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 easily 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.
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 partitioning, 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.

![Diagram of the compiler]

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 di-
rectives. 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:

\texttt{CDISTRIBUTE partitioning-spec-list}

A partitioning-spec is:

\texttt{array-name ( axis-descriptor-list )}

An axis-descriptor is one of:

- \texttt{BLOCK([weight])}
- \texttt{CYCLIC([weight])}
- \texttt{BLOCK_CYCLIC(size [,weight])}
- \texttt{[NOP]}

A weight is:

\texttt{scalar-integer-constant}

A size is:

\texttt{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:

\[
B = \frac{N}{P}
\]

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

\text{CDISTRIBUTE} \ a(\text{BLOCK}),

scatter-partitioned by

\text{CDISTRIBUTE} \ a(\text{CYCLIC}),

or block-scatter-partitioned by

\text{CDISTRIBUTE} \ a(\text{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 \( a(ginz) \) 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>
<thead>
<tr>
<th>Data distribution in task ( k )</th>
<th>Block-partitioning</th>
<th>Scatter-partitioning</th>
<th>Block-scatter-partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a(\text{minLoc} : \text{maxLoc}) ) where, ( \text{minLoc} = k \times B ) ( \text{maxLoc} = \text{minLoc} + B - 1 )</td>
<td>( a(k : N - 1 : P) )</td>
<td>( a((k + i \times P) \times \text{size} : (k + i \times P) \times \text{size} + \text{size} - 1), i = 0, N/\text{size} - 1 )</td>
<td></td>
</tr>
<tr>
<td>Location of data ( a(ginz) )</td>
<td>( k = \text{ginz}/B )</td>
<td>( k = \text{MOD}(\text{ginz}, P) )</td>
<td>( k = \text{MOD}(\text{ginz}/\text{size}, P) )</td>
</tr>
<tr>
<td>Global to local index conversion</td>
<td>( \text{linz} = \text{ginz} - \text{minLoc} )</td>
<td>( \text{linz} = \text{ginz}/P )</td>
<td>( \text{linz} = \text{MOD}(\text{ginz}, \text{size}) + \text{ginz}/(\text{size} \times P) \times \text{size} )</td>
</tr>
<tr>
<td>Local to global index conversion</td>
<td>( \text{ginz} = \text{linz} + \text{minLoc} )</td>
<td>( \text{ginz} = \text{linz} \times P + k )</td>
<td>( \text{ginz} = \text{MOD}(\text{linz}, \text{size}) + (\text{linz}/\text{size} \times P + k) \times \text{size} )</td>
</tr>
</tbody>
</table>
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
\[
a(\text{minLoc}:\text{maxLoc}) = b(\text{minLoc}:\text{maxLoc}) + d(\text{minLoc+c}:\text{maxLoc+c})
\]

It is sequentialized into
\[
\text{do } i = \text{minLoc, maxLoc} \\
\quad a(i) = b(i) + d(i+c)
\]

and with the global-to-local index conversion, we have
\[
\text{do } i = 0, B-1 \\
\quad a\text{Loc}(i) = b\text{Loc}(i) + d\text{Loc}(i+c)
\]

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
\[
\text{do } i = k, N-1, P \\
\quad a(i) = b(i+c) + \text{func}(i)
\]
and with the global-to-local index conversion, we have
\[
\text{do } i = 0, B-1 \\
\quad a\text{Loc}(i) = b\text{Loc}(i+c) + \text{func}(i\cdot P+k)
\]

**Block-scatter-partitioning**

For the array operation
\[
a = b + d(c:N-1+c)
\]
PE \( k \) is assigned the computation
\[
\text{forall } (i = k\cdot \text{size}:N-1:P\cdot \text{size}) \\
\quad a(i:i+\text{size}-1) = b(i:i+\text{size}-1) + d(i+c:i+\text{size}-1+c)
\]
It is sequentialized into
\[
\text{do } i = k\cdot \text{size}, N-1, P\cdot \text{size} \\
\quad \text{do } j = i, i+\text{size}-1 \\
\quad \quad a(j) = b(j) + d(j+c)
\]
and with the global-to-local index conversion, we have
\[
\text{do } i = 0, B-1 \\
\quad a\text{Loc}(i) = b\text{Loc}(i) + d\text{Loc}(i+c)
\]

For the forall loop
\[
\text{forall } (i=0:N-1) \\
\quad a(i) = b(i+c) + \text{func}(i)
\]
PE \( k \) is assigned the computation
\[
\text{forall } (i = k\cdot \text{size}:N-1:P\cdot \text{size}) \\
\quad \text{forall } (j = i, i+\text{size}-1) \\
\quad \quad a(j) = b(j+c) + \text{func}(j)
\]
It is sequentialized into
\[
do \ j = i, i + \text{size}-1 \\
\quad a(j) = b(j+c) + \text{func}(j)
\]

