Compaq Fortran

Parallel Processing Manual for Tru64 UNIX Systems

January 2002

This manual provides information about the Compaq Fortran parallel program development and run-time environment on Compaq Tru64 UNIX systems.

Note: The NUMA parallel processing feature described in this manual is available in the Compaq Fortran software but is not supported.

Revision/Update Information:	This is a new manual.
Software Version:	Compaq Fortran Version 5.5 or higher for Tru64 UNIX Systems

Compaq Computer Corporation
Houston, Texas

First Printing, January 2002

This document is available on CD–ROM.

This document was prepared using DECdocument, Version 3.3-1n.

Contents

6 Compiling and Running HPF Programs

7 Optimizing HPF Programs

A HPF Tutorials: Introduction

B HPF Tutorial: LU Decomposition

C HPF Tutorial: Solving Nearest-Neighbor Problems

D HPF Tutorial: Visualizing the Mandelbrot Set

E HPF Tutorial: Simulating Network Striped Files

Index

Examples

Figures

Tables

Preface

This manual describes the Compaq Fortran parallel processing environment. This environment comprises coding, compiling, linking, and executing Compaq Fortran parallel programs using the Compaq Tru64™ UNIX operating system on Alpha hardware.

This manual brings together explanations of Fortran parallel processing that have appeared in other Compaq Fortran manuals. Chapter 5 contains descriptions of HPF directives that first appeared in the *Digital High Performance Fortran 90 HPF and PSE Manual*.

This manual also contains new material not previously published.

Intended Audience

This manual makes the following assumptions about you, the reader:

- You already have a basic understanding of the Fortran 95/90 language. Tutorial Fortran 95/90 language information is widely available in commercially published books (see the Preface of the *Compaq Fortran Language Reference Manual*).
- You are familiar with the operating system shell commands used during program development and a text editor, such as `emacs` or `vi`. Such information is available in your operating system documentation set or commercially published books.
- You have access to the *Compaq Fortran Language Reference Manual*, which describes the Compaq Fortran language.
- You have access to the *Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems*, which describes the Compaq Fortran programming environment including the compiler options, performance guidelines, run-time I/O, error handling support, and data types.

Structure of This Manual

This manual consists of the following chapters and appendixes:

- Chapter 1, Compaq Fortran Parallel Processing: An Introduction, introduces you to Compaq Fortran parallel processing.
- Chapter 2, Data Layout: An Introduction, introduces directives that distribute data — usually in large arrays — among the memories of processors that share computations.
- Chapter 3, NUMA Architecture: An Introduction, introduces Non Uniform Memory Access (NUMA) architecture.
- Chapter 4, High Performance Fortran (HPF) Software: An Introduction, introduces High Performance Fortran (HPF) software.
- Chapter 5, HPF Essentials, shows ways to distribute data among the memories of processors that share computations.
- Chapter 6, Compiling and Running HPF Programs, explains how to compile and execute HPF programs.
- Chapter 7, Optimizing HPF Programs, explains how to write HPF programs that execute quickly.
- Appendix A, HPF Tutorials: Introduction, introduces the four tutorials in the appendixes.
- Appendix B, HPF Tutorial: LU Decomposition, contains an HPF example program from linear algebra.
- Appendix C, HPF Tutorial: Solving Nearest-Neighbor Problems, contains an HPF example program related to the transfer of heat in a rectangular solid.
- Appendix D, HPF Tutorial: Visualizing the Mandelbrot Set, contains an HPF example program related to the Mandelbrot Set.
- Appendix E, HPF Tutorial: Simulating Network Striped Files, contains an HPF example program that illustrates input/output by simulating network striped files.

Associated Documents

The following documents may also be useful to Compaq Fortran programmers:

- *Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems*

  Describes compiling, linking, running, and debugging Compaq Fortran programs, performance guidelines, run-time I/O, error-handling support, data types, numeric data conversion, calling other procedures and library routines, and compatibility with Compaq Fortran 77 (formerly DEC Fortran).

  In particular, you should see the chapter on "Parallel Compiler Directives and Their Programming Environment" and the appendix on "Parallel Library Routines."

- *Compaq Fortran Language Reference Manual*

  Describes the Compaq Fortran 95/90 source language for reference purposes, including the format and use of statements, intrinsic procedures, and other language elements.

- *Compaq Fortran Installation Guide for Tru64 UNIX Systems*

  Explains how to install Compaq Fortran on the Compaq Tru64 UNIX operating system, including prerequisites and requirements.

- Compaq Fortran Release Notes

  Provide more information on this version of Compaq Fortran, including known problems and a summary of the Compaq Fortran run-time error messages.

  The Release Notes are located in:

  `/usr/lib/cmplrs/fort90/relnotes90`

- Compaq Fortran online reference pages

  Describe the Compaq Fortran software components, including `f95`(1), `f90`(1), `f77`(1), `fpr`(1), `fsplit`(1), `intro`(3f), numerous Fortran library routines listed in `intro`(3f), and numerous parallel Fortran library routines listed in `intro`(3hpf).

- Compaq Tru64 UNIX operating system documentation

  The operating system documentation set includes reference pages for operating system components and a programmer's subkit, in which certain

documents describe the commands, tools, libraries, and other aspects of the programming environment:

  - For programming information, see the *Compaq Tru64 UNIX Programmer's Guide* and the *Compaq Tru64 UNIX Using Programming Support Tools*.
  - For performance information, see the *Compaq Tru64 UNIX System Tuning and Performance*.
  - For an overview of Compaq Tru64 UNIX documentation, see the *Compaq Tru64 UNIX Reader's Guide*.

  For more information, see the Compaq Tru64 UNIX Web site at:

  `http://www.tru64unix.compaq.com/`

- Other layered product documentation

  If you are using a programming-related layered product package from Compaq, consult the appropriate documentation for the layered product package for use of that product.

- High Performance Fortran (HPF)

  See the High Performance Fortran Language Specification, available without charge at the following locations:

  - The HPF Web site at:

    `http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/index.cfm`

  - Anonymous FTP at `ftp.cs.rice.edu` in `/public/HPFF/draft`

- Third-party documentation

  If you are unfamiliar with OpenMP software and will be using OpenMP directives to control parallel execution of your program, Compaq recommends this book: *Parallel Programming in OpenMP* (Rohit Chandra *et. al.*, Morgan Kaufmann, 2000) is a comprehensive introduction to the compiler directives, run-time library routines, and environment variables that comprise OpenMP software. Its International Standard Book Number (ISBN) is 1-55860-671-8. More information about this book is on the Web site for Morgan Kaufmann Publishers at `http://www.mkp.com`.

Compaq Fortran Web Page

The Compaq Fortran home page is at:

`http://www.compaq.com/fortran`

This Web site contains information about software patch kits, example programs, and additional product information.

Communicating with Compaq

If you have a customer support contract and have comments or questions about Compaq Fortran software, you can contact our Customer Support Center (CSC), preferably using electronic means (such as DSNlink). Customers in the United States can call the CSC at 1-800-354-9000.

You can also send comments, questions, and suggestions about the Compaq Fortran product to the following e-mail address: `fortran@compaq.com`. Note that this address is for informational inquiries only and is not a formal support channel.

Conventions Used in This Document

This manual uses the conventions listed in Table 1. Also, example code — such as program `red_black_10.f90` in Section 1.4 — is usually in free source form (where a statement does not have to begin in position 7 of a line).

Table 1 Conventions Used in This Document

Convention	Meaning
%	This manual uses a percent sign (%) to represent the Tru64 UNIX system prompt. The actual user prompt varies with the shell in use.
% **pwd** /usr/usrc/jones	This manual displays system prompts and responses using a monospaced font. Typed user input is displayed in a bold monospaced font.
monospaced	This typeface indicates the name of a command, option, pathname, file name, directory path, or partition. This typeface is also used in examples of program code, interactive examples, and other screen displays.
cat(1)	A shell command name followed by the number 1 in parentheses refers to a command reference page. Similarly, a routine name followed by the number 2 or 3 in parentheses refers to a system call or library routine reference page. (The number in parentheses indicates the section containing the reference page.) To read online reference pages, use the man (1) command. Your operating system documentation also includes reference page descriptions.
newterm	Bold type indicates the introduction of a new term in text.
variable	Italic type indicates important information, a complete title of a manual, or variable information, such as user-supplied information in command or option syntax.
UPPERCASE lowercase	The operating system shell differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
{ }	Large braces enclose lists from which you must choose one item. For example: { STATUS DISPOSE DISP }
[ ]	Square brackets enclose items that are optional. For example: BLOCK DATA [nam]

(continued on next page)

Table 1 (Cont.) Conventions Used in This Document

Convention	Meaning
. . .	A horizontal ellipsis means that the item preceding the ellipsis can be repeated. For example: s[,s] . . .
. . .	A vertical ellipsis in a figure or example means that not all of the statements are shown.
real	This term refers to all floating-point intrinsic data types as a group.
complex	This term refers to all complex floating-point intrinsic data types as a group.
logical	This term refers to all logical data types as a group.
integer	This term refers to all integer data types as a group.
Alpha Alpha systems	The terms Alpha and Alpha systems refer to the Alpha architecture or systems equipped with this 64-bit architecture.
Compaq Tru64 UNIX Compaq Tru64 UNIX systems	The terms Compaq Tru64 UNIX and Compaq Tru64 UNIX systems refer to the Compaq Tru64 UNIX (formerly DIGITAL UNIX) operating system running on Alpha processor hardware.
Fortran	This term refers to language information that is common to ANSI FORTRAN 77, ANSI/ISO Fortran 95/90, and Compaq Fortran.
Fortran 95/90	This term refers to language information that is common to ANSI/ISO Fortran 95 and ANSI/ISO Fortran 90.
`f90`	This command invokes the Compaq Fortran compiler on Tru64 UNIX Alpha systems.
Compaq Fortran 77 DEC Fortran	The term Compaq Fortran 77 (formerly DEC Fortran) refers to language information that is common to the FORTRAN-77 standard and any Compaq Fortran extensions.
Compaq Fortran	The term Compaq Fortran (formerly DIGITAL Fortran 90) refers to language information that is common to the Fortran 95/90 standards and any Compaq Fortran extensions.

1

Compaq Fortran Parallel Processing: An Introduction

This chapter describes:

- Section 1.1, Overview of Parallel Processing
- Section 1.2, Applying Amdahl's Law To Determine Whether To Convert a Serial Program to a Parallel One
- Section 1.3, Example of Selecting Serial or Parallel Processing
- Section 1.4, Serial Example Program red_black_10
- Section 1.5, Architectures That Can Implement Parallel Processing
- Section 1.6, OpenMP Directives on an SMP System: Parallel Program red_black_20

1.1 Overview of Parallel Processing

The fundamental premise of parallel processing is that running a program on multiple processors is faster than running the same program on a single processor. The multiple processors share the work of executing the code.

For appropriate applications, parallel programs can execute dramatically faster than ordinary serial programs. To achieve this desired speed-up, the program must be decomposed so that different data and instructions are distributed among the processors to achieve simultaneous execution.

A further advantage of parallel processing is that a system can be scaled or built up gradually. If, over time, a parallel system becomes too small for the tasks needed, additional processors can be added to meet the new requirements with few or no changes to the source programs and the associated compiler commands.

Ideally, the performance gain of parallel operations should be proportional to the number of processors participating in the computation. In some special cases, the gain is even greater, due to the fact that two processors have twice as much cache memory as one processor. In most cases, however, the gain is somewhat less, because parallel processing inevitably requires a certain amount of communication between processors and synchronization overhead. Minimizing communications costs and idle time among processors is the key to achieving optimized parallel performance.

1.2 Applying Amdahl's Law To Determine Whether To Convert a Serial Program to a Parallel One

One way to determine whether or not a serial Fortran program should be converted to a parallel one is to apply Amdahl's Law. This principle, formalized by computer scientist Gene Amdahl in the 1960s, says that the potential speed-up of the serial program depends on two factors:

- The fraction of execution time that can occur in parallel mode. This number is always less than 1.0, since some execution time must occur in serial mode. For example, a DO loop requires certain setup operations that cannot be done in parallel mode. (The loop's iterations often can be done in parallel mode on more than one processor.)
- The number of processors.

If the fraction of execution time that can occur in parallel mode is p and the number of processors is N, then Amdahl's Law becomes:

$$\text{Speedup}(N,p) = \frac{1}{p/N + 1-p}$$

For example, suppose that the number of processors is 8 and that 60% of a serial program's run-time execution can occur in parallel mode. Then the potential speed-up of this programming environment is:

$$\frac{1}{0.6/8 + 1-0.6} = \frac{1}{0.075 + 0.4} = \frac{1}{0.475} = 2.11$$

If p remains at 0.6 and N doubles to 16, then the potential speedup increases to 2.29. In this case, suppose that a serial program requires 4 hours to execute. The parallel version executes in about (1/2.29 * 4) hours = 1.75 hours.

Figure 1–1 shows Amdahl's Law for various values of p (the fraction of a program that executes in parallel mode) and N (the number of run-time processors).

Figure 1–1 Amdahl's Law: Potential Speedup of Serial Programs

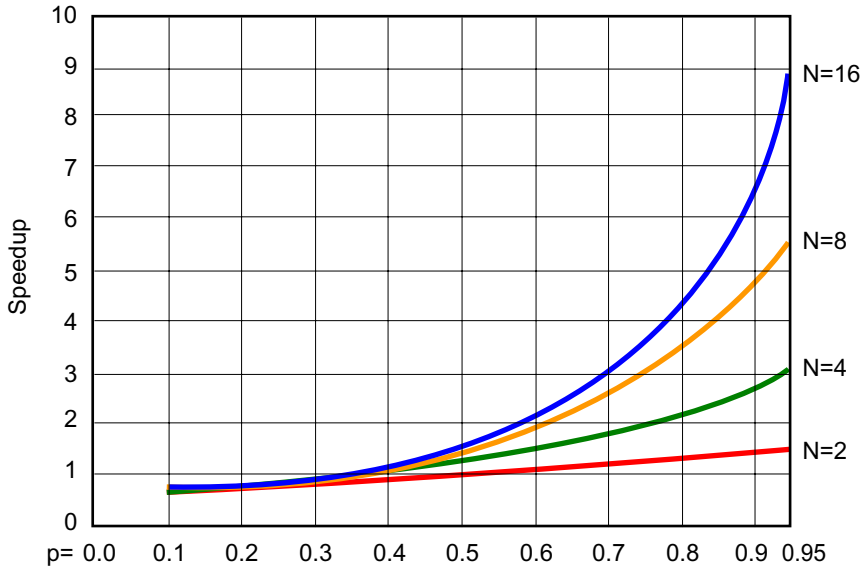


Figure 1–1 shows that relatively little speedup occurs, regardless of the value of N, until p is at least 0.8. Speedup occurs rapidly as p increases from 0.8. Although p never reaches 1.0, the theoretical values of Speedup(2,1), Speedup(4,1), Speedup(8,1), and Speedup(16,1) are 2, 4, 8, and 16, respectively.

As a result of Amdahl's Law, you might adopt a general guideline of not attempting to convert a serial program to a parallel one unless p >= 0.8. Of course, you might find it worthwhile to use a different value of p depending on your circumstances. From Figure 1–1, a serial program that spends 70% of its time executing a few DO loops could, after conversion, run three times faster on a 16-processor system. If the number Speedup(16,0.7)=3.0 saves a large amount of processing time, the conversion effort is probably worthwhile.

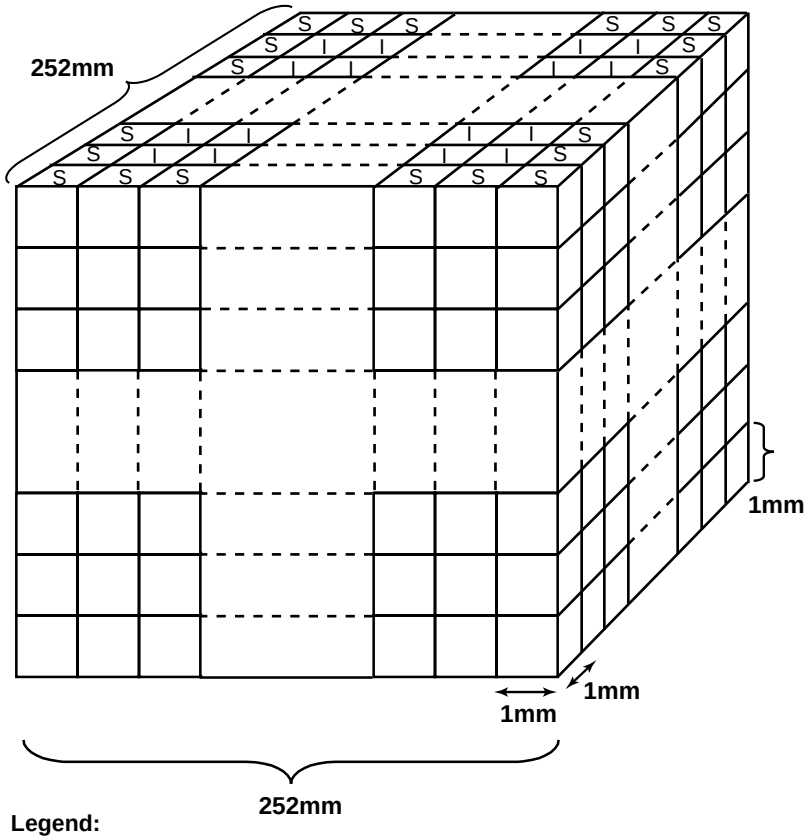
You might need to create an instrumented version of the serial program to find the fraction of elapsed time that its DO loops consume.

1.3 Example of Selecting Serial or Parallel Processing

This example, a typical Fortran serial program, begins by considering a metal cube that is 252 mm wide by 252 mm long by 252 mm high. We make a model of the cube with a grid that divides the cube into 16,003,008 cubes that are 1 mm on a side. The 378,008 cubes along the six sides have an initial temperature of 20 degrees Celsius while the 15,625,000 interior cubes have an initial temperature of 15 degrees Celsius.

Figure 1–2 shows a picture of the metal cube with these contained cubes and temperatures.

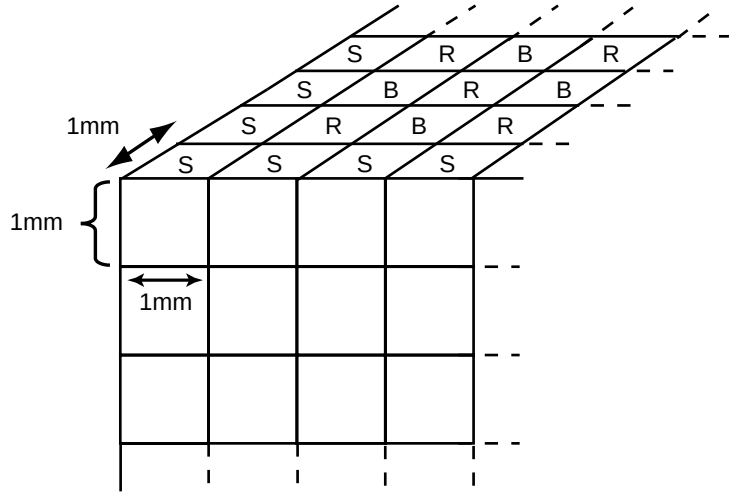
Figure 1–2 Metal Cube with Initial Temperatures



Legend:
S - Edge cube with initial temperature of 20 degrees.
I - Interior cube with initial temperature of 15 degrees.

The interior cubes are arbitrarily colored like a checkerboard, alternating red and black, with the upper left interior cube being red as shown in Figure 1–3.

Figure 1–3 Upper Left Portion of Metal Cube



Legend:

S - Side cube with initial temperature of 20 degrees.
R - Interior cube colored red, with initial temperature of 15 degrees.
B - Interior cube colored black, with initial temperature of 15 degrees.

Heat, constantly applied to the side cubes (labeled S in Figure 1–2 and Figure 1–3), flows to the interior until all the cubes reach their final temperature of 20 degrees.

One reasonable mathematical model of this thermodynamic physical situation says that, after a small amount of time, the new temperature of an interior cube is equal to the average of its upper, lower, north, south, west, and east neighbors' temperatures. This model states that a red cube's new temperature depends only on those of its black (or side) neighbors and a black cube's new temperature depends only on those of its red (or side) neighbors.

The corresponding Fortran expression of the mathematics of a single cube's new temperature is the following, where CUBE(L,M,N) is the temperature of an interior element of the cube:

```
CUBE(L,M,N) = ( CUBE(L+1,M,N) + CUBE(L-1,M,N)   + &
                CUBE(L,M+1,N) + CUBE(L,M-1,N)   + &
                CUBE(L,M,N+1) + CUBE(L,M,N-1) ) * (1.0/6.0)
```

In order to find all the new temperatures of the interior cubes after a short period of time, a Fortran program can make eight passes through array CUBE. The following letters represent part of Figure 1–3 where S is a side cube, R is a red cube, and B is a black cube.

```
S S S . . . .
S R B . . . .
S B R . . . .
. . . .
. . . .
. . . .
```

Each of the eight red or black cubes (in the upper left corner of the interior) is the anchor of a set of 15625000/8 = 2250000 cubes. An anchor cube is the first one whose new temperature is calculated by a pass through the array. If CUBE is declared as a single precision floating-point array whose dimensions are 252 by 252 by 252 via a statement including CUBE(0:251, 0:251, 0:251), then:

- CUBE(0,0,0) is the upper left cube of the top plane
- CUBE(1,1,1) is the red anchor cube for the first pass through the array
- CUBE(1,2,2) is the red anchor cube for the second pass through the array
- CUBE(2,1,2) is the red anchor cube for the third pass through the array
- CUBE(2,2,1) is the red anchor cube for the fourth pass through the array
- CUBE(1,1,2) is the black anchor cube for the fifth pass through the array
- CUBE(1,2,1) is the black anchor cube for the sixth pass through the array
- CUBE(2,1,1) is the black anchor cube for the seventh pass through the array
- CUBE(2,2,2) is the black anchor cube for the eighth pass through the array

The DO loops that will find the new temperatures of the 1953125 cubes anchored by CUBE(1,1,1) during the first pass through the array are (where variable ONE_SIXTH equals 1.0/6.0):

```
DO K = 1, N, 2
   DO J = 1, N, 2
      DO I = 1, N, 2
         CUBE(I,J,K) = ( CUBE(I-1,J,K) + CUBE(I+1,J,K)   + &
                         CUBE(I,J-1,K) + CUBE(I,J+1,K)   + &
                         CUBE(I,J,K-1) + CUBE(I,J,K+1) ) * ONE_SIXTH
      END DO
   END DO
END DO
```

After the first pass, CUBE(1,1,1) will have the value
(20.0 + 15.0 + 20.0 + 15.0 + 20.0 + 15.0) * 1.0/6.0 = 17.5.

The computations in this pair of DO loops can execute in parallel because there are no data dependencies. Recall that a red cube's new temperature depends only on those of its neighboring black (or side) cubes and not on those of any other red cubes.

The program has to decide when to stop by measuring the difference between any two complete passes through array CUBE. The measurement is the sum of the squares of the 16003008 differences. If this sum is less than 0.1, then the program stops.

1.4 Serial Example Program red_black_10

Example 1–1 shows a listing of program `red_black_10.f90`. It contains eight sets of DO statements that will execute in serial mode. As you read the program, note that the current temperatures of the cube are in an array named `X` (instead of CUBE) and that the corresponding previous temperatures are in an array named `X_old`.

Example 1–1 Serial Program red_black_10.f90

```
program red_black_10
integer, parameter           :: n=250           ! 252 x 252 x 252 array
integer, parameter           :: niters=1000     ! display results every
                                                !    1000 iterations
integer, parameter           :: maxiters=200000 ! maximum number
                                                !   of iterations
real, parameter              :: tol = 0.1       ! tolerance
real, parameter              :: one_sixth = (1.0 / 6.0)
real, dimension(0:n+1,0:n+1,0:n+1) :: x         ! current temperatures
real, dimension(0:n+1,0:n+1,0:n+1) :: x_old     ! previous temperatures
integer                      :: count           ! of all iterations
real                         :: start, elapsed, error
integer                      :: i, j, k, iters
```

(continued on next page)

Example 1–1 (Cont.) Serial Program red_black_10.f90

```
! Initialize array x by setting the side elements to 20.0 and
! the n**3 interior elements to 15.0
        do k=0, n+1
           do j=0, n+1
              do i=0, n+1
                 if (i.eq.0   .or. j.eq.0   .or. k.eq.0 .or. &
                     i.eq.n+1 .or. j.eq.n+1 .or. k.eq.n+1) then
                    x(i,j,k) = 20.0
                 else
                    x(i,j,k) = 15.0
                 endif
              end do
           end do
        end do

        print "(A)", ""
        print "(A,i4,A,i4,A,i4,A)", "Starting ",n," x",n," x",n," red-black"
        print "(A)", ""

        x_old = x
        count = 0
        error = huge(error)

! Main loop:
        start = SECNDS(0.0)

        print "(A,2f9.5)", &
              "Initial values of x(125,125,0) and x(125,125,125) are", &
                                 x(125,125,0), x(125,125,125)
        print "(A)", ""

        do while (error > tol)
           do iters = 1, niters

              ! Do red iterations starting at x(1,1,1)
              do k = 1, n, 2
                 do j = 1, n, 2
                    do i = 1, n, 2
                       x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                          + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
                    end do
                 end do
              end do
```

(continued on next page)

Example 1–1 (Cont.) Serial Program red_black_10.f90

```
! Do red iterations starting at x(1,2,2)
do k = 2, n, 2
   do j = 2, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do

! Do red iterations starting at x(2,1,2)
do k = 2, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do

! Do red iterations starting at x(2,2,1)
do k = 1, n, 2
   do j = 2, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do

! Do black iterations starting at x(1,1,2)
do k = 1, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
```

(continued on next page)

Example 1–1 (Cont.) Serial Program red_black_10.f90

```
            ! Do black iterations starting at x(1,2,1)
            do k = 1, n, 2
               do j = 2, n, 2
                  do i = 1, n, 2
                     x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                        + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
                  end do
               end do
            end do

            ! Do black iterations starting at x(2,1,1)
            do k = 1, n, 2
               do j = 1, n, 2
                  do i = 2, n, 2
                     x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                        + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
                  end do
               end do
            end do

            ! Do black iterations starting at x(2,2,2)
            do k = 2, n, 2
               do j = 2, n, 2
                  do i = 2, n, 2
                     x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                        + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
                  end do
               end do
            end do

         end do
         count = count + niters
         print "(A,2f9.5)", &
               "Current values of x(125,125,0) and x(125,125,125) are", &
                                 x(125,125,0), x(125,125,125)
         if (count > maxiters) exit
         error = sqrt(sum(abs(x-x_old)**2))
         print "(A,i6,A,f15.7)", "Iterations completed: ", count, &
                                 "   Relative error: ", error
         print "(A)", ""
         x_old = x
      end do
```

(continued on next page)

Example 1–1 (Cont.) Serial Program red_black_10.f90

```
        elapsed = SECNDS(start)
        print *, 'Number of iterations = ', count
        print *, 'Time elapsed = ', elapsed
        end
```

The compilation and execution commands for this program are:

```
% f90 -o red_black_10.exe red_black_10.f90
% red_black_10.exe > red_black_10.out
```

The output goes to file red_black_10.out for retrieval and display.

The contents of output file red_black_10.out follow:

```
Starting  250 x 250 x 250 red-black

Initial values of x(125,125,0) and x(125,125,125) are 20.00000 15.00000

Current values of x(125,125,0) and x(125,125,125) are 20.00000 15.00000
Iterations completed:   1000   Relative error:    1560.6927490

Current values of x(125,125,0) and x(125,125,125) are 20.00000 15.00000
Iterations completed:   2000   Relative error:       0.0000590

 Number of iterations =         2000
 Time elapsed =    ****.***
```

1.4.1 Analysis of Serial Example Program red_black_10

At this point program red_black_10 gives accurate results. An initial review of the output file from Example 1–1 indicates that most of the computation time is spent inside the eight DO loops. If we can determine that "most of the computation time" is equivalent to:

p >= 0.8

from Section 1.2, then converting red_black_10 from a serial program to a parallel one should show significant speedup.

We will analyze program red_black_10 to see where it spends its processing time. The following commands do this:

1. Compile and link to create executable and listing files:

   ```
   % f90 -o red_black_10.exe -V red_black_10.f90
   ```

The -V option creates a listing file named red_black_10.l. Its contents include the numbered lines in file red_black_10.f90. The most important numbered lines, from the eight DO loops, are shown with comments:

```
 52   x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
 53      + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth

 62   x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
 63      + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth

 72   x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
 73      + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth

 82   x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
 83      + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth

 92   x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
 93      + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth

102   x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
103      + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth

112   x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
113      + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth

122   x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
123      + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
```

2. Use the atom program to create a special version of the executable file:

   ```
   % atom -tool pixie red_black_10.exe
   ```

   This command creates an instrumented version of red_black_10.exe in executable file red_black_10.exe.pixie. It also creates file red_black_10.exe.Addrs with address information.

3. Run the program to obtain count information:

   ```
   % red_black_10.exe.pixie
   ```

   This command gives the same results as executing red_black_10.exe and creates file red_black_10.exe.Counts with count information.

4. Use the prof program to identify lines in the source program that result in large amounts of execution time.

   ```
   % prof -pixie red_black_10.exe
   ```

   This command runs the prof profiler program. It extracts and displays information from files red_black_10.exe.Addrs and red_black_10.exe.Counts. The following command runs prof to place the extracted information into file red_black_10.prof:

   ```
   % prof -pixie red_black_10.exe > red_black_10.prof
   ```

An extract from ASCII file red_black_10.prof follows:

```
line bytes      cycles      %  cum %

  53   636 36813256002   7.60   7.60
  63   624 36813254000   7.60  15.21
  73   624 36813254000   7.60  22.81
  83   624 36813254000   7.60  30.42
  93   624 36813254000   7.60  38.02
 103   624 36813254000   7.60  45.63
 113   624 36813254000   7.60  53.23
 123   624 36813254000   7.60  60.84
  52   372 21843750000   4.51  65.35
  62   372 21843750000   4.51  69.86
  72   372 21843750000   4.51  74.37
  82   372 21843750000   4.51  78.88
  92   372 21843750000   4.51  83.40
 102   372 21843750000   4.51  87.91
 112   372 21843750000   4.51  92.42
 122   372 21843750000   4.51  96.93
```

These 16 lines from file red_black_10.prof complete our analysis of program red_black_10.f90. The sixteenth line shows that the eight array assignment statements cumulatively account for more than 96% of the program's execution time. This is the same as saying, with reference to Section 1.2, that p >= 0.8. So, if we can make these statements execute in parallel, the program should execute significantly faster (given enough processors).

Note

Serial example program red_black_10.f90 serves solely as a foundation for conversion to programs that execute in parallel mode.

It is possible to rewrite it for faster execution. One way would be to combine some of the DO loops into a single loop.

You can access program red_black_10.f90 in the file /usr/lib/cmplrs/fort90/examples/red_black_10.f90.

1.5 Architectures That Can Implement Parallel Processing

The following three hardware architectures can execute parallel Compaq Fortran programs:

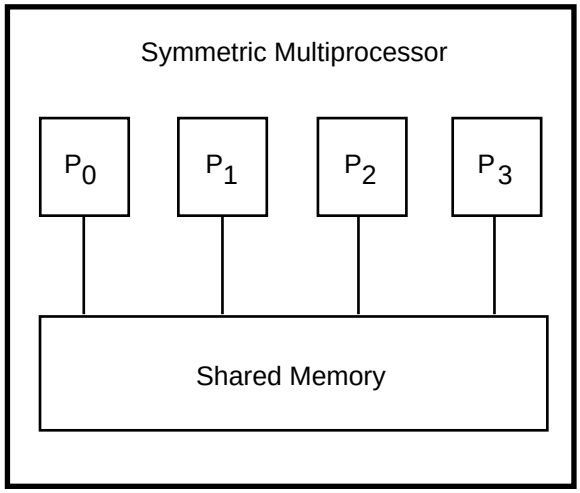
- Symmetric multiprocessor, or SMP (see Section 1.5.1)
- Non-uniform memory access, or NUMA (see Section 1.5.2)

- Distributed memory (see Section 1.5.3)

1.5.1 Symmetric Multiprocessor (SMP) Architecture

Figure 1–4 shows a typical configuration of a symmetric multiprocessor system. Processors P_0, P_1, P_2, and P_3 all share the same memory in this system. The figure illustrates an important principle of SMP systems: Each processor has equal access to all memory locations (ignoring cache effects). For this reason, another name for an SMP system is a Uniform Memory Access (UMA) system.

Figure 1–4 A Typical SMP System



One common method of creating parallel Fortran programs for SMP systems is inserting OpenMP directives. Section 1.6 introduces some of these directives and shows one way of inserting them into serial program `red_black_10.f90` in Example 1–1. The resulting parallel program, `red_black_20.f90`, is compiled and executed.

For an explanation of all OpenMP directives, see the *Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems*.

1.5.2 Non-Uniform Memory Access (NUMA) Architecture

Note

The NUMA architecture is an unsupported feature of Compaq Fortran Version 5.5.

AlphaServer™ GS80, GS160, and GS320 systems consist of one or more SMP modules. An interconnection switch joins multiple SMP modules.

Any processor can access any memory location that is in its local memory or in the memory of another SMP module. For large arrays, the amount of time for a processor to access an array element depends on the element's location:

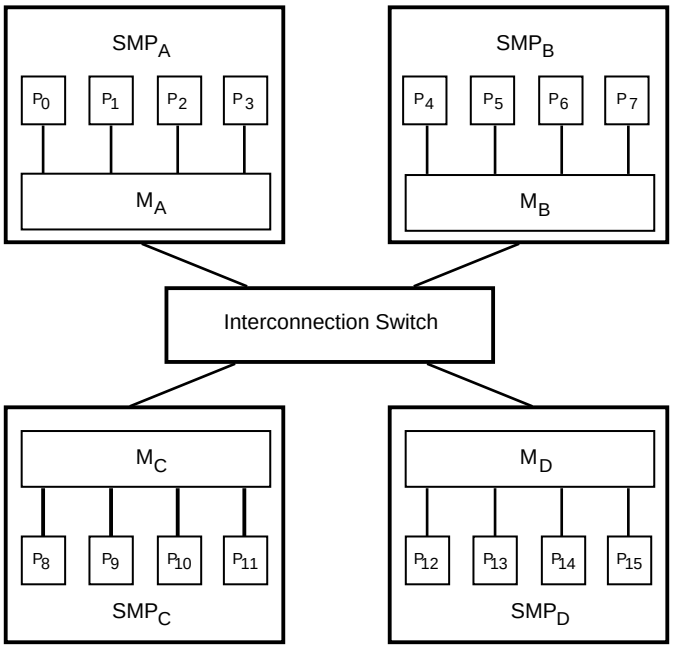
- If it is in the processor's local memory, then access time is small.
- If it is in another SMP's local memory, then access time is much larger.

Because of the difference in access times, AlphaServer GS systems consisting of multiple SMP modules are called Non-Uniform Memory Access (NUMA) systems.

