Compiling High Performance Fortran for Distributed-memory Systems

Jonathan Harris
John A. Bircsak
M. Regina Bolduc
Jill Ann Diewald
Israel Gale
Neil W. Johnson
Shin Lee
C. Alexander Nelson
Carl D. Offner

Digital's DEC Fortran 90 compiler implements most of High Performance Fortran version 1.1, a language for writing parallel programs. The compiler generates code for distributed-memory machines consisting of interconnected workstations or servers powered by Digital's Alpha microprocessors. The DEC Fortran 90 compiler efficiently implements the features of Fortran 90 and HPF that support parallelism. HPF programs compiled with Digital's compiler yield performance that scales linearly or even superlinearly on significant applications on both distributed-memory and shared-memory architectures.

High Performance Fortran (HPF) is a new programming language for writing parallel programs. It is based on the Fortran 90 language, with extensions that enable the programmer to specify how array operations can be divided among multiple processors for increased performance. In HPF, the program specifies only the pattern in which the data is divided among the processors; the compiler automates the low-level details of synchronization and communication of data between processors.

Digital's DEC Fortran 90 compiler is the first implementation of the full HPF version 1.1 language (except for transcriptive argument passing, dynamic remapping, and nested FORALL and WHERE constructs). The compiler was designed for a distributed-memory machine made up of a cluster (or farm) of workstations and/or servers powered by Digital's Alpha microprocessors.

In a distributed-memory machine, communication between processors must be kept to an absolute minimum, because communication across the network is enormously more time-consuming than any operation done locally. Digital's DEC Fortran 90 compiler includes a number of optimizations to minimize the cost of communication between processors.

This paper briefly reviews the features of Fortran 90 and HPF that support parallelism, describes how the compiler implements these features efficiently, and concludes with some recent performance results showing that HPF programs compiled with Digital's compiler yield performance that scales linearly or even superlinearly on significant applications on both distributed-memory and shared-memory architectures.

Historical Background

The desire to write parallel programs dates back to the 1950s, at least, and probably earlier. The mathematician John von Neumann, credited with the invention of the basic architecture of today's serial computers, also invented cellular automata, the precursor of today's massively parallel machines. The continuing motivation for parallelism is provided by the need to solve computationally intense problems in a reasonable time and at an affordable price. Today's parallel machines,

which range from collections of workstations connected by standard fiber-optic networks to tightly coupled CPUs with custom high-speed interconnection networks, are cheaper than single-processor systems with equivalent performance. In many cases, equivalent single-processor systems do not exist and could not be constructed with existing technology.

Historically, one of the difficulties with parallel machines has been writing parallel programs. The work of parallelizing a program was far from the original science being explored; it required programmers to keep track of a great deal of information unrelated to the actual computations; and it was done using ad hoc methods that were not portable to other machines.

The experience gained from this work, however, led to a consensus on a better way to write portable Fortran programs that would perform well on a variety of parallel machines. The High Performance Fortran Forum, an international consortium of more than 100 commercial parallel machine users, academics, and computer vendors, captured and refined these ideas, producing the language now known as High Performance Fortran.[1–3] HPF programming systems are now being developed by most vendors of parallel machines and software. HPF is included as part of the DEC Fortran 90 language.[4]

One obvious and reasonable question is: Why invent a new language rather than have compilers automatically generate parallel code? The answer is straightforward: it is generally conceded that automatic parallelization technology is not yet sufficiently advanced. Although parallelization for particular architectures (e.g., vector machines and shared-memory multiprocessors) has been successful, it is not fully automatic but requires substantial assistance from the programmer to obtain good performance. That assistance usually comes in the form of hints to the compiler and rewritten sections of code that are more parallelizable. These hints, and in some cases the rewritten code, are not usually portable to other architectures or compilers. Agreement was widespread at the HPF Forum that a set of hints could be standardized and done in a portable way. Automatic parallelization technology is an active field of research; consequently, it is expected that compilers will become increasingly adept.[5–12] Thus, these hints are cast as comments—called *compiler directives*—in the source code. HPF actually contains very little new language beyond this; it consists primarily of these compiler directives.

The HPF language was shaped by certain key considerations in parallel programming:

- The need to identify computations that can be done in parallel
- The need to minimize communication between processors on machines with nonuniform memory access costs
- The need to keep processors as busy as possible by balancing the computation load across processors

It is not always obvious which computations in a Fortran program are parallelizable. Although some DO loops express parallelizable computations, other DO loops express computations in which later iterations of the loop require the results of earlier iterations. This forces the computation to be done in order (serially), rather than simultaneously (in parallel). Also, whether or not a computation is parallelizable sometimes depends on user data that may vary from run to run of the program. Accordingly, HPF contains a new statement (FORALL) for describing parallel computations, and a new directive (INDEPENDENT) to identify additional parallel computations to the compiler. These features are equally useful for distributed- or shared-memory machines.

HPF's data distribution directives are particularly important for distributed-memory machines. The HPF directives were designed primarily to increase performance on "computers with nonuniform memory access costs."[1] Of all parallel architectures, distributed memory is the architecture in which the location of data has the greatest effect on access cost. On distributed-memory machines, interprocessor communication is very expensive compared to the cost of fetching local data, typically by several orders of magnitude. Thus the effect of suboptimal distribution of data across processors can be catastrophic. HPF directives tell the compiler how to distribute data across processors; based on knowledge of the algorithm, programmers choose directives that will minimize communication time. These directives can also help achieve good load balance: by spreading data appropriately across processors, the computations on those data will also be spread across processors.

Finally, a number of idioms that are important in parallel programming either are awkward to express in Fortran or are greatly dependent on machine architecture for their efficient implementation. To be useful in a portable language, these idioms must be easy to express and implement efficiently. HPF has captured some of these idioms as library routines for efficient implementation on very different architectures.

For example, consider the Fortran 77 program in Figure 1, which repeatedly replaces each element of a two-dimensional array with the average of its north, south, east, and west neighbors. This kind of computation arises in a number of programs, including iterative solvers for partial differential equations and image-filtering applications. Figure 2 shows how this code can be expressed in HPF.

On a machine with four processors, a single HPF directive causes the array *A* to be distributed across the processors as shown in Figure 3. The program

```
      integer n, number_of_iterations, i,j,k
      parameter(n=16)
      real A(n,n), Temp(n,n)
     ... (Initialize A, number_of_iterations) ...
      do k=1, number_of_iterations
C
        Update non-edge elements only
        do i=2, n-1
          do j=2, n-1
             Temp(i, j)=(A(i, j-1)+A(i, j+1)+A(i+1, j)+A(i-1, j))*0.25
          enddo
        enddo
        do i=2, n-1
          do j=2, n-1
            A(i, j)=Temp(i,j)
          enddo
        enddo
      enddo
```

Figure 1
A Computation Expressed in Fortran 77

```
        integer n, number_of_iterations, i, j, k
        parameter (n=16)
        real A(n, n)
!hpf$   distribute A(block, block)
       ...(Initialize A, number_of_iterations) ...
        do k=1, number_of_iterations
          forall (i=2:n-1, j=2:n-1) !Update non-edge elements only
             A(i, j)=(A(i, j-1)+A(i, j+1)+A(i+1, j)+A(i-1, j))*0.25
          endforall
        enddo
```

Figure 2
The Same Computation Expressed in HPF

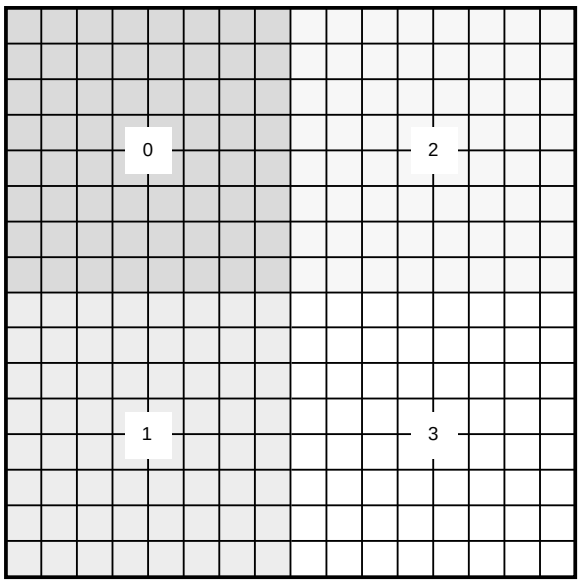


Figure 3
An Array Distributed over Four Processors

executes in parallel on the four processors, with each processor performing the updates to the array elements it owns. This update, however, requires interprocessor communication (or "data motion"). To compute a new value for $A(8, 2)$, which lives on processor 0, the value of $A(9, 2)$, which lives on processor 1, is needed. In fact, processor 0 requires the seven values $A(9, 2)$, $A(9, 3)$, … $A(9, 8)$ from processor 1, and the seven values $A(2, 9)$, $A(3, 9)$, … $A(8, 9)$ from processor 2.[13] Each processor, then, needs seven values apiece from two neighbors. By knowing the layout of the data and the computation being performed, the compiler can automatically generate the interprocessor communication instructions needed to execute the code.