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

\[
\do i = 0, \text{B}-1 \\
\quad \text{aLoc}(i) = \text{bLoc}(i+c) + \text{func}(\text{MOD}(i, \text{size})+(i/\text{size}*\text{P}+\text{k})*\text{size})
\]

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 \textit{where} statements or \textit{forall} loops with masks. They can be translated directly into \textit{if} statements. In many cases, the \textit{if} statements can then be transferred into loop boundaries to reduce overhead.

The \textit{where} statement:

\[
\begin{align*}
\text{where} & \ (\text{mask}) \\
& \ a(0: \text{N}-1) = b(1: \text{N}) \\
\text{elsewhere} & \\
& \ a = d
\end{align*}
\]

can be translated into:

\[
\begin{align*}
\do i = 0, \text{B}-1 \\
& \ \text{if} \ (\text{mask}(i)) \ \text{then} \\
& \quad \text{aLoc}(i) = \text{bLoc}(i+1) \\
& \ \text{else} \\
& \quad \text{aLoc}(i) = \text{dLoc}(i)
\end{align*}
\]

The \textit{forall} loop statement:

\[
\forall i=0: \text{N}-1, \ \text{mask}(i) \\
a(i) = b(i) + \text{func}(i)
\]

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:
\[
\begin{align*}
do i = 0, B-1 \\
    &\text{if (i+minLoc .GE. c)} \\
    &aLoc(i) = bLoc(i-c) + dLoc(i-c)
\end{align*}
\]
and the if statement can be transferred into loop boundary:
\[
\begin{align*}
\text{lbound} &= \text{MAX}(0, c-\text{minLoc}) \\
do i = \text{lbound}, B-1 \\
    aLoc(i) &= bLoc(i-c) + dLoc(i-c)
\end{align*}
\]

The following forall loop
\[
\begin{align*}
\text{forall (i = 1:u)} \\
a(i) &= b(i-c) + d(i-c)
\end{align*}
\]
can be translated into:
\[
\begin{align*}
\text{lbound} &= \text{MAX}(0, 1-\text{minLoc}) \\
\text{ubound} &= \text{MIN}(B-1, u-\text{minLoc}) \\
do i = \text{lbound}, \text{ubound} \\
    aLoc(i) &= bLoc(i-c) + dLoc(i-c)
\end{align*}
\]

The active area can also be specified by a linear combination of indices, such as the following statement:
\[
\begin{align*}
\text{forall (i=11:u1, j=12:u2)} \\
a(i+j*r+c) &= \text{func}(i,j)
\end{align*}
\]
where, \( r > u1 - l1 \), otherwise \( a \) will be multiple-assigned; \( \text{func} \) is a function.

It can be transferred into:
forall (s=l1+12*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+12*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:

\[
\text{forall } (i=0: i\cdot2-1) \\
\quad a(i) = a(i\cdot2-1)
\]

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

\[
\text{forall } (i=0: N-1) \\
\quad a(i) = a(b(i))
\]

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:

\[
\text{forall } (i=0:N-1) \\
\quad a(i) = a(0)
\]

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

\[
\text{forall } (i=0:N-1) \\
\quad a(i) = a(i/c)
\]

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

\[
\text{forall } (i=0:N-1, j=0:N) \\
\quad a(i,j) = a(0,j)
\]

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:

$$\text{desPE}_1 = \frac{\text{minLoc} + c}{B}$$  
$$\text{desPE}_2 = \frac{\text{maxLoc} + c}{B}$$

The index range:

$$\text{lb}_1 = \text{MOD}(c, B)$$

$$\text{ub}_2 = B - 1$$

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:

$$\text{srcPE}_1 = \frac{\text{minLoc} - c}{B}$$  
$$\text{srcPE}_2 = \frac{\text{maxLoc} - c}{B}$$

The index range:

$$\text{lb}_1 = 0$$

$$\text{lb}_2 = \text{MOD}(c, B)$$

$$\text{ub}_2 = \text{MOD}(c - 1, B)$$

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:
\[
\begin{align*}
\text{desPE1} &= (\min\text{Loc}-c)/B \\
\text{The index range:} &= \text{desPE2} = (\max\text{Loc}-c)/B \\
\text{lb1} &= \text{MOD}(N-c, B) \\
\text{ub2} &= B-1
\end{align*}
\]

Receive:
\[
\begin{align*}
\text{srcPE1} &= (\min\text{Loc}+c)/B \\
\text{The index range:} &= \text{srcPE2} = (\max\text{Loc}+c)/B \\
\text{lb1} &= 0 \\
\text{ub2} &= \text{MOD}(N-c-1, B)
\end{align*}
\]

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 \(a(\text{linx})\) becomes the reference to \(tmp(\text{linx}+\text{offset})\), where \(\text{offset} = (j - i) \times B\).