Figure 1–5 shows the configuration of an AlphaServer GS160 NUMA system. In this figure:

- Each of four SMP modules has four processors. (An AlphaServer GS80 system has two SMP modules while an AlphaServer GS320 system has eight SMP modules.)
- Each SMP module has one memory that the four processors share. (While each SMP module always has one memory, the number of sharing processors can be 1, 2, 3, or 4.)
- Each processor can access any memory location within its own SMP module and any memory location on another SMP module.
- Accessing a local memory location is much faster than going through the interconnection switch to access a remote memory location on another SMP module.

Figure 1–5 A Typical NUMA System



Legend:
SMP - Symmetric Multiprocessor
P - Processor
M - Memory of a Symmetric Multiprocessor

VM-0613A-AI

A common method of creating parallel Fortran programs for NUMA systems is inserting OpenMP and data layout directives. In general, OpenMP directives distribute computations across processors; data layout directives place data in specified memories. The OMP NUMA directive, described in Section 3.3, requests the compiler to map iterations of an OpenMP-controlled DO loop to threads. These threads execute on processors local to the data being accessed. The effect of data layout directives is to have the threads access nearby data (on the same SMP) instead of remote data (on another SMP).

For More Information:

- See Chapter 3, NUMA Architecture: An Introduction.

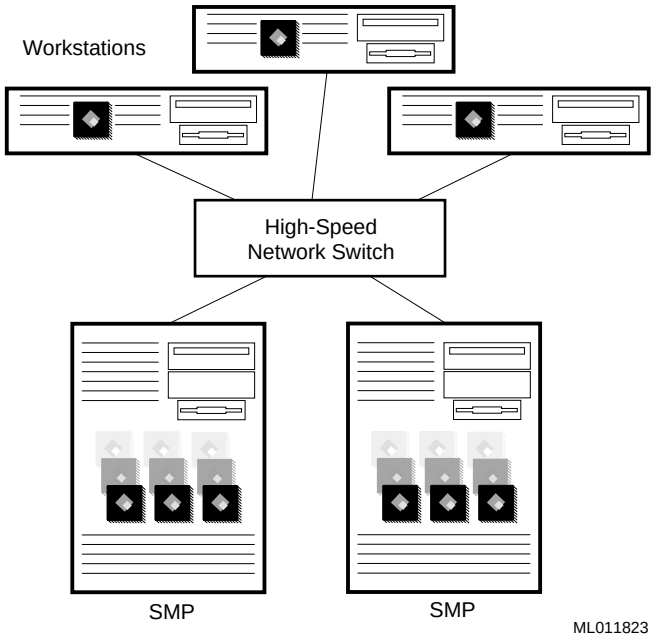
1.5.3 Distributed Memory Architecture

Distributed memory architecture is similar to NUMA architecture, because both can link SMP systems together — even though many distributed memory systems link together uniprocessors. The most significant difference is that communication between systems is slower with distributed memory architecture than with NUMA architecture. Also, the memory is not shared. This means you have to place explicit calls to message passing routines in your program or the compiler has to generate these calls. Distributed memory architectures rely on explicit message passing since the hardware does not support shared memory.

Figure 1–6 shows a typical configuration of a distributed memory system. In this figure:

- Each of the three workstations has one processor.
- Each of the two SMP systems has three processors.
- Each computer can access any memory location within itself and any memory location on another computer. Accessing a distant memory location requires passing a message.
- Accessing a local memory location is much faster than going across the network to access a distant memory location on another computer.

Figure 1–6 A Typical Distributed Memory System



The method described in this manual for creating parallel Fortran programs for distributed systems is to insert High Performance Fortran (HPF) directives.

Chapter 4 introduces HPF directives including DISTRIBUTE. It shows one way of inserting them into serial program `red_black_10.f90` in Example 1–1. The resulting parallel program, `red_black_50.f90`, is compiled and executed.

Chapter 5 contains a thorough explanation of all HPF directives.

1.6 OpenMP Directives on an SMP System: Parallel Program red_black_20

Example 1–2 shows a program named `red_black_20.f90` on an SMP system. The program is a result of the conversion of serial program `red_black_10.f90` in Example 1–1 to a parallel program using OpenMP directives.

Example 1–2 Parallel Program red_black_20.f90, Using OpenMP Directives on an SMP System

```
        program red_black_20
        integer, parameter           :: n=250           ! 252 x 252 x 252 array
        integer, parameter           :: niters=1000     ! display results every
                                                        !    1000 iterations
        integer, parameter           :: maxiters=200000 ! maximum number
                                                        !   of iterations
        real, parameter              :: tol = 0.1       ! tolerance
        real, parameter              :: one_sixth = (1.0 / 6.0)
        real, dimension(0:n+1,0:n+1,0:n+1) :: x         ! current temperatures
        real, dimension(0:n+1,0:n+1,0:n+1) :: x_old     ! previous temperatures
        integer                      :: count           ! of all iterations
        real                         :: start, elapsed, error
        integer                      :: i, j, k, iters

! Initialize array x by setting the side elements to 20.0 and
! the n**3 interior elements to 15.0
        do k=0, n+1
           do j=0, n+1
              do i=0, n+1
                 if (i.eq.0   .or. j.eq.0   .or. k.eq.0 .or. &
                     i.eq.n+1 .or. j.eq.n+1 .or. k.eq.n+1) then
                    x(i,j,k) = 20.0
                 else
                    x(i,j,k) = 15.0
                 endif
              end do
           end do
        end do

        print "(A)", ""
        print "(A,i4,A,i4,A,i4,A)", "Starting ",n," x",n," x",n," red-black"
        print "(A)", ""

        x_old = x
        count = 0
        error = huge(error)

! Main loop:
        start = SECNDS(0.0)

        print "(A,2f9.5)", &
              "Initial values of x(125,125,0) and x(125,125,125) are", &
                               x(125,125,0), x(125,125,125)
        print "(A)", ""

        do while (error > tol)
           do iters = 1, niters
              ! Beginning of a parallel region
              !$omp parallel private(i,j,k)
```

(continued on next page)

Example 1–2 (Cont.) Parallel Program red_black_20.f90, Using OpenMP Directives on an SMP System

```
! Do red iterations starting at x(1,1,1)
!$omp do schedule(static)
do k = 1, n, 2
   do j = 1, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end do

! Do red iterations starting at x(1,2,2)
!$omp do schedule(static)
do k = 2, n, 2
   do j = 2, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end do

! Do red iterations starting at x(2,1,2)
!$omp do schedule(static)
do k = 2, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end do
```

(continued on next page)

Example 1–2 (Cont.) Parallel Program red_black_20.f90, Using OpenMP Directives on an SMP System

```
! Do red iterations starting at x(2,2,1)
!$omp do schedule(static)
do k = 1, n, 2
   do j = 2, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end do

! Do black iterations starting at x(1,1,2)
!$omp do schedule(static)
do k = 1, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end do

! Do black iterations starting at x(1,2,1)
!$omp do schedule(static)
do k = 1, n, 2
   do j = 2, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end do
```

(continued on next page)

Example 1–2 (Cont.) Parallel Program red_black_20.f90, Using OpenMP Directives on an SMP System

```
      ! Do black iterations starting at x(2,1,1)
      !$omp do schedule(static)
      do k = 1, n, 2
         do j = 1, n, 2
            do i = 2, n, 2
               x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                  + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
            end do
         end do
      end do
      !$omp end do

      ! Do black iterations starting at x(2,2,2)
      !$omp do schedule(static)
      do k = 2, n, 2
         do j = 2, n, 2
            do i = 2, n, 2
               x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                  + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
            end do
         end do
      end do
      !$omp end do

      ! End of the parallel region
      !$omp end parallel
   end do
   count = count + niters
   print "(A,2f9.5)", &
         "Current values of x(125,125,0) and x(125,125,125) are", &
                           x(125,125,0), x(125,125,125)
   if (count > maxiters) exit
   error = sqrt(sum(abs(x-x_old)**2))
   print "(A,i6,A,f15.7)", "Iterations completed: ", count, &
                          "   Relative error: ", error
   print "(A)", ""
   x_old = x
end do

elapsed = SECNDS(start)
print *, 'Number of iterations = ', count
print *, 'Time elapsed = ', elapsed
end
```

1.6.1 Explanation of Parallel Program red_black_20

Parallel program red_black_20.f90 differs from serial program red_black_10.f90 as follows:

- Within the do iters = 1, niters loop, the following pair of OpenMP directives is added. This pair defines a parallel region.

  ```
  !$omp parallel private(i,j,k)
  !$omp end parallel
  ```

  These directives cause a team of threads to execute the code in that region. They also give each thread a private copy of variables i and j and k. If no other directives were present to divide the work of executing the code into separate units and to assign each unit to a single thread, then all of the code in the parallel region would be executed redundantly by each thread in the team.

- In the parallel region, each of the eight DO loops that begin with do k = is surrounded by the following pair of OpenMP directives.

  ```
  !$omp do schedule(static)
  !$omp end do
  ```

  These directives partition the set of iterations for the loop into subsets. They also schedule each subset for execution by a single thread within the team of threads that the parallel directive created. The schedule (static) clause schedules the iterations of the DO loops in equal size chunks depending on the number of threads.

  For example, consider the four DO loops that begin with do j = 2, n, 2. Suppose that n is 302 instead of 252; then each loop executes 151 times. If the number of threads is set to 4 (see the setenv OMP_NUM_THREADS command below), then the size of each chunk is an integer near 151/4 = 37.75. These numbers mean that the first thread could process the 38 iterations for j = 2, 4, 6, ..., 76; the second thread could process the 38 iterations for j = 78, 80, 82, ..., 152; the third thread could process the 38 iterations for j = 154, 156, 158, ..., 228; and the fourth thread could process the 37 iterations for j = 230, 232, 234, ..., 302. Static scheduling is the default for DO loops identified by a !$OMP DO directive.

Typically, one thread on each processor executes selected iterations of a DO loop.

The environment, compilation, and execution commands for this program are shown below. The output goes to file red_black_20.out for retrieval and display. Assume that the SMP system is the one in Figure 1–4:

```
% setenv OMP_NUM_THREADS 4
% f90 -o red_black_20.exe -omp red_black_20.f90
% red_black_20.exe > red_black_20.out
```

The first and last parts of output file red_black_20.out are identical to those of file red_black_10.out described in Section 1.4.1 with one exception: the value in the Time elapsed line will almost certainly be different.

The -omp option is required. Without it, the compiler treats OpenMP directives in source file red_black_10.f90 as comments.

You can access program red_black_20.f90 in the file /usr/lib/cmplrs/fort90/examples/red_black_20.f90.

2
Data Layout: An Introduction

This chapter describes:

- Section 2.1, Overview of Data Layout Principles
- Section 2.2, User-Directed Data Migration

2.1 Overview of Data Layout Principles

Even experienced OpenMP programmers may have little experience with data layout principles. OpenMP directives, designed for SMP systems (see Figure 1–4), focus on computations and threads, with the assumption that the amount of time it takes a thread to access a memory location is constant. If your programs execute on NUMA systems, you should read this chapter to learn the basics of data placement so that your programs perform well. If your programs execute only on SMP systems, then this chapter does not apply.

Experienced HPF programmers place their programs' data onto processors so that (given an equal distribution of work across the processors) each processor requires about the same amount of time as any other processor. The use of the DISTRIBUTE directive is very important for this placement. Chapter 5 provides a full explanation of DISTRIBUTE and related HPF directives.

For the best performance of programs with OpenMP directives on NUMA systems, place data as close as possible to the threads that access it. In other words, try to avoid a processor's access of data across the interconnection switch in Figure 1–5. The data accessed by each thread must be near the processor on which the thread is executing. Usually the data is contained in large arrays.

Correct data placement minimizes the amount of time required for a thread to access a memory location.

For example, consider array A(200000,4000,1000) that will not fit completely into any of the memories M_A . . . M_D of Figure 1–5. Suppose that a program contains the following statement, executed by a particular thread:

```
TEMP = A(100000,1200,900)
```

Suppose also that this thread is executing on processor P_{14}. Consequently:

- If array element A(100000,1200,900) is in local memory M_D, then variable TEMP receives its value quickly because time-consuming communication across the interconnection switch does not occur.
- If array element A(100000,1200,900) is not in local memory M_D, then variable TEMP receives its value slowly because time-consuming communication across the interconnection switch (to another memory) must occur.

Getting data close to the threads that access it (that is, getting threads and related data into the same local memory) is basically a two-step process:

1. In your programs, give directives that optimally distribute the data (usually in large arrays) over the memories of a NUMA system.
2. In your programs, give directives that assign threads to the processors whose memories contain the distributed data.

The process occurs in two ways:

- User-directed data migration

  The thread/data connection occurs at run time. See Section 2.2, User-Directed Data Migration and example program `red_black_30.f90` in Section 3.1.

- Manual data placement

  The thread/data connection occurs mostly at compilation time. See example program `red_black_40.f90` in Section 3.2.

For both ways, directives in a source program establish the thread/data connection. The user-directed migration directive is an executable one. The manual placement directive is static with the actual data placement occurring at run time.

2.2 User-Directed Data Migration

Many programs, such as red_black_10.f90 in Section 1.4, have a small number of DO loops that account for a large percentage of execution time. However, when the iterations of the DO loops execute at run time, there is a problem. The threads corresponding to the iterations do not automatically access array elements that are in the same local memory as the module on which the thread is executing.

For example, consider the following program fragment:

```
REAL X(12000000000)    ! Twelve billion elements
INTEGER*8 :: I         ! Access all elements of array X
!$OMP PARALLEL PRIVATE(I)
!$OMP DO SCHEDULE(STATIC)
DO I = 1, 12000000000  ! Twelve billion iterations
   X(I) = SQRT(FLOAT(I)) + SIN(FLOAT(I)) + ALOG(FLOAT(I))
END DO
!$OMP END DO
!$OMP END PARALLEL
```

If it executes on the NUMA system in Figure 1–5 with environment variable OMP_NUM_THREADS set to 16, then each thread contains 12000000000/16 = 750000000 iterations. Suppose that the first thread resides on processor P_0, the second thread resides on processor P_1, . . . , and the sixteenth thread resides on processor P_{15}. Also suppose that the elements of array X fit into memory M_A. Then:

- The first four threads, on processors P_0 through P_3, contain instructions that access data in memory M_A. All data accesses are local — and fast since the interconnection switch is not used.
- The next 12 threads, on processors P_4 through P_{15}, contain instructions that access data in memory M_A. All data accesses are remote — and less fast since the interconnection switch is used.

We want to change the distribution of array X's elements so that they reside in all four memories instead of only in memory M_A. Furthermore, each element should reside in the same memory as the thread whose instructions access the element.

The directives we can use for user-directed data migration are:

- MIGRATE_NEXT_TOUCH (see Section 2.2.1)
- MIGRATE_NEXT_TOUCH_NOPRESERVE (see Section 2.2.2)

Section 3.4 contains expansions of this program fragment into two complete programs.

2.2.1 MIGRATE_NEXT_TOUCH Directive

The MIGRATE_NEXT_TOUCH directive provides a simple way to move pages of data to the memories where threads are accessing those pages. This movement ensures that a thread has a page that it is using in its local memory.

The MIGRATE_NEXT_TOUCH directive takes the following form:

```
!DEC$ MIGRATE_NEXT_TOUCH(var1, var2, ... , varn)
```

In the directive, var_1 through var_n are variables that are usually arrays occupying many pages of memory. Each variable specifies a set of pages that are to be migrated, that is, moved to a new location in physical memory. The set includes every page that contains any byte of the storage for that variable.

Whenever program execution reaches a MIGRATE_NEXT_TOUCH directive at run time, the set of pages for each variable is marked for migration. After a page is marked, the next time a thread references that page, it causes a page fault to occur. The operating system then migrates the page to the memory which the referencing thread is executing on. Finally the operating system unmarks the page, and execution continues with the page in its new location.

If the referencing thread already has the page in its local memory, then no page migration occurs.

In the current example, we could insert the following MIGRATE_NEXT_TOUCH directive just before the DO loop:

```
!DEC$ MIGRATE_NEXT_TOUCH(X)
```

In summary, the MIGRATE_NEXT_TOUCH directive causes the next thread that uses a set of pages to pull those pages near itself. Later references by the same thread will have fast local access to those pages.

Also, moving pages does require some time so it is important to use the directive carefully. For example, placing the MIGRATE_NEXT_TOUCH directive inside an innermost DO loop would likely have a negative effect on performance. For another example, the absence of a sustained association between threads and the data they reference may result in unacceptable overhead as pages repeatedly migrate.

An alternative to repeated run-time movement of pages is manually placing data onto memories by including data distribution directives in a source program.

Note that each of the threads (given a STATIC schedule) will access data that is almost entirely disjoint. That is, there is a sustained association between threads and the data that they access. As a result of this association, the effect of the MIGRATE_NEXT_TOUCH directive is (in this example) to get the pages in the right place so that almost all accesses will be local ones.

2.2.2 MIGRATE_NEXT_TOUCH_NOPRESERVE Directive

The MIGRATE_NEXT_TOUCH_NOPRESERVE directive is similar to the MIGRATE_NEXT_TOUCH directive. Three important differences exist:

- Although it moves the location of pages, the MIGRATE_NEXT_TOUCH_NOPRESERVE directive does not copy the contents of the pages to the new location. For this reason the MIGRATE_NEXT_TOUCH_NOPRESERVE directive is only suitable for situations in which the contents of the specified variables are no longer needed. That is, the contents of these variables will be overwritten before they are read.
- The MIGRATE_NEXT_TOUCH_NOPRESERVE directive affects only those pages that are entirely contained within the storage for the specified variables. For example, suppose that var_1 occupies part of memory page 3001, all of memory pages 3002 through 3038, and part of memory page 3039. Then only pages 3002 through 3038 are moved.
- The MIGRATE_NEXT_TOUCH_NOPRESERVE directive is useful for distributing arrays that are about to be overwritten with new values. Because it does not need to move the contents of pages, the MIGRATE_NEXT_TOUCH_NOPRESERVE directive is typically more efficient than the MIGRATE_NEXT_TOUCH directive.

In summary, the MIGRATE_NEXT_TOUCH_NOPRESERVE directive moves the physical location of a set of pages to the memory where the next reference occurs, without copying the pages' contents.

The MIGRATE_NEXT_TOUCH_NOPRESERVE directive takes the following form:

```
!DEC$ MIGRATE_NEXT_TOUCH_NOPRESERVE(var1, var2, ... , varn)
```

In the directive, var_1 through var_n are variables that are usually arrays occupying many pages of memory.

3

NUMA Architecture: An Introduction

This chapter describes:

- Section 3.1, OpenMP and Page Migration Directives on a NUMA System: Parallel Program red_black_30
- Section 3.2, OpenMP and Data Layout Directives on a NUMA System: Parallel Program red_black_40
- Section 3.3, Restrictions on OpenMP Features With the !DEC$ OMP NUMA Directive
- Section 3.4, Two Short but Complete Example Programs
- Section 3.5, Specifying Memories and Threads per Memory

Note

The NUMA architecture is an unsupported feature of Compaq Fortran 5.5.

3.1 OpenMP and Page Migration Directives on a NUMA System: Parallel Program red_black_30

Example 3–1 shows a program named `red_black_30.f90` on a NUMA system. The program is a result of the conversion of serial program `red_black_10.f90` in Example 1–1 to a parallel program using OpenMP directives and data migration directives.

Example 3–1 Program red_black_30.f90

```
        program red_black_30
        integer, parameter           :: n=250           ! 252 x 252 x 252 array
        integer, parameter           :: niters=1000     ! display results every
                                                        !    1000 iterations
        integer, parameter           :: maxiters=200000 ! maximum number
                                                        !   of iterations
        real, parameter              :: tol = 0.1       ! tolerance
        real, parameter              :: one_sixth = (1.0 / 6.0)
        real, dimension(0:n+1,0:n+1,0:n+1) :: x         ! current temperatures
        real, dimension(0:n+1,0:n+1,0:n+1) :: x_old     ! previous temperatures
        integer                      :: count           ! of all iterations
        real                         :: start, elapsed, error
        integer                      :: i, j, k, iters

! Initialize array x by setting the side elements to 20.0 and
! the n**3 interior elements to 15.0
        do k=0, n+1
           do j=0, n+1
              do i=0, n+1
                 if (i.eq.0   .or. j.eq.0   .or. k.eq.0 .or. &
                     i.eq.n+1 .or. j.eq.n+1 .or. k.eq.n+1) then
                    x(i,j,k) = 20.0
                 else
                    x(i,j,k) = 15.0
                 endif
              end do
           end do
        end do

        print "(A)", ""
        print "(A,i4,A,i4,A,i4,A)", "Starting ",n," x",n," x",n," red-black"
        print "(A)", ""

        x_old = x
        count = 0
        error = huge(error)

! Main loop:
        start = SECNDS(0.0)

        print "(A,2f9.5)", &
              "Initial values of x(125,125,0) and x(125,125,125) are", &
                               x(125,125,0), x(125,125,125)
        print "(A)", ""
```

(continued on next page)

Example 3–1 (Cont.) Program red_black_30.f90

```
! Migrate pages of x near the next thread that touches them
!dec$ migrate_next_touch(x)
do while (error > tol)
   do iters = 1, niters
      ! Beginning of a parallel region
      !$omp parallel private(i,j,k)

      ! Do red iterations starting at x(1,1,1)
      !$omp do schedule(static)
      do k = 1, n, 2
         do j = 1, n, 2
            do i = 1, n, 2
               x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                  + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
            end do
         end do
      end do
      !$omp end do

      ! Do red iterations starting at x(1,2,2)
      !$omp do schedule(static)
      do k = 2, n, 2
         do j = 2, n, 2
            do i = 1, n, 2
               x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                  + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
            end do
         end do
      end do
      !$omp end do

      ! Do red iterations starting at x(2,1,2)
      !$omp do schedule(static)
      do k = 2, n, 2
         do j = 1, n, 2
            do i = 2, n, 2
               x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                  + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
            end do
         end do
      end do
      !$omp end do
```

(continued on next page)

Example 3–1 (Cont.) Program red_black_30.f90

```
! Do red iterations starting at x(2,2,1)
!$omp do schedule(static)
do k = 1, n, 2
   do j = 2, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end do

! Do black iterations starting at x(1,1,2)
!$omp do schedule(static)
do k = 1, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end do

! Do black iterations starting at x(1,2,1)
!$omp do schedule(static)
do k = 1, n, 2
   do j = 2, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end do

! Do black iterations starting at x(2,1,1)
!$omp do schedule(static)
do k = 1, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end do
```

(continued on next page)

Example 3–1 (Cont.) Program red_black_30.f90

```
            ! Do black iterations starting at x(2,2,2)
            !$omp do schedule(static)
            do k = 2, n, 2
               do j = 2, n, 2
                  do i = 2, n, 2
                     x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                        + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
                  end do
               end do
            end do
            !$omp end do

            ! End of the parallel region
            !$omp end parallel
         end do
         count = count + niters
         print "(A,2f9.5)", &
               "Current values of x(125,125,0) and x(125,125,125) are", &
                              x(125,125,0), x(125,125,125)
         if (count > maxiters) exit
         error = sqrt(sum(abs(x-x_old)**2))
         print "(A,i6,A,f15.7)", "Iterations completed: ", count, &
                               "   Relative error: ", error
         print "(A)", ""
         x_old = x
      end do

      elapsed = SECNDS(start)
      print *, 'Number of iterations = ', count
      print *, 'Time elapsed = ', elapsed
      end
```

3.1.1 Explanation of Parallel Program red_black_30

Parallel program red_black_30.f90 differs from parallel program red_black_20.f90 described in Example 1–2 in only one way. In the main loop, before the statement:

```
do while (error > tol)
```

the following directive is added:

```
 !dec$ migrate_next_touch(x)
```

If this program executes on a NUMA system, then the MIGRATE_NEXT_TOUCH directive is in effect for array X(0:251,0:251,0:251). At run time, pages of array X are copied from their locations to a memory near the first thread that accesses each page. This means that after the migration, the thread does not access array elements on that page by going across the interconnection switch in Figure 1–5. The interconnection switch is not used because of the movement of the page to the memory of the thread making the accesses. Of course, no page movement occurs if the page is already in the memory of the accessing thread.

For a full explanation of the MIGRATE_NEXT_TOUCH directive, see Section 2.2.1.

The output of this program goes to file `red_black_30.out` for retrieval and display. Assume that the NUMA system is the one in Figure 1–5. The first three `setenv` commands reflect this architecture since there are 16 threads (one thread for each processor), four memories, and four threads for each memory.

```
% setenv OMP_NUM_THREADS 16
% setenv NUMA_MEMORIES 4
% setenv NUMA_TPM 4
% f90 -o red_black_30.exe -omp -numa red_black_30.f90
% red_black_30.exe > red_black_30.out
```

The first and last parts of output file `red_black_30.out` are identical to those of file `red_black_10.out` described in Section 1.4.1 with one exception: The value in the `Time elapsed` line will almost certainly be different.

The `-omp` and `-numa` command-line options are required. Without the `-omp` option, the compiler treats OpenMP directives (such as !$omp end do) in source file `red_black_30.f90` as comments. Without the `-numa` option, the compiler treats NUMA-related directives (such as !dec$ migrate_next_touch(x)) in source file `red_black_30.f90` as comments.

The `-numa` option enables the other NUMA options (whose names begin with `-numa_`). You cannot specify the `-hpf` option along with the `-numa` option.

The `numa_memories` *n* option specifies how many RADs (which usually correspond to physical memory units) the program uses at run time. On NUMA machines such as the AlphaServer GS320 system, there are multiple physical memory units within a single system.

If the `f90` command does not contain the `-numa_memories` option, then the value of the `NUMA_MEMORIES` environment variable is the number of RADs the program uses at run time.

If the -numa_memories option does not appear and the NUMA_MEMORIES environment variable is not set, then the number is chosen at run time. Including -numa_memories 0 is the same as not including -numa_memories.

If the f90 command does not contain the -numa_tpm option, then the value of the NUMA_TPM environment variable is the number of threads per physical memory unit that will execute NUMA parallel features of a program at run time.

The letters tpm in the option -numa_tpm represent "threads per memory." This option specifies the number of threads per physical memory unit that will execute NUMA parallel features of a program at run time. If this option does not appear in the f90 command and the NUMA_TPM environment variable is not set, then the number of threads per memory created for NUMA parallel features is set at run time. This number will be the number of CPUs in the executing system divided by the number of physical memory units in the executing system.

Note

If you have a choice, use a compiler option instead of its corresponding environment variable. An option gives more information to the compiler and a faster-executing program often results.

You can access program red_black_30.f90 in the file /usr/lib/cmplrs/fort90/examples/red_black_30.f90.

3.2 OpenMP and Data Layout Directives on a NUMA System: Parallel Program red_black_40

Example 3–2 shows a program named red_black_40.f90 on a NUMA system. The program is a result of the conversion of serial program red_black_10.f90 in Example 1–1 to a parallel program using OpenMP directives and data layout directives.

Example 3–2 Parallel Program red_black_40.f90

```
        program red_black_40
        integer, parameter           :: n=250           ! 252 x 252 x 252 array
        integer, parameter           :: niters=1000     ! display results every
                                                        !    1000 iterations
        integer, parameter           :: maxiters=200000 ! maximum number
                                                        !   of iterations
        real, parameter              :: tol = 0.1       ! tolerance
        real, parameter              :: one_sixth = (1.0 / 6.0)
        real, dimension(0:n+1,0:n+1,0:n+1) :: x         ! current temperatures
        real, dimension(0:n+1,0:n+1,0:n+1) :: x_old     ! previous temperatures
        !dec$ distribute (*,*,block) :: x, x_old
        integer                      :: count           ! of all iterations
        real                         :: start, elapsed, error
        integer                      :: i, j, k, iters

! Initialize array x by setting the side elements to 20.0 and
! the n**3 interior elements to 15.0
        do k=0, n+1
           do j=0, n+1
              do i=0, n+1
                 if (i.eq.0   .or. j.eq.0   .or. k.eq.0 .or. &
                     i.eq.n+1 .or. j.eq.n+1 .or. k.eq.n+1) then
                    x(i,j,k) = 20.0
                 else
                    x(i,j,k) = 15.0
                 endif
              end do
           end do
        end do

        print "(A)", ""
        print "(A,i4,A,i4,A,i4,A)", "Starting ",n," x",n," x",n," red-black"
        print "(A)", ""

        x_old = x
        count = 0
        error = huge(error)

! Main loop:
        start = SECNDS(0.0)

        print "(A,2f9.5)", &
              "Initial values of x(125,125,0) and x(125,125,125) are", &
                               x(125,125,0), x(125,125,125)
        print "(A)", ""

        do while (error > tol)
           do iters = 1, niters
```

(continued on next page)

Example 3–2 (Cont.) Parallel Program red_black_40.f90

```
! Do red iterations starting at x(1,1,1)
!dec$ omp numa
!$omp parallel do private(i,j,k)
do k = 1, n, 2
   do j = 1, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end parallel do

! Do red iterations starting at x(1,2,2)
!dec$ omp numa
!$omp parallel do private(i,j,k)
do k = 2, n, 2
   do j = 2, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end parallel do

! Do red iterations starting at x(2,1,2)
!dec$ omp numa
!$omp parallel do private(i,j,k)
do k = 2, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end parallel do
```

(continued on next page)

Example 3–2 (Cont.) Parallel Program red_black_40.f90

```
! Do red iterations starting at x(2,2,1)
!dec$ omp numa
!$omp parallel do private(i,j,k)
do k = 1, n, 2
   do j = 2, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end parallel do

! Do black iterations starting at x(1,1,2)
!dec$ omp numa
!$omp parallel do private(i,j,k)
do k = 1, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end parallel do

! Do black iterations starting at x(1,2,1)
!dec$ omp numa
!$omp parallel do private(i,j,k)
do k = 1, n, 2
   do j = 2, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
!$omp end parallel do
```

(continued on next page)

Example 3–2 (Cont.) Parallel Program red_black_40.f90

```
        ! Do black iterations starting at x(2,1,1)
        !dec$ omp numa
        !$omp parallel do private(i,j,k)
        do k = 1, n, 2
           do j = 1, n, 2
              do i = 2, n, 2
                 x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                    + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
              end do
           end do
        end do
        !$omp end parallel do

        ! Do black iterations starting at x(2,2,2)
        !dec$ omp numa
        !$omp parallel do private(i,j,k)
        do k = 2, n, 2
           do j = 2, n, 2
              do i = 2, n, 2
                 x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
                    + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
              end do
           end do
        end do
        !$omp end parallel do

     end do
     count = count + niters
     print "(A,2f9.5)", &
           "Current values of x(125,125,0) and x(125,125,125) are", &
                           x(125,125,0), x(125,125,125)
     if (count > maxiters) exit
     error = sqrt(sum(abs(x-x_old)**2))
     print "(A,i6,A,f15.7)", "Iterations completed: ", count, &
                            "   Relative error: ", error
     print "(A)", ""
     x_old = x
  end do

  elapsed = SECNDS(start)
  print *, 'Number of iterations = ', count
  print *, 'Time elapsed = ', elapsed
  end
```

3.2.1 Explanation of Parallel Program red_black_40

Parallel program red_black_40.f90 differs from serial program red_black_10.f90 described in Example 1–1 as follows:

- In the data declarations, after the statements that define equal size arrays x and x_old, the following statement is added:

  ```
  !dec$ distribute (*,*,block)  ::  x, x_old
  ```

  This statement distributes the elements of the arrays across the memories. The * keyword in the first dimension keeps the elements of a column in the same memory. For example, the elements in the first column of array x are x(0,1,1), x(1,1,1), x(2,1,1), . . . , x(251,1,1); they are together in a memory. The * keyword in the second dimension keeps the elements of a row in the same memory. Together these two * keywords keep each plane of the arrays in the same memory. The block keyword in the third dimension distributes the set of planes over different memories. Thus these keywords combine to divide the arrays x and x_old into two dimensional planes. The first plane goes to one memory, the second plane goes to the next memory, the last plane goes to the last memory, and so forth.

  How does the compiler know how many planes to divide the arrays x and x_old onto? The answer is the number of memories in the NUMA system. This example uses the -numa_memories *n* command-line option to specify the number.

- In the parallel region, each of the eight DO loops starting with do k = begins with the following pair of directives:

  ```
  !dec$ omp numa
  !$omp parallel do private(i,j,k)
  ```

  and ends with the following directive:

  ```
  !$omp end parallel do
  ```

  These directives tell the compiler to schedule the iterations of the following do loop onto threads that are executing on the same modules (that is, SMPs) as the data that the threads access. Each thread will have its own copies of loop variables i and j and k.

The output of this program goes to file red_black_40.out for retrieval and display. Assume that the NUMA system is the one in Figure 1–5.

```
% f90 -o red_black_40.exe \
-omp -numa -numa_memories 4 -numa_tpm 4 red_black_40.f90
% red_black_40.exe > red_black_40.out
```

The first and last parts of output file red_black_40.out are identical to those of file red_black_10.out at the end of Section 1.4, with one exception: The value in the Time elapsed line will almost certainly be different.

The -omp option is required. Without it, the compiler treats OpenMP directives in source file red_black_40.f90 as comments. Similarly, the -numa option is required. Without it, the compiler treats the NUMA-related directives !dec$ omp numa) in source file red_black_40.f90 as comments.

The -numa_tpm 4 option, where tpm represents "threads per memory," reflects the NUMA system in Figure 1–5. Each memory has four local processors and you should assign one thread to each processor.

Note that a statement such as

```
% setenv OMP_NUM_THREADS 16
```

does *not* accompany the previous compilation and execution commands. Section 3.3 explains that the OMP NUMA directive results in overriding the value of environment variable OMP_NUM_THREADS.

You can access program red_black_40.f90 in the file /usr/lib/cmplrs/fort90/examples/red_black_40.f90.

3.3 Restrictions on OpenMP Features With the !DEC$ OMP NUMA Directive

The !DEC$ OMP NUMA directive has several effects on the way that programs with OpenMP directives execute. In order that iterations are executed by a thread on the memory containing the data being accessed, it is necessary to bind each thread to a memory. It is also necessary to have at least one thread bound to each of the memories that contain data to be accessed.

Because of these and other requirements that occur when the compiler generates code for NUMA parallel loops, the !DEC$ OMP NUMA directive modifies the behavior of some OpenMP directives. It also imposes some restrictions on the features of OpenMP that may be used with it. These modifications and restrictions are:

- NUMA cannot be used with separate PARALLEL and DO directives.

  You can use the !DEC$ OMP NUMA directive only with the combined form of !$OMP PARALLEL DO directive. You cannot use it with separate !$OMP PARALLEL and !$OMP DO directives.

This restriction occurs because the !DEC$ OMP NUMA directive affects both the set of threads that is created and the way iterations are scheduled. When it generates code for the DO loop, the compiler needs to know how the set of threads was bound to memories.

- Number of threads used in NUMA PARALLEL DO constructs

  The !DEC$ OMP NUMA directive overrides the standard OpenMP mechanisms for specifying the number of threads that are used by the !$OMP PARALLEL DO directive that follows. Because of this override, an error occurs if you specify the NUM_THREADS clause on a NUMA PARALLEL DO directive.