Even for seemingly simple cases, the communication instructions can be complex. Figure 4 shows the communication instructions that are generated for the code that implements the FORALL statement for a distributed-memory parallel processor.

Processor 0	Processor 1	Processor 2	Processor 3
SEND $A(8, 2)\ldots A(8, 8)$ to Processor 1	SEND $A(9, 2)\ldots A(9, 8)$ to Processor 0	SEND $A(2, 9)\ldots A(8, 9)$ to Processor 0	SEND $A(9, 9)\ldots A(15, 9)$ to Processor 1
SEND $A(2, 8)\ldots A(8, 8)$ to Processor 2	SEND $A(9, 8)\ldots A(15, 8)$ to Processor 3	SEND $A(8, 9)\ldots A(8, 15)$ to Processor 3	SEND $A(9, 9)\ldots A(9, 9)$ to Processor 2
RECEIVE $A(9, 2)\ldots A(9, 8)$ from Processor 1	RECEIVE $A(8, 2)\ldots A(8, 8)$ from Processor 0	RECEIVE $A(2, 8)\ldots A(8, 8)$ from Processor 0	RECEIVE $A(9, 8)\ldots A(15, 8)$ from Processor 1
RECEIVE $A(2, 9)\ldots A(8, 9)$ from Processor 2	RECEIVE $A(9, 9)\ldots A(15, 9)$ from Processor 3	RECEIVE $A(9, 9)\ldots A(9, 15)$ from Processor 3	RECEIVE $A(8, 9)\ldots A(8, 15)$ from Processor 2

Figure 4
Compiler-generated Communication for a FORALL Statement

Although the communication needed in this simple example is not difficult to figure out by hand, keeping track of the communication needed for higher-dimensional arrays, distributed onto more processors, with more complicated computations, can be a very difficult, bug-prone task. In addition, a number of the optimizations that can be performed would be extremely tedious to figure out by hand. Nevertheless, distributed-memory parallel processors are programmed almost exclusively today by writing programs that contain explicit hand-generated calls to the SEND and RECEIVE communication routines. The difference between this kind of programming and programming in HPF is comparable to the difference between assembly language programming and high-level language programming.

This paper continues with an overview of the HPF language, a discussion of the machine architecture targeted by the compiler, the architecture of the compiler itself, and a discussion of some optimizations performed by its components. It concludes with recent performance results, showing that HPF programs compiled with Digital's compiler scale linearly in significant cases.

Overview of the High Performance Fortran Language

High Performance Fortran consists of a small set of extensions to Fortran 90. It is a data-parallel programming language, meaning that parallelism is made possible by the explicit distribution of large arrays of data across processors, as opposed to a control-parallel language, in which threads of computation are distributed. Like the standard Fortran 77, Fortran 90, and C models, the HPF programming model contains a single thread of control; the language itself has no notion of process or thread.

Conceptually, the program executes on all the processors simultaneously. Since each processor contains only a subset of the distributed data, occasionally a processor may need to access data stored in the memory of another processor. The compiler determines the actual details of the interprocessor communication needed to support this access; that is, rather than being specified explicitly, the details are implicit in the program.

The compiler translates HPF programs into low-level code that contains explicit calls to SEND and RECEIVE message-passing routines. All addresses in this translated code are modified so that they refer to data local to a processor. As part of this translation, addressing expressions and loop bounds become expressions involving the processor number on which the code is executing. Thus, the compiler needs to generate only one program: the generated code is parametrized by the processor number and so can be executed on all processors with appropriate results on each processor. This generated code is called explicit single-program multiple-data code, or explicit-SPMD code.

In some cases, the programmer may find it useful to write explicit-SPMD code at the source code level. To accommodate this, the HPF language includes an escape hatch called EXTRINSIC procedures that is used to leave data-parallel mode and enter explicit-SPMD mode.

We now describe some of the HPF language extensions used to manage parallel data.

Distributing Data over Processors

Data is distributed over processors by the DISTRIBUTE directive, the ALIGN directive, or the default distribution.

The DISTRIBUTE Directive For parallel execution of array operations, each array must be divided in memory, with each processor storing some portion of the array in its own local memory. Dividing the array into parts is known as distributing the array. The HPF DISTRIBUTE directive controls the distribution of arrays across each processor's local memory. It does this by specifying a mapping pattern of data objects onto processors. Many mappings are possible; we illustrate only a few.

Consider first the case of a 16×16 array A in an environment with four processors. One possible specification for A is

```
        real A(16, 16)
!hpf$   distribute A(*, block)
```

The asterisk (*) for the first dimension of A means that the array elements are not distributed along the first (vertical) axis. In other words, the elements in any given column are not divided among different processors, but are assigned as a single block to one processor. This type of mapping is referred to as serial distribution. Figure 5 illustrates this distribution.

The BLOCK keyword for the second dimension means that for any given row, the array elements are distributed over each processor in large blocks. The blocks are of approximately equal size—in this case, they are exactly equal—with each processor holding one block. As a result, A is broken into four contiguous groups of columns, with each group assigned to a separate processor.

Another possibility is a (*, CYCLIC) distribution. As in (*, BLOCK), all the elements in each column are assigned to one processor. The elements in any given row, however, are dealt out to the processors in round-robin order, like playing cards dealt out to players around a table. When elements are distributed over n processors, each processor contains every nth column, starting from a different offset. Figure 6 shows the same array and processor arrangement, distributed CYCLIC instead of BLOCK.

As these examples indicate, the distributions of the separate dimensions are independent.

A (BLOCK, BLOCK) distribution, as in Figure 3, divides the array into large rectangles. In that figure, the array elements in any given column or any given row are divided into two large blocks: Processor 0 gets A(1:8, 1:8), processor 1 gets A(9:16, 1:8), processor 2 gets A(1:8, 9:16), and processor 3 gets A(9:16,9:16).

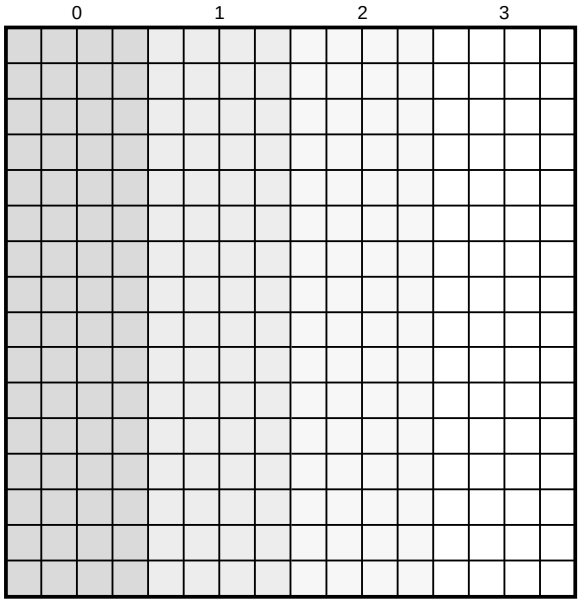


Figure 5
A (*, BLOCK) Distribution

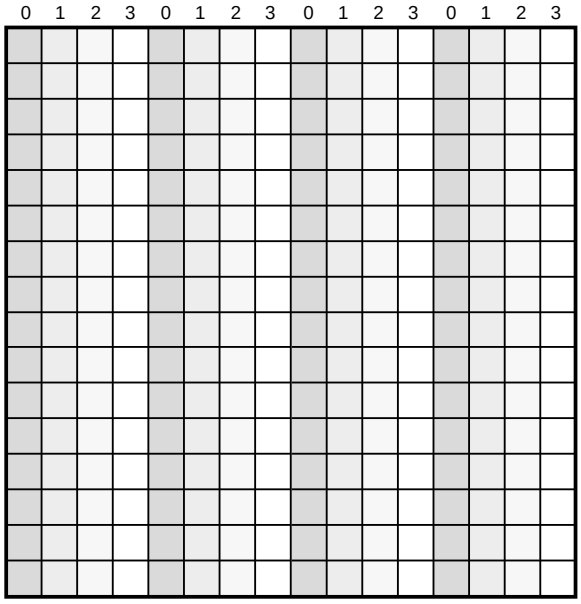


Figure 6
A (*, CYCLIC) Distribution

The ALIGN Directive The ALIGN directive is used to specify the mapping of arrays relative to one another. Corresponding elements in aligned arrays are always mapped to the same processor; array operations between aligned arrays are in most cases more efficient than array operations between arrays that are not known to be aligned.