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran90</td>
<td>CSHIFT EOSHIFT</td>
<td>DOTPRODUCT ALL, ANY</td>
<td>SPREAD REPLICATE</td>
<td>PACK UNPACK RESHAPE</td>
<td>MATMUL</td>
</tr>
<tr>
<td></td>
<td>DIAGONAL PROJECT</td>
<td>COUNT MAXVAL, MINVAL</td>
<td></td>
<td>TRANSPOSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SUM FIRSTLOC, LASTLOC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPRESS/ CrOS III Specific</td>
<td>KXCHAN KXREAD KXWRIT</td>
<td>KXCOMB</td>
<td>KXBROD</td>
<td>KXREAD KXWRIT</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td>Crystal_router</td>
<td>Crystal_accumulator</td>
<td>Crystal_router</td>
<td>Crystal_router</td>
<td></td>
</tr>
</tbody>
</table>

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 \texttt{minLoc-incrm} and \texttt{incrm-offset} can be pulled out of the loop.

Before common expression extraction:

```fortran
    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

\[
tmp2 = \text{incrm}\text{-offset}
\]

do s = lb1, ub1
    if (MOD(s+tmp1,incrm) .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 \textit{accurate approach} and the \textit{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:

\[
call \text{csend}(\text{gtype}+2*k+1,\text{indRow},\text{intSize},\text{allNode},\text{npid})
\]
\[
call \text{csend}(\text{gtype}+2*k,\text{fac},\text{realSize}+N,\text{allNode},\text{npid})
\]
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”.

\[
\begin{align*}
\text{fac}(N) &= \text{REAL}(\text{indxRow}) \\
\text{call csend(gtype+k,fac,realSize*(N+1),allNode,npid)}
\end{align*}
\]

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 \textit{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

\[
\text{indx} = -1
\]

is translated into

\[
\begin{align*}
\text{do } i &= 0, N-1 \\
\text{indx}(i) &= -1 \\
\text{end do}
\end{align*}
\]
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

indx = -1
do i = 0, N-1
   indx(i) = -1
end do
do k = 0, N-1
   iTmp = MAXLOC(ABS(a(:,k)), MASK = indx .EQ. -1)
   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
   maxNum = a(indxRow,k)
   if (k/B .EQ. thisPE) maxNum = aLoc(indxRow,k-minLoc)
   indx(indxRow) = k
   if (k/B .EQ. thisPE) then
      call csend(ctype+2*k+1,indxRow,intValue,allNode,npid)
   else
      call crecv(ctype+2*k+1,indxRow,intValue)
   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) &  
a(i,j) = a(i,j) - fac(i) * row(j)  
end do  

C  
if (k/B .EQ. thisPE) then  
  call csend(gtype+2*k,fac,realSize*N,allNode,nnpid)  
else  
  call crecv(gtype+2*k,fac,realSize*N)  
endif  

lb = MAX(0, k-minLoc)  
do i = 0, N-1  
do j = lb, B-1  
  if (indx(i) .EQ. -1) aLoc(i,j) = aLoc(i,j)-fac(i)*rowLoc(j)  
end do  
end do  

integer function MaxLoc(x,n,mask)  
exter n  
exteral x(0:n-1)  
exteral logical mask(0:n-1)  
exteral 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

\[ \text{tmp} = \text{ABS}(a(:,k)) \]

is translated into

\[
\begin{align*}
\text{if } (k/B \ .\ EQ. \ thisPE) \text{ then} \\
\text{do } i = 0, N-1 \\
\quad \text{tmp}(i) = \text{ABS}(aLoc(i,k-minLoc)) \\
\text{end do} \\
\text{end if}
\end{align*}
\]

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

\[ \text{row} = a(\text{indxRow},:) \]

is translated into

\[
\begin{align*}
\text{do } j = 0, B-1 \\
\quad \text{rowLoc}(j) = aLoc(\text{indxRow},j) \\
\text{end do}
\end{align*}
\]

The following statement is to be duplicated:

\[ \text{indx}(\text{indxRow}) = k \]

However, the value of \( \text{indxRow} \) is not available at every PE. Therefore, a pair of communication calls, \text{csend} and \text{crecv}, are inserted to broadcast \( \text{indxRow} \) to all the PEs.

\[
\begin{align*}
\text{if } (k/B \ .\ EQ. \ thisPE) \text{ then} \\
\quad \text{call csend(gtype+2*k+1,indxRow,intSize,allNode,npid)} \\
\text{else} \\
\quad \text{call crecv(gtype+2*k+1,indxRow,intSize)} \\
\text{endif}
\end{align*}
\]

\[ \text{indx}(\text{indxRow}) = k \]
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
lbounbd = MAX(0, k-minLoc)
do i = 0, N-1
  do j = lbounbd, B-1
    if (indx(i) .EQ. -1) then
      aLoc(i,j) = aLoc(i,j) - fac(i) * rowLoc(j)
    endif
  end do
end do
```