  Instead of using the value specified in the most recent call to omp_set_num_threads or the value specified in the OMP_NUM_THREADS environment variable, the !DEC$ OMP NUMA directive uses a set of threads that is determined by the layout of the data that is used in the loop and the nest of NUMA PARALLEL DO constructs that are present.

  In the simplest case — where the loop operates on an array that is distributed in only one dimension and there are no nested NUMA PARALLEL DO loops — the number of threads is set to the number of NUMA memories times the number of threads per memory.

  In more complex cases involving the distribution of more than one dimension of an array and the use of nested NUMA PARALLEL DO loops, the compiler chooses an appropriate subset of this full set of NUMA threads to use at each level.

- Mixing NUMA and non-NUMA parallel constructs

  A NUMA PARALLEL DO construct may not be executed within the dynamic extent of a non-NUMA parallel region. Similarly, within the dynamic extent of a NUMA PARALLEL DO construct, non-NUMA parallel regions may not be executed. The same program may execute both NUMA and non-NUMA parallel constructs; however, one kind of parallel construct must be completed before beginning the other kind.

- Orphaning of NUMA PARALLEL DO constructs

  When nested NUMA PARALLEL DO constructs are used, the compiler needs to see all of the nested levels at one time in order to assign an appropriate subset of the NUMA threads to each level. Consequently a NUMA PARALLEL DO construct may not occur in a subprogram that is called within any other NUMA PARALLEL DO construct. That is, when NUMA PARALLEL DO constructs are nested, all levels of the nest must occur lexically within the same subprogram.

3.4 Two Short but Complete Example Programs

Many programs, such as red_black_10.f90 in Section 1.4, have a small number of DO loops that account for a large percentage of execution time. However, when the iterations of the DO loops execute at run time, there is a problem. The threads corresponding to the iterations do not automatically access array elements that are in the same local memory as the module on which the thread is executing.

For example, consider the following program fragment:

```
REAL X(12000000000)    ! Twelve billion elements
INTEGER*8 :: I         ! Access all elements of array X
!$OMP PARALLEL PRIVATE(I)
!$OMP DO SCHEDULE(STATIC)
DO I = 1, 12000000000  ! Twelve billion iterations
   X(I) = SQRT(FLOAT(I)) + SIN(FLOAT(I)) + ALOG(FLOAT(I))
END DO
!$OMP END DO
!$OMP END PARALLEL
```

Assume that its NUMA system is half of the one in Figure 1–5.

We expand the program fragment to a complete program, for execution on the 8-processor NUMA system, in two ways described below:

- Section 3.4.1, Program TWELVE_BILLION_A
- Section 3.4.2, Program TWELVE_BILLION_B

3.4.1 Program TWELVE_BILLION_A

Program TWELVE_BILLION_A (source file twelve_billion_a.f90) contains OpenMP directives and a user-directed page migration directive:

```
PROGRAM TWELVE_BILLION_A   ! Twelve billion elements
REAL X(12000000000)
INTEGER*8 :: I             ! Access all elements of array X
!DEC$ MIGRATE_NEXT_TOUCH_NOPRESERVE(X)
!$OMP PARALLEL DO PRIVATE(I) SCHEDULE(STATIC)
DO I = 1, 12000000000      ! Twelve billion iterations
   X(I) = SQRT(FLOAT(I)) + SIN(FLOAT(I)) + ALOG(FLOAT(I))
END DO
!$OMP END PARALLEL DO
PRINT *, 'X(1) = ', X(1)
END
```

This program uses the MIGRATE_NEXT_TOUCH_NOPRESERVE directive because the contents of array X do not have to be preserved as its pages move from one memory to another.

The following commands compile and execute the program:

```
% setenv NUMA_MEMORIES 2
% setenv NUMA_TPM 4
% f90 -o twelve_billion_a.exe -omp -numa twelve_billion_a.f90
% twelve_billion_a.exe
```

3.4.2 Program TWELVE_BILLION_B

Program TWELVE_BILLION_B (source file twelve_billion_b.f90) contains an OpenMP directive, a distribution directive, and a directive that assigns loop iterations onto threads:

```
 PROGRAM TWELVE_BILLION_B   ! Twelve billion elements
 REAL X(12000000000)
 INTEGER*8 :: I             ! Access all elements of array X
 !DEC$ DISTRIBUTE BLOCK :: X
 !DEC$ OMP NUMA
 !$OMP PARALLEL DO PRIVATE(I)
 DO I = 1, 12000000000      ! Twelve billion iterations
    X(I) = SQRT(FLOAT(I)) + SIN(FLOAT(I)) + ALOG(FLOAT(I))
 END DO
 !$OMP END PARALLEL DO
 PRINT *, 'X(1) = ', X(1)
 END
```

This program uses the DISTRIBUTE directive to guide the compiler as it places the contents of array X onto the memories.

When your program contains OpenMP directives and the DISTRIBUTE directive, it should also contain the OMP NUMA directive.

The following commands compile and execute the program:

```
% setenv OMP_NUM_THREADS 8
% setenv NUMA_MEMORIES 2
% setenv NUMA_TPM 4
% f90 -o twelve_billion_b.exe -omp -numa twelve_billion_b.f90
% twelve_billion_b.exe
```

3.5 Specifying Memories and Threads per Memory

You can specify the number of NUMA memories by the size of the array specified in a !DEC$ MEMORIES directive or by using the -numa_memories option with a non-zero value in the f90 command or by the value of the NUMA_MEMORIES environment variable. Or, you can leave this number completely unspecified; then at run time it will take on the default value for the executing system. This value is the number of RADs (Resource Affinity Domains) in the current partition of the Tru64 UNIX operating system. Usually a RAD corresponds to one physical memory unit.

An example of the first method is:

```
!DEC$ MEMORIES M(8)
!DEC$ DISTRIBUTE A(BLOCK) ONTO M
```

You can specify the number of threads per memory by using the `-numa_tpm` option with a non-zero value in the `f90` command or by the value of the `NUMA_TPM` environment variable. Or, you can leave this number completely unspecified; then at run time it will take on the default value for the executing system. This value is the number of CPUs in the current partition of the Tru64 UNIX operating system divided by (using integer division) the number of RADs.

If the `omp_set_num_threads` routine is called, it affects any OpenMP directives that are not modified by the !DEC$ OMP NUMA directive in the usual way. However, this routine has no effect on the number of threads, used by PARALLEL DO constructs, that the !DEC$ OMP NUMA directive modifies.

4

High Performance Fortran (HPF) Software: An Introduction

This chapter describes:

- Section 4.1, HPF Directives on a Distributed Memory System: Parallel Program red_black_50
- Section 4.2, What is HPF?
- Section 4.3, Parallel Programming Models

4.1 HPF Directives on a Distributed Memory System: Parallel Program red_black_50

Example 4–1 shows a program named `red_black_50.f90` on a distributed memory system. The program is a result of the conversion of serial program `red_black_10.f90` in Example 1–1 to a parallel program using HPF directives.

When the Compaq Fortran compiler processes programs with HPF directives, it generates code that uses Message Passing Interface (MPI) software from an MPI library. It no longer generates code that uses the Parallel Software Environment (PSE) library routines.

Example 4–1 Parallel Program red_black_50.f90

```
        program red_black_50
        integer, parameter           :: n=250           ! 252 x 252 x 252 array
        integer, parameter           :: niters=1000     ! display results every
                                                        !    1000 iterations
        integer, parameter           :: maxiters=200000 ! maximum number
                                                        !   of iterations
        real, parameter              :: tol = 0.1       ! tolerance
        real, parameter              :: one_sixth = (1.0 / 6.0)
        real, dimension(0:n+1,0:n+1,0:n+1) :: x         ! current temperatures
        real, dimension(0:n+1,0:n+1,0:n+1) :: x_old     ! previous temperatures
        !hpf$ distribute (*,*,block) :: x, x_old
        integer                      :: count           ! of all iterations
        real                         :: start, elapsed, error
        integer                      :: i, j, k, iters

! Initialize array x by setting the side elements to 20.0 and
! the n**3 interior elements to 15.0
        do k=0, n+1
           do j=0, n+1
              do i=0, n+1
                 if (i.eq.0   .or. j.eq.0   .or. k.eq.0 .or. &
                     i.eq.n+1 .or. j.eq.n+1 .or. k.eq.n+1) then
                    x(i,j,k) = 20.0
                 else
                    x(i,j,k) = 15.0
                 endif
              end do
           end do
        end do

        print "(A)", ""
        print "(A,i4,A,i4,A,i4,A)", "Starting ",n," x",n," x",n," red-black"
        print "(A)", ""

        x_old = x
        count = 0
        error = huge(error)

! Main loop:
        start = SECNDS(0.0)

        print "(A,2f9.5)", &
              "Initial values of x(125,125,0) and x(125,125,125) are", &
                               x(125,125,0), x(125,125,125)
        print "(A)", ""

        do while (error > tol)
           do iters = 1, niters
```

(continued on next page)

Example 4–1 (Cont.) Parallel Program red_black_50.f90

```
! Do red iterations starting at x(1,1,1)
!hpf$ independent, new(i,j,k)
do k = 1, n, 2
   do j = 1, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do

! Do red iterations starting at x(1,2,2)
!hpf$ independent, new(i,j,k)
do k = 2, n, 2
   do j = 2, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do

! Do red iterations starting at x(2,1,2)
!hpf$ independent, new(i,j,k)
do k = 2, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do

! Do red iterations starting at x(2,2,1)
!hpf$ independent, new(i,j,k)
do k = 1, n, 2
   do j = 2, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
```

(continued on next page)

Example 4–1 (Cont.) Parallel Program red_black_50.f90

```
! Do black iterations starting at x(1,1,2)
!hpf$ independent, new(i,j,k)
do k = 1, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do

! Do black iterations starting at x(1,2,1)
!hpf$ independent, new(i,j,k)
do k = 1, n, 2
   do j = 2, n, 2
      do i = 1, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do

! Do black iterations starting at x(2,1,1)
!hpf$ independent, new(i,j,k)
do k = 1, n, 2
   do j = 1, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do

! Do black iterations starting at x(2,2,2)
!hpf$ independent, new(i,j,k)
do k = 2, n, 2
   do j = 2, n, 2
      do i = 2, n, 2
         x(i,j,k) = (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) &
            + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) * one_sixth
      end do
   end do
end do
```

(continued on next page)

Example 4–1 (Cont.) Parallel Program red_black_50.f90

```
   end do
   count = count + niters
   print "(A,2f9.5)", &
         "Current values of x(125,125,0) and x(125,125,125) are", &
                          x(125,125,0), x(125,125,125)
   if (count > maxiters) exit
   error = sqrt(sum(abs(x-x_old)**2))
   print "(A,i6,A,f15.7)", "Iterations completed: ", count, &
                          "   Relative error: ", error
   print "(A)", ""
   x_old = x
end do

elapsed = SECNDS(start)
print *, 'Number of iterations = ', count
print *, 'Time elapsed = ', elapsed
end
```

4.1.1 Explanation of Parallel Program red_black_50

Parallel program `red_black_50.f90` differs from serial program `red_black_10.f90` described in Example 1–1 as follows:

- In the data declarations, after the statements that define equal size arrays x and x_old, the following statement is added:

  `!hpf$ distribute (*,*,block) :: x, x_old`

  This statement distributes the elements of the arrays across the memories. The * keyword in the first dimension keeps the elements of a column in the same memory. For example, the elements in the first column of array x are x(0,1,1), x(1,1,1), x(2,1,1), . . . , x(251,1,1); they are together in a memory. The * keyword in the second dimension keeps the elements of a row in the same memory. Together these two * keywords keep each plane of the arrays in the same memory. The `block` keyword in the third dimension distributes the set of planes over different memories. Thus these keywords combine to divide the arrays x and x_old into two dimensional planes. The first plane goes to one memory, the second plane goes to the next memory, . . . , and the last plane goes to the last memory.

  How does the compiler know how many planes to divide the arrays x and x_old onto? The answer is the number of memories in the NUMA system. In Figure 1–6, the number is 5. The `-distribute 5` option in the `f90` command specifies the number.

- Each of the eight loop nests is now preceded by the directive `!hpf$ independent, new(i,j,k)`. This informs the compiler that the DO loop that follows it can be executed in parallel. The compiler generates code that has the `x(i,j,k) =` calculations occur in parallel mode over *n* systems, where the value of *n* comes from the `-distribute` *n* option in the `f90` command line.

Based on the distributed memory system in Figure 1–6, the compilation and execution commands for this program are:

```
% limit stacksize unlimited
% limit datasize unlimited
% f90 -o red_black_50.exe \
-distribute 5 -hpf -hpf_target cmpi red_black_50.f90
% dmpirun -np 5 red_black_50.exe > red_black_50.out
```

The first two commands prevent stack size and data size errors. See Section 7.12 for more information.

The compilation command reflects the following:

- Run-time distributed memory machine with five systems (`-distribute 5`)
- Presence of HPF directives in the source program (`-hpf`)
- Version of MPI that comes with Compaq Fortran (`-hpf_target cmpi`)

The execution command reflects the following:

- Presence of software from an MPI library in the executable file (`dmpirun`)
- Run-time distributed memory machine with the number of processors equal to five systems (`-np 5`)

The output goes to file `red_black_50.out` for retrieval and display.

The first and last parts of output file `red_black_50.out` are identical to those of file `red_black_10.out` described in Section 1.4.1 with one exception: The value in the `Time elapsed` line will almost certainly be different.

You can specify Compaq Fortran MPI with environment variable DECF90_HPF_TARGET instead of with the `-hpf_target cmpi` option. In this case, remove `-hpf_target cmpi` from the `f90` command and precede the command with the following statement:

```
% setenv DECF90_HPF_TARGET cmpi
```

Section 6.2 describes how to combine HPF programs with Compaq Fortran MPI.

In general, the -distribute *n* option:

- Requires the -hpf option
- Appears in the listing file as -numnodes instead of as -distribute
- Has the same effect as -hpf *n*, when it appears with the -hpf option

When the -distribute *n* option appears without *n*, the default value of *n* is 0, which means that the number of separate systems is determined at run time.

In general, the -hpf option:

- Cannot be used with the -omp option
- Cannot be used with the -numa option
- Has the same effect as -hpf *n*, when it appears with the -distribute *n* option

In general, the -hpf_target option requires the -hpf option.

You can access program red_black_50.f90 in the file /usr/lib/cmplrs/fort90/examples/red_black_50.f90.

4.2 What is HPF?

High Performance Fortran (HPF) is a set of extensions to the Fortran 90 standard that permits programmers to specify how data is to be distributed across multiple processors. HPF's constructs allow programmers to indicate potential parallelism at a relatively high level without entering into the low-level details of message-passing and synchronization. When an HPF program is compiled, the compiler assumes responsibility for scheduling the parallel operations on the physical machines, thus reducing the time and effort required for parallel program development. For appropriate applications, parallel programs can execute dramatically faster than ordinary Fortran programs.

HPF is implemented as an integral component of the Compaq Fortran compiler. HPF programs compiled with the Compaq Fortran compiler can be executed serially on a single-processor Alpha system or in parallel on a multiple-processor Alpha system running Compaq's MPI software.

HPF gives programmers the ability to specify data distribution and data parallel operations at a high level. The compiler takes care of the details of the parallel execution. However, you must provide enough information to the compiler to ensure that data is distributed among the participating processors in the most efficient manner.

4.3 Parallel Programming Models

The design of parallel programs begins with the choice of a programming model that governs the overall structure of the program. Several models of parallelism can be used in parallel applications, for example:

- Data parallelism

  Operations are applied to many elements of an array (or other data structure). An example of this would be adding the corresponding elements of two arrays to produce a third array.

- Task parallelism

  Conceptually different operations are performed at the same time. An example of this would be a series of filters used in image processing.

- Master-slave parallelism

  One process assigns subtasks to other processes. An example of this would be a numerical integration program that decomposed the problem domain in a master process, leaving the work of integrating the resulting subdomains to a set of slave processes.

All of these types of parallelism, and others as well, are useful in certain applications. It is difficult, however, to support all of these models in the same language. HPF concentrates primarily on data parallel computations which form a widely useful class. To provide some access to other models of parallelism, an HPF program can contain what are known as extrinsic procedures. The extrinsic procedures can be written for other programming paradigms or even in another programming language, such as C or assembly language. This language feature also allows for the use of existing libraries.

4.3.1 Data Parallel Programming

The data parallel programming model is based on the premise that many large scale programs have a "natural" parallelism at a fine-grain level, such as performing the same operation on all the elements of an array.

To perform such fine-grained parallel operations, data parallel programs rely on three basic structural features:

- Global data

  All processors "see" the same set of variables when accessing data. Array declarations declare the entire size of an array, not the portion on a single processor as in many task parallel languages.

The data mapping component of HPF describes how an array can be divided among processors according to regular patterns. Communication between processors occurs when two data items are referenced together but are not stored on the same processor. By carefully matching data mapping to the requirements of the algorithms used in a given program, you can minimize the communication that occurs when a program executes. Minimizing communication should be a prime objective of HPF programming because communication is very time-consuming compared to other operations.

- Single-threaded control structure

  Data parallel operations are executed in order. When the program reaches a data parallel construct, many operations can be executed at once. Nevertheless, all processors are governed by the same single thread of logical control. Although different processors may operate on separate sections of data, processors do not fork into separate routines or processes.

- Loosely synchronous parallel execution

  Although all processors in data parallel programs execute the same program, the processors are not necessarily processing the exact same instruction at the same time. Instead, the processors operate independently, except when synchronization events (such as message communications) intervene. It is called "loosely synchronous" because these occasional synchronization events typically cause the processors to stay in the same general location in the program.

  It is commonly believed that data parallel programs require barrier synchronizations between loops or routines. However, barrier synchronizations are costly in terms of performance and are not always logically necessary. The primary reason a processor needs to pause is data dependency, not routine boundaries. Frequently, a processor may cross a routine boundary and continue computing for some time before reaching a point where the result of another processor's computation is logically required. The executables produced by Compaq Fortran frequently postpone synchronization events until they are required by data dependency. This leads to a significant performance gain over indiscriminate barrier synchronization.

4.3.2 HPF and Data Parallelism

HPF contains features for specifying data parallel operations and for mapping data across processors.

The program must specify sections of code to be considered by the compiler for parallelization by supplying supplemental high-level data partitioning information.

When the program is compiled, the complex details of communications and synchronization involved in coordinating the parallel operations are generated by the compiler automatically, thus eliminating the need for manual insertion of explicit message-passing calls.

An application can be developed and run on a single workstation, and run on a distributed memory system of any size.

5
HPF Essentials

This chapter describes:

- Section 5.1, HPF Basics
- Section 5.2, HPF Directives
- Section 5.3, Minimum Requirements for Parallel Execution
- Section 5.4, Data Parallel Array Operations
- Section 5.5, Data Mapping
- Section 5.6, Subprograms in HPF
- Section 5.7, Intrinsic and Library Procedures
- Section 5.8, Extrinsic Procedures

For more information about the HPF language, see the High Performance Fortran Language Specification at the locations described in Associated Documents in the Preface.

For a more technical presentation of information specifically about efficient use of Compaq Fortran's implementation of HPF, see Chapter 7.

5.1 HPF Basics

HPF is a set of extensions to Fortran intended to facilitate writing parallel Fortran programs. Appropriately written HPF programs that are run in an HPF-capable environment, such as the distributed memory system in Figure 1–6 with underlying Message Passing Interface (MPI) software, can execute at a dramatically faster speed than Fortran programs run in a single-processor environment. (Details of the HPF/MPI connection are in Section 6.2.)

HPF is especially useful for programs that can be expressed as large scale array operations. They form a major class of computationally intensive programs. HPF programming involves inserting directives that advise the compiler about potentially parallelizable array operations. Lower level details of parallelization, such as message passing and synchronization, are automated by the compiler and invisible to the programmer.

Array operations are usually expressed with data parallel statements such as FORALL structures or Fortran 90 whole array or array subsection assignments in order to be parallelized. In the current version of Compaq Fortran, array operations expressed as DO loops can be parallelized by using the INDEPENDENT directive. See Section 5.4.4. Also, array operations expressed as DO loops in Fortran 77 code can usually be easily converted to array assignment statements or to FORALL structures. See Section 7.2 for more information.

Compaq Fortran parallelizes array operations when these array operations are accompanied by HPF directives. The compiler uses the information given in the directives to spread data storage and computation across a cluster or multiprocessor server. For a large class of computationally intensive problems, using HPF achieves a dramatic increase in performance over serial (nonparallel) execution.

5.1.1 When to Use HPF

For many programs, HPF can produce enormous performance gains, with speed-up in near direct proportion to the number of processors. However, some algorithms are not suitable for HPF implementation.

There is no formula or completely general rule for determining when HPF is useful because the achievable degree of parallel speed-up is highly algorithm-dependent. Nevertheless, the following considerations can serve as a rough guide.

5.1.1.1 Existing Code

Existing codes are good candidates for conversion to HPF under the following circumstances:

- The computationally intensive kernel of the program must be expressible as an operation (or operations) on a large array (or arrays).
- The existing code should spend a long time performing the array operations.
- Codes already written in Fortran 90 syntax are easy to convert to HPF.

- Existing HPF codes written for other vendors' compilers or translators may need minor modifications, such as adding interface blocks (see Section 5.6.2).
- Codes written to run well on vector machines generally perform well when converted to HPF. In particular, this means codes with large DO loops that have no inter-iteration dependencies.
- Thread-based parallel programs, or programs that rely on a process spawning other processes, are *not* suitable for coding in HPF. However, this type of code can be incorporated into an HPF program through use of EXTRINSIC subroutines.

5.1.1.2 New Code

For new code, HPF is generally useful in the following cases:

- Problems utilizing iterative solution methods
- Signal processing
- Image processing
- Modeling
- Grid-based problems in general, especially translationally invariant grid operations — solution methods where large parts of a grid are uniformly operated on
- In general, most problems expressible as operations on large arrays

5.2 HPF Directives

These are examples of HPF directives:

```
!HPF$ DISTRIBUTE A(BLOCK, BLOCK)
!HPF$ ALIGN B(I) WITH C(I)
!HPF$ PROCESSORS P(8)
```

HPF directives are preceded by the tag `!HPF$` to identify them to the compiler. Because this tag begins with an exclamation mark (!), all HPF directives are syntactically Fortran comments. Except for a syntax check at compile time, HPF directives are ignored (treated like comments) in source code not explicitly compiled for execution on a distributed memory system, and have no effect on the meaning of the program.

When compiled with the -hpf switch (see Section 6.1.1.1, -hpf [~] Option — Compile for Parallel Execution), the compiler uses the HPF directives to create a parallelized version of the program. In a parallel environment, correctly used HPF directives affect only the performance of a program, not its meaning or results. Incorrect use of HPF directives can inadvertently change the meaning of the code. The result can be generation of an illegal program by the compiler.

HPF directives must follow certain syntax rules in order to produce meaningful results. For example, the !HPF$ tag must begin in column 1 in fixed source form, but may be indented in free source form. A number of other syntax rules apply.

For More Information:

- On the syntax of HPF directives, see the *Compaq Fortran Language Reference Manual*.

Table 5–1 lists the HPF directives and the sections in this chapter that explain them.

Table 5–1 HPF Directives and HPF-Specific Attribute

HPF Directive	Where Documented
ALIGN	Section 5.5.3
DISTRIBUTE	Section 5.5.6
INDEPENDENT	Section 5.4.4
INHERIT	Section 5.6.5
PROCESSORS	Section 5.5.5
SHADOW	Section 5.5.7
TEMPLATE	Section 5.5.4

5.3 Minimum Requirements for Parallel Execution

In order to achieve performance gains from using HPF, programs must be written so that they execute in parallel. In order to be compiled to execute in parallel, certain minimum requirements must be met. Code that does not meet these requirements is not parallelized and is compiled to run serially (with no parallel speed-up).

- Array operations are parallelized only on arrays that either:
  - Have been explicitly distributed using the DISTRIBUTE directive

  - Are ultimately aligned with an array or template that has been explicitly distributed using the DISTRIBUTE directive.

- Only **data parallel array assignment statements** are parallelized. The phrase "data parallel array assignment statements" refers to:
  - Fortran 90 whole array or array subsection assignment statements
  - FORALL structures
  - DO loops with the INDEPENDENT attribute
  - Certain Compaq Fortran array intrinsic functions and library routines

Section C.5.1 emphasizes the importance of data distribution and explains how to easily change a distribution. See also Section 5.5.

5.4 Data Parallel Array Operations

This section explains Fortran 90 array terminology, array assignment syntax, and FORALL structures.

5.4.1 Array Terminology

An array consists of elements that extend in one or more dimensions to represent columns, rows, planes, and so on. The number of dimensions in an array is called the **rank** of the array. The number of elements in a dimension is called the **extent** of the array in that dimension. The **shape** of an array is its rank and its extent in each dimension. The **size** of an array is the product of the extents.

```
REAL, DIMENSION(10, 5:24, -5:M) :: A
REAL, DIMENSION(0:9, 20, M+6)   :: B
```

This example uses entity-oriented declaration syntax. The rank of *A* is 3, the shape of *A* is (10, 20, (M+6)), the extent of *A* in the second dimension is 20, and the size of *A* is 10 * 20 * (M+6).

Arrays can be zero-sized if the extent of any dimension is zero (certain restrictions apply to programs containing zero-sized arrays). The rank must be fixed at the time the program is written, but the extents in any dimension and the upper and lower bounds do not have to be fixed until the array comes into existence. Two arrays are **conformable** if they have the same shape, that is, the same rank and the same extents in corresponding dimensions; A and B are conformable.

For More Information:

- On entity-oriented declaration syntax, see Section 5.4.6.
- On restrictions applying to programs containing zero-sized arrays, see Section 7.1.

5.4.2 Fortran 90 Array Assignment

Fortran 90 array assignment statements allow operations on entire arrays to be expressed more simply than was possible in Fortran 77. These array assignment statements are parallelized by the Compaq Fortran compiler for increased performance. A DO loop that is used to accomplish an array assignment will be parallelized only if it is marked with the INDEPENDENT directive.

For More Information:

- On the INDEPENDENT directive, see Section 5.4.4.

5.4.2.1 Whole Array Assignment

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 each element of *C* with 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 triply-nested loops, such as:

```
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
```

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

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

If the `f90` command includes the `-hpf` option, then routines coded in array assignment syntax are parallelized by the Compaq Fortran compiler for parallel execution. DO loops are parallelized only if they are marked with the INDEPENDENT directive.

For More Information:

- On the INDEPENDENT directive, see Section 5.4.4.

5.4.2.2 Array Subsections

You can reference parts of arrays ("array subsections") using a notation known as subscript triplet notation. In subscript triplet notation, up to three parameters are specified for each dimension of the array. When a range of values is intended, the syntax of a subscript triplet is:

```
[a]:[b][:c]
```

Where *a* is the lower bound, *b* is the upper bound, and *c* is the stride (increment). The first colon is mandatory when a range of values is specified, even if *a*, *b* and *c* are all omitted. Default values are used when any (or all) of *a*, *b*, or *c* are omitted, as follows:

- The default value for *a* is the declared lower bound for that dimension.
- The default value for *b* is the declared upper bound for that dimension.
- The default value for *c* is 1.

When a single value, rather than a range of values, is desired for a given dimension, a single parameter is specified, with no colons.

For example, consider the following code fragment, composed of an array declaration and an array subsection assignment:

```
REAL A(100, 100)
A(1,1:100:2) = 7
```

The assignment statement assigns a value of *7* to all the elements in the subsection of the array represented by the expression *A(1,1:100:2)*. For the first dimension of the expression, the *1* is a single parameter, specifying a constant value of *1* for the first dimension. For the second dimension, the notation *1:100:2* is a subscript triplet in which *1* is the lower bound, *100* is the upper bound, and *2* is the stride. Therefore, the array subsection assignment in the code fragment assigns a value of *7* to the odd elements of the first row of *A*.

In the same array *A*, the four elements *A(1,1)*, *A(100,1)*, *A(1, 100)*, and *A(100, 100)* reference the four corners of *A*. *A(1:100:99, 1:100:99)* is a 2 by 2 array section referencing all four corners. *A(1, :)* references the entire first row of *A*, because the colon is a place holder referencing the entire declared range of the second dimension. Similarly, *A(100,:)* references the entire last row of *A*.

As seen in Section 5.4.2.1, many whole array assignments can be expressed in a single line in Fortran 90. Similarly, many array subsection assignments can also be done in a single line. For example, consider the array subsection assignment expressed by this Fortran 77 DO loop:

```
DO x = k+1, n
  A(x, k) = A(x, k) / A(k, k)
END DO
```

Using Fortran 90 array assignment syntax, this same assignment requires only a single line:

```
A(k+1:n, k) = A(k+1:n, k) / A(k, k)
```

Fortran 90 array assignment syntax can also be used to assign a scalar to every element of an array:

```
REAL A(16, 32), S
A = S/2
```

For More Information:

- On array specifications (explicit shape, assumed shape, and so on), see Section 5.6.1
- On specifying a section or subset of an array, see the *Compaq Fortran Language Reference Manual* on Array Elements and Sections and the WHERE Statement.

5.4.3 FORALL

The FORALL statement is part of the ANSI Fortran 95 standard. FORALL is a natural idiom for expressing parallelism, and is parallelized by the Compaq Fortran compiler for parallel execution on a distributed memory system.

FORALL is a more 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 array section. It can, however, be expressed in a FORALL statement:

```
REAL, DIMENSION(n, n) :: A
FORALL (i=1:n)  A(i, i) = 1
```

The FORALL/END FORALL structure can be used to include multiple assignment statements:

```
FORALL (i=k+1:n, j=k+1:n)
  A(i, j) = A(i, j) - A(i, k)*A(k, j)
  B(i, j) = A(i, j) + 1
END FORALL
```

In a FORALL/END FORALL structure, each line is computed separately. A FORALL/END FORALL structure produces exactly the same result as a separate FORALL statement for each line. The previous FORALL/END FORALL structure is equivalent to the following:

```
FORALL (i=k+1:n, j=k+1:n) A(i, j) = A(i, j) - A(i, k)*A(k, j)
FORALL (i=k+1:n, j=k+1:n) B(i, j) = A(i, j) + 1
```

Although FORALL structures serve the same purpose as some DO loops did in Fortran 77, a FORALL structure is an assignment statement (not a loop), and in many cases produces a different result from an analogous DO loop because of its different semantics. For a comparison of DO loops and FORALL structures, see Section B.2.3.

5.4.4 INDEPENDENT Directive

Some DO loops are eligible to be tagged with the INDEPENDENT directive, which allows for parallel execution. This is useful for converting pre-existing Fortran 77 code to HPF.

A loop is eligible be tagged INDEPENDENT if the iterations can be performed in any order (forwards, backwards, or even random) and still produce the "same" result. More precisely: A loop may be tagged INDEPENDENT if no array element (or other atomic data object) is assigned a value by one iteration and read or written by any other iteration. (Note that the REDUCTION and NEW keywords relax this definition somewhat. There are restrictions involving I/O, pointer assignment/nullification, and ALLOCATE/DEALLOCATE statements. For details, see the High Performance Fortran Language Specification.)

For example:

```
!HPF$ INDEPENDENT
      DO I=1, 100
        A(I) = B(I)
      END DO
```

Place the INDEPENDENT directive on the line immediately before the DO loop you wish to mark.

When DO loops are nested, you must evaluate each nesting level separately to determine whether it is eligible for the INDEPENDENT directive. For example:

```
      DO n = 100, 1, -1
!HPF$   INDEPENDENT, NEW(j)
        DO i = k+1, n
!HPF$     INDEPENDENT
          DO j = k+1, n
            A(i, j) = A(i, j) - A(i, k)*A(k, j) + n
          END DO
        END DO
      END DO
```

In this code fragment, each of the two inner DO loops can be marked INDEPENDENT, because the iterations of these loops can be performed in any random order without affecting the results. However, the outer loop cannot be marked independent, because its iterations must be performed in sequential order or the results will be altered.

The NEW(j) keyword tells the compiler that in each iteration, the inner DO loop variable `j` is unrelated to the `j` from the previous iteration. Compaq's compiler currently requires the NEW keyword in order to parallelize nested INDEPENDENT DO loops.

The three parallel structures (Fortran 90 array syntax, FORALL, and INDEPENDENT DO loops) differ from each other in syntax and semantics. Each has advantages and disadvantages. For a comparison among them, see Section B.2.3.

A number of restrictions must be adhered to for INDEPENDENT DO loops to be successfully parallelized.

Unlike FORALLs, INDEPENDENT DO loops can contain calls to procedures that are not PURE. However, special ON HOME RESIDENT syntax must be used for INDEPENDENT loops that contain procedure calls.

For More Information:

- For a comparison between the three parallel structures (Fortran 90 array syntax, FORALL, and INDEPENDENT DO loops), see Section B.2.3.
- On restrictions that must be followed for INDEPENDENT DO loops to be successfully parallelized, see the Release Notes.
- On the special restrictions that apply to INDEPENDENT loops that contain procedure calls, see the Release Notes.

5.4.5 Vector-Valued Subscripts

Vector-valued subscripts provide a more general way to select a subset of array elements than subscript triplet notation. (Subscript triplet notation is explained in Section 5.4.2.2.) A vector-valued subscript is a one-dimensional array of type INTEGER (a vector) that is used as a subscript for one dimension of another array. The elements of this index vector select the elements of the indexed array to be in the subsection. For example, consider the following code fragment:

```
INTEGER A(3)
INTEGER B(6, 4)
FORALL (i=1:3) A(i) = 2*i - 1
B(A, 3) = 12
```

In this code fragment, the FORALL statement assigns the values *(/1, 3, 5/)* to the index vector *A*. The assignment statement uses these three values to decide which elements of *B* to assign a value of *12*. Using these values, it assigns a value of *12* to *B(1, 3)*, *B(3, 3)*, and *B(5, 3)*.

A vector-valued subscript with duplicate values must not occur on the left-hand side of an assignment statement because this could lead to indeterminate program results. For example, the following code fragment is illegal:

```
INTEGER A(4)
INTEGER B(0:5, 4)
FORALL (i=1:4) A(i) = (i-2)*(i-3)
FORALL (i=1:4) B(A(i), 4) = i        ! Illegal assignment !
```

In this example, the first FORALL statement assigns to *A* the values *(/2, 0, 0, 2/)*. However, the values that are assigned in the second FORALL statement are impossible to predict. The second FORALL statement assigns two different values to *B(2, 4)*, and two different values to *B(0, 4)*. Unlike a DO loop, which makes assignments in a predictable sequential order, a FORALL construct is a parallel structure that can assign values to many array elements simultaneously. It is impossible to predict which of the duplicate values assigned to these elements will remain after the execution of the statement is completed.

Because it is costly in terms of performance for the compiler to check for duplicate elements in vector-valued subscripts, illegal code does not necessarily generate an error message. It is up to the programmer to avoid this mistake.

The HPF library routine COPY_SCATTER permits duplicate values on the left side of an assignment statement. COPY_SCATTER is subject to certain restrictions and can produce indeterminate program results. See the online `man` page for `copy_scatter`.