The most common use of ALIGN is to specify that the corresponding elements of two or more arrays be mapped identically, as in the following example:

```
!hpf$   align A(i) with B(i)
```

This example specifies that the two arrays *A* and *B* are always mapped the same way. More complex alignments can also be specified. For example:

```
!hpf$   align E(i) with F(2*i-1)
```

In this example, the elements of *E* are aligned with the odd elements of *F*. In this case, *E* can have at most half as many elements as *F*.

An array can be aligned with the interior of a larger array:

```
        real A(12, 12)
        real B(16, 16)
!hpf$   align A(i, j) with B(i+2, j+2)
```

In this example, the 12×12 array *A* is aligned with the interior of the 16×16 array *B* (see Figure 7). Each interior element of *B* is always stored on the same processor as the corresponding element of *A*.

The Default Distribution Variables that are not explicitly distributed or aligned are given a default distribution by the compiler. The default distribution is not specified by the language: different compilers can choose different default distributions, usually based on constraints of the target architecture. In the DEC Fortran 90 language, an array or scalar with the default distribution is completely replicated. This decision was made because the large arrays in the program are the significant ones that the programmer has to distribute explicitly to get good performance. Any other arrays or scalars will be small and generally will benefit from being replicated since their values will then be available everywhere. Of course, the programmer retains complete control and can specify a different distribution for these arrays.

Replicated data is cheap to read but generally expensive to write. Programmers typically use replicated data for information that is computed infrequently but used often.

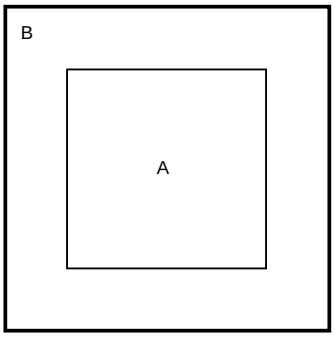


Figure 7
An Example of Array Alignment

Data Mapping and Procedure Calls

The distribution of arrays across processors introduces a new complication for procedure calls: the interface between the procedure and the calling program must take into account not only the type and size of the relevant objects but also their mapping across processors. The HPF language includes special forms of the ALIGN and DISTRIBUTE directives for procedure interfaces. These allow the program to specify whether array arguments can be handled by the procedure as they are currently distributed, or whether (and how) they need to be redistributed across the processors.

Expressing Parallel Computations

Parallel computations in HPF can be identified in four ways:

- Fortran 90 array assignments
- FORALL statements
- The INDEPENDENT directive, applied to DO loops and FORALL statements
- Fortran 90 and HPF intrinsics and library functions

In addition, a compiler may be able to discover parallelism in other constructs. In this section, we discuss the first two of these parallel constructions.

Fortran 90 Array Assignment In Fortran 77, operations on whole arrays can be accomplished only through explicit DO loops that access array elements one at a time. Fortran 90 array assignment statements allow operations on entire arrays to be expressed more simply.

In Fortran 90, the usual intrinsic operations for scalars (arithmetic, comparison, and logical) can be applied to arrays, provided the arrays are of the same shape. For example, if *A, B,* and *C* are two-dimensional arrays of the same shape, the statement $C = A + B$ assigns to each element of *C* a value equal to the sum of the corresponding elements of *A* and *B*.

In more complex cases, this assignment syntax can have the effect of drastically simplifying the code. For instance, consider the case of three-dimensional arrays, such as the arrays dimensioned in the following declaration:

```
real D(10, 5:24, -5:M), E(0:9, 20, M+6)
```

In Fortran 77 syntax, an assignment to every element of D requires triple-nested loops such as the example shown in Figure 8.

In Fortran 90, this code can be expressed in a single line:

```
D = 2.5*D+E+2.0
```

The FORALL Statement The FORALL statement is an HPF extension to the American National Standards Institute (ANSI) Fortran 90 standard but has been included in the draft Fortran 95 standard.

```
do i = 1, 10
  do j = 5, 24
    do k = -5, M
        D(i, j, k) = 2.5*D(i, j, k) + E(i-1, j-4, k+6) + 2.0
    end do
  end do
end do
```

Figure 8
An Example of a Triple-nested Loop

FORALL is a generalized form of Fortran 90 array assignment syntax that allows a wider variety of array assignments to be expressed. For example, the diagonal of an array cannot be represented as a single Fortran 90 array section. Therefore, the assignment of a value to every element of the diagonal cannot be expressed in a single array assignment statement. It can be expressed in a FORALL statement:

```
real, dimension(n, n) :: A
forall (i = 1:n) A(i, i) = 1
```

Although FORALL structures serve the same purpose as some DO loops do in Fortran 77, a FORALL structure is a parallel assignment statement, not a loop, and in many cases produces a different result from an analogous DO loop. For example, the FORALL statement

```
forall (i = 2:5) C(i, i) = C(i-1, i-1)
```

applied to the matrix

$$C = \begin{bmatrix} 11 & 0 & 0 & 0 & 0 \\ 0 & 22 & 0 & 0 & 0 \\ 0 & 0 & 33 & 0 & 0 \\ 0 & 0 & 0 & 44 & 0 \\ 0 & 0 & 0 & 0 & 55 \end{bmatrix}$$

produces the following result:

$$C = \begin{bmatrix} 11 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 \\ 0 & 0 & 22 & 0 & 0 \\ 0 & 0 & 0 & 33 & 0 \\ 0 & 0 & 0 & 0 & 44 \end{bmatrix}$$

On the other hand, the apparently similar DO loop

```
do i = 2, 5
  C(i, i) = C(i-1, i-1)
end do
```

produces

$$C = \begin{bmatrix} 11 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 \\ 0 & 0 & 11 & 0 & 0 \\ 0 & 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 0 & 11 \end{bmatrix}$$

This happens because the DO loop iterations are performed sequentially, so that each successive element of the diagonal is updated before it is used in the next iteration. In contrast, in the FORALL statement, all the diagonal elements are fetched and used before any stores happen.

The Target Machine

Digital's DEC Fortran 90 compiler generates code for clusters of Alpha processors running the Digital UNIX operating system. These clusters can be separate Alpha workstations or servers connected by a fiber distributed data interface (FDDI) or other network devices. (Digital's high-speed GIGAswitch/FDDI system is particularly appropriate.[14]) A shared-memory, symmetric multiprocessing (SMP) system like the AlphaServer 8400 system can also be used. In the case of an SMP system, the message-passing library uses shared memory as the message-passing medium; the generated code is otherwise identical. The same executable can run on a distributed-memory cluster or an SMP shared-memory cluster without recompiling. DEC Fortran 90 programs use the execution environment provided by Digital's Parallel Software Environment (PSE), a companion product.[3,15] PSE is responsible for invoking the program on multiple processors and for performing the message passing requested by the generated code.

The Architecture of the Compiler

Figure 9 illustrates the high-level architecture of the compiler. The curved path is the path taken when compiler command-line switches are set for compiling programs that will not execute in parallel, or when the scoping unit being compiled is declared as EXTRINSIC(HPF_LOCAL).

Figure 9 shows the front end, transform, middle end, and GEM back end components of the compiler. These components function in the following ways:

- The front end parses the input code and produces an internal representation containing an abstract syntax tree and a symbol table. It performs extensive semantic checking.[16]

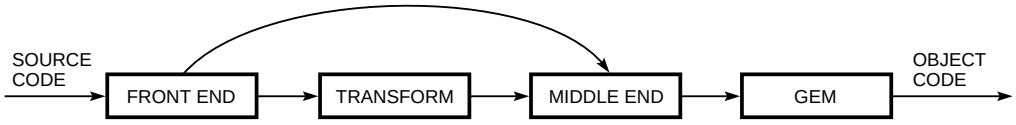


Figure 9
Compiler Components

- The transform component performs the transformation from global-HPF to explicit-SPMD form. To do this, it localizes the addressing of data, inserts communication where necessary, and distributes parallel computations over processors.
- The middle end translates the internal representation into another form of internal representation suitable for GEM.
- The GEM back end, also used by other Digital compilers, performs local and global optimization, storage allocation, code generation, register allocation, and emits binary object code.[17]

In this paper, we are mainly concerned with the transform component of the compiler.

