
ubound= MIN(B-1, u1+u2*r+c-minLoc)
do s =lbound, ubound
  if (MOD(s+minLoc-c,r) .LE. u1)
    aLoc(s) = func(MOD(s+minLoc-c,r), (s+minLoc-c)/r)
```

5. Communication Insertion

After a program is partitioned into tasks, communication primitives must be inserted when dependencies exist between tasks. Generally, whenever data requested by a statement are not local, a *receive* is inserted before the statement, and a *send* is inserted at the source PE that holds the data. The *send* is usually inserted after the statement that generates the data. Compared to the *receive* insertion, the *send* insertion is more difficult, since it may not know which data element needs to be sent. Information of *send* may be:

- compile-time available;
- runtime available; or
- runtime not available.

When information of *send* is compile-time available, the destination of a data item can be calculated for the *send* primitive. The following is an example:

```
forall (i=0:N-1)
  a(i) = a(i*2-1)
```

A data item $a(j)$ (j is an odd number) will be sent to the location of $(j+1)/2$ in the range of 0 to $N-1$. In the following statement

```
forall (i=0:N-1)
  a(i) = a(b(i))
```

when b is a replicated array, the information of *send* is available at runtime, since each PE has a copy of array b. If b is a distributed array, information of *send* is not available at runtime, because a PE holds only a part of array b and is not able to know where the local data a should be sent.

Data communication can be done in two ways:

1. by sending local data to the PEs that need them, then receiving the data. This is called *send-receive* (SR) communication.

2. by requesting data from a PE, sending data upon the requests, then receiving the data. This is called *request-send-receive* (RSR) communication.

The SR communication requires compile-time or runtime information for dependency analysis. It cannot be used if information is not available at compile-time or runtime. The RSR communication is more general, but it involves longer communication latency. Moreover, pooling or interruption techniques must be implemented to receive requests. A *broadcast* communication can be used for substitution of the RSR communication. That is, whenever a data item could be requested by other PEs, it is simply broadcast to all PEs. However, this method is not scalable and causes heavy network traffic.

If all PEs request the same data, a *broadcast* is inserted in the source PE and a *receive* in each PE except the source PE. A *broadcast* can also be implemented with EXPRESS routine *KXBROD* [29] or Crystal_router [30]. An example of broadcast pattern is shown in the following statement:

```
forall (i=0:N-1)
  a(i) = a(0)
```

When different data items are broadcast to different groups of PEs, it is called *multicasting.* For example, the following statement has a multicast pattern:

```
forall (i=0:N-1)
  a(i) = a(i/c)
```

where c is a constant. A data item $a(j)$ is sent to locations $j * c + k, k = 0, 1, 2, ..., c - 1$, and $j * c + k$ is in the range of 0 to $N - 1$. Another example is shown in the following statement:

```
forall (i=0:N-1, j=0:N)
  a(i,j) = a(0,j)
```

The multicast is along the first dimension. This kind of multicasting is also called *spread.*

Data transferring from one PE to another PE will be packed together at the source PE and unpacked at the destination PE. Packing and unpacking reduces the number of communications, but introduces extra overhead.

In the following, we show how to insert communication primitives by using an example of simple shift communication.

Shift communication $(i - c)$ or $(i + c)$

In this simple shift pattern, a destination PE may receive messages from, at most, two source PEs, and accordingly, a source PE may send messages to, at most, two destination

PEs. In the source PE, we need to find out destination PEs and whether data are requested there. In the destination PE, we check to determine if there are any data requested by other PEs, and if so, the source PEs are determined. If data are in the same PE, there is no message to be transferred. If two source PEs or two destination PEs are the same, only one message is transferred.

The source and destination PEs are calculated as follows:

For $(i - c)$:

```
Send:
    desPE1 = (minLoc+c)/B            desPE2 = (maxLoc+c)/B
    The index range:                 The index range:
    lb1 = MOD(c, B)                  lb2 = 0
    ub2 = B-1                        ub2 = MOD(c-1, B)
    Test if desPE1 is thisPE and if so, do not send message.
    Test if desPE2 is thisPE and if desPE2 is desPE1, if so, do not
      send message.
    Test whether any data request is in the range on desPE and if so, send
      the array block.