where, *lbounbd* 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.

30
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
do k = 0, N-1
  iTmp = MAXLOC(ABS(a(:,k)), MASK = indx .EQ. -1)
  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)
  end if
  maxNum = a(indxRow,k)
  maxNum = aLoc(indxRow,kLoc)

Figure 5: Optimized Fortran77+MP code for Gaussian elimination.
fac = a(:,k) / maxNum
    do i = 0, N-1
        fac(i) = aLoc(i,kLoc) / maxNum
    end do

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))
end if

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

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

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

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
        end if
    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.)

<table>
<thead>
<tr>
<th></th>
<th>Number of PEs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Hand1</td>
<td>85.4</td>
</tr>
<tr>
<td>Hand2</td>
<td>73.4</td>
</tr>
<tr>
<td>Comp1</td>
<td>80.0</td>
</tr>
</tbody>
</table>

the difference was the index calculation. We optimized “Hand1” into “Hand2”, changing the following code segment:

From

\[
\begin{align*}
    & \text{do } \ i = 0, \ N-1 \\
    & \quad \text{do } \ j = \text{start, numCol-1} \\
    & \quad \quad a(i,j) = a(i,j) - \text{fac}(i) \times y(\text{maxRow},j) \\
    & \quad \text{end do} \\
    & \text{end do}
\end{align*}
\]

to:

\[
\begin{align*}
    & \text{do } \ j = 0, \ B-1 \\
    & \quad \text{row}(j) = y(\text{maxRow},j) \\
    & \text{end do} \\
    & \text{do } \ i = 0, \ N-1 \\
    & \quad \text{do } \ j = \text{start, numCol-1} \\
    & \quad \quad a(i,j) = a(i,j) - \text{fac}(i) \times \text{row}(j) \\
    & \quad \text{end do} \\
    & \text{end do}
\end{align*}
\]

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.)

<table>
<thead>
<tr>
<th></th>
<th>Number of PEs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 4 8 16</td>
</tr>
<tr>
<td>Hand1</td>
<td>13.0 6.67 3.42 1.75 0.91</td>
</tr>
<tr>
<td>Comp1</td>
<td>18.8 10.1 5.36 2.84 1.50</td>
</tr>
</tbody>
</table>

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.)

<table>
<thead>
<tr>
<th></th>
<th>Number of PEs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 4 8 16</td>
</tr>
<tr>
<td>Hand1</td>
<td>71.7 35.9 17.9 8.98 4.83</td>
</tr>
<tr>
<td>Hand2</td>
<td>66.5 33.3 16.7 8.38 4.26</td>
</tr>
<tr>
<td>Comp1</td>
<td>139.6 69.1 35.5 18.1 9.40</td>
</tr>
<tr>
<td>Comp2</td>
<td>66.6 33.5 16.8 8.45 4.32</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Implementation</th>
<th>Size (lines)</th>
<th>Machine</th>
<th>Performance (megaflops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original C Code</td>
<td>1500</td>
<td>CRAY X-MP (1 CPU)</td>
<td>~ 1</td>
</tr>
<tr>
<td>Fortran90</td>
<td>600</td>
<td>CM-2</td>
<td>66</td>
</tr>
<tr>
<td>Fortran77 by Hand from Fortran90</td>
<td>1500</td>
<td>CRAY Y-MP (1 CPU)</td>
<td>20</td>
</tr>
<tr>
<td>Fortran+MP by Hand from Fortran90</td>
<td>1650</td>
<td>NCUBE-1 (16 nodes)</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NCUBE-2 (16 nodes)</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEL i5860 (16 nodes)</td>
<td>80</td>
</tr>
</tbody>
</table>

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 resulted in a portable code running well on the CRAY, the Connection Machine, and hypercubes. Notice 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