An Overview of Transform

Figure 10 shows the transform phases discussed in this paper. These phases perform the following key tasks:

- LOWER. Transforms array assignments so that they look internally like FORALL statements.
- DATA. Fills in the data space information for each symbol using information from HPF directives where available. This determines where each data object lives, i.e., how it is distributed over the processors.
- ITER. Fills in the iteration space information for each computational expression node. This determines where each computation takes place and indicates where communication is necessary.
- ARG. Pulls functions in the interior of expressions up to the statement level. It also compares the mapping of actual arguments to that of their corresponding dummies and generates remapping into compiler-generated temporaries if necessary.

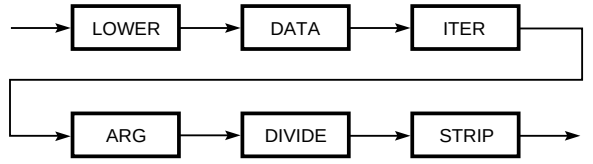


Figure 10
The Transform Phases

- DIVIDE. Pulls all communication inside expressions (identified by ITER) up to the statement level and identifies what kind of communication is needed. It also ensures that information needed for flow of control is available at each processor.
- STRIP. Turns global-HPF code into explicit-SPMD code by localizing the addressing of all data objects and inserting explicit SEND and RECEIVE calls to make communication explicit. In the process, it performs strip mining and loop optimizations, vectorizes communication, and optimizes nearest-neighbor computations.

Transform uses the following main data structures:

- Symbol table. This is the symbol table created by the front end. It is extended by the transform phase to include dope information for array and scalar symbols.
- Dotree. Transform uses the dotree form of the abstract syntax tree as an internal representation of the program.
- Dependence graph. This is a graph whose nodes are expression nodes in the dotree and whose edges represent dependence edges.
- Data spaces. A data space is associated with each data symbol (i.e., each array and each scalar). The data space information describes how each data object is distributed over the processors. This information is derived from HPF directives.
- Iteration spaces. An iteration space is associated with each computational node in the dotree. The iteration space information describes how computations are distributed over the processors. This information is not specified in the source code but is produced by the compiler.

The interrelationship of these data structures is discussed in Reference 18. The data and iteration spaces are central to the processing performed by transform.

The Transform Phases

LOWER

Since the FORALL statement is a generalization of a Fortran 90 array assignment and includes it as a special case, it is convenient for the compiler to have a uniform representation for these two constructions. The

LOWER phase implements this by turning each Fortran 90 array assignment into an equivalent FORALL statement (actually, into the dotree representation of one). This uniform representation means that the compiler has far fewer special cases to consider than otherwise might be necessary and leads to no degradation of the generated code.

DATA

The DATA phase specifies where data lives. Placing and addressing data correctly is one of the major tasks of transform. There are a large number of possibilities:

When a value is available on every processor, it is said to be *replicated*. When it is available on more than one but not all processors, it is said to be *partially replicated*. For instance, a scalar may live on only one processor, or on more than one processor. Typically, a scalar is replicated—it lives on all processors. The replication of scalar data makes fetches cheap because each processor has a copy of the requested value. Stores to replicated scalar data can be expensive, however, if the value to be stored has not been replicated. In that case, the value to be stored must be sent to each processor.

The same consideration applies to arrays. Arrays may be replicated, in which case each processor has a copy of an entire array; or arrays may be partially replicated, in which case each element of the array is available on a subset of the processors.

Furthermore, arrays that are not replicated may be distributed across the processors in several different fashions, as explained above. In fact, each dimension of each array may be distributed independently of the other dimensions. The HPF mapping directives, principally ALIGN and DISTRIBUTE, give the programmer the ability to specify completely how each dimension of each array is laid out. DATA uses the information in these directives to construct an internal description or *data space* of the layout of each array.

ITER

The ITER phase determines where the intermediate results of calculations should live. Its relationship to DATA can be expressed as:

- DATA decides where parallel data lives.
- ITER decides where parallel computations happen.

Each array has a fixed number of dimensions and an extent in each of those dimensions; these properties together determine the shape of an array. After DATA has finished processing, the shape and mapping of each array is known. Similarly, the result of a computation has a particular shape and mapping. This shape may be different from that of the data used in the computation. As a simple example, the computation

```
A(:,:,3) + B(:,:,3)
```

has a two-dimensional shape, even though both arrays A and B have three-dimensional shapes. The data space data structure is used to describe the shape of each array and its layout in memory and across processors; similarly, *iteration space* is used to describe the shape of each computation and its layout across processors. One of the main tasks of transform is to construct the iteration space for each computation so that it leads to as little interprocessor communication as possible: this construction happens in ITER. The compiler's view of this construction and the interaction of these spaces are explained in Reference 18.

Shapes can change within an expression: while some operators return a result having the shape of their operands (e.g., adding two arrays of the same shape returns an array of the same shape), other operators can return a result having a different shape than the shape of their operands. For example, reductions like SUM return a result having a shape with lower rank than that of the input expression being reduced.

One well-known method of determining where computations happen is the "owner-computes" rule. With this method, all the values needed to construct the computation on the right-hand side of an assignment statement are fetched (using interprocessor communication if necessary) and computed on the processor that contains the left-hand-side location. Then they are stored to that left-hand-side location (on the same processor on which they were computed). Thus a description of where computations occur is derived from the output of DATA. There are, however, simple examples where this method leads to less than optimal performance. For instance, in the code

```
        real A(n, n), B(n, n), C(n, n)
!hpf$   distribute A(block, block)
!hpf$   distribute B(cyclic, cyclic)
!hpf$   distribute C(cyclic, cyclic)

        forall (i=1:n, j=1:n)
          A(i, j) = B(i, j) + C(i, j)
        end forall
```

the owner-computes rule would move B and C to align with A, and then add the moved values of B and C and assign to A. It is certainly more efficient, however, to add B and C together where they are aligned with each other and then communicate the result to where it needs to be stored to A. With this procedure, we need to communicate only one set of values rather than two. The compiler identifies cases such as these and generates the computation, as indicated here, to minimize the communication.

ARG

The ARG phase performs any necessary remapping of actual arguments at subroutine call sites. It does this by comparing the mapping of the actuals (as determined by ITER) to the mapping of the corresponding dummies (as determined by DATA).

In our implementation, the caller performs all remapping. If remapping is necessary, ARG exposes that remapping by inserting an assignment statement that remaps the actual to a temporary that is mapped the way the dummy is mapped. This guarantees that references to a dummy will access the correct data as specified by the programmer. Of course, if the parameter is an OUT argument, a similar copy-out remapping has to be inserted after the subroutine call.

DIVIDE

The DIVIDE phase partitions ("divides") each expression in the dotree into regions. Each region contains computations that can happen without interprocessor communication. When region R uses the values of a subexpression computed in region S, for example, interprocessor communication is required to remap the computed values from their locations in S to their desired locations in R. DIVIDE makes a temporary mapped the way region R needs it and makes an explicit assignment statement whose left-hand side is that temporary and whose right-hand side is the subexpression computed in region S. In this way, DIVIDE makes explicit the interprocessor communication that is implicit in the iteration space information attached to each expression node.

DIVIDE also performs other processing:

- DIVIDE replicates expressions needed to manage control flow, such as an expression representing a bound of a DO loop or the condition in an IF statement. Consequently, each processor can do the necessary branching.
- For each statement requiring communication, DIVIDE identifies the kind of communication needed.

  Depending on what knowledge the two sides of the communication (i.e., the sender and the receiver) have, we distinguish two kinds of communication:
  - Full knowledge. The sender knows what it is sending and to whom, and the receiver knows what it is receiving and from whom.
  - Partial knowledge. Either the sender knows what it is sending and to whom, or the receiver knows what it is receiving and from whom, but the other party knows nothing.

    This kind of message is typical of code dealing with irregular data accesses, for instance, code with array references containing vector-valued subscripts.

STRIP

The STRIP phase (shortened from "strip miner"; probably a better term would be the "localizer") takes the statements categorized by DIVIDE as needing communication and inserts calls to library routines to move the data from where it is to where it needs to be.