Receive:
    srcPE1 = (minLoc-c)/B            srcPE2 = (maxLoc-c)/B
    The index range:                 The index range:
    lb1 = 0                          lb2 = MOD(c, B)
    ub2 = MOD(c-1, B)                ub2 = B-1
    Test if desPE1 is thisPE and if so, do not receive message.
    Test if desPE2 is thisPE and if desPE2 is desPE1, if so, do not
      receive message.
    Test whether any data request is in the ranges on thisPE and if so,
      receive the corresponding array blocks.
```

For $(i + c)$:

```
Send:
    desPE1 = (minLoc-c)/B            desPE2 = (maxLoc-c)/B
    The index range:                 The index range:
    lb1 = MOD(N-c, B)                lb2 = 0
    ub2 = B-1                        ub2 = MOD(N-c-1, B)

Receive:
    srcPE1 = (minLoc+c)/B            srcPE2 = (maxLoc+c)/B
    The index range:                 The index range:
    lb1 = 0                          lb2 = MOD(N-c, B)
    ub2 = MOD(N-c-1, B)              ub2 = B-1

    The sending and receiving tests are the same as above.
```

The data items to be sent to the other PE are packed into an array. When an array *a* in task *i* is transferred to array *tmp* in task *j*, the reference to *a(linx)* becomes the reference to *tmp(linx+offset)*, where $offset = (j - i) * B$.

6. Intrinsic Functions

In Fortran90, there are many intrinsic functions. The intrinsic functions that may cause communication can be divided into five categories as shown in Table 2.

We will build a subroutine library to translate the corresponding functions. Each intrinsic function may be compiled into different subroutines for different partition styles. On general principles, intrinsic functions can be implemented with the Crystal_router or Crystal_accumulator [30]. However, some of the intrinsic functions can be directly mapped into EXPRESS routines [29]. These are the commercially supported versions of software originally developed at Caltech [30]. Express runs on a network of UNIX workstations, as well as on multicomputers, such as those from INTEL and NCUBE.

Table 2: Fortran90 Intrinsic Functions

		1. Sending & receiving	2. Reduction	3. Multicasting	4. Irregular operations	5. Special routines
Fortran90		CSHIFT EOSHIFT DIAGONAL PROJECT	DOTPRODUCT ALL, ANY COUNT MAXVAL, MINVAL PRODUCT SUM FIRSTLOC, LASTLOC MAXLOC, MINLOC	SPREAD REPLICATE	PACK UNPACK RESHAPE TRANSPOSE	MATMUL
EXPRESS/ CrOS III	Specific	KXCHAN KXREAD KXWRIT	KXCOMB	KXBROD	KXREAD KXWRIT	
	General	Crystal_router	Crystal_accumulator	Crystal_router	Crystal_router	

For intrinsic functions in the first category, data are transferred with *send* and *receive* primitives. However, several data items can be packed together to reduce the number of communications. *CSHIFT* and *EOSHIFT* can be implemented with the EXPRESS routine *KXCHAN*, and *DIAGONAL* and *PROJECT* with the EXPRESS routines *KXWRIT* and *KXREAD*. *DIAGONAL* can also be implemented with the subroutine *fold* [30]. In the second category, data are processed with a reduction tree [31]. These intrinsic functions can be implemented with the EXPRESS routine *KXCOMB* or the subroutine *combine* [30]. They can also be implemented with Crystal_accumulator. The third category uses multiple broadcast trees to spread data. They can be implemented with the EXPRESS routine *KXBROD* or Crystal_router. The fourth category is difficult to implement due to its irregular operations. We must discover the individual data elements to be transferred, and pack the data that will be sent to the same PE. These intrinsic functions can be implemented with the EXPRESS routines *KXWRIT* and *KXREAD*, but a more efficient implementation can be obtained with Crystal_router. The fifth category will be implemented using existing research on parallel matrix algorithms.

Although it will be nontrivial to implement these intrinsics, it is critical for our project that the essential primitives have already been developed for MIMD machines.

7. Optimization

Performance can be improved with optimization. A knowledge-based approach can be used for optimization. The knowledge base can be built in an incremental fashion. First, optimization rules for most frequently used structures are collected. As more rules are added to the knowledge base, more different codes can be optimized.

Optimization for the computation components include:

- applying an STM (single assignment to multiple assignment) transformation, which condenses some arrays to reduce memory usage, and to avoid unnecessary copying;
- extracting common expressions and conditions out of loops;
- integrating some conditions into loop boundaries; and
- reordering statements and combining loops to increase granularity between communications.

Next is an example of extracting common expressions out of loops. Two expressions *minLoc-incrm* and *incrm-offset* can be pulled out of the loop.

Before common expression extraction:

```
do s = lbl, ubl
  if (MOD(s+minLoc-incrm,incrm2) .LT. incrm) then
    if ((s-incrm) .GE. 0) then
      x(s) = x(s-incrm) - term2(s)
    else
      x(s) = xbuf(s-incrm+offset) - term2(s)
    endif
  endif
end do
```

After common expression extraction:

```
tmp1 = minLoc-incrm
tmp2 = incrm-offset
do s = lb1, ub1
  if (MOD(s+tmp1,incrm2) .LT. incrm) then
    if ((s-incrm) .GE. 0) then
      x(s) = x(s-incrm) - term2(s)
    else
      x(s) = xbuf(s-tmp2) - term2(s)
    endif
  endif
end do
```

There are two extreme approaches for communication, the *accurate approach* and the *broadcast approach*. The accurate approach does sophisticated analysis and transfers only data needed by other PEs. The broadcast approach does little analysis and simply broadcasts data to all PEs, whether or not they are useful. A realistic approach falls in between. It devotes a reasonable amount of effort for analysis and reduces most unnecessary communication. The analysis and optimizations may include:

- eliminating unnecessary communications;
- identifying the PEs that really need the data and sending the data to them instead of broadcasting;
- identifying the data request and sending only the segments of data instead of the whole array; and
- combining communications that can be sent at the same time.

We use a simple example in the Gaussian elimination program for communication combination:

```
call csend(gtype+2*k+1,indxRow,intSize,allNode,npid)
call csend(gtype+2*k,fac,realSize*N,allNode,npid)
```

These two communication calls can be combined after packing two messages. Note that integer "indxRow" must be converted into a real number before packing and converted back into an integer after unpacking at the destination PE. The variable "fac" will be declared as an array with size of "$N+1$" instead of "N" to pack "indxRow".

```
fac(N) = REAL(indxRow)
call csend(gtype+k,fac,realSize*(N+1),allNode,npid)
```

8. An Introductory Example: Gaussian Elimination

We use Gaussian elimination as an example for translating a Fortran90 program into a Fortran+MP program. The Fortran90 code is shown in Figure 3, and the hand-compiled Fortran+MP code is shown in Figure 4. The hand-compiled code implements rules stated above. Note that the size of the Fortran90 code is much smaller than that of the Fortran+MP code. The former has 19 lines, and the latter has 68 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* 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 Fortran90 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
```

```
      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

CDISTRIBUTE a(,BLOCK)
CDISTRIBUTE row(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: Fortran90 code for Gaussian elimination.

```
        thisNode = mynode()
        numNode = numnodes()
        B  = NN/numNode
        minCol  = thisNode*B
        maxCol  = minCol + B - 1
        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
CDISTRIBUTE a(,BLOCK)
CDISTRIBUTE row(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-minLoc))
            end do
            indxRow = MaxLoc(tmp, N, mask)
          end if
C         maxNum = a(indxRow,k)
          if (k/B .EQ. thisPE) maxNum  = aLoc(indxRow,k-minLoc)
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
```

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

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

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-minLoc)
          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.)

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

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

When it is 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.

```
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* loop

```
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*:

```
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-minLoc)
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 the Fortran90 code. We can optimize this code for better performance. The optimized code is shown in Figure 5. We have performed three kinds of optimizations:

1. Common expression extraction

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

```
      thisNode = mynode()
      numNode = numnodes()
      B  = NN/numNode
      minCol  = thisNode*B
      maxCol  = minCol + B - 1

      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
CDISTRIBUTE a(,BLOCK)
CDISTRIBUTE row(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 - minLoc
        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)
```