It then localizes parallel assignments coming from vector assignments and FORALL constructs. In other words, each processor has some (possibly zero) number of array locations that must be stored to. A set of loops is generated that calculates the value to be stored and stores it. The bounds for these loops are dependent on the distribution of the array being assigned to and the section of the array being assigned to. These bounds may be explicit numbers known at compile time, or they may be expressions (when the array size is not known at compile time). In any case, they are exposed so that they may be optimized by later phases. They are not calls to run-time routines.

The subscripts of each dimension of each array in the statement are then rewritten in terms of the loop variable. This modification effectively turns the original global subscript into a local subscript. Scalar subscripts are also converted to local subscripts, but in this case the subscript expression does not involve loop indices. Similarly, scalar assignments that reference array elements have their subscripts converted from global addressing to local addressing, based on the original subscript and the distribution of the corresponding dimension of the array. They do not require strip loops. For example, consider the code fragment shown in Figure 11a.

Here k is some variable whose value has been assigned before the FORALL. Let us assume that A and B have been distributed over a 4×5 processor array in such a way that the first dimensions of A and B are distributed CYCLIC over the first dimension of the processor array (which has extent 4), and the second dimensions of A and B are distributed BLOCK over the second dimension of the processor array (which has extent 5). (The programmer can express this through a facility in HPF.) The generated code is shown in Figure 11b.

If the array assigned to on the left-hand side of such a statement is also referenced on the right-hand side, then replacing the parallel FORALL by a DO loop may violate the "fetch before store" semantics of the original statement. That is, an array element may be assigned to on one iteration of the DO loop, and this new value may subsequently be read on a later iteration. In the original meaning of the statement, however, all values read would be the original values.

This problem can always be resolved by evaluating the right-hand side of the statement in its entirety into a temporary array, and then—in a second set of DO loops—assigning that temporary to the left-hand side. We use dependence analysis to determine if such a problem occurs at all. Even if it does, there are cases in which loop transformations can be used to eliminate the need for a temporary, as outlined in Reference 19.

```
        real A(100, 20), B(100, 20)
!hpf$   distribute A(cyclic, block), B(cyclic, block)
        ...
        forall (i = 2:99)
          A(i, k) = B(i, k)
        end forall
```

(a) Code Fragment

```
m = my_processor()
...
if k mod 5 = ⌊m/4⌋ then
   do i = (if m mod 4 = 0 then 2 else 1), (if m mod 4 = 3 then 24 else 25)
     A(i, ⌊k/5⌋) = B(i, ⌊k/5⌋)
   end do
end if
```

(b) Pseudocode Generated for Code Fragment

Figure 11
Code Fragment and Pseudocode Generated for Code Fragment

(Some poor implementations always introduce the temporary even when it is not needed.)

Unlike other HPF implementations, ours uses compiler-generated inlined expressions instead of function calls to determine local addressing values. Furthermore, our implementation does not introduce barrier synchronization, since the sends and receives generated by the transform phase will enforce any necessary synchronization. In general, this is much less expensive than a naive insertion of barriers. The reason this works can be seen as follows: first, any value needed by a processor is computed either locally or nonlocally. If the value is computed locally, the normal control flow guarantees correct access order for that value. If the value is computed nonlocally, the generated receive on the processor that needs the value causes the receiving processor to wait until the value arrives from the sending processor. The sending processor will not send the value until it has computed it, again because of normal control-flow. If the sending processor is ready to send data before the receiving processor is ready for it, the sending processor can continue without waiting for the data to be received. Digital's Parallel Software Environment (PSE) buffers the data until it is needed.[15]

Some Optimizations Performed by the Compiler

The GEM back end performs the following optimizations:

- Constant folding
- Optimizations of arithmetic IF, logical IF, and block IF-THEN-ELSE
- Global common subexpression elimination
- Removal of invariant expressions from loops
- Global allocation of general registers across program units
- In-line expansion of statement functions and routines
- Optimization of array addressing in loops
- Value propagation
- Deletion of redundant and unreachable code
- Loop unrolling
- Software pipelining to rearrange instructions between different unrolled loop iterations
- Array temporary elimination

In addition, the transform component performs some important optimizations, mainly devoted to improving interprocessor communication. We have implemented the following optimizations:

Message Vectorization

The compiler generates code to limit the communication to one SEND and one RECEIVE for each array being moved between any two processors. This is the most obvious and basic of all the optimizations that a compiler can perform for distributed-memory architectures and has been widely studied.[20–22]

If the arrays *A* and B are laid out as in Figure 12 and if *B* is to be assigned to *A*, then array elements *B*(4), *B*(5), and *B*(6), all of which live on processor 6, should be sent to processor 1. Clearly, we do not want to generate three distinct messages for this. Therefore, we collect these three elements and generate one message containing all three of them. This example involves full knowledge.

Communications involving partial knowledge are also vectorized, but they are much more expensive because the side of the message without initial knowledge has to be informed of the message. Although there are several ways to do this, all are costly, either in time or in space.

We use the same method, incidentally, to inline the HPF XXX_SCATTER routines. These new routines have been introduced to handle a parallel construct that could cause more than one value to be assigned to the same location. The outcome of such cases is determined by the routine being inlined. For instance, SUM_SCATTER simply adds all the values that arrive at each location and assigns the final result to that location. Although this is an example of interprocessor communication with partial knowledge, we can still build up messages so that only a minimum number of messages are sent.

In some cases, we can improve the handling of communications with partial knowledge, provided they occur more than once in a program. For more information, please see the section Run-time Preprocessing of Irregular Data Accesses.

Strip Mining and Loop Optimizations

Strip mining and loop optimizations have to do with generating efficient code on a per-processor basis, and so in some sense can be thought of as conventional. Generally, we follow the processing detailed in Reference 19 and summarized as:

- Strip mining obstacles are eliminated where possible by loop transformations (loop reversal or loop interchange).
- Temporaries, if introduced, are of minimal size; this is achieved by loop interchange.
- Exterior loop optimization is used to allow reused data to be kept in registers over consecutive iterations of the innermost loop.
- Loop fusion enables more efficient use of conventional optimizations and minimizes loop overhead.

Nearest-neighbor Computations

Nearest-neighbor computations are common in code written to discretize partial differential equations. See the example given in Figure 2.

If we have, for example, 16 processors, with the array *A* distributed in a (BLOCK, BLOCK) fashion over the processors, then conceptually, the array is distributed as in Figure 13, where the arrows indicate communication needed between neighboring processors. In fact, in this case, each processor needs to see values only from a narrow strip (or "shadow edge") in the memory of its neighboring processors, as shown in Figure 14.

The compiler identifies nearest-neighbor computations (the user does not have to tag them), and it alters the addressing of each array involved in these computations (throughout the compilation unit). As a result, each processor can store those array elements that are needed from the neighboring processors. Those array elements are moved in (using message vectorization) at the beginning of the computation, after which the entire computation is local.

Recognizing nearest-neighbor statements helps generate better code in several ways:

- Less run-time overhead. The compiler can easily identify the exact small portion of the array that needs to be moved. The communication for nearest-neighbor assignments is extremely regular: At each step, each processor is sending an entire shadow edge to precisely one of its neighbors. Therefore the communication processing overhead is greatly reduced. That is, we are able to generate

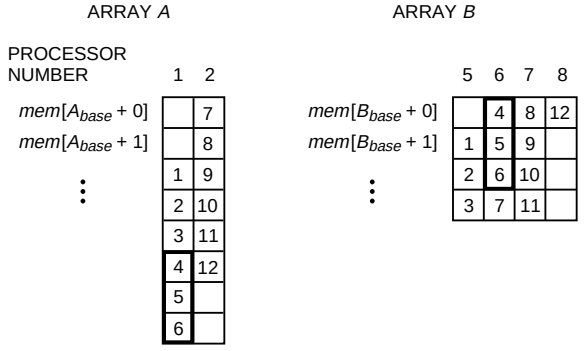


Figure 12
Two Arrays in Memory

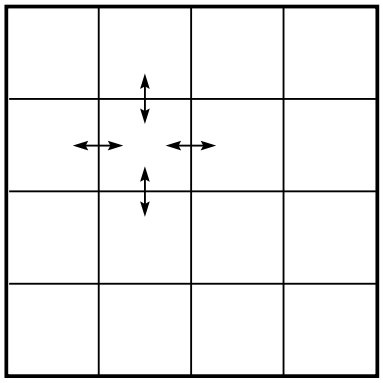


Figure 13
A Nearest-neighbor Communication Pattern

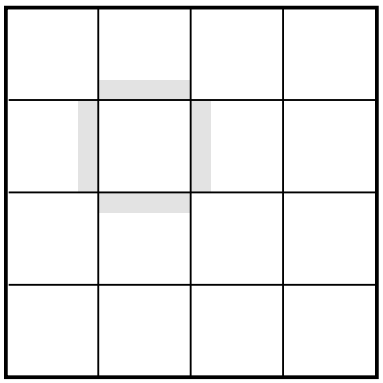


Figure 14
Shadow Edges for a Nearest-neighbor Computation

communication involving even less overhead than general communication involving full knowledge.

- No local copying. If shadow edges were not used, then the following standard processing would take place: For each shifted-array reference on the right-hand side of the assignment, shift the entire array; then identify that part of the shifted array that lives locally on each processor and create a local temporary to hold it. Some of that temporary (the part representing our shadow edge) would be moved in from a neighboring processor, and the rest of the temporary would be copied locally from the original array. Our processing eliminates the need for the local temporary and for the local copy, which is substantial for large arrays.
- Greater locality of reference. When the actual computation is performed, greater locality of reference is achieved because the shadow edges (i.e., the received values) are now part of the array, rather than being a temporary somewhere else in memory.
- Fewer messages. Finally, the optimization also makes it possible for the compiler to see that some messages may be combined into one message, thereby reducing the number of messages that must be sent. For instance, if the right-hand side of the assignment statement in the above example also contained a term $A(i+1, j+1)$, even though overlapping shadow edges and an additional shadow edge would now be in the diagonally adjacent processor, no additional communication would need to be generated.

Reductions

The SUM intrinsic function of Fortran 90 takes an array argument and returns the sum of all its elements. Alternatively, SUM can return an array whose rank is one less than the rank of its argument, and each of whose values is the sum of the elements in the argument along a line parallel to a specified dimension. In either case, the rank of the result is less than that of the argument; therefore, SUM is referred to as a reduction intrinsic. Fortran 90 includes a family of such reductions, and HPF adds more.

We inline these reduction intrinsics in such a way as to distribute the work as much as possible across the processors and to minimize the number of messages sent.

In general, the reduction is performed in three basic steps:

1. Each processor locally performs the reduction operation on its part of the reduction source into a buffer.
2. These partial reduction results are combined with those of the other processors in a "logarithmic" fashion (to reduce the number of messages sent).
3. The accumulated result is then locally copied to the target location.

Figure 15 shows how the computations and communications occur in a complete reduction of an array distributed over four processors. In this figure, each vertical column represents the memory of a single processor. The processors are thought of (in this case) as being arranged in a 2×2 square; this is purely for conceptual purposes—the actual processors are typically connected through a switch.

First, the reduction is performed locally in the memory of each processor. This is represented by the vertical arrows in the figure. Then the computations are accumulated over the four processors in two steps: the two parallel curved arrows indicate the interprocessor communication in the first step, followed by the communication indicated by the remaining curved arrow in the second step. Of course, for five to eight processors, three communication steps would be needed, and so on.

Although this basic idea never changes, the actual generated code must take into account various factors. These include (1) whether the object being reduced

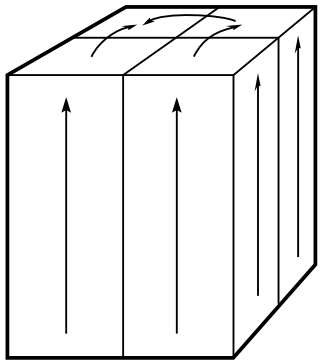


Figure 15
Computations and Communication for a Complete Reduction over Four Processors

is replicated or distributed, (2) the different distributions that each array dimension might have, and (3) whether the reduction is complete or partial (i.e., with a DIM argument).

Run-time Preprocessing of Irregular Data Accesses

Run-time preprocessing of irregular data accesses is a popular technique.[23] If an expression involving the same pattern of irregular data access is present more than once in a compilation unit, additional run-time preprocessing can be used to good effect. An abstract example would be code of the form:

```
call setup(U, V, W)
do i = 1, n_time_steps, 1
   do i = 1, n, 1
      A(V(i)) = A(V(i)) + B(W(i))
   enddo
   do i = 1, n, 1
      C(V(i)) = C(V(i)) + D(W(i))
   enddo
   do i = 1, n, 1
      E(V(i)) = E(V(i)) + F(W(i))
   enddo
enddo
```

which could be written in HPF as:

```
call setup(U, V, W)
do i = 1, n_time_steps, 1
   A = sum_scatter(B(W(1:n)), A, V(1:n))
   C = sum_scatter(D(W(1:n)), C, V(1:n))
   E = sum_scatter(F(W(1:n)), E, V(1:n))
enddo
```

To the compiler, the significant thing about this code is that the indirection vectors V and W are constant over iterations of the loop. Therefore, the compiler computes the source and target addresses of the data that has to be sent and received by each processor once at the top of the loop, thus paying this price one time. Each such statement then becomes a communication with full knowledge and is executed quite efficiently with message vectorization.

Other Communication Optimizations

The processing needed to set up communication of array assignments is fairly expensive. For each element of source data on a processor, the value of the data and the target processor number are computed. For each target data on a processor, the source processor number and the target memory address are computed. The compiler and run time also need to sort out local data that do not involve communication, as well as to vectorize the data that are to be communicated.

We try to optimize the communication processing by analyzing the iteration space and data space of the array sections involved. Examples of the patterns of operations that we optimize include the following:

- Contiguous data. When the source or target local array section on each processor is in contiguous memory addresses, the processing can be optimized to treat the section as a whole, instead of computing the value or memory address of each element in the section.

  In general, array sections belong to this category if the last vector dimension is distributed BLOCK or CYCLIC and the prior dimensions (if any) are all serial.

  If the source and target array sections satisfy even more restricted constraints, the processing overhead may be further reduced. For example, array operations that involve sending a contiguous section of BLOCK or CYCLIC distributed data to a single processor, or vice versa, belong to this category and result in very efficient communication processing.
- Unique source or target processor. When a processor only sends data to a unique processor, or a processor only receives data from a unique processor, the processing can be optimized to use that unique processor number instead of computing the processor number for each element in the section. This optimization also applies to target arrays that are fully replicated.
- Irregular data access. If all indirection vectors are fully replicated for an irregular data access, we can actually implement the array operation as a full-knowledge communication instead of a more expensive partial-knowledge communication.

  For example, the irregular data access statement

  ```
  A(V(:)) = B(:)
  ```

  can be turned into a regular remapping statement if V is fully replicated and A and B are both distributed.

  Furthermore, if B is also fully replicated, the statement is recognized as a local assignment, removing the communication processing overhead altogether.

Performance

In this section, we examine the performance of three HPF programs. One program applies the shallow-water equations, discretized using a finite difference scheme to a specific problem; another is a conjugate-gradient solver for the Poisson equation, and the third is a three-dimensional finite difference solver. These programs are not reproduced in this paper, but they can be obtained via the World Wide Web at http://www.digital.com/info/hpc/f90/.

The Shallow-water Benchmark

The shallow-water equations model atmospheric flows, tides, river and coastal flows, and other phenomena. The shallow-water benchmark program uses these equations to simulate a specific flow problem. It models variables related to the pressure, velocity, and vorticity at each point of a two-dimensional mesh that

is a slice through either the water or the atmosphere. Partial differential equations relate the variables. The model is implemented using a finite-difference method that approximates the partial differential equations at each of the mesh points.[24] Models based on partial differential equations are at the core of many simulations of physical phenomena; finite difference methods are commonly used for solving such models on computers.

The shallow-water program is a widely quoted benchmark, partly because the program is small enough to examine and tune carefully, yet it performs real computation representative of many scientific simulations. Unlike SPEC and other benchmarks, the source for the shallow-water program is not controlled.

The shallow-water benchmark was written in HPF and run in parallel on workstation farms using PSE. There is no explicit message-passing code in the program. We modified the Fortran 90 version that Applied Parallel Research used for its benchmark data. The F90/HPF version of the program takes advantage of the new features in Fortran 90 such as modules. The Fortran 77 version of the program is an unmodified version from Applied Parallel Research.

The resulting programs were run on two hardware configurations: as many as eight 275-megahertz (MHz) DEC 3000 Model 900 workstations connected by a GIGAswitch system, and an eight-processor AlphaServer 8400 (300-MHz) system using shared-memory as the messaging medium. Table 1 gives the speedups obtained for the 512×512-sized problem, with ITMAX set to 50.