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

```
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
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.)

2. Loop fusion

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

3. Reordering

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

9. 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
- sequential Fortran77 code modified with a parallel algorithm, if necessary
- Fortran90 (CMFortran) code
- hand-written Fortran77+MP code (initially run on iPSC/2)
- hand-compiled Fortran77+MP code from Fortran90 code (iPSC/2)

Now, we have three test programs in our test suite: Gaussian elimination, FFT, and the N-body problem. Performance on iPSC/2 is shown in Tables 3, 4, and 5, respectively. The "Hand" programs are hand-written codes and the "Comp" programs are hand-compiled codes.

For the Gaussian elimination with partial pivoting shown in Table 3, the program has been block-partitioned in columns. Essentially, the Fortran90 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

Table 3: 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
Compl	80.0	50.2	26.6	13.8	7.72

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" Fortran90 code revealed a possible improvement that we could apply to our hand-written code.

In Table 4, we used the FFT algorithm in [30] with modification. We applied vector communication and reduced repeated computation. There was a 50% degradation in per-

Table 4: 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

formance for the "Comp1" code, since it tested for possible communication patterns and involved larger overhead. These tests were eliminated in the hand-written code, since the user knew they were not necessary.

Table 5: 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 5 is for the N-body problem using the algorithm in [30]. Note that the example is the simple $O(N^2)$ algorithm and not the more challenging $O(N(\log N))$ approach [32]. "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.

Finally, in Table 6, we come to a "real", although small in term of code size, problem. The original climate modeling code has been used in production on CRAY and SUN com-

Table 6: Climate Modeling Code [33]

Implementation	Size (lines)	Machine	Performance (megaflops)
Original C Code	1500	CRAY X-MP (1 CPU)	~ 1
Fortran90	600	CM-2	66
Fortran77 by Hand from Fortran90	1500	CRAY Y-MP (1 CPU)	20
Fortran+MP by Hand from Fortran90	1650	NCUBE-1 (16 nodes)	3.3
		NCUBE-2 (16 nodes)	20
		INTEL i860 (16 nodes)	80

puters [34]. This project contained an interesting division of labor. An application expert first rewrote the C code in Fortran90. Computer scientists without in-depth knowledge of the application performed further conversions into Fortran77 and Fortran+MP [33]. In this case, we believed that no automatic method could have parallelized the original C code, but that our planned automatic approach would be able to perform the MIMD parallelization from Fortran90. This project result in a portable code running well on the CRAY, the Connection Machine, and hypercubes. Note that we even improved the sequential performance (line 1 vs. line 3 of Table 6) by an order of magnitude. The original C code made extensive use of pointers, which had several repercussions. It made vectorization difficult on the CRAY; it made the code impossible to parallelize automatically as the "structure of problem" had been expressed in dynamic pointer values; and it made the code difficult to port except by the application expert.

Our initial experiments are sufficiently encouraging. We believe that a language like Fortran90 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.

10. Conclusion

Fortran90 is a language that can naturally represent the parallelism of many applications, especially that with static and regular array structures. This language can be extended to represent applications with irregular and dynamic structures.

Fortran90 can be compiled for both SIMD and MIMD parallel machines. It unifies the programming environments of different parallel computers. We have discussed the essential issues of building a Fortran90 compiler for distributed memory parallel computers. With this compiler, a program written in Fortran90 can be compiled into fairly efficient target codes for many regular applications.

Acknowledgments

The authors thank Wei Shu for her contribution in building the test suite, and Diane Purser and Betty Laplante for their editorial efforts. 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] G.C. Fox. Parallel computing comes of age: Supercomputer level parallel computations at Caltech. *Concurrency: Practice and Experience*, 1(1):63–103, September 1989.

[2] Steven W. Otto, Adam Kolawa, and Anthony Hey. Performance of the Mark II Caltech/JPL hypercube. Technical Report C3P-188, California Institute of Technology, August 1985.

[3] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[4] A. H. Karp. Programming for parallelism. *IEEE Computer*, pages 43–57.

[5] A. H. Karp and R. G. Babb II. A comparison of 12 parallel fortran dialects. *IEEE Software*, pages 52–67, September 1988.

[6] 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.

[7] 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.

[8] L. W. Tucker and G. G. Robertson. Architecture and applications of the Connection Machine. *IEEE Computer*, pages 26–38, August 1988.

[9] Clive Baillie, Ed Felten, and David Walker. Benchmarking the Connection Machine. Technical Report C3P-443, California Institute of Technology, July 1987.

[10] G.C. Fox. What have we learnt from using real parallel machines to solve real problems? In G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 897–955. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988.

[11] 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.

[12] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. Cedar Fortran and its compiler. Technical Report CSRD Report No. 966, University of Illinois at Urbana-Champaign, January 1990.

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

[14] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* The MIT Press, 1989.

[15] 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.

[16] C. Koelbel, P. Mehrotra, and J.V. Rosendale. Semi-automatic process partitioning for parallel computation. *International Journal of Parallel Programming*, 16(5):365–382, 1987.

[17] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

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

[19] M. C. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 2(2):171–208, October 1988.

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

[21] M.J. Quinn. Compiling SIMD programs for MIMD architectures. In *Proc. of International Conference on Computer Languages*, March 1990.

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

[23] 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.

[24] G. Fox and W. Furmanski. Load balancing by a neural network. Technical Report C3P-363, California Institute of Technology, 1986.

[25] J. Saltz, R. Mirchandaney, R. Smith, D. Nicol, and K. Crowley. The PARTY parallel runtime system. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 1987. held in Los Angeles, CA.

[26] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graph and compiler optimizations. In *Proc. of 8th ACM Symp. Principles on Programming Lang.*, January 1981.

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

[28] 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.

[29] ParaSoft Corp. Express Fortran reference guide. Technical Report Version 3.0, 1990.

[30] 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.

[31] C.T. Ho. *Optimal Communication Primitives and Graph Embeddings.* PhD thesis, Yale Universite, May 1989.

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

[33] G. L. Keppenne, M. Ghil, G. C. Fox, J. W. Flower, A. Kowala, P. N. Papaccio, J. J. Rosati, J. F. Shepanski, F. G. Spasaro, and J. O. Dickey. Parallel processing applied to climate modeling. Technical Report SCCS-22, Syracuse University, November 1990.

[34] G. L. Keppenne. *Bifurcations, Strange Attractors and Low-frequency Atmospheric Dumamics.* PhD thesis, Universite Catholique de Louvain, 1989.