The speedups in each line are relative to the DEC Fortran 77 code, compiled with the DEC Fortran version 3.6 compiler and run on one processor. The DEC Fortran 90 -wsf compiler is the DEC Fortran 90 version 1.3 compiler with the -wsf option ("parallelize HPF for a workstation farm") specified. Both compilers use version 3.58 of the Fortran RTL. The operating system used is Digital UNIX version 3.2.

Table 1 indicates that this HPF version of shallow water scales very well to eight processors. In fact, we are getting apparent superlinear speedup in some cases. This is due in part to optimizations that the DEC Fortran 90 compiler performs that the serial compiler does not, and in part to cache effects: with more processors, there is more cache. On the shared-memory machine, we are getting apparent superlinear speedups even when compared to the DEC Fortran 90 -wsf compiler's one-processor code; this is likely due to cache effects. The program appears to scale well beyond eight processors, though we have not made a benchmark-quality run on more than eight identical processors.

For purposes of comparison, Table 2 gives the published speedups from Applied Parallel Research on the shallow-water benchmark for the IBM SP2 and Intel Paragon parallel architectures. The speedups shown are relative to the one-processor version of the code. This table indicates that the scaling achieved by the DEC Fortran 90 compiler on Alpha workstation farms is comparable to that achieved by Applied Parallel Research on dedicated parallel systems with high-speed parallel interconnects.

A Conjugate-gradient Poisson Solver

The Poisson partial differential equation is a workhorse of mathematical physics, occurring in problems

Table 2
Speedups of HPF Shallow-water Code on IBM's and Intel's Parallel Architectures

	Number of Processors				
	8	4	3	2	1
IBM SP2	7.50	3.81	—	1.97	1.00
Intel Paragon	7.38	3.84	—	1.95	1.00

Table 1
Speedups of DEC Fortran 90/HPF Shallow-water Equation Code

	DEC Fortran 90 -wsf Compiler					DEC Fortran 77 Compiler
	Number of Processors					
	8	4	3	2	1	1
Eight 275-MHz, DEC 3000 Model 900 workstations in a GIGAswitch farm	8.57	3.13	2.19	1.59	1.00	1.00
Eight-processor, 300-MHz, shared-memory SMP AlphaServer 8400 systems	10.6	5.30	3.86	1.97	1.12	1.00

of heat flow and electrostatic or gravitational potential. We have investigated a Poisson solver using the conjugate-gradient algorithm. The code exercises both the nearest-neighbor optimizations and the inlining abilities of the DEC Fortran 90 compiler.[25]

Table 3 gives the timings and speedup obtained on a 1000 × 1000 array. The hardware and software configurations are identical to those used for the shallow-water timings.

Red-black Relaxation

A common method of solving partial differential equations is red-black relaxation.[26] We used this method to solve the Poisson equation in a three-dimensional cube. We compare the parallelization of this algorithm for a distributed-memory system (a cluster of Digital Alpha workstations) with Parallel Virtual Machine (PVM), which is an explicit message-passing library, and with HPF.[27] These algorithms are based on codes written by Klose, Wolton, and Lemke and made available as part of the suite of GENESIS distributed-memory benchmarks.[28]

Table 4 gives the speedups obtained for both the HPF and PVM versions of the program, which solves a 128 × 128 × 128 problem, on a cluster of DEC 3000 Model 900 workstations connected by an FDDI/GIGAswitch system. The speedups shown are relative to DEC Fortran 77 code written for and run on a single processor. This table shows that the HPF version performs somewhat better than the PVM version.

There is a significant difference in the complexity of the programs, however. The PVM code is quite intricate, because it requires that the user be responsible for the block partitioning of the volume, and then for explicitly copying boundary faces between processors. By contrast, the HPF code is intuitive and far more easily maintained. The reader is encouraged to obtain the codes (as described above) and compare them.

Table 4
Speedups of DEC Fortran 90/HPF and DEC Fortran 77/PVM on Red-black Code

	Number of Processors			
	8	4	2	1
DEC Fortran 77				1.00
DEC Fortran 77/PVM	7.01	3.73	1.79	—
DEC Fortran 90/HPF	8.04	4.10	1.95	1.05

In conclusion, we have shown that important algorithms familiar to the scientific and technical community can be written in HPF. HPF codes scale well to at least eight processors on farms of Alpha workstations with PSE and deliver speedups competitive with other vendors' dedicated parallel architectures.

Acknowledgments

Significant help from the following people has been essential to the success of this project: High Performance Computing Group engineering manager Jeff Reyer; the Parallel Software Environment Group led by Ed Benson and including Phil Cameron, Richard Warren, and Santa Wiryaman; the Parallel Tools Group managed by Tomas Lofgren and including David LaFrance-Linden and Chuck Wan; the Digital Fortran 90 Group led by Keith Kimball; David Loveman for discussions of language issues; Ned Anderson of the High Performance Computing Numerical Library Group for consulting on numerical issues; Brendan Boulter of Digital Galway for the conjugate-gradient code and help with benchmarking; Bill Celmaster, for writing the PVM version of the red-black benchmark and its related description; Roland Belanger for benchmarking assistance; and Marco Annaratone for useful technical discussions.

Table 3
Speedups of DEC Fortran 90/HPF on Conjugate-gradient Poisson Solver

	DEC Fortran 90 -wsf Compiler					DEC Fortran 77 Compiler
	Number of Processors					
	8	4	3	2	1	1
Eight 275-MHz, DEC 3000 Model 900 workstations in a GIGAswitch farm	14.1	8.38	5.20	2.52	1.07	1.00
Eight-processor, 300-MHz, shared-memory SMP AlphaServer 8400 systems	17.0	9.02	6.87	4.51	0.98	1.00

References and Notes

1. High Performance Fortran Forum, "High Performance Fortran Language Specification, Version 1.0," *Scientific Programming,* vol. 2, no. 1 (1993). Also available as Technical Report CRPC-TR93300, Center for Research on Parallel Computation, Rice University, Houston, Tex.; and via anonymous ftp from titan.cs.rice.edu in the directory public/HPFF/draft; version 1.1 is the file hpf_v11.ps.

2. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel, *The High Performance Fortran Handbook* (Cambridge, Mass.: MIT Press, 1994).

3. *Digital High Performance Fortran 90 HPF and PSE Manual* (Maynard, Mass.: Digital Equipment Corporation, 1995).

4. *DEC Fortran 90 Language Reference Manual* (Maynard, Mass.: Digital Equipment Corporation, 1994).

5. E. Albert, K. Knobe, J. Lukas, and G. Steele, Jr., "Compiling Fortran 8x Array Features for the Connection Machine Computer System," *Symposium on Parallel Programming: Experience with Applications, Languages, and Systems, ACM SIGPLAN,* July 1988.

6. K. Knobe, J. Lukas, and G. Steele, Jr., "Massively Parallel Data Optimization," *Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation, IEEE,* George Mason University, October 1988.

7. K. Knobe, J. Lukas, and G. Steele, Jr., "Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines," *Journal of Parallel and Distributed Computing,* vol. 8 (1990): 102–118.

8. K. Knobe and V. Natarajan, "Data Optimization: Minimizing Residual Interprocessor Data Motion on SIMD Machines," *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation, IEEE,* University of Maryland, October 1990.

9. M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Systems,* vol. 3, no. 2 (1992): 179–193.

10. M. Gupta and P. Banerjee, "PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers," *ICS93: The Seventh ACM International Conference on Supercomputing,* Japan, 1993.

11. S. Chatterjee, J. Gilbert, and R. Schreiber, "The Alignment-distribution Graph," *Sixth Annual Workshop on Languages and Compilers for Parallel Computing,* 1993.

12. J. Anderson and M. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM Press,* vol. 28 (1993): 1290–1317.

13. The seven values $A(9, 2)$, $A(9, 3)$, … $A(9, 8)$ can be expressed concisely in Fortran 90 as $A(9, 2{:}8)$.

14. R. Souza et al., "GIGAswitch System: A High-performance Packet-switching Platform," *Digital Technical Journal,* vol. 6, no. 1 (1994): 9–22.

15. E. Benson, D. LaFrance-Linden, R. Warren, and S. Wiryaman, "Design of Digital's Parallel Software Environment," *Digital Technical Journal,* vol. 7, no. 3 (1995, this issue): 24–38.

16. D. Loveman, "The DEC High Performance Fortran 90 Compiler Front End," *Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation,* pages 46–53, McLean, Virginia, February 1995. IEEE.

17. D. Blickstein et al., "The GEM Optimizing Compiler System," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue, 1992): 121–136.

18. C. Offner, "A Data Structure for Managing Parallel Operations," *Proceedings of the 27th Hawaii International Conference on System Sciences, Volume II: Software Technology* (IEEE Computer Society Press, 1994): 33–42.

19. J. Allen and K. Kennedy, "Vector Register Allocation," *IEEE Transactions on Computers,* vol. 41, no. 10 (1992): 1290–1317.

20. S. Amarasinghe and M. Lam, "Communication Optimization and Code Generation for Distributed Memory Machines," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM Press,* vol. 28 (1993): 126–138.

21. C.-W. Tseng, "An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines," Ph.D. thesis, Rice University, Houston, Tex., 1993. Available as Rice COMP TR93-199.

22. A. Rogers, "Compiling for Locality of Reference," Technical Report TR91-1195, Ph.D. thesis, Cornell University, Ithaca, N.Y., 1991.

23. J. Saltz, R. Mirchandaney, and K. Crowley, "Run-time Parallelization and Scheduling of Loops," *IEEE Transactions on Computers* (1991): 603–611.

24. R. Sadourney, "The Dynamics of Finite-difference Models of the Shallow-water Equations," *Journal of Atmospheric Sciences,* vol. 32, no. 4 (1975).

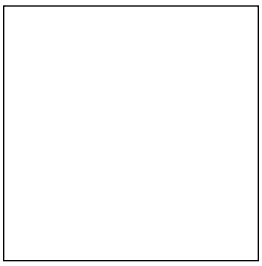
25. B. Boulter, "Performance Evaluation of HPF for Scientific Computing," *Proceedings of High Performance Computing and Networking, Lecture Notes in Computer Science 919* (Springer-Verlag, 1995).

26. W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in Fortran: The Art of Scientific Computing* (Cambridge: Cambridge University Press, 2d edition, 1992).

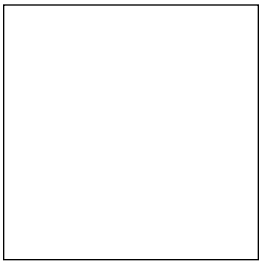
27. A. Geist, *PVM: Parallel Virtual Machine* (Cambridge, Mass.: MIT Press, 1994).

28. A. Hey, "The GENESIS Distributed Memory Benchmarks," *Parallel Computing,* vol. 17, no. 10–11 (1991): 1275–1283.

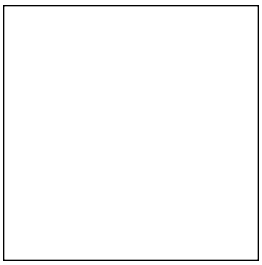
Biographies



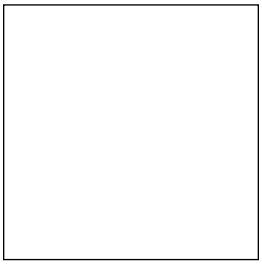
Jonathan Harris
Jonathan Harris is a consulting engineer in the High Performance Computing Group and the project leader for the transform (HPF parallelization) component of the DEC Fortran 90 compiler. Prior to the High Performance Fortran project, he designed the instruction set for the DECmpp, a 16K processor machine that became operational in 1987. He also helped design a compiler and debugger for the machine, contributed to the processor design, and invented parallel algorithms, some of which were patented. He obtained an M.S. in computer science in 1985 as a Digital Resident at the University of Illinois; he has been with Digital since 1977.



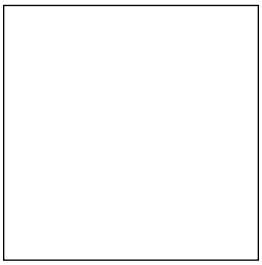
John A. Bircsak
A principal software engineer in Digital's High Performance Computing Group, John Bircsak contributed to the design and development of the transform component of the DEC Fortran 90 compiler. Before joining Digital in 1991, he was involved in the design and development of compilers at Compass, Inc.; prior to that, he worked on compilers and software tools at Raytheon Corp. He holds a B.S.E. in computer science and engineering from the University of Pennsylvania (1984) and an M.S. in computer science from Boston University (1990).



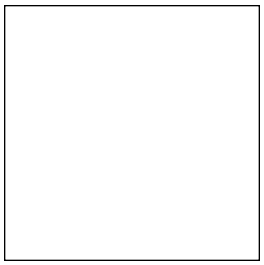
M. Regina Bolduc
Regina Bolduc joined Digital in 1991; she is a principal software engineer in the High Performance Computing Group. Regina was involved in the development of the transform and front end components of the DEC Fortran 90 compiler. Prior to this work, she was a senior member of the technical staff at Compass, Inc., where she worked on the design and development of compilers and compiler-generator tools. Regina received a B.A. in mathematics from Emmanuel College in 1957.



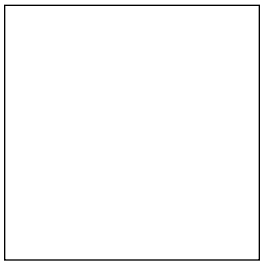
Jill Ann Diewald
Jill Diewald contributed to the design and implementation of the transform component of the DEC Fortran 90 compiler. She is a principal software engineer in the High Performance Computing Group. Before joining Digital in 1991, Jill was a technical coordinator at Compass, Inc., where she helped design and develop compilers and compiler-related tools. Prior to that position, she worked at Innovative Systems Techniques and Data Resources, Inc. on programming languages that provide economic analysis, modeling, and database capabilities for the financial marketplace. She has a B.S. in computer science from the University of Michigan.



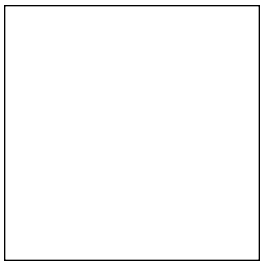
Israel Gale
Israel Gale is a principal writer in the High Performance Computing Group and the author of Digital's High Performance Fortran Tutorial. He joined Digital in 1994 after receiving an A.M. degree in Near Eastern Languages and Civilizations from Harvard University.



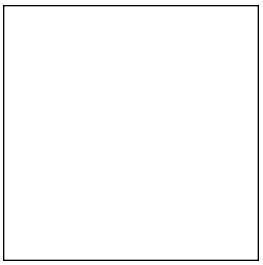
Neil W. Johnson
Before coming to Digital in 1991, Neil Johnson was a staff scientist at Compass, Inc. He has more than 30 years of experience in the development of compilers, including work on the vectorization and optimization phases and tools for compiler development. As a principal software engineer in Digital's High Performance Computing Group, he has worked on the development of the front-end phase for the DEC Fortran 90 compiler. He is a member of ACM and holds B.A. (magna cum laude) and M.A. degrees in mathematics from Concordia College and the University of Nebraska, respectively.



Shin Lee
Shin Lee is a principal software engineer in Digital's High Performance Computing Group. She contributed to the design and development of the transform component of the DEC Fortran 90 compiler. Before joining Digital in 1991, she worked on the design and development of compilers at Encore Computer Corporation and Wang Labs, Inc. She received a B.S. in chemistry from National Taiwan University and an M.S. in computer science from Michigan State University.



C. Alexander Nelson
In 1991, Alex Nelson came to Digital to work on the SIMD compiler for the MasPar machine. He is a principal software engineer in the High Performance Computing Group and helped design and implement the transform component of the DEC Fortran 90 compiler. Prior to this work, he was employed as a software engineer at Compass, Inc. and a systems architect at Incremental Systems. He received an M.S. in computer science from the University of North Carolina in 1987 and an M.S. in chemistry (cum laude) from Davidson College in 1985. He is a member of Phi Beta Kappa.



Carl D. Offner
As a principal software engineer in Digital's High Performance Computing Group, Carl Offner has primary responsibility for the high-level design of the transform component of the DEC Fortran 90 compiler. He is also a member of the Advanced Development Group working on issues of parallelizing DO loops. Before joining Digital in 1993, Carl worked at Intel and at Compass, Inc. on compiler development. Before that, he taught junior high and high school mathematics for 16 years. Carl represents Digital at the High Performance Fortran Forum. He is a member of ACM, AMS, and MAA and holds a Ph.D. in mathematics from Harvard University.