5.4.6 Entity-Oriented Declaration Syntax

In Fortran 90, arrays can be organized either by attribute, as in FORTRAN 77, or by entity. The :: notation is used in the entity-oriented declaration form, in which you can group the type, the attributes, and the optional initialization value of an entity into a single statement. For example:

```
INTEGER, DIMENSION(4), PARAMETER :: PERMUTATION = (/1,3,2,4/)
```

5.4.7 SEQUENCE and NOSEQUENCE Directives

The SEQUENCE directive indicates that data objects in a procedure depend on array element order or storage association. The SEQUENCE directive warns the compiler not to map data across processors. You can use the SEQUENCE directive with or without a list of arrays.

The form of the directive without a list of arrays is:

```
!HPF$ SEQUENCE
```

This form of the directive instructs the compiler to assume that all arrays in this procedure depend on sequence association.

The form of the directive with a list of arrays is:

```
!HPF$ SEQUENCE X, Y, Z
```

This directive instructs the compiler that only *X*, *Y*, and *Z* rely on sequence association.

On non-NUMA systems, arrays with the SEQUENCE attribute may not be named in a DISTRIBUTE or ALIGN directive. Array operations involving such unmapped arrays are performed serially, with no parallel speed-up. Also, DISTRIBUTE or ALIGN directives may not appear in the same procedure as a SEQUENCE directive. An error message is generated at compile time if an array with the SEQUENCE attribute is improperly named in a DISTRIBUTE or ALIGN directive.

In programs compiled with the `-hpf` option, element order and storage association apply only when explicitly requested with the SEQUENCE directive. When the `-hpf` option is not used, sequence association is always assumed and supported.

The NOSEQUENCE directive asserts that named data objects, or all data objects in a procedure, do not depend on array element order or storage association. The form of this directive is:

```
!HPF$ NOSEQUENCE
```

The NOSEQUENCE directive is the default when the -hpf option is used. The SEQUENCE directive is the default when the -hpf option is not used.

5.4.8 Out of Range Subscripts

In older versions of Fortran, some programmers developed the practice of using out of range subscripts, as in the following (illegal) example:

```
REAL A(50, 50)
DO i = 1, 2500
  A(i, 1) = 8
END DO
```

This code is illegal, although it can produce correct results in nonparallel implementations of Fortran. Referencing an out of range subscript does not necessarily generate an error message. However, in cases where the variable referenced is distributed, use of such code causes an application to stall or produce incorrect results when executed in parallel on a distributed memory system.

The -check_bounds option may not be used together with the -hpf option. To check an HPF program for out-of-range subscripts, use the -check_bounds option in a serial compilation (that is, without using the -hpf option).

5.5 Data Mapping

Proper data mapping is critical for the performance of any HPF program. The discussion of data mapping is divided as follows:

- Section 5.5.1, Data Mapping Basics
- Section 5.5.2, Illustrated Summary of HPF Data Mapping
- Section 5.5.3, ALIGN Directive
- Section 5.5.4, TEMPLATE Directive
- Section 5.5.5, PROCESSORS Directive
- Section 5.5.6, DISTRIBUTE Directive

Section 5.5.6 includes an extensive set of figures showing many of the major distributions for one- and two-dimensional arrays.

For More Information:

- See Section C.5.1, Deciding on a Distribution
- See Section 5.3, Minimum Requirements for Parallel Execution

5.5.1 Data Mapping Basics

HPF is designed for **data parallel programming**, a programming model in which the work of large-scale uniform array operations is divided up among a number of processors in order to increase performance. In data parallel programming, each array is split up into parts, and each part is stored on a different processor. In most cases, it is most efficient for operations to be performed by the processor storing the data in its local memory. If the arrays are mapped onto the processors in such a way that each processor has most of the information necessary to perform a given array operation on the part of the array stored locally, each processor can work independently on its own section of the array at the same time the other processors are working on other sections of the array. In this manner, an array operation is completed more quickly than the same operation performed by a single processor. In the optimal case, the speed-up **scales linearly**; in an environment with *n* processors, the operation is completed *n* times faster than with a single processor.

5.5.2 Illustrated Summary of HPF Data Mapping

This section explains HPF's basic concepts and models of data mapping. After A brief code fragment illustrates a sample data mapping, followed by a series of figures that represent this mapping schematically.

In HPF's data mapping model, arrays are aligned into groups, which are distributed onto an abstract processor arrangement. The underlying software environment (in this case, MPI with the Tru64 UNIX operating system) maps this processor arrangement onto the physical processors in the cluster.

HPF data mapping can be thought of as occurring in the following five stages:

1. Array and template declaration (standard Fortran declarations and TEMPLATE directive)
2. Array alignment (ALIGN)
3. Declaration of abstract processor arrangement (PROCESSORS)
4. Distribution onto the abstract processor arrangement (DISTRIBUTE)
5. Mapping the abstract processor arrangement onto the physical processors (compile-time and run-time command-line options)

Although the program must explicitly specify array declaration and distribution (stages 1 and 4) in order to successfully map an array, it is not usually necessary for the program to specify all five stages.

It is easiest to summarize these five stages pictorially. Each of the illustrations on the following pages show one of the five stages. The illustrations of the first four stages are based on the code fragment in Example 5–1.

Example 5–1 Code Fragment for Mapping Illustrations

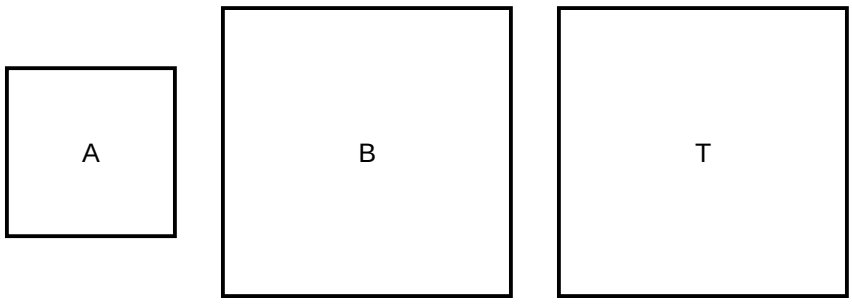
```
      REAL A(12, 12)     ! Array
      REAL B(16, 16)     ! and template
!HPF$ TEMPLATE T(16,16)  ! declarations
!HPF$ ALIGN B WITH T                ! Array
!HPF$ ALIGN A(i, j) WITH T(i+2, j+2) ! alignment
!HPF$ PROCESSORS P(2, 2)       ! Declaration of processor arrangement
!HPF$ DISTRIBUTE T(BLOCK, BLOCK) ONTO P    ! Array distribution
```

The code fragment in Example 5–1 does not do anything; it represents only the mapping of data in preparation for some other array operations not specified here.

Assume that Example 5–1 is part of a larger program whose source code is called `foo.f90`, and whose executable file is `foo.out`. The following two command lines show user control over the fifth stage of HPF data mapping, at compile time, and at run time:

```
% f90  -hpf 4  -o foo.out  foo.f90
% foo.out  -peers 4  -on Fred,Greg,Hilda,Ingrid
```

Stage 1: Array Declaration (Required) and Template Declaration (Optional)



MLO-011940

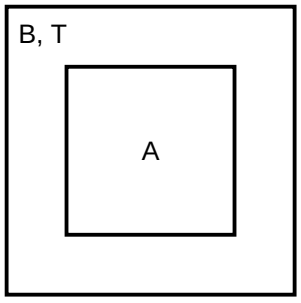
```
      REAL A(12, 12)
      REAL B(16, 16)
!HPF$ TEMPLATE T(16, 16)
```

Array declaration, which is the same as in a nonparallel Fortran environment, is mandatory. Template declaration is optional. Templates are used for data alignment. (See Section 5.5.4.)

For More Information:

- On array declarations, see the *Compaq Fortran Language Reference Manual*.
- On templates, see Section 5.5.4.

Stage 2: Array Alignment (optional)



MLO-011941

```
!HPF$ TEMPLATE T(16,16)
!HPF$ ALIGN B WITH T
!HPF$ ALIGN A(i, j) WITH T(i+2, j+2)
```

When two arrays will be interacting with one another, it is usually advantageous to use the ALIGN directive. The ALIGN directive ensures that corresponding elements of two arrays are always stored on the same processor.

Arrays are lined up together onto a **template**. A template describes an index space with a specified shape, but with no content. A template can be thought of as "an array of nothings". No storage is allocated for templates. In this example, the two arrays *A* and *B* are aligned with the template *T* using the ALIGN directive. *B* is aligned with the whole template *T*, whereas *A* is aligned with only part of *T*.

In the ALIGN directives, arrays *A* and *B* are the **alignees**, and template *T* is the *align target*. The subscripts *i* and *j* are **dummy variables**, which do not represent any specific values. They refer to all the valid subscript values for *A*. They are used to specify the correspondence between elements of *A* and elements of *T*.

The explicit naming of templates is required only for certain specialized alignments. In most cases, it is possible to use a template implicitly by aligning the arrays with one another (see Section 5.5.6.10).

For More Information:

- On the TEMPLATE directive, see Section 5.5.4.
- On the ALIGN directive, see Section 5.5.3.

Stage 3: Declaration of Abstract Processor Arrangement (optional)

MLO-011942

```
!HPF$ PROCESSORS P(2, 2)
```

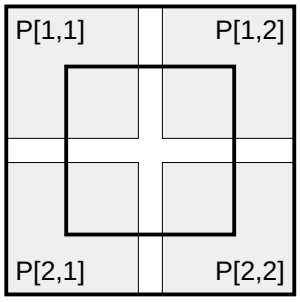
This stage of the data mapping process defines a conceptual arrangement of the processors in preparation for a DISTRIBUTE directive (see Stage 4). Processor arrangements are called "abstract" because at compile time the processors in the arrangement are not yet identified with particular physical processors. If the program does not contain a PROCESSORS directive, the compiler defines an appropriate default processor arrangement.

When processor arrangements are explicitly declared, they must be defined to conform with both the anticipated array distribution (stage 4), and the anticipated size of the distributed memory system on which the program will be run. In this example, a 2 by 2 arrangement is used, because it is two dimensional (to support the two-dimensional BLOCK, BLOCK distribution used in stage 4), and contains a total of four processors (to conform to the distributed memory system that used in stage 5 at run time). It is also possible to determine the size of the processor arrangement dynamically at run time using the intrinsic function NUMBER_OF_PROCESSORS( ).

For More Information:

- On the PROCESSORS directive, see Section 5.5.5
- On using NUMBER_OF_PROCESSORS( ) to dimension a processor arrangement at run time, see Section 5.5.5.

Stage 4: Distribution of the Arrays onto the Processor Arrangement (required)



MLO-011943

```
DISTRIBUTE T(BLOCK, BLOCK) ONTO P
```

Distribution means dividing up the storage of the arrays among the processors. This usually means that each processor has only a subsection of each array in its own local memory. You must explicitly select a distribution in order to achieve any parallel speed-up. Proper selection of a distribution is absolutely critical for application performance.

There are a very large number of possible distributions, many of which are explained in Section 5.5.6. This example uses (BLOCK, BLOCK) distribution, one of a large number of possibilities.

In the case of the example distribution (BLOCK, BLOCK), it is useful to visualize the arrays as superimposed over the processor arrangement. However, other distributions require more complex visualizations. A number of example illustrations can be found in Section 5.5.6.

Because arrays *A* and *B* have already been aligned with template *T*, the distribution of both arrays is implied when *T* is distributed.

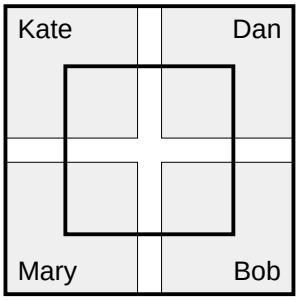
When templates are not explicitly named, array names can be used in place of template names in DISTRIBUTE directives. See Section 5.5.6.10.

The ONTO clause may be used only when a processor arrangement has been explicitly declared. When an ONTO clause is not specified, the array is distributed onto a default processor arrangement. See Section 5.5.6.11.

For More Information:

- On using array names in DISTRIBUTE directives instead of template names, see Section 5.5.6.10.
- On using (or omitting) the ONTO clause, see Section 5.5.6.11.
- Illustration and explanation of a number of specific distributions can be found in Section 5.5.6.

Stage 5: Mapping of the Processor Arrangement onto Physical Processors



MLO-011944

This final stage of data distribution is handled transparently in a system-dependent way at run time. Environment variables or command-line options can be used to include or exclude particular machines. These are described in Chapter 6, Compiling and Running HPF Programs.

In this example, the program is run in an environment comprising workstations named Kate, Mary, Dan, and Bob. If desired, you can specify the hosts to be included in the execution, as in the example. It is usually better to leave out this specification so that members can be selected based on load-balancing considerations.

The `-hpf` *n* compile-time command line option controls the number of processors that the program is designed to use:

```
% f90 -hpf 4 -o a.out a.f90
```

The number of processors *[n]* specified by the `-hpf` option must be equal to the number of processors specified in the PROCESSORS directive.

The -peers command-line option can be used at run time to specify the number of processors to be used, and the -on command-line option can be used to specify particular hosts:

```
% a.out -peers 4 -on Kate,Mary,Dan,Bob
```

In this example, the number of peers is equal to the number of hosts specified with -on. However, in some cases the number of hosts will be less than the number of peers, such as when the -virtual option is used.

For More Information:

- On compile-time and run-time command-line options, see Chapter 6.

5.5.3 ALIGN Directive

The ALIGN directive is used to specify that certain data objects are to be mapped in the same way as certain other data objects. Corresponding elements in aligned arrays are always mapped to the same processor; array operations between aligned arrays are usually more efficient than array operations between arrays that are not known to be aligned.

Compaq recommends that you do not attempt to align arrays by using matching DISTRIBUTE directives. You must use the ALIGN directive to guarantee that corresponding elements are mapped to the same processor in every supported run-time environment.

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 WITH B
```

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

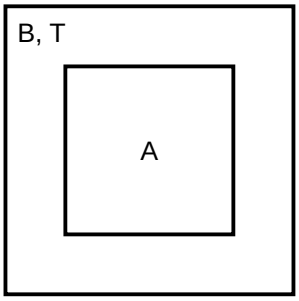
```
!HPF$ ALIGN E(i) WITH F(2*i-1)
```

In this example, the elements of *E* correspond to the odd elements of *F*. In this case, *E* can have a maximum of half as many elements as *F*.

As shown in the example given in Section 5.5.2, an array can be aligned with the interior of a larger array or template:

```
      REAL A(12, 12)
      REAL B(16, 16)
!HPF$ TEMPLATE T(16,16)
!HPF$ ALIGN B WITH T
!HPF$ ALIGN A(i, j) WITH T(i+2, j+2)
```

In this example, the 16 × 16 array *B* is aligned with the template *T* of the same size, and the 12 × 12 array *A* is aligned with the interior of *T*. Because *A* and *B* are both aligned with the template *T*, *A* and *B* are said to be **indirectly aligned**. Each interior element of *B* is always stored on the same processor as the corresponding element of *A*:



MLO-011941

When an asterisk (*) is specified for a given dimension, it specifies that alignment occurs between the non-asterisk dimensions of the alignee and the align target. For example:

```
!HPF$ ALIGN P(i) WITH Q(*, i)
```

In this example, *P* is aligned with the second dimension (with every row) of *Q*. Each element of *P* is available on the same processor as every element in the corresponding column of *Q*. This means that any given element `P(i)` is available on each processor that stores any element in the i^{th} column of *Q*. Depending on the mapping of *Q*, *P* may need to be partially or fully replicated onto all processors in order to achieve this result.

When a whole array is aligned, the ALIGN directive can be written either *with* an align subscript, like this:

```
!HPF$ ALIGN b(i) WITH c(i)
```

or *without* an align subscript, like this:

```
!HPF$ ALIGN b WITH c
```

These two forms have slightly different semantics. When an align subscript is used, the align target is permitted to be larger than the alignee. Also, elements whose subscripts are equal are aligned, regardless of what the lower bound of each array happens to be.

When an align subscript is not used, the alignee and the align target must be exactly the same size. Corresponding elements are aligned beginning with the lower bound of each array, regardless of whether the subscripts of the corresponding elements are equal.

Using (or not using) an align subscript can have an effect on performance when the arrays are allocatable. For examples and detailed explanation, see Section 7.8.

Other more complex alignments are possible.

For more information, see the High Performance Fortran Language Specification.

Circular alignments are not permitted. For example, the following code is illegal:

```
!HPF$ ALIGN A WITH B
!HPF$ ALIGN B WITH A  ! Illegal circular alignment!
```

Each array can be the alignee (to the left of the WITH) only once. When a given set of data objects are aligned with each other, the object array or template) that is never an alignee (is never to the left of the WITH) is known as the **ultimate align target**. Only the ultimate align target is permitted to appear in a DISTRIBUTE directive. The other arrays that are aligned with the ultimate align target are implicitly distributed together with the ultimate align target.

The ALIGN directive causes data objects to be mapped across processors only if the the ultimate align target appears in a DISTRIBUTE directive. For more information, see Section 5.3.

Because the ALIGN directive implicitly determines the distribution of the aligned arrays, it has a direct effect on how much or little communication occurs among the processors. A poorly chosen alignment can cause severe application performance degradation, whereas a well chosen alignment can cause dramatic improvement in performance.

For More Information:

- On the syntax of the ALIGN directive, see the *Compaq Fortran Language Reference Manual*.
- On the performance consequences of using (or not using) an align subscript, see Section 7.8.

5.5.4 TEMPLATE Directive

A template is an empty array space (or an array of nothings). A template is used as an **align target** (the object after WITH in an ALIGN directive), and can be distributed with the DISTRIBUTE directive.

For most programs, declaration of an explicit template is not necessary. When you do not explicitly declare a template, you can use an array name in place of a template name in the ALIGN and DISTRIBUTE directives. For an example, see Section 5.5.6.10.

Because they have no content, no storage space is allocated for templates. Templates are declared in the specification part of a scoping unit with the !HPF$ TEMPLATE directive.

Templates cannot be in COMMON. Two templates declared in different scoping units are always distinct even if they are given the same name. Templates cannot be passed through the subprogram argument interface. For an example of passing an array that is aligned with a template to a subprogram, see Section 5.6.5.

Some specialized alignments require the use of an explicit template. For example, an explicit template is needed when a particular array needs to be distributed over only some of the processors in the executing cluster partition. This cannot be done by declaring a smaller processor arrangement, because processor arrangements must always have exactly the same number of processors as the executing cluster partition. However, an array can be restricted to a subset of the partition with the following technique: A template is distributed over a full-sized processor arrangement, after which an array can be aligned with a slice of the template. For instance:

```
!HPF$ PROCESSORS P(4, 4)
!HPF$ TEMPLATE T(4, 4)
!HPF$ DISTRIBUTE(BLOCK, BLOCK) ONTO P :: T
!HPF$ ALIGN A(J) WITH T(J, 1)
```

This technique is used in an Input/Output (I/O) optimization explained in Section 7.11.4.

Another instance where explicit declaration of a template is useful is a program where smaller arrays are to be aligned with a larger index space but no single array spans the entire index space. For example, if four $n \times n$ arrays are aligned to the four corners of a TEMPLATE of size $(n + 100) \times (n + 100)$:

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) :: inclusive(n+100,n+100)
      REAL, DIMENSION(n,n)  ::NW, NE, SW, SE
!HPF$ ALIGN NW(i,j) WITH inclusive( i , j)
!HPF$ ALIGN NE(i,j) WITH inclusive( i , j+100 )
!HPF$ ALIGN SW(i,j) WITH inclusive( i+100, j )
!HPF$ ALIGN SE(i,j) WITH inclusive( i+100, j+100 )
```

In this example, the template `inclusive` allows the four smaller arrays to be aligned together and distributed even though no single one of them spans the entire index space.

- For information on the syntax of the TEMPLATE directive, see the *Compaq Fortran Language Reference Manual*.

5.5.5 PROCESSORS Directive

Rather than distributing arrays directly onto physical processors, HPF uses **abstract processor arrangements**, which allow distributions to be expressed without reference to any particular hardware configuration. This greatly improves the portability of parallel programs.

The use of processor arrangements also permits a greater variety of data mappings to be expressed. For instance, in a program written for 16 processors, processor arrangements can be declared not only of shape 4×4, but also 2×8, 8×2, 1×16, or 16×1. Even though all of these shapes have the same number of processors, each shape results in a different data mapping because the distribution in any given dimension of an array is determined by the extent of the processor arrangement in that dimension (see Section 5.5.6.4).

Here are examples of declarations of a one-, two-, and three-dimensional processor arrangement:

```
!HPF$ PROCESSORS P(4)
!HPF$ PROCESSORS Q(4,6)
!HPF$ PROCESSORS R(4,3,3)
```

Only one PROCESSORS directive can appear in a program.

PROCESSORS, like TEMPLATE, is an optional directive. If an array is distributed without an explicit processor arrangement (see Section 5.5.6.11), the compiler creates a default processor arrangement.

The total number of processors in a processor arrangement (the product of the values specified for each of its dimensions) must be equal to the number of peers specified at compile time and run time. If a general program that can run on any number of processors is desired, the processor arrangement must be dimensioned dynamically using the intrinsic function NUMBER_OF_PROCESSORS():

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
```

In order to produce a general program that can run on any number of processors, use the `-hpf` option at compile time without any numerical argument.

Any number of peers is allowed, but performance is improved in some cases if the number of processors is a power of two.

You can simulate a cluster larger than the number of available physical processors (CPUs) with the `-virtual` run-time option. Although a processor arrangement smaller than the number of peers in the executing partition is not permitted, the storage of an array can be restricted to a subset of the partition using the TEMPLATE directive.

Like array elements, processors in each dimension of an abstract processor arrangement are by default indexed starting with 1. This is a different numbering system from that used for physical processors in which physical processors (referred to as **peers**) are numbered starting with 0.

For More Information:

- On the syntax of the PROCESSORS directive, see the *Compaq Fortran Language Reference Manual*.
- On using compiler options, see Section 6.1.1.
- On compiling a program to run on any number of processors, see Section 6.1.1.1.
- On whether the number of processors should be a power of two, see Section 6.1.1.1.
- On using the `-virtual` run-time option to simulate a cluster larger than the number of available CPUs, see Section 6.1.1.1.
- On restricting the storage of an array to a subset of the cluster partition, see Section 5.5.4.

5.5.6 DISTRIBUTE Directive

The choice of an appropriate distribution for any given algorithm is critical to application performance. A carefully chosen DISTRIBUTE directive can improve the performance of HPF code by orders of magnitude over otherwise identical code with a poorly chosen DISTRIBUTE directive.

All HPF data mappings are constructed from various combinations of two basic types of parallel distribution: BLOCK and CYCLIC.

The DISTRIBUTE directive has two basic forms, shown in the following two example lines:

```
!HPF$ DISTRIBUTE A(CYCLIC, BLOCK)
!HPF$ DISTRIBUTE A(CYCLIC, BLOCK) ONTO P
```

Use the ONTO clause when a template or array is distributed onto an explicitly named processor arrangement. Use the DISTRIBUTE directive without an ONTO clause when a processor arrangement is not explicitly named.

The template or array named in a DISTRIBUTE directive must be an **ultimate align target**.

For More Information:

- On ultimate align targets, see Section 5.5.3.

5.5.6.1 Explanation of the Distribution Figures

In the distribution figures on the following pages, each distribution is shown in two views: **array view** and **processor view**. For each distribution, a code fragment is given showing the declaration of an array *A* and an abstract processor arrangement *P*, followed by the distribution of *A* onto *P*. The code fragment, which describes both views, is printed under the array view.

In array view, the data mapping is shown with a series of boxes, each box representing one array element. The large letter in each box represents the name of the workstation which stores that array element in its memory.

The pattern formed by the large letters is the most important feature of the array views. Although the arrays declared in the code fragments are artificially small (by several orders of magnitude), they are large enough to show the broad patterns that appear in more realistically sized arrays. For example, in BLOCK, BLOCK distribution the array elements are always divided into (roughly) square blocks (see Figure 5–5); in *, BLOCK they are grouped in broad vertical stripes (see Figure 5–17); in *, CYCLIC they are grouped in narrow vertical stripes (see Figure 5–19).

Note about Array Views

These figures are meant to illustrate data mapping only. No information is given about the values of array elements.

The processor view is a different way of representing the same code fragment. In processor view, the processors are positioned to show each processor's place in the abstract processor arrangement. The processors are lined up in a single row when the processor arrangement is one dimensional, and in a rectangular pattern when the processor arrangement is two dimensional.

Note about Processor Views

The physical processors are shown organized according to the abstract processor arrangement as an aid to conceptualization only. No information is given about the actual connectivity of the processors or configuration of the network.

In processor view, each processor contains a list of the array elements stored on that processor according to the given code fragment. Also included is a schematic representation of the physical storage sequence of the array elements. This information is useful when EXTRINSIC(HPF_LOCAL) routines are used. The array elements are stored on the processor in the order listed. Each black or white box represents the storage space for one array element. Each black box represents one array element. Some processors have white boxes, representing unused storage space. Unused storage space occurs because all processors allocate the same amount of storage space for a given array, even though the number of elements stored on each processor is not necessarily equal.

Note

Physical storage sequence in HPF is processor-dependent. The information about physical storage sequence given in the distribution illustrations describes the current implementation of Compaq Fortran. Programs that depend on this information may not be portable to other HPF implementations.

Note about Rows and Columns

When referring to elements of a two-dimensional array or processor arrangement, this manual refers to the first subscript as varying with vertical movement through the array, and the second subscript as varying with horizontal movement. In other words, the first axis is vertical and the second axis is horizontal. This notation is patterned after matrix notation in mathematics, where the elements in the first row of a matrix M are referred to as M_{11}, M_{12}, M_{13} . . . , the second

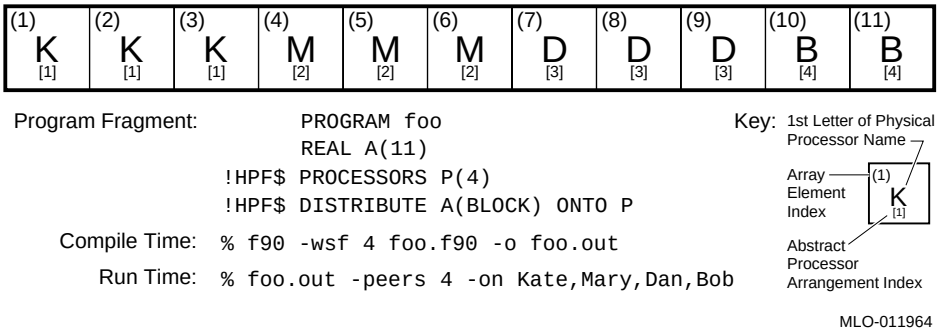
row as M_{21}, M_{22}, M_{23}, and so on. This terminology is used for both arrays and processor arrangements. Array element subscripts should not be confused with Cartesian ordered pairs (x, y), in which x varies with horizontal movement, and y varies with vertical movement.

5.5.6.2 BLOCK Distribution

In BLOCK distribution of a one-dimensional array, the array elements are distributed over each processor in large blocks. To the extent possible, each processor is given an equal number of array elements. If the number of elements is not evenly divisible by the number of processors, all processors have an equal number of elements except for the last processor, which has fewer elements than the others. (The one exception to this rule is the case where the number of elements in the array is relatively small compared to the number of processors in the distributed memory system. In that case, it is possible that one or more processors have zero elements. Nevertheless, the rule still holds for those processors with a non-zero number of elements.)

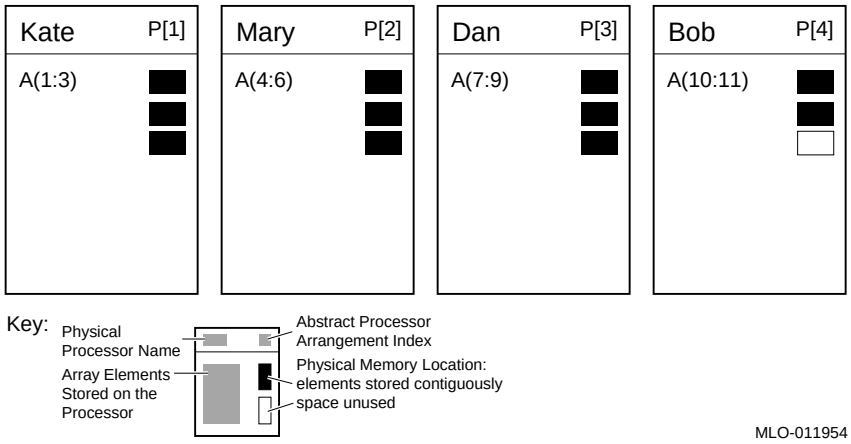
Figure 5–1 is the array view of an 11 element one-dimensional array *A* distributed (BLOCK) onto four processors. The processor view is shown in Figure 5–2. The first three processors (Kate, Mary, and Dan) each get 3 elements, and the last processor (Bob) receives 2.

Figure 5–1 BLOCK Distribution — Array View



MLO-011964

Figure 5–2 BLOCK Distribution — Processor View



The formulas that generate the information found in Figures 5–1 and 5–2 can be found in the High Performance Fortran Language Specification.

See Section 5.5.6.1 for a detailed explanation of the format of the figures found in this chapter.

5.5.6.3 CYCLIC Distribution

In cyclic distribution, the array elements are dealt out to the processors in round-robin order, like playing cards dealt out to players around the table. When elements are distributed over *n* processors, each processor, starting from a different offset, contains every n^{th} column. Figure 5–3 is the array view of the same array and processor arrangement, distributed CYCLIC, instead of BLOCK. The processor view is shown in Figure 5–4.

Figure 5–3 CYCLIC Distribution — Array View

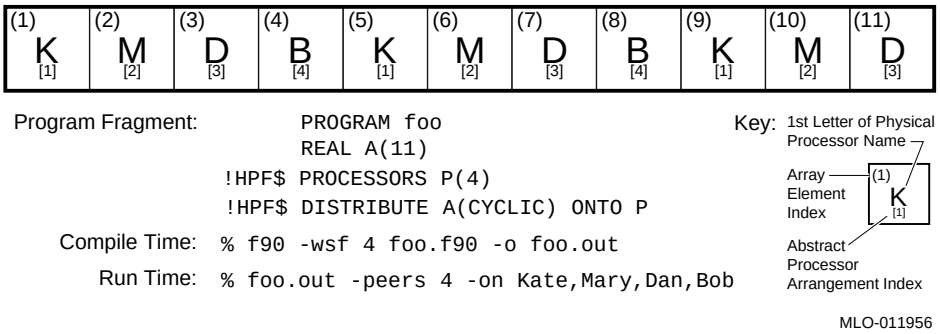
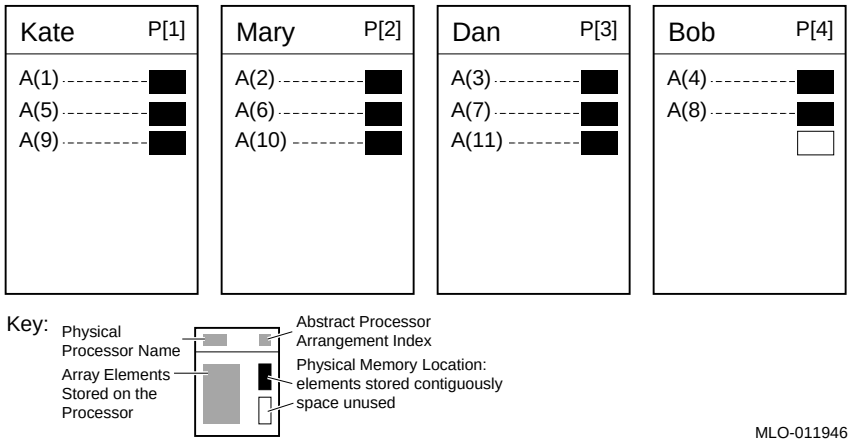


Figure 5–4 CYCLIC Distribution — Processor View



The formulas that generate the information found in Figures 5–3 and 5–4 can be found in the High Performance Fortran Language Specification.

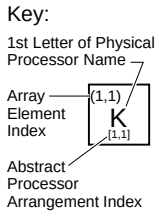
See Section 5.5.6.1 for a detailed explanation of the format of the figures found in this chapter.

5.5.6.4 BLOCK, BLOCK Distribution

When multidimensional arrays are distributed, the pattern of distribution is figured independently for each dimension based on the shape of the processor array. For example, when both dimensions are distributed BLOCK, the array is divided into large rectangles. BLOCK, BLOCK distribution is shown in Figures 5–5 and 5–6.

Figure 5–5 BLOCK, BLOCK Distribution — Array View

(1,1) K [1,1]	(1,2) K [1,1]	(1,3) K [1,1]	(1,4) K [1,1]	(1,5) D [1,2]	(1,6) D [1,2]	(1,7) D [1,2]
(2,1) K [1,1]	(2,2) K [1,1]	(2,3) K [1,1]	(2,4) K [1,1]	(2,5) D [1,2]	(2,6) D [1,2]	(2,7) D [1,2]
(3,1) K [1,1]	(3,2) K [1,1]	(3,3) K [1,1]	(3,4) K [1,1]	(3,5) D [1,2]	(3,6) D [1,2]	(3,7) D [1,2]
(4,1) K [1,1]	(4,2) K [1,1]	(4,3) K [1,1]	(4,4) K [1,1]	(4,5) D [1,2]	(4,6) D [1,2]	(4,7) D [1,2]
(5,1) M [2,1]	(5,2) M [2,1]	(5,3) M [2,1]	(5,4) M [2,1]	(5,5) B [2,2]	(5,6) B [2,2]	(5,7) B [2,2]
(6,1) M [2,1]	(6,2) M [2,1]	(6,3) M [2,1]	(6,4) M [2,1]	(6,5) B [2,2]	(6,6) B [2,2]	(6,7) B [2,2]
(7,1) M [2,1]	(7,2) M [2,1]	(7,3) M [2,1]	(7,4) M [2,1]	(7,5) B [2,2]	(7,6) B [2,2]	(7,7) B [2,2]



Program Fragment:

```
      PROGRAM foo
      REAL A(7,7)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO P
```

Compile Time:

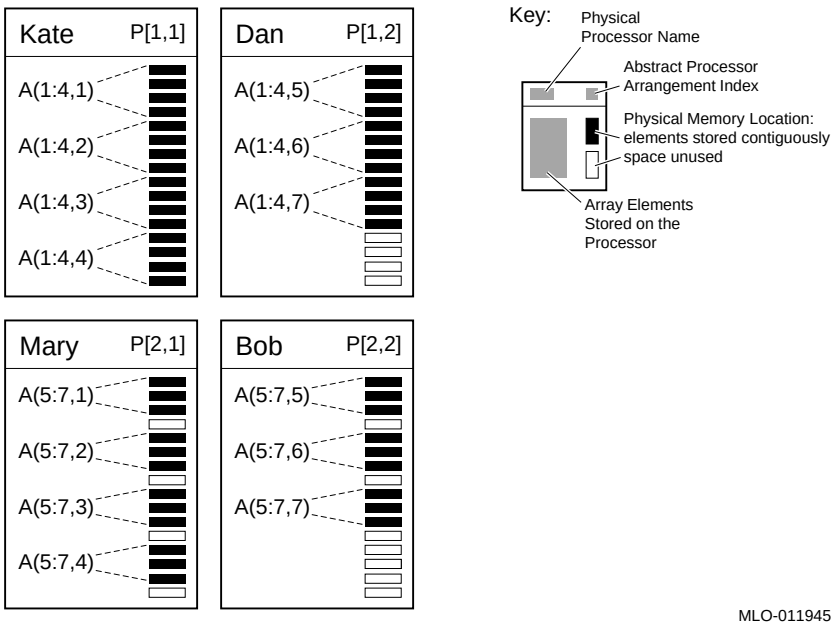
```
% f90 -wsf 4 foo.f90 -o foo.out
```

Run Time:

```
% foo.out -peers 4 -on Kate,Mary,Dan,Bob
```

MLO-011955

Figure 5–6 BLOCK, BLOCK Distribution — Processor View



Because each dimension is distributed independently, any single column (or row) of Figure 5–5 considered in isolation resembles a one-dimensional array, distributed (BLOCK).

You may wonder why any given column (or row) of a two-dimensional array distributed (BLOCK, BLOCK) onto four processors (as in Figure 5–5) is divided into only two blocks, whereas a one-dimensional array distributed BLOCK onto the same number of processors is divided into four blocks (as in Figure 5–1).

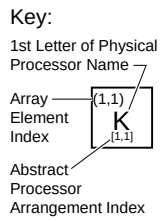
The answer is that the number of blocks in any dimension is determined by the extent of the processor arrangement in that dimension. In Figure 5–1, the array is divided into four blocks because the processor arrangement has an extent of four. However, in BLOCK, BLOCK distribution, the processor arrangement is 2 by 2 (see Figure 5–6). Therefore, each column (or row) has two blocks, because the processor arrangement has an extent of 2 in each dimension.

5.5.6.5 CYCLIC, CYCLIC Distribution

CYCLIC, CYCLIC distribution produces a sort of checkerboard effect in which no element is on the same processor as its immediate neighbors. See Figures 5–7 and 5–8.

Figure 5–7 CYCLIC, CYCLIC Distribution — Array View

(1,1) K [1,1]	(1,2) D [1,2]	(1,3) K [1,1]	(1,4) D [1,2]	(1,5) K [1,1]	(1,6) D [1,2]	(1,7) K [1,1]
(2,1) M [2,1]	(2,2) B [2,2]	(2,3) M [2,1]	(2,4) B [2,2]	(2,5) M [2,1]	(2,6) B [2,2]	(2,7) M [2,1]
(3,1) K [1,1]	(3,2) D [1,2]	(3,3) K [1,1]	(3,4) D [1,2]	(3,5) K [1,1]	(3,6) D [1,2]	(3,7) K [1,1]
(4,1) M [2,1]	(4,2) B [2,2]	(4,3) M [2,1]	(4,4) B [2,2]	(4,5) M [2,1]	(4,6) B [2,2]	(4,7) M [2,1]
(5,1) K [1,1]	(5,2) D [1,2]	(5,3) K [1,1]	(5,4) D [1,2]	(5,5) K [1,1]	(5,6) D [1,2]	(5,7) K [1,1]
(6,1) M [2,1]	(6,2) B [2,2]	(6,3) M [2,1]	(6,4) B [2,2]	(6,5) M [2,1]	(6,6) B [2,2]	(6,7) M [2,1]
(7,1) K [1,1]	(7,2) D [1,2]	(7,3) K [1,1]	(7,4) D [1,2]	(7,5) K [1,1]	(7,6) D [1,2]	(7,7) K [1,1]



Program Fragment:

```
      PROGRAM foo
      REAL A(7,7)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE A(CYCLIC,CYCLIC) ONTO P
```

Compile Time:

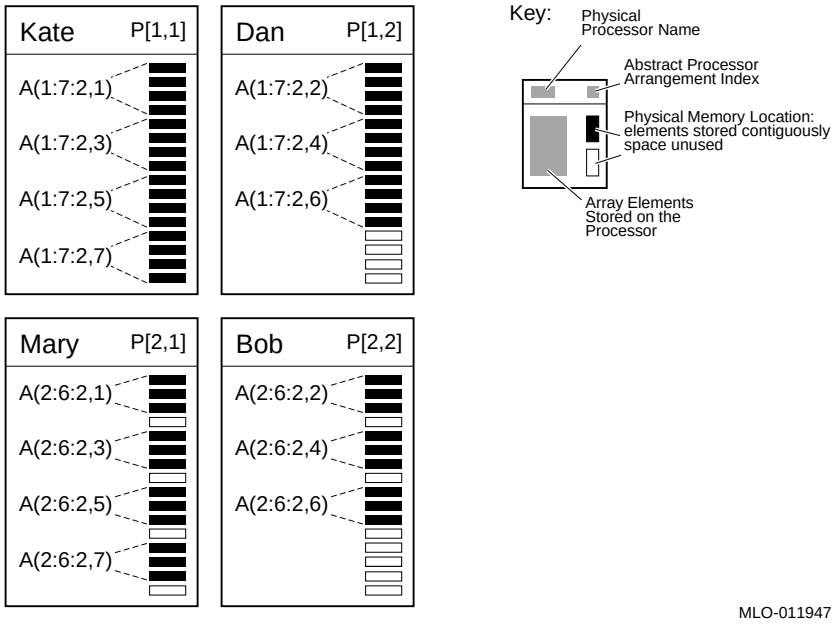
```
% f90 -wsf 4 foo.f90 -o foo.out
```

Run Time:

```
% foo.out -peers 4 -on Kate,Mary,Dan,Bob
```

MLO-011957

Figure 5–8 CYCLIC, CYCLIC Distribution — Processor View



Each dimension is distributed independently. This means that any single column (or row) of Figure 5–7 considered in isolation resembles a one-dimensional array with a CYCLIC distribution.

For an explanation of why each column (or row) alternates between two (rather than four) processors, see Section 5.5.6.4.

The formulas that generate the information found in Figures 5–7 and 5–8 can be found in the High Performance Fortran Language Specification.

A visually-oriented technique for reproducing the results of these formulas for two-dimensional distributions can be found in Section 5.5.6.9.

See Section 5.5.6.1 for a detailed explanation of the format of the figures found in this chapter.

5.5.6.6 CYCLIC, BLOCK Distribution

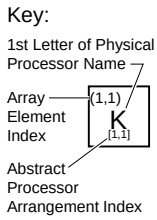
It is not necessary for multidimensional arrays to have the same distribution in each dimension. In CYCLIC, BLOCK distribution, any row considered in isolation is divided into blocks (as in BLOCK distribution), but elements in any column alternate between processors (as in CYCLIC distribution).

This manual refers to the first dimension as vertical and the second dimension as horizontal. (CYCLIC, BLOCK) distribution means that elements are distributed cyclically along the vertical axis, and in blocks along the horizontal axis.

CYCLIC, BLOCK distribution is shown in Figures 5–9 and 5–10.

Figure 5–9 CYCLIC, BLOCK Distribution — Array View

(1,1) K [1,1]	(1,2) K [1,1]	(1,3) K [1,1]	(1,4) K [1,1]	(1,5) D [1,2]	(1,6) D [1,2]	(1,7) D [1,2]
(2,1) M [2,1]	(2,2) M [2,1]	(2,3) M [2,1]	(2,4) M [2,1]	(2,5) B [2,2]	(2,6) B [2,2]	(2,7) B [2,2]
(3,1) K [1,1]	(3,2) K [1,1]	(3,3) K [1,1]	(3,4) K [1,1]	(3,5) D [1,2]	(3,6) D [1,2]	(3,7) D [1,2]
(4,1) M [2,1]	(4,2) M [2,1]	(4,3) M [2,1]	(4,4) M [2,1]	(4,5) B [2,2]	(4,6) B [2,2]	(4,7) B [2,2]
(5,1) K [1,1]	(5,2) K [1,1]	(5,3) K [1,1]	(5,4) K [1,1]	(5,5) D [1,2]	(5,6) D [1,2]	(5,7) D [1,2]
(6,1) M [2,1]	(6,2) M [2,1]	(6,3) M [2,1]	(6,4) M [2,1]	(6,5) B [2,2]	(6,6) B [2,2]	(6,7) B [2,2]
(7,1) K [1,1]	(7,2) K [1,1]	(7,3) K [1,1]	(7,4) K [1,1]	(7,5) D [1,2]	(7,6) D [1,2]	(7,7) D [1,2]



Program Fragment:

```
      PROGRAM foo
      REAL A(7,7)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE A(CYCLIC,BLOCK) ONTO P
```

Compile Time:

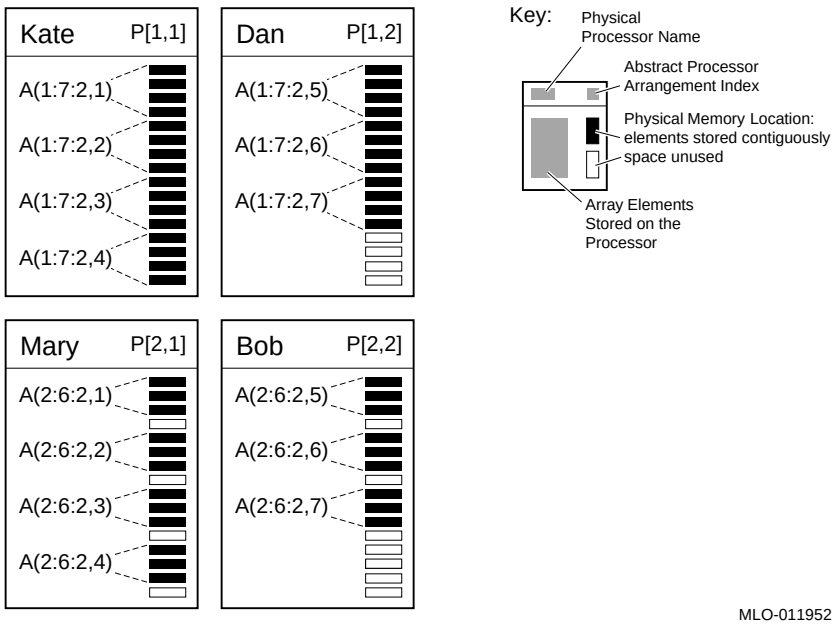
```
% f90 -wsf 4 foo.f90 -o foo.out
```

Run Time:

```
% foo.out -peers 4 -on Kate,Mary,Dan,Bob
```

MLO-011962

Figure 5–10 CYCLIC, BLOCK Distribution — Processor View



The formulas that generate the information found in Figures 5–9 and 5–10 can be found in the High Performance Fortran Language Specification.

A visually-oriented technique for reproducing the results of these formulas for two-dimensional distributions can be found in Section 5.5.6.9.

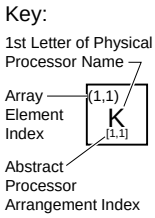
See Section 5.5.6.1 for a detailed explanation of the format of the figures found in this chapter.

5.5.6.7 BLOCK, CYCLIC Distribution

BLOCK, CYCLIC distribution is analogous to CYCLIC, BLOCK, with the opposite orientation. See Figures 5–11 and 5–12.

Figure 5–11 BLOCK, CYCLIC Distribution — Array View

(1,1) K [1,1]	(1,2) D [1,2]	(1,3) K [1,1]	(1,4) D [1,2]	(1,5) K [1,1]	(1,6) D [1,2]	(1,7) K [1,1]
(2,1) K [1,1]	(2,2) D [1,2]	(2,3) K [1,1]	(2,4) D [1,2]	(2,5) K [1,1]	(2,6) D [1,2]	(2,7) K [1,1]
(3,1) K [1,1]	(3,2) D [1,2]	(3,3) K [1,1]	(3,4) D [1,2]	(3,5) K [1,1]	(3,6) D [1,2]	(3,7) K [1,1]
(4,1) K [1,1]	(4,2) D [1,2]	(4,3) K [1,1]	(4,4) D [1,2]	(4,5) K [1,1]	(4,6) D [1,2]	(4,7) K [1,1]
(5,1) M [2,1]	(5,2) B [2,2]	(5,3) M [2,1]	(5,4) B [2,2]	(5,5) M [2,1]	(5,6) B [2,2]	(5,7) M [2,1]
(6,1) M [2,1]	(6,2) B [2,2]	(6,3) M [2,1]	(6,4) B [2,2]	(6,5) M [2,1]	(6,6) B [2,2]	(6,7) M [2,1]
(7,1) M [2,1]	(7,2) B [2,2]	(7,3) M [2,1]	(7,4) B [2,2]	(7,5) M [2,1]	(7,6) B [2,2]	(7,7) M [2,1]



Program Fragment:

```
      PROGRAM foo
      REAL A(7,7)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE A(BLOCK,CYCLIC) ONTO P
```

Compile Time:

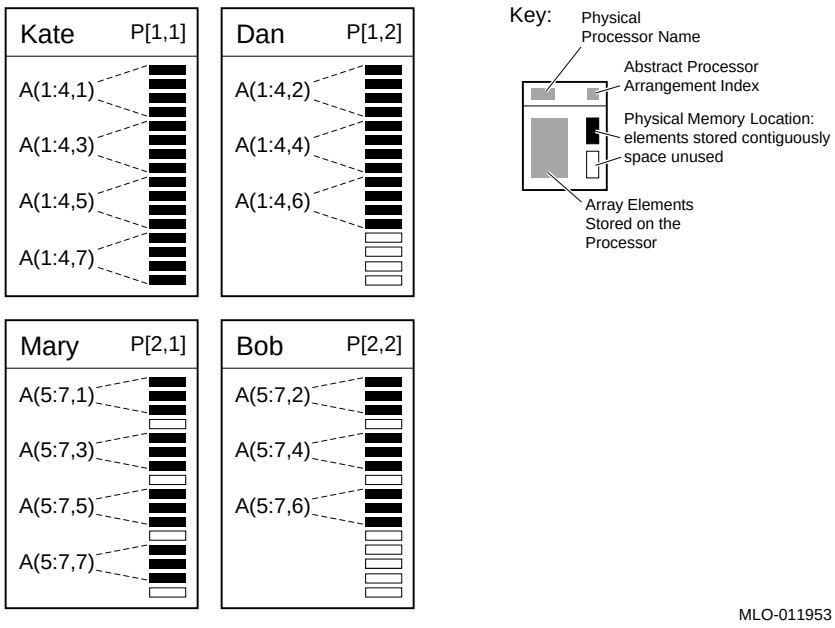
```
% f90 -wsf 4 foo.f90 -o foo.out
```

Run Time:

```
% foo.out -peers 4 -on Kate,Mary,Dan,Bob
```

MLO-011963

Figure 5–12 BLOCK, CYCLIC Distribution — Processor View



The formulas that generate the information found in Figures 5–11 and 5–12 can be found in the High Performance Fortran Language Specification.

A visually-oriented technique for reproducing the results of these formulas for two-dimensional distributions can be found in Section 5.5.6.9.

See Section 5.5.6.1 for a detailed explanation of the format of the figures found in this chapter.

5.5.6.8 Asterisk Distributions

When an asterisk (*) occurs inside the parentheses of a DISTRIBUTE directive, it refers to array elements not being distributed along one of the axes. In other words, array elements along the axis marked with an asterisk in the DISTRIBUTE directive are not divided up among different processors, but assigned as a single block to one processor. This type of mapping is sometimes called "on processor" distribution. It can also be referred to as "collapsed" or "serial" distribution.

For example, in (BLOCK, *) distribution, the asterisk for the second dimension means that each row is assigned as a single block to one processor. (In this manual, the second dimension is referred to as horizontal. See the "Note about Rows and Columns" in Section 5.5.6.1.)

Even though (BLOCK, *) distribution is used for two-dimensional arrays, it is considered a one-dimensional distribution, because only the first dimension is distributed. It must therefore be distributed onto a one-dimensional processor arrangement. The general rule for this is the following: The rank of the processor arrangement must be equal to the number of non-asterisk dimensions in the DISTRIBUTE directive.

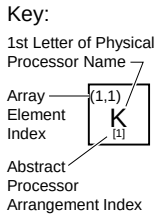
Figures 5–13 and 5–14 depict (BLOCK, ∗) distribution. Figures 5–15, 5–16, 5–17, 5–18, 5–19, and 5–20 show other combinations of CYCLIC and BLOCK with ∗.

The formulas that generate the information found in these figures can be found in the High Performance Fortran Language Specification.

See Section 5.5.6.1 for a detailed explanation of the format of the figures found in this chapter.

Figure 5–13 BLOCK,* Distribution — Array View

(1,1) K [1]	(1,2) K [1]	(1,3) K [1]	(1,4) K [1]	(1,5) K [1]	(1,6) K [1]	(1,7) K [1]
(2,1) K [1]	(2,2) K [1]	(2,3) K [1]	(2,4) K [1]	(2,5) K [1]	(2,6) K [1]	(2,7) K [1]
(3,1) M [2]	(3,2) M [2]	(3,3) M [2]	(3,4) M [2]	(3,5) M [2]	(3,6) M [2]	(3,7) M [2]
(4,1) M [2]	(4,2) M [2]	(4,3) M [2]	(4,4) M [2]	(4,5) M [2]	(4,6) M [2]	(4,7) M [2]
(5,1) D [3]	(5,2) D [3]	(5,3) D [3]	(5,4) D [3]	(5,5) D [3]	(5,6) D [3]	(5,7) D [3]
(6,1) D [3]	(6,2) D [3]	(6,3) D [3]	(6,4) D [3]	(6,5) D [3]	(6,6) D [3]	(6,7) D [3]
(7,1) B [4]	(7,2) B [4]	(7,3) B [4]	(7,4) B [4]	(7,5) B [4]	(7,6) B [4]	(7,7) B [4]



Program Fragment:

```
      PROGRAM foo
      REAL A(7,7)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(BLOCK,*) ONTO P
```

Compile Time:

```
% f90 -wsf 4 foo.f90 -o foo.out
```

Run Time:

```
% foo.out -peers 4 -on Kate,Mary,Dan,Bob
```

MLO-011960

Figure 5–14 BLOCK, * Distribution — Processor View

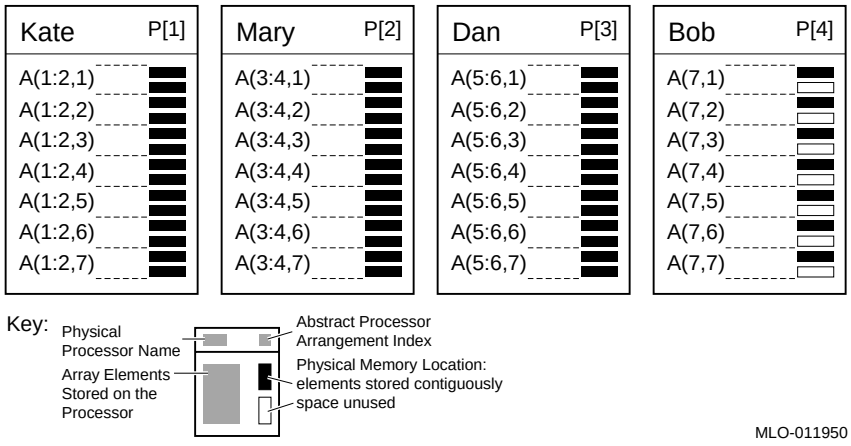
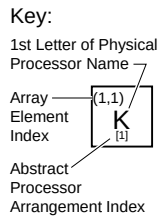


Figure 5–15 CYCLIC, * Distribution — Array View

(1,1) K [1]	(1,2) K [1]	(1,3) K [1]	(1,4) K [1]	(1,5) K [1]	(1,6) K [1]	(1,7) K [1]
(2,1) M [2]	(2,2) M [2]	(2,3) M [2]	(2,4) M [2]	(2,5) M [2]	(2,6) M [2]	(2,7) M [2]
(3,1) D [3]	(3,2) D [3]	(3,3) D [3]	(3,4) D [3]	(3,5) D [3]	(3,6) D [3]	(3,7) D [3]
(4,1) B [4]	(4,2) B [4]	(4,3) B [4]	(4,4) B [4]	(4,5) B [4]	(4,6) B [4]	(4,7) B [4]
(5,1) K [1]	(5,2) K [1]	(5,3) K [1]	(5,4) K [1]	(5,5) K [1]	(5,6) K [1]	(5,7) K [1]
(6,1) M [2]	(6,2) M [2]	(6,3) M [2]	(6,4) M [2]	(6,5) M [2]	(6,6) M [2]	(6,7) M [2]
(7,1) D [3]	(7,2) D [3]	(7,3) D [3]	(7,4) D [3]	(7,5) D [3]	(7,6) D [3]	(7,7) D [3]



Program Fragment:

```
      PROGRAM foo
      REAL A(7,7)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(CYCLIC,*) ONTO P
```

Compile Time:

```
% f90 -wsf 4 foo.f90 -o foo.out
```

Run Time:

```
% foo.out -peers 4 -on Kate,Mary,Dan,Bob
```

MLO-011958

Figure 5–16 CYCLIC, * Distribution — Processor View

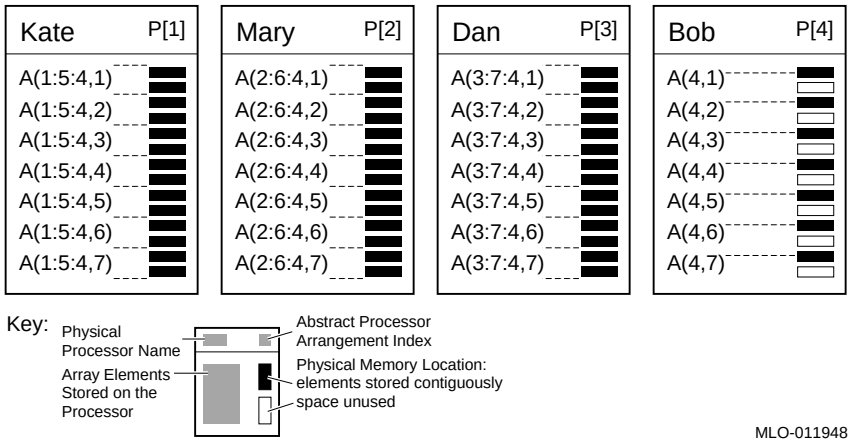
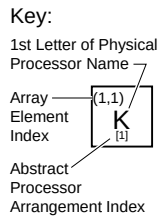


Figure 5–17 *, BLOCK Distribution — Array View

(1,1) K [1]	(1,2) K [1]	(1,3) M [2]	(1,4) M [2]	(1,5) D [3]	(1,6) D [3]	(1,7) B [4]
(2,1) K [1]	(2,2) K [1]	(2,3) M [2]	(2,4) M [2]	(2,5) D [3]	(2,6) D [3]	(2,7) B [4]
(3,1) K [1]	(3,2) K [1]	(3,3) M [2]	(3,4) M [2]	(3,5) D [3]	(3,6) D [3]	(3,7) B [4]
(4,1) K [1]	(4,2) K [1]	(4,3) M [2]	(4,4) M [2]	(4,5) D [3]	(4,6) D [3]	(4,7) B [4]
(5,1) K [1]	(5,2) K [1]	(5,3) M [2]	(5,4) M [2]	(5,5) D [3]	(5,6) D [3]	(5,7) B [4]
(6,1) K [1]	(6,2) K [1]	(6,3) M [2]	(6,4) M [2]	(6,5) D [3]	(6,6) D [3]	(6,7) B [4]
(7,1) K [1]	(7,2) K [1]	(7,3) M [2]	(7,4) M [2]	(7,5) D [3]	(7,6) D [3]	(7,7) B [4]



Program Fragment:

```
      PROGRAM foo
      REAL A(7,7)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(*,BLOCK) ONTO P
```

Compile Time:

```
% f90 -wsf 4 foo.f90 -o foo.out
```

Run Time:

```
% foo.out -peers 4 -on Kate,Mary,Dan,Bob
```

MLO-011959

Figure 5–18 *, BLOCK Distribution — Processor View

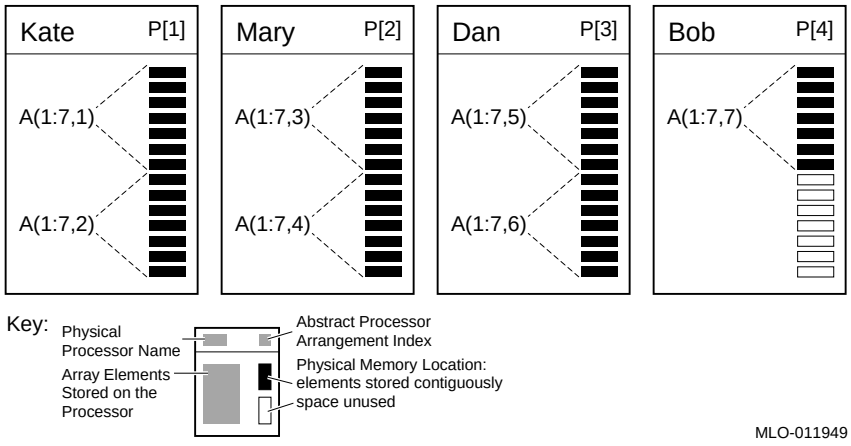


Figure 5–19 *, CYCLIC Distribution — Array View

(1,1) K [1]	(1,2) M [2]	(1,3) D [3]	(1,4) B [4]	(1,5) K [1]	(1,6) M [2]	(1,7) D [3]
(2,1) K [1]	(2,2) M [2]	(2,3) D [3]	(2,4) B [4]	(2,5) K [1]	(2,6) M [2]	(2,7) D [3]
(3,1) K [1]	(3,2) M [2]	(3,3) D [3]	(3,4) B [4]	(3,5) K [1]	(3,6) M [2]	(3,7) D [3]
(4,1) K [1]	(4,2) M [2]	(4,3) D [3]	(4,4) B [4]	(4,5) K [1]	(4,6) M [2]	(4,7) D [3]
(5,1) K [1]	(5,2) M [2]	(5,3) D [3]	(5,4) B [4]	(5,5) K [1]	(5,6) M [2]	(5,7) D [3]
(6,1) K [1]	(6,2) M [2]	(6,3) D [3]	(6,4) B [4]	(6,5) K [1]	(6,6) M [2]	(6,7) D [3]
(7,1) K [1]	(7,2) M [2]	(7,3) D [3]	(7,4) B [4]	(7,5) K [1]	(7,6) M [2]	(7,7) D [3]

Key:

1st Letter of Physical Processor Name

Array Element Index — (1,1) K [1]

Abstract Processor Arrangement Index

Program Fragment:

```
      PROGRAM foo
      REAL A(7,7)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(*,CYCLIC) ONTO P
```

Compile Time:

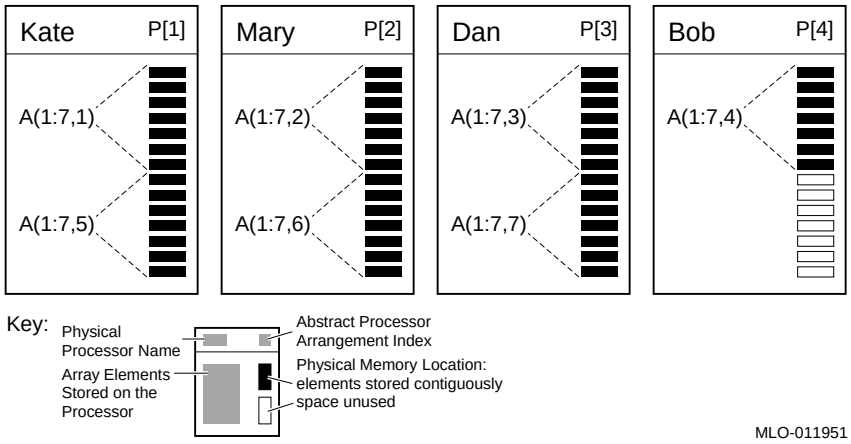
```
% f90 -wsf 4 foo.f90 -o foo.out
```

Run Time:

```
% foo.out -peers 4 -on Kate,Mary,Dan,Bob
```

MLO-011961

Figure 5–20 *, CYCLIC Distribution — Processor View



5.5.6.9 Visual Technique for Computing Two-Dimensional Distributions

Figure 5–21 presents a visually-oriented technique for constructing the array view of two-dimensional distributions. In this technique, the elements in the upper left-hand corner of the array are assigned in the same pattern as the processor arrangement. Figure 5–21 shows how this pattern is expanded and/or repeated to construct the appropriate array view.

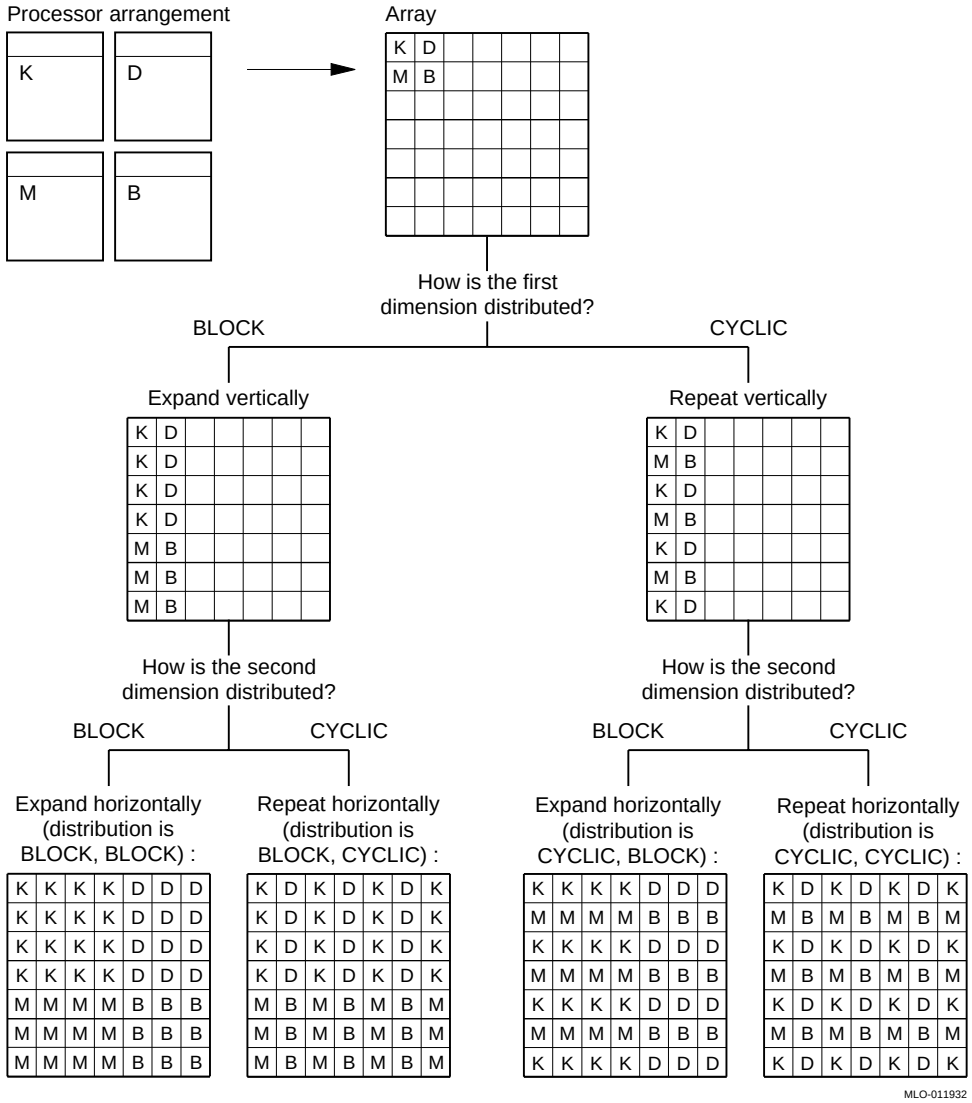
This technique can be used for all two-dimensional distributions.

This technique cannot be used to figure distributions containing an asterisk (*), distributions of one-dimensional arrays, or distributions of arrays with more than two dimensions.

This manual refers to the first axis as vertical for both arrays and processor arrangements. See the "Note about Rows and Columns" in Section 5.5.6.1.

Precise formulas that are valid for all distributions can be found in the High Performance Fortran Language Specification.

Figure 5–21 Visual Technique for Computing Two-Dimensional Distributions



5.5.6.10 Using DISTRIBUTE Without an Explicit Template

When a template is not explicitly named in a TEMPLATE directive, the name of an array takes the place of the name of the template in the DISTRIBUTE directive. Any array can be distributed as long as it is never an alignee (to the left of the keyword WITH) in an ALIGN directive. The following two versions of the code fragment from Example 5–1 are equivalent when compiled for four processors:

- With an explicit TEMPLATE:

```
      REAL A(12, 12)
      REAL B(16, 16)
!HPF$ TEMPLATE T(16,16)
!HPF$ ALIGN B WITH T
!HPF$ ALIGN A(i, j) WITH T(i+2, j+2)
!HPF$ PROCESSORS P(2, 2)
!HPF$ DISTRIBUTE T(BLOCK, BLOCK) ONTO P
```

- Without an explicit TEMPLATE:

```
      REAL A(12, 12)
      REAL B(16, 16)
!HPF$ ALIGN A(i, j) WITH B(i+2, j+2)
!HPF$ PROCESSORS P(2, 2)
!HPF$ DISTRIBUTE B(BLOCK, BLOCK) ONTO P
```

5.5.6.11 Using DISTRIBUTE Without an Explicit PROCESSORS Directive

When the PROCESSORS directive is not used, the ONTO clause must be omitted from the DISTRIBUTE directive. The code fragment from Example 5–1 without a PROCESSORS directive looks like this:

```
      REAL A(12, 12)
      REAL B(16, 16)
!HPF$ ALIGN A WITH B(i+2, j+2)
!HPF$ DISTRIBUTE B(BLOCK, BLOCK)
```

When the DISTRIBUTE directive is used without an ONTO clause, the compiler provides a default processor arrangement. The compiler attempts to select an efficient shape for the default processor arrangement, but you should not rely on the arrangement being any one particular shape. In the above code fragment, for example, possible processor arrangements shapes are 4×1, 1×4, or 2×2. If you want one particular shape, use the PROCESSORS directive and distribute the array with an ONTO clause.

5.5.6.12 Deciding on a Distribution

There is no completely general rule for determining which data mapping is most efficient, because optimal data distribution is highly algorithm-dependent. In most parallel programming environments, communication between processors is more time-consuming than computation by a huge margin. Therefore, the primary goal in choosing a data mapping is to minimize communication between processors. A secondary goal is to balance the computational load among the processors.

Array assignments in which the calculation of each array element requires information only from its near neighbors generally run faster with a BLOCK distribution, because this allows the processor calculating any given array element to have all of the necessary data in its own memory in most cases. The Compaq Fortran compiler includes an optimization which minimizes communication even along the edges of blocks in nearest-neighbor calculations. For an example of a nearest neighbor calculation, see Appendix C.

When the calculation requires information from distant elements in the array, a CYCLIC distribution is frequently faster because it improves load-balancing, dividing the work more evenly among processors. See Section B.3.3.

Some algorithms are column-oriented or row-oriented in the data they require for each calculation. These algorithms frequently benefit from a distribution with an asterisk (*) in one of the dimensions.

The distribution figures in this chapter can be used as a guide to the basic distribution choices for two-dimensional arrays; more complex distributions such as CYCLIC(*n*) (CYCLIC with an *intr-expr*) are also supported. See the High Performance Fortran Language Specification for documentation of these complex distributions.

It is worthwhile to make an initial guess at a distribution and then try a few alternatives to see which performs best. One of the advantages of HPF is that changing distributions is an easy change to the coding of the program.

For More Information:

- On selecting a distribution, see Chapters B and C.
- On the nearest neighbor optimization, see Sections C.5.2 and 6.1.1.5.
- On the conditions that allow the compiler to recognize a statement as a nearest-neighbor calculation, see Section 7.6.

5.5.7 SHADOW Directive for Nearest-Neighbor Algorithms

The Compaq Fortran compiler performs an optimization of nearest-neighbor algorithms to reduce communications. **Shadow edges** are allocated to hold copies of array elements that are near neighbors to each processor's edge elements.

The SHADOW directive can be used to manually set shadow-edge widths for each array dimension. When you do not list an array in a SHADOW directive, the compiler sizes the shadow edge automatically, based on your algorithm. For example:

```
      REAL A(1000, 1000, 1000)
!HPF$ DISTRIBUTE A(BLOCK, BLOCK, BLOCK)
!HPF$ SHADOW A(3,2,0)
```

In this example, shadow edges 3 array elements wide will be allocated for the first dimension of array A. Shadow edges 2 array elements wide will be allocated for the second dimension of A. No shadow storage will be allocated for the third dimension, because a shadow-edge width of 0 is specified.

When an array is not listed in a SHADOW directive, the compiler automatically sizes the shadow edge for all dimensions. You will usually obtain the full performance benefit of the nearest neighbor optimization by relying on the compiler's automatic shadow-edge sizing.

The primary use of the SHADOW directive is preventing copy in/copy out when arrays used in nearest-neighbor computations are passed through the procedure interface. If you want to conserve memory by limiting the sizes of shadow-edge widths, it is usually preferable to use the `-nearest_neighbor` compile-time command-line option.

However, there are some situations where shadow-edge widths should be set manually:

- For any array involved in nearest-neighbor calculations in both a subprogram (other than a contained subprogram) and its caller.
- For any array involved in nearest-neighbor calculations that was declared in the specification part of a module.
- For any POINTER array involved in nearest-neighbor calculations.

In these cases, setting shadow-edge widths manually leads to more efficient memory usage and prevents unnecessary local copying of data.

You can limit shadow-edge widths to a certain maximum value with the `-nearest_neighbor` option.

The nearest neighbor optimization can be disabled with the `-nonearest_neighbor` command-line option. This option has the same effect as setting all shadow-edge widths to zero (0).

For More Information:

- On the nearest neighbor optimization, see Sections C.5.2 and 6.1.1.5.
- On the conditions that allow the compiler to recognize a statement as a nearest-neighbor computation, see Section 7.6.

5.6 Subprograms in HPF

Parallel programming introduces a new complication for procedure calls: the interface between the procedure and the calling program must take into account not only the type, kind, rank, and size of the relevant objects, but also their mapping across the processors in a cluster.

For More Information:

- On features for handling subprograms in Compaq Fortran, see the *Compaq Fortran Language Reference Manual*.

5.6.1 Assumed-Size Array Specifications

Compaq Fortran supports assumed-size array specifications. However, arguments passed using assumed-size dummies are not handled in parallel, and typically degrade performance.

An **assumed-size array** is a dummy array argument whose size is assumed from its associated actual. Its rank and extents may differ from its actual. Only its size is assumed, and only in the last dimension. Assumed-size dummies are always mapped as serial replicated. Using assumed-size dummies in HPF programs can cause major performance degradation.

For More Information:

- On the definition and syntax of assumed-size array specifications, see the *Compaq Fortran Language Reference Manual*.

5.6.2 Explicit Interfaces

In HPF, the mapping of a dummy argument in a called routine is usually required to be visible to the calling routine in an **explicit interface**. An explicit interface consists of one of the following:

1. USE association — the calling routine may contain a USE statement referring to a module that contains the called routine, or contains an explicit interface for the called routine.

2. Host association — the calling routine may call a routine contained in the same scope.
3. Explicit interface block — the calling routine may contain an interface block describing the called routine. The interface block must contain dummy variable declarations and mapping directives that match the routine it describes.

Enclosing subroutines with a MODULE/END MODULE statement is an easy way to provide explicit interfaces for subroutines.

The High Performance Fortran Language Specification permits the explicit interface to be omitted in some cases (roughly speaking, when the dummy can get the contents of the actual without inter-processor communication). However, Compaq strongly recommends using explicit interfaces whenever a dummy is mapped. This is good programming practice and provides more information to the compiler. The compiler often produces more efficient executables when it is provided with more information.

For More Information:

- On providing explicit interfaces for legacy code, see Section 5.6.3.

5.6.3 Module Program Units

Module program units replace the old BLOCK DATA and COMMON techniques for passing data to subprograms.

Modules are useful for structuring a program into libraries of:

- Commonly used procedures
- Encapsulated derived data types and their defined operators and assignment
- Packages of related global data definitions

Modules can be a very easy way to provide explicit interfaces with very little programming effort. The HPF language generally requires explicit interfaces when dummy arguments are mapped.

Simply enclose your subroutines with MODULE and END MODULE statements. Multiple subroutines can be enclosed in the same module. Then add a USE statement to each calling scope. In this manner, by adding as few as three lines of source code, explicit interfaces can be provided for an entire program.

The following is an example of a module that contains a procedure that can be called from a main program with a USE statement:

```
MODULE FUNCTION_TO_BE_INTEGRATED
CONTAINS

  PURE REAL FUNCTION F(X)       ! FUNCTION TO BE INTEGRATED
    REAL, INTENT(IN) :: X
    F = 4 / (1.0 + X**2)
  END FUNCTION F

END MODULE FUNCTION_TO_BE_INTEGRATED
```

In this example, the function F(X) can be defined in the calling scope with a USE statement:

```
USE FUNCTION_TO_BE_INTEGRATED
```

Within the scope where the USE statement appeared, the function F(X) has an explicit interface.

For More Information:

- On HPF language rules for when an explicit interface is needed, see Section 5.6.2.
- For another example of the use of modules, see Appendix E.

5.6.4 PURE Attribute

In HPF, a **pure function** or **pure subroutine** is one that produces no side effects and makes no reference to mapped variables other than its actual argument. This means that a pure function's only effect on the state of a program is to return a value, and a pure subroutine's only effect on the state of a program is to modify INTENT(OUT) and INTENT(INOUT) parameters.

User-defined functions may be called inside a FORALL structure only if they are pure functions. Subroutines called by PURE functions must be pure. Because a FORALL structure is an extended assignment statement (not a loop), there is no way to directly express a subroutine call from within a FORALL structure; however, a PURE function that is called within a FORALL structure may itself call a pure subroutine.

Assigning the PURE attribute to a function allows that function to be called inside a FORALL structure. Assigning the PURE attribute to a subroutine lets that subroutine be called inside a PURE function.

The PURE attribute is required only for functions called within a FORALL structure. Functions called in Fortran 90 array assignment statements or INDEPENDENT DO loops do not need to be pure.

The PURE attribute was designed to avoid two separate problems that are otherwise possible in FORALL structures:

- Cases of program indeterminacy
- Processor synchronization irregularities

Therefore, constraints on PURE functions and subroutines include restrictions on both side effects and data mapping. The features necessary to permit a function or subroutine to be assigned the PURE attribute in Compaq Fortran include:

- Not modifying the value of any global variable
- Not referencing any impure function or subroutine
- Not performing any Input/Output
- Not assigning the SAVE attribute, even to a dummy variable
- Not containing any ALIGN, DISTRIBUTE, or INHERIT directives
- Not mentioning any variable name that appears in a DISTRIBUTE or ALIGN directive anywhere in the program
- Not mentioning any variable name having sequence, storage, pointer, host, or use association with another variable name that appears in a DISTRIBUTE or ALIGN directive anywhere in the program
- All dummy arguments in a PURE function or subroutine (except procedure arguments and arguments with the POINTER attribute) must have INTENT(IN).

Some additional prohibitions apply to PURE functions that do not apply to PURE subroutines. Most notably, a PURE subroutine can modify its arguments, whereas a PURE function cannot do so. These additional prohibitions are listed in the High Performance Fortran Language Specification.

Because PURE functions and subroutines, like all Fortran functions and subprograms, may be compiled separately, the compiler has no way of evaluating the accuracy of a program's assertion that a procedure is PURE. The programmer must take responsibility for checking these conditions. Illegal use of the PURE attribute is not detected by the compiler, and may result in incorrect program results.

The following is an example use of the PURE attribute:

```
PURE FUNCTION DOUBLE(X)
  REAL, INTENT(IN) :: X
  DOUBLE = 2 * X
END FUNCTION DOUBLE
```

5.6.5 Transcriptive Distributions and the INHERIT Directive

Transcriptive mapping is used to handle the case when the mapping of dummy arguments is not known at compile time. The compiler makes a generalized version of the subprogram that can accept whatever mapping the arguments have when passed in.

Transcriptive distribution is specified with an asterisk. For example:

```
!HPF$ DISTRIBUTE A *
```

This specifies that the dummy argument should be distributed in the same way as the actual.

There is no transcriptive form of the ALIGN directive. Transcriptive alignment is specified with the INHERIT directive. The INHERIT attribute specifies that a dummy argument should be aligned and distributed in the same way as the actual, if the actual has been named in an ALIGN directive.

Using transcriptive mappings forces the compiler to generate code that is generalized for any possible alignment, which may be less efficient. The best performance is obtained by explicitly specifying data mapping for the subprogram.

The following example shows two ways of passing a mapped actual to a subroutine, one with transcriptive mapping, and one with explicit mapping:

```
! With Transcriptive Mapping          | ! With Explicit Mapping
! -------------------------           | ! -------------------------------
      PROGRAM foo                     |       PROGRAM foo
      INTEGER T(100, 100)             |       INTEGER T(100, 100)
      INTEGER U(50, 50)               |       INTEGER U(50, 50)
!HPF$ DISTRIBUTE T(BLOCK, BLOCK)      | !HPF$ DISTRIBUTE T(BLOCK, BLOCK)
!HPF$ ALIGN U(I,J) WITH T(I+50,J+50)| !HPF$ ALIGN U(I,J) WITH T(I+50,J+50)
       .                              |        .
       .                              |        .
       .                              |        .
                                      |
      CALL bar(U)                     |       CALL bar(T, U)
       .                              |        .
       .                              |        .
       .                              |        .
                                      |
      CONTAINS                        |       CONTAINS
                                      |
      SUBROUTINE bar(R)               |       SUBROUTINE bar(Q, R)
      INTEGER R(:, :)                 |       INTEGER Q(:, :)
!HPF$ INHERIT R                       |       INTEGER R(:, :)
       .                              | !HPF$ DISTRIBUTE Q (BLOCK, BLOCK)
       .                              | !HPF$ ALIGN R(I, J) WITH Q(I+50, J+50)
       .                              |        .
       .                              |        .
       .                              |        .
                                      |
      END SUBROUTINE bar              |       END SUBROUTINE bar
      END PROGRAM foo                 |       END PROGRAM foo
```

In the preceding example, the array *U* was aligned with another array *T*. When INHERIT is used, there is no need to mention this alignment in the subroutine, because the INHERIT directive makes sure that the mapping of the actual is fully preserved, including alignment. When explicit distributions are given in the subroutine, the align target (Q) must be passed to the subroutine and the alignment must be specified in the subroutine.

If *U* were aligned with a template instead of an array, the template could not be passed as an argument to the subroutine, because templates cannot be passed through the interface (see Section 5.5.4). When explicit directives are used, the template must be declared in the subroutine.

However, when the INHERIT directive is used, there is no need to declare the template in the subroutine. See the following example:

```
! With Transcriptive Mapping        | ! With Explicit Mapping
! -------------------------        | ! --------------------------------
      PROGRAM foo                   |       PROGRAM foo
!HPF$ TEMPLATE T(100, 100)          | !HPF$ TEMPLATE T(100, 100)
      INTEGER U(50, 50)             |       INTEGER U(50, 50)
!HPF$ DISTRIBUTE T(BLOCK, BLOCK)    | !HPF$ DISTRIBUTE T(BLOCK, BLOCK)
!HPF$ ALIGN U(I,J) WITH T(I+50,J+50)| !HPF$ ALIGN U(I,J) WITH T(I+50,J+50)
       .                            |        .
       .                            |        .
       .                            |        .
                                    |
      CALL bar(U)                   |       CALL bar(U)
       .                            |        .
       .                            |        .
       .                            |        .
                                    |
      CONTAINS                      |       CONTAINS
                                    |
      SUBROUTINE bar(R)             |       SUBROUTINE bar(R)
      INTEGER R(:, :)               |       INTEGER R(:, :)
!HPF$ INHERIT R                     | !HPF$ TEMPLATE Q(100, 100)
       .                            | !HPF$ DISTRIBUTE Q (BLOCK, BLOCK)
       .                            | !HPF$ ALIGN R(I, J) WITH Q(I+50, J+50)
       .                            |        .
       .                            |        .
       .                            |        .
                                    |
      END SUBROUTINE bar            |       END SUBROUTINE bar
      END PROGRAM foo               |       END PROGRAM foo
```

Note that in the template declaration, it was necessary to use constant values (100, 100) rather than assumed-shape syntax (:, :) because templates cannot be passed through the interface.

Another possibility would be to align R directly with the template T from the main program. Because bar is a contained procedure, T is available through host association. T would also be available to a module procedure through use association. However, this might be considered undesirable programming practice.

5.7 Intrinsic and Library Procedures

Fortran 90 defines over 100 built-in or intrinsic procedures, some inherited from Fortran 77, some new (see the *Compaq Fortran Language Reference Manual*). In addition, HPF introduces new intrinsic procedures. HPF also defines a library module HPF_LIBRARY that adds further to the power of the language.

5.7.1 Intrinsic Procedures

HPF adds the following to the standard Fortran 90 intrinsic procedures:

- System inquiry functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE
- The optional DIM argument to the Fortran 90 intrinsic functions MAXLOC and MINLOC
- A new elemental intrinsic function ILEN to compute the number of bits needed to store an integer value

All intrinsic procedures are PURE, except for RAN and SECNDS.

The intrinsic procedures that reference the system clock require special consideration in parallel HPF programs, because more than one system clock might be referenced if multiple hosts are involved in the execution. Only SECNDS synchronizes the processors to ensure consistency (at some cost in performance). The other time intrinsics do not guarantee consistency of clocks of different systems.

For More Information:

- On HPF library procedures, see the appropriate man pages.
- For a complete list of the intrinsic procedures provided with Compaq Fortran, see the *Compaq Fortran Language Reference Manual*.

5.7.2 Library Procedures

Compaq Fortran anticipates many operations that are valuable for parallel algorithm design. HPF adds a standard library of functions which includes:

- Mapping inquiry subroutines to determine the actual data mapping of arrays at run time
- Bit manipulation functions LEADZ, POPCNT, and POPPAR
- New array reduction functions IALL, IANY, IPARITY, and PARITY
- Array combining scatter functions, one for each reduction function
- Array prefix and suffix functions, one each for each reduction function
- Two array sorting functions

For More Information:

- For a complete list of the library procedures provided with Compaq Fortran, see the *Compaq Fortran Language Reference Manual*.

5.8 Extrinsic Procedures

HPF provides a mechanism by which HPF programs may call procedures written in other parallel programming styles or other programming languages. Because such procedures are themselves outside HPF, they are called **extrinsic procedures**.

In Compaq Fortran, there are three kinds of EXTRINSIC procedures:

- EXTRINSIC(HPF) (data parallel)
- EXTRINSIC(HPF_LOCAL) (explicit SPMD)
- EXTRINSIC(HPF_SERIAL) (single processor)

In Compaq Fortran, the keywords EXTRINSIC(HPF), EXTRINSIC(HPF_LOCAL), and EXTRINSIC(HPF_SERIAL) can be used as prefixes for modules, block data program units, functions, and subroutines.

These three kinds of EXTRINSIC procedures are explained in Section 5.8.1.

5.8.1 Programming Models and How They Are Specified

The following list describes programming models:

- Data Parallel Model [EXTRINSIC(HPF)]

  This is a regular HPF procedure. It is the default programming model, producing the same result as if the EXTRINSIC prefix is not used at all. The language contains directives which you may use to tell the compiler how data is distributed across processors. Parallelism is attained by having processors operate simultaneously on different portions of this data. However, from the program's viewpoint, there is only a single thread of control.

  The source code does not ordinarily contain SEND or RECEIVE calls. If they are present in the source, they refer to communication with some exterior program (operating on another set of processors).

  Procedures written in this fashion are referred to as **global HPF** (or simply "HPF") procedures.

- Explicit SPMD Model [EXTRINSIC(HPF_LOCAL)]

  A program is written containing explicit SENDs and RECEIVEs and references to GET_HPF_MYNODE (see the online reference page for GET_HPF_MYNODE), and is loaded onto all processors and run in parallel. Storage for all arrays and scalar variables in the program is privatized; identical storage for each such data object exists on each processor. In general, the storage for an array *A* on distinct processors represents distinct slices through a global array with which you are really

concerned (only you are aware of these global arrays, however; they have no representation in the source code).

This is a multithreaded programming model, where the same program runs on each processor. Different processors execute different instructions, because the program on each processor is parameterized by references to the processor number, and because parallel data on different processors in general has different values. Since the data on each processor consists of different slices of global arrays, this programming model is referred to as **single-program, multiple-data** or SPMD.

Each array element really has two kinds of addresses: a global address and a two-part address which consists of a particular processor and a local memory address in that processor. It is up to you to handle the translation between these two forms of addressing. Similarly, it is up to you to insert whatever SENDs and RECEIVEs are necessary. This involves figuring out which array elements have to be sent or received and where they have to go or come from. This requires you to explicitly translate between global and local addressing.

Within the context of Compaq Fortran, the explicit SPMD model is supported by EXTRINSIC(HPF_LOCAL) procedures. These are procedures coded as shown previously, but they also have inquiry library routines available to them. These enable you to retrieve information about global arrays corresponding to dummy arrays in the procedure.

- Single Processor Model [EXTRINSIC(HPF_SERIAL)]

  This is the conventional Fortran programming model. A program is written to execute on one processor which has a single linear memory. Sequence and storage association holds in this model, because it implements conventional Fortran 90.

  In this implementation, a single processor procedure may execute on any processor, with two exceptional cases:

  – If it is the main program (a Compaq Fortran program compiled without the `-hpf` option), it always executes on processor 0.

  – It may additionally be declared to be EXTRINSIC(HPF_SERIAL). This is an indication that it may be called from a global HPF procedure, and that the compiler generates code in the global procedure to move all the arguments to a single processor before the call and back afterwards.

The compiler can be invoked in several ways, corresponding to these different programming models:

- When invoked with the `-hpf` switch but without an EXTRINSIC declaration, the compiler expects a global HPF source program and generates the correct addressing and message-passing in the emitted object code.

  The compiler produces code containing explicit SENDs and RECEIVEs and references to the processor number. In effect, the compiler accepts a global HPF program and emits explicit SPMD object code.

  This implements the data parallel programming model.

- Without the `-hpf` switch and without an EXTRINSIC declaration, the compiler generates addressing for a single linear memory. Such a procedure can be used to implement the explicit SPMD programming model by containing programmer-written message passing as needed. It can also be used to implement the single processor model. There is no way to discover by inspecting the source code of such a procedure which programming model is intended. The programming model is determined by how the procedure is invoked: on one processor or simultaneously on many.

- If the procedure is declared as EXTRINSIC(HPF_LOCAL) (if this procedure is called from a global HPF procedure, this declaration must also be visible in an explicit interface in the calling procedure), the procedure becomes an explicit SPMD procedure. The HPF_LOCAL declaration also has two other effects:

  - If the calling procedure is global HPF, it is a signal to the compiler when compiling the calling procedure to localize the passed parameters. That is, only the part of each passed parameter which lives on a given processor is passed as the actual parameter to the instance of the HPF_LOCAL procedure which is called on that processor.

  - It makes available to the HPF_LOCAL procedure a library of inquiry routines (the HPF Local Routine Library) which enable the local procedure to extract information about the global HPF data object corresponding to a given local dummy parameter. This information can be used (explicitly, by the programmer) to set up interprocessor communication as needed.

- If the procedure is declared as EXTRINSIC(HPF_SERIAL) the compiler processes it as any other single-processor Fortran 90 procedure. If the procedure is called from a global HPF procedure, the EXTRINSIC(HPF_SERIAL) declaration must be visible to the calling procedure in an explicit interface. The compiler generates code in the

calling procedure that (before the call) moves all arguments to the processor on which the subprogram executes, and copies them back after the call if necessary.

An EXTRINSIC(HPF_LOCAL) or an EXTRINSIC(HPF_SERIAL) declaration in a procedure overrides the `-hpf` switch — as if that switch were not invoked for the procedure. This makes it possible to have an HPF_LOCAL or HPF_SERIAL subprogram in the same file as that procedure. The compiler is invoked with the `-hpf` switch, but that switch has no effect on the compilation of the HPF_LOCAL or HPF_SERIAL subprocedure.

5.8.2 Who Can Call Whom

A single processor procedure which is the main program always executes on processor 0. Other than that, there is no restriction on where a single processor procedure executes.

A single processor procedure can call a single processor procedure.

A global HPF procedure can call:

- A global HPF procedure
- An HPF_LOCAL procedure
- An HPF_SERIAL procedure

An HPF_LOCAL procedure can call:

- An explicit SPMD procedure, which might be another HPF_LOCAL procedure
- A single processor procedure, which runs on the processor from which it is called (an example of this could be a scalar library procedure)

An explicit SPMD procedure that is not an HPF_LOCAL procedure (such as subroutine bar in Section 5.8.2.1) can call:

- Another explicit SPMD procedure (which cannot be an HPF_LOCAL procedure)
- A single processor procedure, which runs on the processor from which it is called

This relaxes some of the restrictions in Annex A of the High Performance Fortran Language Specification.

5.8.2.1 Calling Non-HPF Subprograms from EXTRINSIC(HPF_LOCAL) Routines

According to the High Performance Fortran Language Specification, EXTRINSIC(HPF_LOCAL) routines are only allowed to call other EXTRINSIC(HPF_LOCAL) routines, EXTRINSIC(F90_LOCAL) routines, or other extrinsic routines that preserve EXTRINSIC(HPF) semantics.

Compaq Fortran does not currently support the optional extrinsic prefix EXTRINSIC(F90_LOCAL). However, Compaq relaxes the restriction given in the High Performance Fortran Language Specification and allows (non-HPF) Fortran routines to be called from EXTRINSIC(HPF_LOCAL) routines. This is done by calling the non-HPF subprogram without an EXTRINSIC prefix, as in the following example:

```
! The main program is an EXTRINSIC(HPF) routine
      PROGRAM MAIN
      INTERFACE
         EXTRINSIC(HPF_LOCAL) SUBROUTINE foo
         END SUBROUTINE foo
      END INTERFACE

      CALL foo()
      END

! foo is an EXTRINSIC(HPF_LOCAL) routine

      EXTRINSIC(HPF_LOCAL) SUBROUTINE foo
      INTERFACE
        SUBROUTINE bar(B)
        REAL B(100)
        END SUBROUTINE bar
      END INTERFACE
      REAL A(100)

      CALL bar(A)
      PRINT *, A(1)
      END SUBROUTINE foo

! bar is declared without an EXTRINSIC prefix and not compiled with -hpf

      SUBROUTINE bar(B)
      REAL B(100)

      B = 1.0
      END SUBROUTINE bar
```

The non-HPF routine `bar` is called from an EXTRINSIC(HPF_LOCAL) routine. It is declared without using an EXTRINSIC prefix and is not compiled with `-hpf`. This is the only method of calling existing routines with non-assumed-shape arguments from EXTRINSIC(HPF_LOCAL) routines. This can be useful, for example, if you wish to call an existing routine written in Fortran 77, or in C.

5.8.3 Requirements on the Called EXTRINSIC Procedure

HPF requires a called EXTRINSIC(HPF_LOCAL) or EXTRINSIC(HPF_SERIAL) procedure to satisfy the following behavioral requirements:

- The overall implementation must behave as if all actions of the caller preceding the subprogram invocation are completed before any action of the subprogram is executed; and as if all actions of the subprogram are completed before any action of the caller following the subprogram invocation is executed.
- IN/OUT intent restrictions declared in the interface for the extrinsic subroutine must be obeyed.
- Replicated variables, if updated, must be updated consistently. If a variable accessible to a local subprogram has a replicated representation and is updated by (one or more copies of) the local subroutine, all copies of the replicated data must have identical values when the last processor returns from the local procedure.
- No HPF variable is modified unless it could be modified by an EXTRINSIC(HPF) procedure with the same explicit interface.
- When a subprogram returns and the caller resumes execution, all objects accessible to the caller after the call are mapped exactly as they were before the call. As with an ordinary HPF subprogram, actual arguments may be copied or remapped in any way as long as the effect is undone on return from the subprogram.
- Exactly the same set of processors is visible to the HPF environment before and after the subprogram call.

For More Information:

- On the EXTRINSIC prefix, see the High Performance Fortran Language Specification.

5.8.4 Calling C Subprograms from HPF Programs

To write EXTRINSIC routines in C (or other non-HPF languages), you must make your subprogram conform to Fortran calling conventions. In particular, the subprogram may have to access information passed through dope vectors.

For More Information:

- On mixed-language programming in general, see the *Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems*.

6

Compiling and Running HPF Programs

This chapter describes:

- Section 6.1, Compiling HPF Programs
- Section 6.2, HPF Programs with MPI

Note

You should use the `-hpf` option instead of its predecessor, the `-wsf` option. Similarly, the `-hpf_target` option is preferred to the `-wsf_target` option and the `-nohpf_main` option is preferred to the `-nowsf_main`. Finally, you should use the environment variable `DECF90_HPF_TARGET` instead of `DECF90_WSF_TARGET`. If you use `DECF90_WSF_TARGET`, you get a warning message and the value of `DECF90_HPF_TARGET` is used.

The compiler gives a warning message whenever it sees one of the options `-wsf`, `-wsf` *n*, `-wsf_target`, or `-nowsf_main`. Your program will continue to compile, however. The compiler also replaces `wsf` text with `hpf` when it creates the listing file.

6.1 Compiling HPF Programs

The Compaq Fortran compiler can be used to produce either standard applications that execute on a single processor (serial execution), or parallel applications that execute on multiple processors. Parallel applications are produced by using the Compaq Fortran compiler with the `-hpf` option.

Note

In order to achieve parallel execution, Fortran programs must be written with HPF (High Performance Fortran) directives and without reliance on sequence association. For information about HPF, see Chapters 5 and 7. The HPF Tutorial is contained in Appendixes A, B, C, D, and E.

6.1.1 Compile-Time Options for High Performance Fortran Programs

This section describes the Compaq Fortran command-line options that are specifically relevant to parallel HPF programs.

6.1.1.1 -hpf [*nn*] Option — Compile for Parallel Execution

Specifying the `-hpf` option indicates that the program should be compiled to execute in parallel on multiple processors.

HPF directives in programs affect program execution only if the `-hpf` option is specified at compile time. If the `-hpf` option is omitted, HPF directives are checked for syntax, but otherwise ignored.

Specifying `-hpf` with a number as an argument optimizes the executable for that number of processors. For example, specifying `-hpf 4` generates a program for 4 processors. Specifying `-hpf` without an argument produces a more general program that can run on any arbitrary number of processors. Using a numerical argument results in superior application performance.

For best performance, do not specify an argument to `-hpf` that is greater than the number of CPUs that will be available at run time. Relying on the `-virtual` run-time option to simulate a cluster larger than the number of available processors usually causes degradation of application performance.

Any number of processors is allowed. However, performance may be degraded in some cases if the number of processors is not a power of two.

The `-nearest_neighbor` and `-show hpf` options can be used only when `-hpf` is specified.

When parallel programs are compiled and linked as separate steps (see the documentaton of the `-c` option in the *Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems*), the `-hpf` option must be used with the `f90` command both at compile time and link time. If `-hpf` is used with a numerical argument, the same argument must be used at compile time and link time.

For More Information:

- On processor arrangements, see Section 5.5.5.
- On compiling and linking as separate steps, see the documentaton of the `-c` option in the *Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems*.

6.1.1.2 -assume bigarrays Option — Assume Nearest-Neighbor Arrays are Large

Certain nearest-neighbor computations can be better optimized when the compiler assumes that the number of elements in the relevant nearest-neighbor arrays is big enough. An array is big enough if any two array elements shadow-edge-width apart in any distributed dimension are in the same processor or in adjacent processors. The `-assume bigarrays` option permits the compiler to make this assumption. This assumption is true in the typical case.

The `-assume bigarrays` option is automatically specified when `-fast` is specified. When `-assume bigarrays` is wrongly specified, references to small arrays in nearest-neighbor computations will fail with an error message.

For More Information:

- On `-fast`, see Section 6.1.1.4.

6.1.1.3 -assume nozsize Option — Omit Zero-Sized Array Checking

An array (or array section) is zero-sized when the extent of any of its dimensions takes the value zero or less than zero. When the `-hpf` option is specified, the compiler is required to insert a series of checks to guard against irregularities (such as division by zero) in the generated code that zero-sized data objects can cause. Depending upon the particular application, these checks can cause noticeable (or even major) degradation of performance.

The `-assume nozsize` option causes the compiler to omit these checks for zero-sized arrays and array sections. This option is automatically selected when the `-fast` option is selected.

The `-assume nozsize` option may not be used when a program references any zero-sized arrays or array sections. An executable produced with the `-assume nozsize` option may fail or produce incorrect results when it references any zero-sized arrays or array sections.

You can insert a run-time check into your program to ensure that a given line is not executed if an array or array section referenced there is zero-sized. This will allow you to specify `-assume nozsize` even when there is a possibility of a zero-sized array reference in that line.

For More Information:

- On using run-time checks for zero-sized data objects, see Section 7.1.

6.1.1.4 -fast Option — Set Options to Improve Run-Time Performance

The `-fast` option activates options that improve run-time performance. A full list of the options set by `-fast` can be found on the `f90`(1) reference page.

Among the options set by the `-fast` option are the `-assume nozsize` option and the `-assume bigarrays` option. This means that the restrictions that apply to these options also apply to the `-fast` option.

For More Information:

- On the `-assume nozsize` option, see Section 6.1.1.3.
- On the `-assume bigarrays` option, see Section 6.1.1.2.

6.1.1.5 -nearest_neighbor [*nn*] and -nonearest_neighbor Options — Nearest Neighbor Optimization

The compiler's nearest-neighbor optimization is enabled by default. The `-nearest_neighbor` option is used to modify the limit on the extra storage allocated for nearest neighbor optimization.

The `-nonearest_neighbor` option is used to disable nearest neighbor optimization.

The compiler automatically determines the correct shadow-edge widths on an array-by-array, dimension-by-dimension basis. You can also set shadow-edge widths manually by using the SHADOW directive. You must use the SHADOW directive to preserve the shadow edges when nearest-neighbor arrays are passed as arguments.

The optional *nn* field specifies the maximum allowable shadow-edge width in order to set a limit on how much extra storage the compiler may allocate for nearest-neighbor arrays. The nearest-neighbor optimization is not performed for array dimensions needing a shadow-edge width greater than *nn*.

When programs are compiled with the `-hpf` option, the default is `-nearest_neighbor 10`.

The `-nonearest_neighbor` option disables the nearest-neighbor optimization. It is equivalent to specifying `-nearest_neighbor 0`.

For More Information:

- On using the SHADOW directive to specify shadow-edge width, see Section 5.5.7.
- On Compaq Fortran's nearest neighbor optimization, see Section C.5.2.
- On the conditions that allow the compiler to recognize a statement as a nearest-neighbor computation, see Section 7.6.

6.1.1.6 -nohpf_main Option — Compiling Parallel Objects to Link with a Non-Parallel Main Program

Use the `-nohpf_main` option to incorporate parallel routines into non-parallel programs.

When you incorporate parallel routines into non-parallel programs, some routines must be compiled with `-nohpf_main`, and some should be compiled without `-nohpf_main`.

For More Information:

- On mixed-language programming in general, see the *Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems*.

6.1.1.7 -show hpf—Show Parallelization Information

The `-show hpf` option sends information related to parallelization to standard error and to the listing (if one is generated with `-V`). These flags are valid only if the `-hpf` flag is specified. You can use this information to help you tune your program for better performance.

This option has several forms:

- `-show hpf_comm` includes detailed information about statements which cause interprocessor communication to be generated. This option typically generates a very large number of messages.
- `-show hpf_indep` includes information about the optimization of DO loops marked with the INDEPENDENT directive. Every marked loop will be acknowledged and an explanation given for any INDEPENDENT DO loop that was not successfully parallelized.
- `-show hpf_nearest` includes information about arrays and statements involved in optimized nearest-neighbor computations. Messages are generated only for statements that are optimized. This option allows you to check whether statements that you intended to be nearest-neighbor are successfully optimized by the compiler. It is also useful for finding out shadow-edge widths that were automatically generated by the compiler.

- -show hpf_punt gives information about distribution directives that were ignored and statements that were not handled in parallel. For more information on serialized routines, see Section 7.4.
- -show hpf_temps gives information about temporaries that were created at procedure interfaces.
- -show hpf_all is the same as specifying all the other -show hpf_ options.
- -show hpf generates a selected subset of those messages generated by the other -show hpf_ options. It is designed to provide the most important information while minimizing the number of messages. It provides the output of -show hpf_indep, -show hpf_nearest, and -show hpf_punt, as well as selected messages from -show hpf_comm.

It is usually best to try using -show hpf first. Use the others only when you need a more detailed listing.

-show can take only one argument. However, the -show flags can be combined by specifying -show multiple times. For example:

```
% f90 -hpf -show hpf_nearest -show hpf_comm -show hpf_punt foo.f90
```

For More Information:

- On routines and statements that are not handled in parallel, see Section 7.4.
- For an example of the output from -show hpf, see Section 7.13.

6.1.2 Consistency of Number of Peers

When linking is done as a separate step from compiling, the Compaq Fortran compiler requires all objects to be compiled with the same argument to the -hpf option's optional *[nn]* field. If objects were compiled for an inconsistent number of processors, the following error message occurs:

```
Unresolved:
_hpf_compiled_for_nn_nodes_
```

If you do not know which object was compiled for the wrong number of processors, the incorrectly compiled object can be identified using the UNIX nm utility.

For More Information:

- On doing compiling and linking as separate steps, see the -c option in the *Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems*.

6.2 HPF Programs with MPI

HPF programs execute with the support of Message Passing Interface (MPI) software.

The HPF/MPI connection enables development of programs that use both HPF and MPI together in the same application. There are many tools available for debugging, profiling, and visualizing the execution of MPI programs.

For More Information:

- See the Web site for the Message Passing Interface Forum at:

  `http://www.mpi-forum.org/`

6.2.1 Overview of HPF and MPI

The Compaq Fortran compiler generates code that uses MPI as its message-passing library. The compiler provides a choice of three different variants of MPI:

- One for Compaq's SC supercomputer systems
- One that supports shared-memory and Memory Channel interconnects
- Public domain MPI for other interconnects that include Ethernet and FDDI

To write HPF programs that also call or use MPI (such as distributed-memory libraries that invoke MPI), you must use the MPI-based run-time library. The compiler's MPI run-time library uses its own private MPI "communicator," and thus will not interfere with other MPI code.

6.2.2 Compiling HPF Programs for MPI

You specify MPI support for HPF programs by including the option `-hpf_target` with an MPI selection (argument *target*) in the command to the Fortran 90/95 compiler. For example, to select Compaq MPI:

```
% f90 -hpf 2 -hpf_target cmpi -c lu.f90
```

To invoke both the compiler and linker:

```
% f90 -hpf 2 -hpf_target cmpi -o lu lu.f90
```

Table 6–1 shows the possible values of *target*.

Table 6–1 Summary of MPI Versions

target	**Explanation**
smpi	SC (Quadrics) MPI This MPI comes installed on SC-series systems. It works with the SC's RMS software that provides a set of commands for launching MPI jobs, scheduling these jobs on SC clusters, and performing other miscellaneous tasks.
cmpi	Compaq MPI This MPI is a version that is specifically tuned for Alpha systems. It is distributed as a Compaq layered product. Compaq MPI supports only Memory Channel clusters and shared-memory (SMP) machines.
gmpi	Generic MPI This target is for use with MPICH V1.2.0 or other compatible libraries. MPICH is a public domain implementation of the MPI specification that is available for many platforms. You can obtain this implementation from http://www-unix.mcs.anl.gov/mpi/mpich/. MPICH V1.2.0 supports many interconnection networks including Ethernet, FDDI, and other hardware. **Note:** Using Compaq Fortran and HPF with this MPI is officially unsupported.

If the command to the Fortran 90/95 compiler includes -hpf_target *target*, then the command must also include -hpf.

Another way of specifying the version of MPI to the compiler, instead of using the option -hpf_target, is to set the environment variable DECF90_HPF_TARGET to a value in the first column of Table 6–1. For example, the command:

```
% f90 -hpf 2 -hpf_target cmpi -c lu.f90
```

is equivalent to the commands:

```
% setenv DECF90_HPF_TARGET cmpi
% f90 -hpf 2 -c lu.f90
```

If an f90 command contains -hpf_target with a value (such as cmpi) and environment variable DECF90_HPF_TARGET is set to a different value, then the value in the f90 command overrides the value of the environment variable.

6.2.3 Linking HPF Programs with MPI

You specify MPI support for HPF programs by including the option -hpf_target with an MPI selection (argument *target*) in the link command. For example:

```
% f90 -hpf 2 -hpf_target cmpi -o lu lu.o
```

The values of *target* come from Table 6–1.

If you specified generic MPI at compilation time, either by including the -hpf_target gmpi option or by setting the environment variable DECF90_HPF_TARGET to gmpi, you must specify a path to the desired generic MPI library during linking. Do this in one of these ways:

- Set the environment variable DECF90_GMPILIB to the path of the desired generic MPI library to link with.
- In the link command line, include -l (possibly along with -L) with the path of the desired generic MPI library to link with. Or explicitly add the library to the link command line.

An example of a link command for a generic MPI library is:

```
% f90 -hpf 2 -hpf_target gmpi -o lu lu.o /usr/users/me/libmpich.a
```

In addition, you must have the Developer's Tool Kit software installed on your system to link properly with the option -hpf_target gmpi.

Finally, programs linked with -hpf_target and an MPI target must be linked with -call_shared (which is the default); the -non_shared option does not link correctly.

6.2.4 Running HPF Programs Linked with MPI

You can use these commands to execute program files created with the various -hpf_target options:

- prun

  The prun command executes program files created with the -hpf_target smpi option. Include the -n *n* option in the command line, where *n* is the same value of -hpf *n* in the compilation command line. Or, if no value was given with the -hpf option, then set *n* to the desired number of peers. Also include the name of the program file.

  In the following example, the compilation command line included -hpf 4:

  ```
  % prun -n 4 -N 4 heat8
  ```

- dmpirun

  The dmpirun command executes program files created with the -hpf_target cmpi option. Include the -n *n* option in the command line, where *n* is the same value of -hpf *n* in the compilation command line. Or, if no value was given with the -hpf option, then set *n* to the desired number of peers. Also include the name of the program file.

  In the following example, the compilation command line included -hpf 4:

  ```
  % dmpirun -n 4 heat8
  ```

The reference page dmpirun contains a full description of this command.

- mpirun

  The mpirun command executes program files created with the -hpf_target gmpi option. Include the -np *n* option in the command line, where *n* is the same value of -hpf *n* in the compilation command line. Also include the name of the program file. The mpirun command varies according to where you installed the generic MPI.

  In the following example, the compilation command line included -hpf 4:

```
% /usr/users/me/mpirun -np 4 heat8
```

6.2.5 Cleaning Up After Running HPF Programs Linked with MPI

Execution of the dmpirun command (but not the prun and mpirun commands) might leave temporary files behind. To delete them and to make sure that memory is freed, use the mpiclean command:

```
% mpiclean
```

6.2.6 Changing HPF Programs for MPI

There is only one change you should make to Fortran source files before compiling them for MPI. If a module contains an EXTRINSIC (HPF_LOCAL) statement and it executes on a system different from peer 0, then its output intended for stdout goes instead to /dev/null. Change such modules or your execution commands to have the extrinsic subroutine do input/output only from peer 0.

7

Optimizing HPF Programs

This chapter describes:

- Section 7.1, -fast Compile-Time Option
- Section 7.2, Converting Fortran 77 Programs to HPF
- Section 7.3, Explicit Interfaces
- Section 7.4, Nonparallel Execution of Code and Data Mapping Removal
- Section 7.5, Compile Speed
- Section 7.6, Nearest-Neighbor Optimization
- Section 7.7, Compiling for a Specific Number of Processors
- Section 7.8, Avoiding Unnecessary Communications Setup for Allocatable or Pointer Arrays
- Section 7.9, USE Statements HPF_LIBRARY and HPF_LOCAL_LIBRARY
- Section 7.10, Forcing Synchronization
- Section 7.11, Input/Output in HPF
- Section 7.12, Stack and Data Space Usage
- Section 7.13, -show hpf Option
- Section 7.14, Timing
- Section 7.15, Spelling of the HPF Directives

7.1 -fast Compile-Time Option

Unless there is a possibility that your program contains zero-sized arrays or array sections, the `-fast` option (or the `-assume nozsize` option) should always be specified at compile time. If neither of these options is selected, the compiler is required to insert a series of checks to guard against irregularities (such as division by zero) in the generated code that zero-sized data objects can cause. Depending upon the particular application, these checks can cause noticeable (or even major) degradation of performance.

The `-fast` or `-assume nozsize` compile-time options may not be used in a program where lines containing any zero-sized arrays or array sections are executed. If any line containing zero-sized arrays is executed in a program compiled with either of these options, incorrect program results occur.

If it is suspected that an array or array section named on a certain program line may be zero-sized, a run-time check can be performed that prevents execution of that line whenever the array or array section is zero-sized. The difference between the UBOUND and LBOUND of the array or array section is less than or equal to zero if the array or array section is zero-sized. If the executions of all occurrences of zero-sized arrays or array sections are avoided using a run-time check such as this, the program may be compiled with the `-fast` or `-assume nozsize` compiler options.

For More Information:

- See Section 6.1.1.4, -fast Option — Set Options to Improve Run-Time Performance
- See Section 6.1.1.3, -assume nozsize Option — Omit Zero-Sized Array Checking

7.2 Converting Fortran 77 Programs to HPF

Take the following steps to port applications from Fortran 77 to Compaq Fortran with HPF for parallel execution:

1. Change compilers to Compaq Fortran.
   a. Recompile the code as is using the Compaq Fortran compiler in scalar mode (that is, without using the `-hpf` option). Compaq Fortran supports a substantial number of, but not all possible, extensions to Fortran 77. This identifies the worst nonstandard offenders, if any, so you can remove them from the scalar code base.
   b. Test and validate that the scalar code produces the correct answers.

c. Recompile using the `-hpf` option (but without changing the source code), test, and validate for a distributed memory system (of any size). You should expect no performance improvement; this simply validates that there are no anomalies between scalar and parallel.

2. Find the "hot spots."

   Identify the routines that use the most time. Do this by profiling the scalar code compiled without `-hpf`. Section 1.4.1 contains an example of profiling a serial program.

3. Fortran 90-ize the hot spots.

   a. Convert global COMMON data used by these routines into MODULE data. This should be straightforward if the data consists of "global objects," but hard if the data is "storage" that is heavily equivalenced. This involves changes in non-hotspot routines that use the same data. A first step is simply to place the COMMON statements into MODULEs and replace INCLUDEs by USEs.

   b. Eliminate the use of Fortran 77 sequence association, linear memory assumptions, pointers that are addresses (such as Cray pointers), array element order assumptions (column-wise storage), and so on.

   c. Actual arguments that look like array elements and really are array sections must be replaced by array sections. In order to accomplish this, you must change calling sequences to pass array arguments by assumed shape. This involves changes on the caller and callee sides. The routines must have explicit interfaces, most easily provided by putting the routines in a module.

   d. Mark intensive computation, such as nested DO loops, with the INDEPENDENT directive when possible. In some cases, you will need to change DO loops to Fortran 90 array assignments or HPF FORALL constructs in order to achieve parallelism.

   e. Recompile, test, and validate that the scalar code produces the correct answers.

   f. Recompile using the `-hpf` and `-show hpf` options, test, and validate for a distributed memory system (of any size including 1). You should expect no performance improvement; this simply validates that there are no anomalies between scalar and parallel.

4. Use data decomposition.

   a. Analyze the usage of data in the hot spots for desired ALIGNment locality and DISTRIBUTE across multiple processors. Annotate code with HPF directives.

b. Recompile, test, and validate that the scalar code produces the correct answers.

c. Recompile using the `-hpf` and `-show hpf` options, test, and validate for a distributed memory system. Pay particular attention to messages produced by `-show hpf`. Replace any serialized constructs identified by the compiler with parallelizable constructs. Pay attention to motion (that is, interprocessor communication) introduced by the compiler (identified by `-show hpf`). Make sure it agrees with the motion you expect from the HPF directives you thought you wrote.

d. If the performance did not improve as expected, analyze using the profiler, modify the code, and return to step 4b.

5. The rest of the code:

   Repeat steps 3 and 4 for more of the code. Ideally, the structure of the whole program should be rethought with Fortran 90 modules in mind. Use of Fortran 90 constructs allows for a significant improvement in the readability and maintainability of the code.

7.3 Explicit Interfaces

In many cases, the High Performance Fortran Language Specification requires an explicit interface when passing mapped objects through the procedure interface. An explicit interface is one of the following:

- An interface block
- A module
- A contained procedure

Explicit interfaces must specify any pointer, target, or allocatable attribute as well as array distributions. It is illegal in Compaq Fortran to omit pointer or target attributes in an interface block. Such an illegal program does not necessarily generate an error message, but program results may be incorrect.

Compaq recommends the use of modules. The use of modules eliminates the need for explicit interface blocks at each subroutine call, and produces code that is easy to write and easy to maintain. Also, compilation time is generally reduced for programs written with modules.

For More Information:

- On explicit interfaces, see Section 5.6.2.
- On modules as an easy way to provide explicit interfaces, see Section 5.6.3.

7.4 Nonparallel Execution of Code and Data Mapping Removal

The use of certain constructs causes some portion (possibly all) of a program not to execute in parallel. Nonparallel execution occurs in the following situations:

- All I/O operations are serialized through a single processor. See Section 7.11 and Section 7.11.1.
- Date and time intrinsics are not handled in parallel. The reason for single-processor execution of these routines is to ensure that a reliable time-stamp is returned.

If one of these situations applies to an expression, the entire statement containing the expression is executed in a nonparallel fashion. This can cause performance degradation. Compaq recommends avoiding the use of constructs or variables to which the above conditions apply in the computationally intensive kernel of a routine or program.

7.5 Compile Speed

The compiler runs more quickly over files that are not too large. The use of modules can aid in separate compilation and avoids the need to write interface blocks.

7.6 Nearest-Neighbor Optimization

By default, the nearest-neighbor compiler optimization is always on. This optimization recognizes constructs that perform regular, short-distance communication of neighboring array elements, and generates more efficient code. This kind of code occurs often when solving partial differential equations using a number of common methods.

The compiler automatically detects nearest-neighbor computations, and allocates shadow edges that are optimally sized for your algorithm. You can also size the shadow edges manually using the SHADOW directive. This is necessary to preserve the shadow edges when nearest-neighbor arrays are passed as arguments.

If the additional storage required for this optimization cannot be tolerated, you can adjust the maximum allowable shadow-edge width using the `-nearest_neighbor` command-line option, or completely disable the nearest-neighbor optimization using the `nonearest_neighbor` option.

There are a number of conditions that must be satisfied in order to take advantage of the nearest-neighbor optimization. See the Release Notes.

The -show hpf option indicates which statements were recognized as nearest-neighbor statements.

For More Information:

- On the nearest neighbor optimization, see Sections C.5.2, 5.5.7, and 6.1.1.5.
- On sizing the shadow edges manually using the SHADOW directive, see Section 5.5.7.
- On the -show hpf option, see Section 6.1.1.7.

7.7 Compiling for a Specific Number of Processors

Compile for a specific number of nodes (that is, specify a value to the -hpf option) for production code, if possible. If you compile using -hpf but do not specify a value, only one dimension of any array is distributed (unless a PROCESSORS directive was used). If the performance of your code depends on distributing more than one dimension, it executes more slowly if you do not specify a value with the -hpf option. In addition, even code that only distributes one dimension of an array may execute more slowly if no value was specified with -hpf because addressing expressions may take longer to evaluate.

7.8 Avoiding Unnecessary Communications Setup for Allocatable or Pointer Arrays

One of the important optimizations in Compaq's HPF compiler is that set-up for data communications is eliminated when it can be proven at compile time that communications will not be necessary. Eliminating communications set-up can provide a significant performance improvement.

Although the removal of communications set-up is a Compaq-specific optimization, the proofs that communications set-up is unnecessary are general proofs based on the rules of the HPF language.

When allocatable (or pointer) arrays are used, the sizes and lower bounds of each array dimension are not known at compile time. Nevertheless, it is often possible to write ALIGN directives that give enough information to allow the compiler to prove when communication is not necessary.

The key is to know whether to write an ALIGN directive *with* an align subscript, like this:

```
!HPF$ ALIGN B(i) WITH C(i)
```

or *without* an align subscript, like this:

```
!HPF$ ALIGN B WITH C
```

These two forms have slightly different semantics. When an align subscript is used:

- The align target (C in our example) is permitted to be larger than the alignee (B in our example).
- Elements whose subscripts are equal are aligned, regardless of what the lower bound of each array happens to be.

When an align subscript is not used:

- The alignee (B) and the align target (C) must be exactly the same size.
- Corresponding elements are aligned beginning with the lower bound of each array, regardless of whether the subscripts of the corresponding elements are equal.

The rule of thumb is: When allocatable or pointer arrays are used in a FORALL assignment or INDEPENDENT DO loop, use an ALIGN directive *with* an align subscript. When allocatable or pointer arrays are used in a whole array assignment, use an ALIGN directive *without* an align subscript.

Example 7–1 illustrates this rule of thumb by comparing the two forms of the ALIGN directive.

Example 7–1 Avoiding Communication Set-up with Allocatable Arrays

```
      SUBROUTINE NO_SUBSCRIPT(n)    |       SUBROUTINE SUBSCRIPT(n)
                                    |
      INTEGER i                     |       INTEGER i
      REAL         :: A, B, C       |       REAL         :: A, B, C
      ALLOCATABLE  :: A, B, C       |       ALLOCATABLE  :: A, B, C
      DIMENSION(:) :: A, B, C       |       DIMENSION(:) :: A, B, C
                                    |
!HPF$ DISTRIBUTE C(BLOCK)           | !HPF$ DISTRIBUTE C(BLOCK)
!HPF$ ALIGN A WITH C                | !HPF$ ALIGN A(i) WITH C(i)
!HPF$ ALIGN B WITH C                | !HPF$ ALIGN B(i) WITH C(i)
                                    |
! Local                            | ! May require communication
      A = B                         |       A = B
                                    |
! May require communication         | ! Local
      FORALL (i=1:n) A(i) = B(i)    |       FORALL (i=1:n) A(i) = B(i)
!HPF$ INDEPENDENT                   | !HPF$ INDEPENDENT
      DO i = 1, n                   |       DO i = 1, n
        A(i) = B(i)                 |         A(i) = B(i)
      END DO                        |       END DO
                                    |
! Local                            | ! Local
      A = C                         |        A = C
                                    |
      END SUBROUTINE NO_SUBSCRIPT   |        END SUBROUTINE SUBSCRIPT
```

The statements commented as "Local" can be proven to require no communication, and the compiler will eliminate communications set-up. Communications set-up cannot be removed for the statements commented as "May require communication." Of course, even when communications set-up is performed, no superfluous data motion will occur if communication turns out at run time to be unnecessary.

Table 7–1 explains why communication may be (or is not) needed for each statement in Example 7–1.

Table 7–1 Explanation of Example 7–1

Statement:	Routine: NO_SUBSCRIPT	Routine: SUBSCRIPT
`A = B`	The "no subscript" form of the ALIGN directive requires both `A` and `B` to be the same size as `C`. `A` and `B` are therefore aligned with each other.	The "subscript" form of the ALIGN directive allows `C` to be larger than `A` or `B`. Since the lower bounds of the three arrays are unknown and can potentially be different from one another, it is possible that `A` is aligned with a different part of `C` than `B` is.

`FORALL (i=1:n) A(i)=B(i)`
 	
`!HPF$ INDEPENDENT`
`DO i = 1, n`
`A(i) = B(i)`
`END DO`
	Even though `A` and `B` must both be the same size, their lower bounds may be different. If `n` is smaller than the extent of the arrays, and `A` and `B` have different lower bounds, then `A(1:n)` is not aligned with `B(1:n)`.	The "subscript" form of the ALIGN directive guarantees that elements whose subscripts are equal are aligned. Even if `C` is larger than `A` and `B`, and even if `A` is aligned with a different part of `C` than `B` is, the sections `A(1:n)` and `B(1:n)` are both aligned with `C(1:n)`, and are therefore aligned with each other.

`A = C`	The "no subscript" form of the ALIGN directive requires `A` to be the same size as `C`. The whole-array assignment syntax also requires this. Corresponding elements are therefore aligned, whether or not the two arrays have the same lower bound.	The "subscript" form of the ALIGN directive states that all elements whose subscripts are equal are aligned. The whole-array assignment syntax requires `A` to be the same size as `C`. Therefore, `A` and `C` are aligned, and have the same lower bound.

For More Information:

- On the ALIGN directive, see Section 5.5.3 and the High Performance Fortran Language Specification.
- On allocatable arrays, see the *Compaq Fortran Language Reference Manual*.

7.9 USE Statements HPF_LIBRARY and HPF_LOCAL_LIBRARY

The HPF language specification states that the HPF_LIBRARY and HPF_LOCAL_LIBRARY routines are only available if USE HPF_LIBRARY or USE HPF_LOCAL_LIBRARY statements are issued. The Compaq Fortran compiler relaxes this restriction. The USE statements, if present, are accepted, but are not required.

7.10 Forcing Synchronization

The following routines automatically force synchronization of the processors when called from global HPF routines:

- Intrinsic subroutines:

  DATE_AND_TIME
  SYSTEM_CLOCK
  DATE
  IDATE
  TIME

- Intrinsic functions:

  SECNDS

These routines do not force synchronization when called from EXTRINSIC(HPF_LOCAL) and EXTRINSIC(HPF_SERIAL) routines.

Synchronization of processors can have an effect on performance. Therefore, it is preferable to avoid synchronization of the processors in the computationally intensive kernel of a program.

7.11 Input/Output in HPF

In Compaq Fortran, all I/O operations are serialized through a single processor. The cluster consists of members from which are chosen a set of peer processors on which the parallel application executes. These are formally numbered starting from 0 as Peer 0, Peer 1, and so on.

All I/O operations are serialized through Peer 0. This means that all data in an output statement which is not available on Peer 0 is copied to a buffer on Peer 0. Consider the following example:

```
      INTEGER, DIMENSION(10000000) :: A
!HPF$ DISTRIBUTE A(BLOCK)
      PRINT *, A
```

Since *A* is distributed BLOCK, not all of the values exist on Peer 0. Therefore the entire array *A* is copied from the various peers into a rather large buffer on Peer 0. The print operation is executed by Peer 0 only.

Input behaves in a similar manner: data being read is copied from the file to a buffer on Peer 0 and distributed according to the data mapping of its destination. This makes I/O slow.

7.11.1 General Guidelines for I/O

The following list contains some guidelines to minimize the performance degradation caused by I/O. If your program reads or writes large volumes of data, read this carefully. When choosing among these guidelines, remember that generally speaking, computation is faster than communication, which is faster than I/O.

The guidelines are:

- Avoid I/O operations in the computationally intensive kernel of the program. This is always wise even if only reading a single value.
- Try to avoid having to dump data part way through the program to temporary files. You can either use more memory or simulate parallel I/O.

  To use more memory you can try either:

  - Attaching more memory to the processors you are using.
  - Distributing your data over more processors, decreasing the memory requirement for each individual processor (carefully to avoid increasing communications costs).
- Make sure that Peer 0 has enough real memory to handle the I/O buffers that are generated.
- Do I/O to a disk attached to Peer 0. Use of a file server requires additional communication while use of a disk on another peer not only requires communication but the communication competes with other Farm jobs.
- Consider reading/writing into variables that are stored only on Peer 0.

For More Information:

- On simulating parallel I/O, see Section E.1
- On reading and writing to variables stored only on peer 0, see Section 7.11.4.

7.11.2 Specifying a Particular Processor as Peer 0

To specify a particular processor as Peer 0, use the `-on` command-line switch when executing the program. Ideally, you want Peer 0 to be a processor with a lot of real memory and attached to the file system to which you are reading or writing. For example, to make a processor named FRED Peer 0:

```
% a.out -peers 4 -on FRED,...
```

This specifies that the first processor (peer 0) should be FRED. The three periods (...) specify that the other peers should automatically be selected, based on load-balancing considerations. No spaces are allowed before the three periods or in between them.

7.11.3 Printing Large Arrays

To avoid running out of memory and/or swamping the network, the best way to print a huge array is to print it in sections. For example, the array *A* can be printed in 1000 element sections. This causes the compiler to generate a buffer on Peer 0 sufficient for only 1000 elements, instead of an array sufficient for 10000000 elements.

```
DO i=1,10000,1000
  PRINT *, a(i:i+1000-1)
ENDDO
```

It is possible that there may not be enough stack space available for the 1000000 element buffer needed to print all of A at once. In this case, a segmentation fault may occur, or the program may behave differently each time it is executed, depending upon how much stack space is available on the node that is peer 0 for that run. If segmentation faults occur at run time, try increasing the stack space. (See Section 7.12.) However, a much more efficient solution is to print the array in sections instead of all at once.

7.11.4 Reading and Writing to Variables Stored Only on Peer 0

Another way to speed up I/O is to read/write from data which is distributed in such a way that it is stored on peer 0 only. Since I/O happens only on peer 0, no buffers are needed.

The following code:

```
INTEGER jseq
REAL seq_array(n)

DO i = 1, n
  READ(*) jseq
  seq_array(i) = REAL(jseq) - 0.5
ENDDO
```

can be re-written as:

```
      REAL seq_array(n)
      REAL tmp(n)
!HPF$ TEMPLATE t(n)
!HPF$ DISTRIBUTE t(BLOCK)
!HPF$ ALIGN tmp  WITH t(1)

      DO i = 1, n
        READ(*) tmp(i)
      ENDDO

      seq_array = REAL(tmp) - 0.5
```

The first program reads into *jseq*. Because the I/O only occurs on peer 0, the value on peer 0 must subsequently be sent to all the other processors. In this program the read happens repeatedly, because it is located in a loop. This causes a dramatic reduction in performance.

In the second example a temporary array was created, *tmp*, that is distributed in such a way that it is stored only on peer 0. There is no need to send the values on peer 0 to all the other processors because *tmp*, which was read into directly, resides only on peer 0. Later, outside the loop, a single array assignment of the entire *tmp* array into *seq_array* is done.

The re-written code is faster, because I/O operations are done uninterrupted, rather than being repeatedly interleaved with assignment statements. The part of the speed-up also comes from the assignment statement, because the broadcasting of values from peer 0 to *seq_array* is done all at once and more easily optimized by the compiler.

The template *t* was required in order to arrange for *tmp* to reside only on peer 0. The TEMPLATE directive creates a template *t*, which is distributed BLOCK.

```
!HPF$ TEMPLATE t(n)
!HPF$ DISTRIBUTE t(BLOCK)
```

The entire array *tmp* is aligned with only the first element of *t*. Since *t* is distributed BLOCK, this first element resides on peer 0. The effect is that all of array *tmp* is serial on peer 0:

```
!HPF$ ALIGN tmp WITH t(1)
```

7.11.5 Use Array Assignment Syntax instead of Implied DO

If the relevant data isn't available on peer 0, using Fortran 90 array syntax is more efficient than using implied DO loops in the I/O statements.

For example, the READ statement in the following code fragment is inefficient in HPF, generating a separate read and copy-in/copy-out onto peer 0 for each array element:

```
        INTEGER a(n), b(n)
!HPF$ DISTRIBUTE a(BLOCK), b(CYCLIC)
        READ(UNIT=filein,fmt=*) (a(i), i=1,n), b(1), b(2), ... b(n)
```

The above code fragment would be optimized in a serial Fortran 90 program, but generates a large number of small, inefficient I/O calls when compiled with the `-hpf` option for parallel execution.

When you need efficient I/O in a parallel program, you will achieve better results by explicitly copying to and from variables available only on peer 0 (effectively doing the copyin/copyout yourself - as described in Section 7.11.4).

An easier option is to convert the implied DO loop to Fortran 90 array syntax, like this:

```
        READ(UNIT=FILEIN,FMT=*) A(1:N), B(1:N)
```

In parallel HPF programs, Fortran 90 array syntax is generally better optimized than implied DO loops.

7.11.6 IOSTAT and I/O with Error Exits–Localizing to Peer 0

The performance of I/O with error exits or IOSTAT (ERR, END, or IOSTAT) can be improved by specifying that all the relevant variables are distributed so that they are stored on Peer 0 only.

```
        SUBROUTINE writeout(a, itotal, n, nrecs)
        INTEGER a(n,5), itotal, istat
!HPF$ TEMPLATE t(1)
!HPF$ DISTRIBUTE t(BLOCK)
!HPF$ ALIGN WITH t(1) :: istat, itotal
!
!HPF$ TEMPLATE t2(n,5)
!HPF$ DISTRIBUTE t2(BLOCK, BLOCK)
!HPF$ ALIGN a WITH t2(1,1)

        DO i=1,NRECS
          WRITE(*,ERR=800,IOSTAT=istat) a(i,3), itotal
        ENDDO
        GOTO 900
800     PRINT *, 'Error in WRITE, IOSTAT=', istat
900     END
```

Notice in this example that in addition to the variables being written, the variable `istat` also needs to be local to peer 0.

For More Information:

- On making variables local to Peer 0, see Section 7.11.4.

7.12 Stack and Data Space Usage

Exceeding the available stack or data space on a processor can cause the program execution to fail. The failure takes the form of a segmentation violation, which results in an error status of -117. This problem can often be corrected by increasing the stack and data space sizes. Use the following `csh` commands to increase the sizes of the stack and data space (other shells require different commands):

```
% limit stacksize unlimited
% limit datasize  unlimited
```

This increases the size available for the buffer to the maximum allowed by the current kernel configuration. See your system administrator to raise these limits further.

7.13 -show hpf Option

Take advantage of the information given by the `-show hpf` option. This option prints information about interprocessor communication generated by the compiler, statements that are recognized as nearest-neighbor, statements that are not handled in parallel, and other information. This is useful in determining what the compiler is doing with your program.

For example, consider this test program:

```
PROGRAM test
INTEGER a(100), b(100), c(100)
!HPF$ DISTRIBUTE (BLOCK) :: a,c
!HPF$ DISTRIBUTE b(CYCLIC)
c= a+b
END PROGRAM test
```

When this program is compiled with the `-show hpf_comm` option, the following output is generated:

```
% f90 -hpf -show hpf_comm test.f90

f90: Info:
f90: Info: test.f90, line 5:
f90: Info:     Communication needed because
f90: Info:     the target is distributed differently than the source.
f90: Info:         Target: @1(1:100)
f90: Info:         Source: B
f90: Info:     Temp @1(1:100) has distribution (block)
```

This tells you that since A and C are distributed BLOCK, the compiler re-maps B to BLOCK in order to do the statement in parallel. The "at" sign (@) indicates a compiler-generated temporary array. If you see more communication generated than you expect, you need to check your program to verify that the HPF directives you have issued are what you intended.

For More Information:

- On the various forms of the `-show hpf` option, see Section 6.1.1.7.

7.14 Timing

Processors are never synchronized explicitly by the compiler, except when certain intrinsics are invoked (see Section 7.10). However, this means that if you issue calls to timing routines not included in the list of synchronized intrinsics (see Section 7.10) you may not get the results you expect. Consider the following (illegal) code:

```
REAL elapsed_time

CALL start_timer()

<statements being timed>

elapsed_time = stop_timer()
PRINT *, elapsed_time
```

In this code fragment, `start_timer` and `stop_timer` are fictitious names for some user-written or operating-system-supplied routines other than the timing intrinsics mentioned.

The variable `elapsed_time` is not explicitly distributed, so it is replicated on all processors. Replication is the default distribution in parallel Compaq Fortran programs. The `stop_timer` routine, which returns its result in `elapsed_time`, is called on all processors. However, since `elapsed_time` is replicated, the print statement prints the peer 0 value. Due to the unsynchronized nature of the code, peer 0 may reach the `stop_timer` call either before or after other processors have finished executing the code being timed, so the value it prints does not reflect the true elapsed time. This program is *not* a legal HPF program, because the `stop_timer` routine could return different values

on different processors; it is illegal to modify a replicated value differently on different processors.

There are two problems here. First, the values stored in elapsed_time differ on different processors. Second, Peer 0 may reach the timing at a different time than the other processors.

To solve the first problem, make the timer routines EXTRINSIC(HPF_SERIAL) routines; they only execute on peer 0. To solve the second problem, force a synchronization before calling the timer routines. For example, this causes the call to stop_timer to be delayed until all the processors finish executing the code being timed. The HPF library routine HPF_SYNCH is used for this purpose.

The following code fragment returns the desired results:

```
REAL elapsed_time
INTERFACE
  EXTRINSIC(HPF_SERIAL) FUNCTION start_timer()
  END FUNCTION
  REAL EXTRINSIC(HPF_SERIAL) FUNCTION stop_timer()
  END FUNCTION
END INTERFACE

CALL hpf_sync()
CALL start_timer()

 <statements being timed>

CALL hpf_sync()
elapsed_time=stop_timer()
PRINT *, elapsed_time
```

Alternatively, you can use the SECNDS intrinsic, which is automatically serialized.

7.15 Spelling of the HPF Directives

The identifying tag !HPF$ must be spelled correctly. Misspelling the tag is a common programming mistake which is difficult to detect. For example, if you leave the dollar sign ($) off of the end of the tag, the entire line is treated as a comment. For example:

```
!HPF DISTRIBUTE a(BLOCK)
```

This misspelled line is interpreted as a comment, because it begins with an exclamation mark (!). This does not always affect program results, but it can cause performance degradation.

A

HPF Tutorials: Introduction

This appendix begins a set of tutorials about High Performance Fortran (HPF) for Fortran programmers. No previous knowledge of HPF is assumed. The tutorials contain general information about HPF, as well as some information about special characteristics of Compaq's implementation of HPF.

The tutorials are found in the following appendixes:

- Appendix B, HPF Tutorial: LU Decomposition

  Introduces the FORALL construct, the INDEPENDENT directive, and HPF data distribution.

- Appendix C, HPF Tutorial: Solving Nearest-Neighbor Problems

  Discusses BLOCK distribution, and Compaq Fortran's optimization of nearest neighbor problems.

- Appendix D, HPF Tutorial: Visualizing the Mandelbrot Set

  Presents the use of the PURE attribute, the use of non-Fortran subprograms within an HPF program, and the use of non-parallel HPF subprograms.

- Appendix E, HPF Tutorial: Simulating Network Striped Files

  Presents techniques for parallel input/output, as well as local subroutines (parameterized by processor), and passing distributed arrays through the procedure interface.

B

HPF Tutorial: LU Decomposition

This appendix describes the parallelization of an algorithm for the well-known Gaussian elimination method of factoring a matrix. This matrix operation, also known as **LU decomposition**, demonstrates both the ease of use and the power of Compaq Fortran used with its High Performance Fortran (HPF) extensions. This small but typical problem introduces basic HPF constructs, including DISTRIBUTE, FORALL, and INDEPENDENT.

In LU decomposition, a square matrix is factored into two matrices *L* and *U*, where *L* is lower triangular with ones on its diagonal, and *U* is upper triangular.

B.1 Using LU Decomposition to Solve a System of Simultaneous Equations

Factoring a matrix in this manner is useful for solving large systems of *n* simultaneous equations in *n* unknowns. This section gives an abridged explanation of the application of LU decomposition to solving equations.

Although HPF achieves performance gains only for large matrices, (for the meaning of "large," see Section 5.1.1), the following artificially small example of a system of 3 equations in 3 variables can be used for the purpose of illustration:

$$x_1 + 2x_2 + 3x_3 = 14$$

$$2x_1 - x_2 + x_3 = 3$$

$$3x_1 + 4x_2 - x_3 = 8$$

Systems of *n* equations in *n* unknowns can be represented in matrix notation as a single equation. This equation consists of an *n* by *n* array *A* of coefficients, an *n*-element vector *x* of variables, and an *n*-element vector *b* of constants. Our example is expressed in this notation:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 1 \\ 3 & 4 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 14 \\ 3 \\ 8 \end{bmatrix}$$

$$A \cdot x = b$$

Using a Gaussian elimination technique, A can be factored (decomposed) into the following two matrices L and U:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 0.4 & 1 \end{bmatrix} \quad and \quad U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -5 & -5 \\ 0 & 0 & -8 \end{bmatrix}$$

After the matrix A of coefficients is factored into the lower and upper triangular matrices L and U, values for the vector x of variables can be determined easily:

Since $A \cdot x = b$ and $A = L \cdot U$, therefore

$$x = (L \cdot U)^{-1} \cdot b$$

or

$$x = U^{-1} \cdot L^{-1} \cdot b$$

In effect, the application of U^{-1} and L^{-1} is performed by the processes of forward elimination (for L^{-1}) and back substitution (for U^{-1}). Consequently, the computation is easily done in two steps, by:

- Calculating an intermediate vector equal to $L^{-1} \cdot b$ using forward elimination
- Applying U^{-1} to this vector by back substitution to yield the solution vector x.

Once A has been factored into L and U, this two-step procedure can be used repeatedly to solve the system of equations for different values of b.

B.2 Coding the Algorithm

A standard algorithm for LU decomposition, described in *Numerical Recipes*,[1] transforms a square matrix "in place," storing the values for all the elements of L and U in the same space in memory where the original square matrix was stored. This can be done by overlapping the two arrays so that the mandatory zeros on the opposite sides of both L and U, and the ones on the diagonal of L, are not explicitly represented in memory. The algorithm transforms the array A in the previous example into the following array:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -5 & -5 \\ 3 & 0.4 & -8 \end{bmatrix}$$

[1] William H. Press [et al.], *Numerical Recipes in FORTRAN : The Art of Scientific Computing*. 2nd ed. Cambridge University Press, 1992.

The lower triangle of this array contains all the significant elements of *L*, and its upper triangle contains all the significant elements of *U*.

The algorithm for accomplishing this transformation is constructed of three controlling structures:

- A sequential DO loop moves down the diagonal from $A(1,1)$ to $A(n-1,n-1)$ in order. For each diagonal element $A(k,k)$, the following operations are performed (on successively smaller portions of the matrix):
  - Column normalization — The elements in the column below the diagonal element $A(k,k)$ are divided by the diagonal element.
  - Submatrix modification — A submatrix is defined containing all the elements of *A* that are below and to the right of $A(k,k)$, not including the column and row that contain $A(k,k)$. The value of each element $A(i,j)$ in the submatrix is modified by subtracting $A(i,k) * A(k,j)$.

For the sake of simplicity, the issue of pivoting is ignored here although the algorithm can be unstable without it.

B.2.1 Fortran 77 Style Code

In Fortran 77 syntax, the algorithm (without pivoting), is coded as follows:

```
DO k = 1, n-1
  DO x = k+1, n                     ! Column
    A(x, k) = A(x, k) / A(k, k)     ! Normalization
  END DO                            !
  DO i = k+1, n                             !
    DO j = k+1, n                           ! Submatrix
      A(i, j) = A(i, j) - A(i, k)*A(k, j) ! Modification
    END DO                                  !
  END DO                                    !
END DO
```

Like all Fortran 77 code, this code is compiled and executed correctly by Compaq Fortran. However, the compiler does not recognize it as parallelizable, and compiles it to run serially with no parallel speed-up.

B.2.2 Parallelizing the DO Loops

In order to achieve parallel speed-up, eligible DO loops should be changed to one of these:

- DO loops marked with the INDEPENDENT directive
- Fortran 90 array assignment statements, or
- their extended form, Fortran 95 FORALL structures.

Some caution is required, because these three parallel constructs are not the same as a non-parallel DO loop. In many cases, simply re-writing a DO loop into one of these three forms can result in different answers or even be illegal. In other cases, the three forms are equivalent (for a comparison among the three, see Section B.2.3).

In our example, the column normalization DO loop can be expressed any of these three ways:

- INDEPENDENT DO loop:

```
!HPF$ INDEPENDENT
      DO x = k+1, n
        A(x, k) = A(x, k) / A(k, k)
      END DO
```

- Fortran 90 array assignment statement:

```
A(k+1:n, k) = A(k+1:n, k) / A(k, k)
```

- FORALL statement:

```
FORALL (i=k+1:n) A(i, k) = A(i, k) / A(k, k)
```

The submatrix modification DO loop is too complex to be expressed by a single array assignment statement. However, it can be marked with the INDEPENDENT directive or changed into a FORALL:

- INDEPENDENT version:

```
!HPF$   INDEPENDENT, NEW(j)
        DO i = k+1, n
!HPF$     INDEPENDENT
          DO j = k+1, n
            A(i, j) = A(i, j) - A(i, k)*A(k, j)
          END DO
        END DO
```

  The NEW(j) keyword tells the compiler that in each iteration, the inner DO loop variable `j` is unrelated to the `j` from the previous iteration. The Compaq Fortran compiler currently requires the NEW keyword in order to parallelize nested INDEPENDENT DO loops.

- FORALL version:

```
      FORALL (i=k+1:n, j=k+1:n)
        A(i, j) = A(i, j) - A(i, k)*A(k, j)
      END FORALL
```

Putting column normalization and submatrix modification together, here are two versions of the complete parallelized algorithm:

- Fortran 90/95 syntax version:

```
      DO k = 1, n-1
        A(k+1:n, k) = A(k+1:n, k) / A(k, k)   ! Column Normalization
        FORALL (i=k+1:n, j=k+1:n)                 ! Sub-
          A(i, j) = A(i, j) - A(i, k)*A(k, j)     !  matrix
        END FORALL                                ! Modification
      END DO
```

- DO INDEPENDENT version:

```
      DO k = 1, n-1
!HPF$   INDEPENDENT
        DO x = k+1, n                    ! Column
          A(x, k) = A(x, k) / A(k, k)    ! Normalization
        END DO                           !
!HPF$   INDEPENDENT, NEW(j)
        DO i = k+1, n                            !
!HPF$     INDEPENDENT                            !
          DO j = k+1, n                          ! Submatrix
            A(i, j) = A(i, j) - A(i, k)*A(k, j) ! Modification
          END DO                                 !
        END DO                                   !
      END DO
```

For More Information:

- On the INDEPENDENT directive, see Section 5.4.4.
- On FORALL, see Section 5.4.3
- On Fortran 90 array assignment syntax, see Section 5.4.2.

B.2.3 Comparison of Array Syntax, FORALL, and INDEPENDENT DO

Although Fortran 90/95 array syntax or FORALLs can serve the same purpose as DO loops did in Fortran 77, FORALLs and array assignments are parallel assignment statements, not loops, and in many cases produce a result different from analogous DO loops.

It is crucial to understand the semantic difference between DO and parallel assignment statements such as FORALL or Fortran 90 array assignment. Statements inside DO loops are executed immediately with each iteration. If a DO loop contains an assignment, an assignment will occur with each iteration.

In contrast, a FORALL specifies that the right-hand side of an assignment is computed for every iteration before any stores are done.

For example, consider the following array *C*:

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

Applying the FORALL statement

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

to this array produces the following result:

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

However, the following apparently similar DO loop

```
DO i = 2, 5
  C(i, i) = C(i-1, i-1)
END DO
```

produces a completely different result:

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

Because a DO loop assigns new values to array elements with each iteration of the loop, you must take into account that later iterations of the loop are operating on an array that has already been partially modified. In the above example, by the time the DO loop is ready to assign a value to element $C(3, 3)$, element $C(2, 2)$ has already been changed from its original value. In the FORALL structure, on the other hand, no assignments are made until the right side of the assignment statement has been computed for every case.

Some operations require the use of DO loops rather than FORALL structures. For example, in the previous LU decomposition code, the outer DO loop that moves down the diagonal is a sequential operation in which a FORALL structure cannot be used. Later iterations of the loop rely upon the fact that the array has already been partially modified.

Some DO loops are eligible to be tagged with the INDEPENDENT directive, which allows for parallel execution. Loosely speaking, a loop can be tagged INDEPENDENT if the iterations can be performed in any order (forwards, backwards, or even random) and still produce the same result. More precisely: A loop can be tagged INDEPENDENT if no array element (or other atomic data object) is assigned a value by one iteration and read or written by any other iteration. (The REDUCTION and NEW keywords relax this definition somewhat. There are restrictions involving I/O, pointer assignment/nullification, and ALLOCATE/DEALLOCATE statements. For details, see the High Performance Fortran Language Specification.)

Here is an example:

```
!HPF$ INDEPENDENT
      DO I=1, 100
        A(I) = B(I)
      END DO
```

Each of the three parallel structures (Fortran 90 array syntax, FORALL, and INDEPENDENT DO loops) has advantages and disadvantages:

- Fortran 90 array syntax is concise and can be more readable than the other forms for simple assignments. Also, unlike FORALL, function calls contained within this syntax do not need to be PURE. However, the complex subscript expressions needed in some cases can make this form less readable than FORALL. Also, this syntax is limited to assignments, and to cases that can be expressed as a whole array or an array section.
- FORALL can express certain cases that cannot be expressed as an array section (such as the diagonal of an array). Also, FORALL can be used to express some assignments that would not be eligible for the INDEPENDENT directive if expressed as DO loops. For example, consider the following DO loop:

  ```
  DO i=1, 100
    A(i) = A(i+1) + A(i)
  END DO
  ```

  Expressed as a DO loop, this computation is not INDEPENDENT and cannot be parallelized, because the result will vary if the iterations are not performed in sequential order.

  However, the same computation can be performed in parallel with this FORALL assignment:

  ```
  FORALL (i=1:100)
    A(i) = A(i+1) + A(i)
  END DO
  ```

FORALL guarantees that the computation will be done as if the right-hand side were computed for all 100 iterations before any stores are done, which in this particular case yields the same answers as if a DO loop were used.

The limitations of FORALL are that it can contain only assignment statements and can contain function calls only if the function is PURE.

- The main advantage to an INDEPENDENT DO loop is that it can contain executable statements other than assignments. Also, function calls from inside an INDEPENDENT DO are not required to be PURE. (There are a number of restrictions on function calls inside INDEPENDENT DO loops. See the Release Notes.)

  The disadvantage of INDEPENDENT DO is that some cases (such as the example in the previous bullet) can be expressed as a FORALL, but not as an INDEPENDENT DO. Also, in some cases using FORALL results in better optimization than INDEPENDENT DO.

For More Information:

- On the INDEPENDENT directive, see Section 5.4.4.
- On FORALL, see Section 5.4.3
- On Fortran 90 array assignment syntax, see Section 5.4.2.
- On PURE, see Section 5.6.4.

B.3 Directives Needed for Parallel Execution

In order to achieve parallel execution, data must be distributed by means of the DISTRIBUTE directive. Programs without any DISTRIBUTE directives are always compiled to run serially.

For parallel execution of array operations, each array must be split up in memory, with each processor storing some portion of the array in its own local memory. Splitting up the array into parts is known as **distributing** the data. The DISTRIBUTE directive controls the distribution of arrays across each processor's local memory.

Because communication of data is very time consuming, a distribution of data that minimizes communication between processors is absolutely critical for application performance.

B.3.1 DISTRIBUTE Directive

The DISTRIBUTE directive specifies a mapping pattern of data objects onto processors. It is used with the two keywords BLOCK and CYCLIC, which specify the distribution pattern.

Note

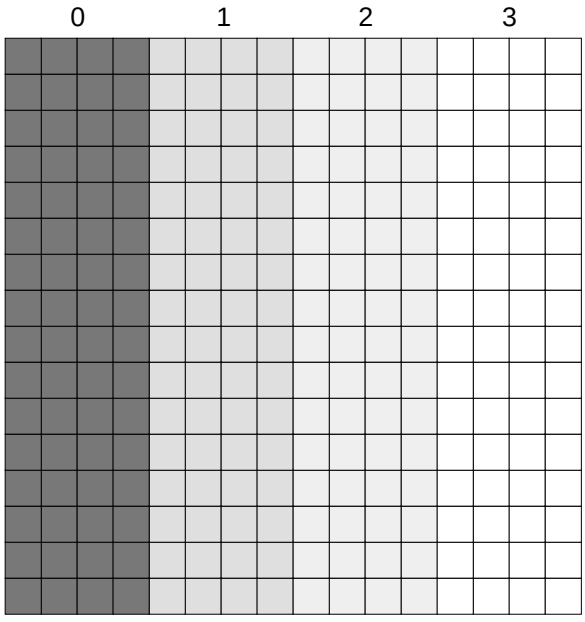
In Fortran expressions referring to elements of a two-dimensional array, the first subscript varies with vertical movement through the array, and the second subscript varies with horizontal movement. This notation is patterned after matrix notation in mathematics, where the elements in the first row of a matrix M are referred to as M_{11}, M_{12}, M_{13} . . . , the second row as M_{21}, M_{22}, M_{23}, and so on. The array can be thought of as a grid with vertical and horizontal axes; the origin is in the upper-left-hand corner; the first axis is vertical, and the second axis is horizontal. Fortran array element subscripts should not be confused with Cartesian ordered pairs (x, y), in which x varies with horizontal movement, and y varies with vertical movement.

The use of the DISTRIBUTE directive is best explained by examining some example distributions. Consider the case of a 16×16 array A in an environment with 4 processors. Here is one possible specification for A:

```
REAL A(16,16)
!HPF$ DISTRIBUTE A(*, BLOCK)
```

Figure B–1 shows this distribution.

Figure B–1 Distributing an Array (*, BLOCK)

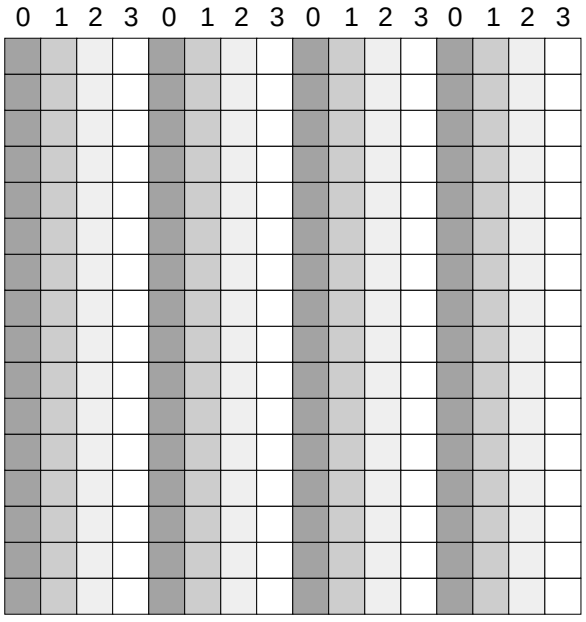


The asterisk (∗) for the first dimension of *A* means that the array elements are not distributed along the first (vertical) axis. In other words, the elements in any given column are not divided up among different processors, but assigned as a single block to one processor. This type of mapping is sometimes called "on processor" distribution. It can also be referred to as "collapsed" or "serial" 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, with each processor assigned to only one block. As a result, A is broken into four contiguous groups of columns, with each group assigned to one processor.

Another possibility is (∗, CYCLIC) distribution. As in (∗, BLOCK), the elements in any given column are assigned as a single block to one processor. However, the elements in any given row are dealt out to the processors in round-robin order, like playing cards dealt out to players around the table. When elements are distributed over *n* processors, each processor, starting from a different offset, contains every n^{th} column. Figure B–2 shows the same array and processor arrangement, distributed CYCLIC instead of BLOCK.

Figure B–2 Distributing an Array (*, CYCLIC)



The pattern of distribution is figured independently for each dimension: the elements in any given column of the array are distributed according to the keyword for the first dimension, and the elements in any given row are distributed according to the keyword for the second dimension. For example, in an array distributed (BLOCK, CYCLIC), the elements in any given column are laid out in blocks, and the elements in any given row are laid out cyclically, as in Figure B–3.

Figure B–3 Distributing an Array (BLOCK, CYCLIC)

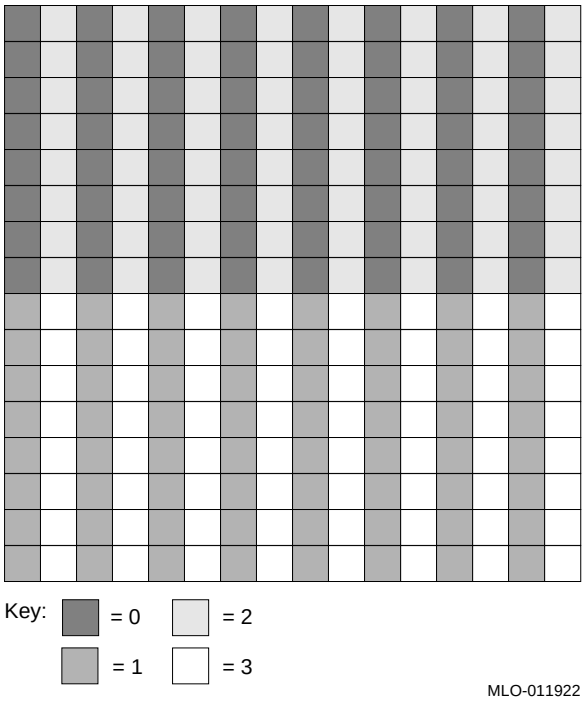
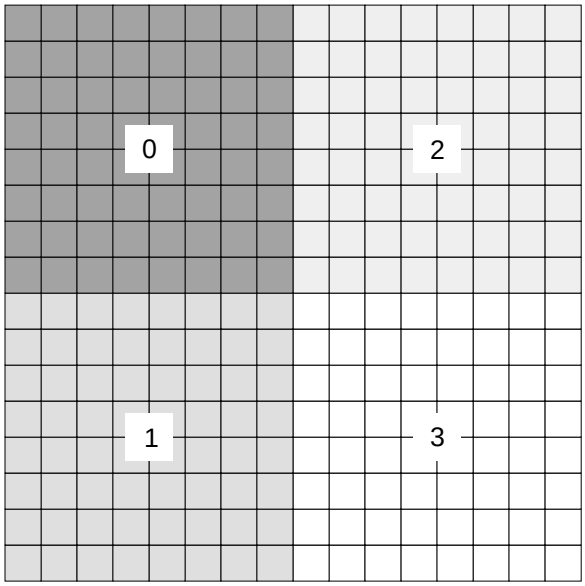


Figure B–4 shows an example array distributed (BLOCK, BLOCK).

Figure B–4 Distributing an Array (BLOCK, BLOCK)



MLO-011939

BLOCK, BLOCK distribution divides the array into large rectangles. The array elements in any given column or any given row are divided into two large blocks. In the above example, 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).

Note

Physical processors (referred to as peers) are numbered starting with 0: peer 0, peer 1, peer 2, and so on. This numbering system is different from that used for abstract processor arrangements, for which the numbering begins at 1 by default. See Section 5.5.5 for more information.

For More Information:

- On each dimension being distributed independently, see Section 5.5.6.4
- On (BLOCK, CYCLIC) distribution, see Section 5.5.6.7.
- On 1-based vs. 0-based numbering of processors, see Section 5.5.5, PROCESSORS Directive.
- For further illustration and explanation of distribution possibilities, see Section 5.5.

B.3.2 Deciding on a Distribution

There is no simple rule for computing data distribution because optimal distribution is highly algorithm-dependent. When the best-performing distribution is not obvious, it is possible to find a suitable distribution through trial and error, because the DISTRIBUTE directive affects only the performance of a program (not the meaning or result). In many cases, however, you can find an appropriate distribution simply by answering the following questions:

- Does the algorithm have a row-wise or column-wise orientation?
- Does the calculation of an array element make use of distant elements in the array, or does it need information primarily from its near neighbors in the array?

If the algorithm is oriented toward a certain dimension, the DISTRIBUTE directive can be used to map the data appropriately. For example, (*, BLOCK) is vertically oriented, whereas (BLOCK, *) is horizontally oriented (for detailed distribution illustrations, see Section 5.5.6).

Nearest-neighbor calculations generally run faster with a BLOCK distribution, because this lets the processor calculating any given array element have all of the necessary data in its own memory in most cases. The Compaq Fortran compiler includes an optimization which minimizes communication in nearest-neighbor calculations even along the edges of blocks (see Section C.5.2). For an example of a nearest neighbor calculation, see Appendix C, HPF Tutorial: Solving Nearest-Neighbor Problems.

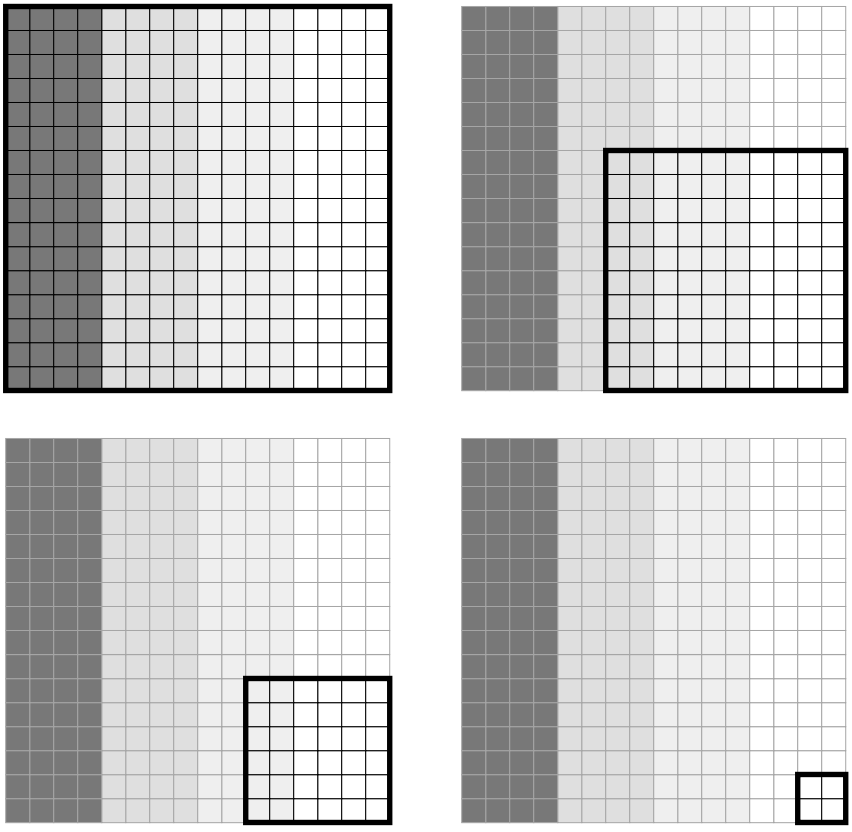
When the calculation of an array element requires information from distant elements in the array, a CYCLIC distribution is frequently faster because of load-balancing considerations. This turns out to be the case for LU decomposition.

B.3.3 Distribution for LU Decomposition

In the LU decomposition example, the submatrix modification uses information from other columns and rows, so it has neither a row-wise nor column-wise orientation. The column normalization statement, however, has an entirely column-wise orientation, needing information from a single column of the matrix only. Therefore, a column-wise orientation is preferred, either (∗, BLOCK) or (∗, CYCLIC).

Both of these structures make use of distant elements in the array, which means that little advantage would be gained from a block distribution. On the other hand, there is much to be gained from using a cyclic distribution in the case of our algorithm. To see why this is the case, see Figure B–5, which depicts the computation with (∗, BLOCK) distribution. (The illustration shows a 16 by 16 array worked on by four processors.)

Figure B–5 LU Decomposition with (*, BLOCK) Distribution

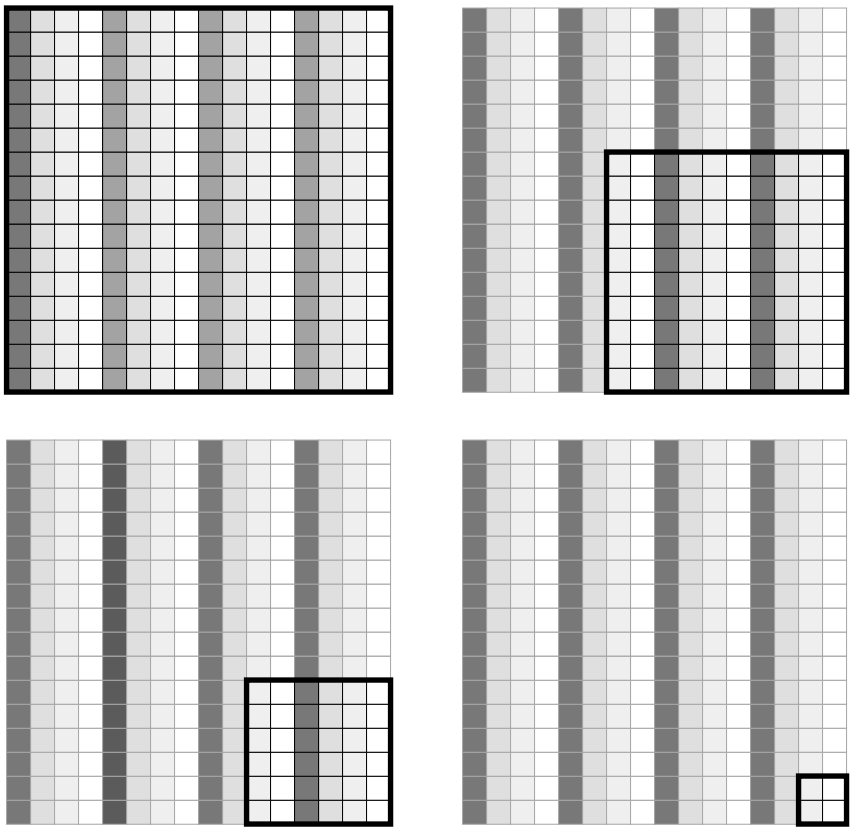


MLO-011936

Computation is done on progressively smaller submatrices in the lower right-hand corner of the array. The first panel of the figure shows the first iteration of the DO loop in which the entire array is worked on by all four processors. The second panel shows the seventh iteration, by which time Peer 0 is completely idle because none of the elements of the submatrix are stored in its memory. The third panel of the figure shows the eleventh iteration of the DO loop, by which time both Peer 0 and Peer 1 are idle. The fourth panel shows the fifteenth iteration, where only Peer 3 is working, with the other three processors idle. For most of time spent in the DO loop, one or more processors are left idle.

In contrast, (∗, CYCLIC) distribution has all four processors active until only 3 out of 16 columns remain to be completed (see Figure B–6). This load balancing consideration makes (∗, CYCLIC) distribution the better choice for this algorithm.

Figure B–6 LU Decomposition with (*, CYCLIC) Distribution



MLO-011935

B.3.3.1 Parallel Speed-Up

If you are familiar with the low-level details of parallel programming, you might wonder how any speed-up is achieved with the LU decomposition algorithm, because the sub-matrix modification appears to require a separate communication for each element in the submatrix. If the submatrix were 1000 by 1000, this would mean one million communications for each iteration of the outer DO loop. This would clearly cost considerably more time than

any speed-up achieved through parallelization, because message start-up time overwhelmingly overshadows the sending of the actual data for small messages. However, the Compaq Fortran compiler minimizes this communication cost through **communications vectorization**. Instead of sending one separate message for each array element, messages with the same destination are packaged in very large bundles or "vectors," greatly reducing message start-up overhead.

Even though LU decomposition is not a completely (or "embarrassingly") parallel computation, parallel speed-up for this algorithm with the Compaq Fortran compiler is excellent. With sufficiently large arrays, parallel speed-up comes close to **scaling linearly**: with performance increasing in near direct proportion to the number of processors used.

B.4 Packaging the Code

Source code of an executable program for the LU decomposition of a square matrix can be found in the file `/usr/examples/hpf/lu.f90`. This source code includes facilities for timing the LU decomposition kernel.

C

HPF Tutorial: Solving Nearest-Neighbor Problems

This appendix presents an example of a nearest-neighbor problem, an important class of problems for which High Performance Fortran is useful.

As an example of such a problem, this appendix concerns the problem of heat flow through a homogeneous two-dimensional object, when the edges are held at a constant temperature. The code presented approximates the steady state heat distribution after the heat flow has stabilized, given the initial temperature distribution and the boundary conditions. An iterative approach is used that gives estimates (increasingly accurate with each iteration) of the final temperature of a sampling of points distributed evenly across the object.

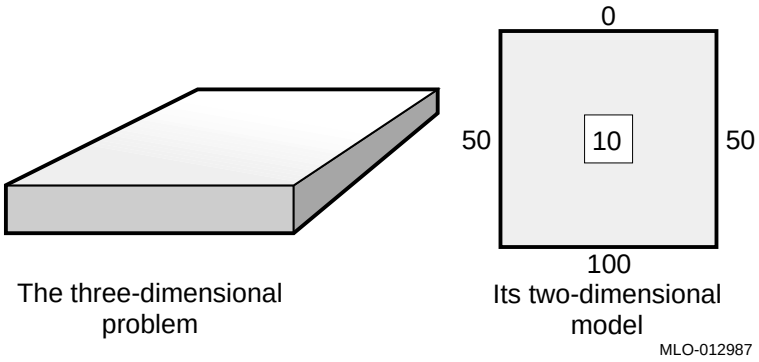
C.1 Two-Dimensional Heat Flow Problem

This example uses a slab of some homogeneous material 10 cm by 10 cm by 1 cm, completely insulated on the top and bottom surfaces, but with the edges exposed. Initially, the slab is at one uniform temperature throughout: 10 degrees Celsius. Then heat sources are applied to the four uninsulated edges that hold them at constant temperatures, specifically 100 degrees and 0 degrees on the two ends and 50 degrees on both of the two sides.

The goal of this appendix is to write a program that will answer this question: What is the temperature distribution in the slab when it becomes stable?

This can be treated as a two-dimensional problem, as shown in Figure C–1.

Figure C–1 Three-Dimensional Problem and Its Two-Dimensional Model



C.2 Jacobi's Method

Jacobi's method, one of the oldest approaches to solving this problem, is a finite-difference method that superimposes a grid over the problem space and calculates temperature values at the grid points. The finer the grid, the more accurate the approximation, and the larger the problem.

In the grid approximation that is a discrete version of the physical problem, the heat flow into any given point at a given moment is the sum of the four temperature differences between that point and each of the four points surrounding it. Translating this into an iterative method, the correct solution can be found if the temperature of a given grid point at a given iteration is taken to be the average of the temperatures of the four surrounding grid points at the previous iteration.

From the point of view of numerical analysis, Jacobi's method is a poor approach to this problem because its rate of convergence is quite slow compared with other methods. It is useful for the purposes of this tutorial, however, because the algorithm is simple, allowing us to focus attention upon the general issue of coding nearest-neighbor algorithms in HPF, rather than upon the particular details of a complex algorithm.

For the purpose of this example, think of each point on the grid as an element in a two-dimensional array. The elements around the edge of the array (the first and last row and column) remain fixed at the boundary conditions (the temperatures of the exposed edges), and the interior (non-edge) elements of the array are updated with each iteration.

If $slab^k(i, j)$ represents the temperature of interior grid-point i, j at iteration k, then $slab^{k+1}(i, j)$ (the temperature of grid-point i, j at iteration $k + 1$) is the average of the temperatures of the four surrounding grid points at iteration k. The average of the four surrounding points is obtained with the following equation:

$$slab^{k+1}(i, j) = (slab^k(i, j-1) + slab^k(i, j+1) + slab^k(i-1, j) + slab^k(i+1, j))/4$$

C.3 Coding the Algorithm

In order to represent Jacobi's method in Fortran 77 syntax (that is, with DO loops), the program must explicitly define a temporary array to hold the results of the intermediate computations of each iteration. (Note that algorithms that modify the array "in place," without the use of temporaries, actually accelerate the convergence. We have chosen Jacobi's method only because of the simplicity of the algorithm.)

At the end of each iteration, this temporary array must be copied back onto the main array, as in the following code (where `n` is the number of rows and columns in the grid):

```
DO k = 1, number_of_iterations
 DO i = 2, n-1                ! Update non-edge
  DO j = 2, n-1               ! elements only
   temp(i, j) = (slab(i, j-1)+slab(i-1, j)+slab(i+1, j)+slab(i, j+1))/4
  END DO
 END DO
 DO i = 2, n-1
  DO j = 2, n-1
   slab(i, j) = temp(i, j)
  END DO
 END DO
END DO
```

The outer loop is not eligible for the INDEPENDENT directive, because the same array elements that are assigned a value in one iteration are read and written in other iterations.

However, all the inner loops are INDEPENDENT, because in each of the inner loops, any array element that is assigned a value in one iteration is never read or written in another iteration of that loop.

```
DO k = 1, number_of_iterations
 !HPF$ INDEPENDENT, NEW(j)
 DO i = 2, n-1                ! Update
  !HPF$ INDEPENDENT           ! non-edge
  DO j = 2, n-1               ! elements only
   temp(i, j) = (slab(i, j-1)+slab(i-1, j)+slab(i+1, j)+slab(i, j+1))/4
  END DO
 END DO
 !HPF$ INDEPENDENT, NEW(j)
 DO i = 2, n-1
  !HPF$ INDEPENDENT
  DO j = 2, n-1
   slab(i, j) = temp(i, j)
  END DO
 END DO
END DO
```

The NEW keyword asserts that each iteration of the marked loop should have a private instance of the named variable. Therefore, a variable name that is listed as NEW can be assigned and used in more than one iteration, because the variable of that name for any given iteration is distinct and unrelated to the variable of the same name in any other iteration.

The NEW keyword is generally required whenever any DO loop (whether INDEPENDENT or not) is nested inside an INDEPENDENT loop. This is because the DO statement in the inner loop is considered an assignment to its DO variable. If the DO variable of the inner loop were not listed as NEW, it would be assigned in more than one iteration of the outer loop. This would disqualify the outer loop from being marked INDEPENDENT.

The algorithm can be expressed more concisely by using FORALL, instead of DO loops:

```
DO k = 1, number_of_iterations
  FORALL (i=2:n-1, j=2:n-1)      ! Non-edge elements only
    slab(i, j) = (slab(i, j-1)+slab(i-1, j)+slab(i+1, j)+slab(i, j+1))/4
  END FORALL
END DO
```

There is no need to explicitly define a temporary array to hold intermediate results, because a FORALL structure computes all values on the right side of the assignment statement before making any changes to the left side.

For More Information:

- On the nearest-neighbor optimization, see Section C.5.2.
- For a full comparison between FORALL structures and DO loops, see Section B.2.3.

C.4 Illustration of the Results

Although parallel execution generally produces performance gains only for large arrays, this example uses a small grid size of 8 by 8 for ease of illustration. (For the meaning of "large," see Section 5.1.1.) Adding two extra rows and columns to hold the boundary conditions, we need a 10 by 10 array. Since the choice of initial values for the interior grid points has no effect on the final steady-state values, they are all arbitrarily initialized at 10 degrees. The edge elements are initialized at 0, 50, and 100 degrees, which are the boundary conditions of the example. This yields the following initial array:

50	0	0	0	0	0	0	0	0	50
50	10	10	10	10	10	10	10	10	50
50	10	10	10	10	10	10	10	10	50
50	10	10	10	10	10	10	10	10	50
50	10	10	10	10	10	10	10	10	50
50	10	10	10	10	10	10	10	10	50
50	10	10	10	10	10	10	10	10	50
50	10	10	10	10	10	10	10	10	50
50	10	10	10	10	10	10	10	10	50
50	100	100	100	100	100	100	100	100	50

After one iteration, the following is produced:

50	0	0	0	0	0	0	0	0	50
50	17.5	7.5	7.5	7.5	7.5	7.5	7.5	17.5	50
50	20	10	10	10	10	10	10	20	50
50	20	10	10	10	10	10	10	20	50
50	20	10	10	10	10	10	10	20	50
50	20	10	10	10	10	10	10	20	50
50	20	10	10	10	10	10	10	20	50
50	20	10	10	10	10	10	10	20	50
50	42.5	32.5	32.5	32.5	32.5	32.5	32.5	42.5	50
50	100	100	100	100	100	100	100	100	50

After 203 iterations, the steady-state solution is achieved to two decimal places. Notice that values reflected about the vertical axis of symmetry are the

same and that values reflected about the horizontal axis of symmetry have the property that they sum to 100.

$$\begin{bmatrix}
50 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 50 \\
50 & 26.37 & 17.83 & 14.48 & 13.28 & 13.28 & 14.48 & 17.83 & 26.37 & 50 \\
50 & 37.66 & 30.45 & 26.82 & 25.34 & 25.34 & 26.82 & 30.45 & 37.66 & 50 \\
50 & 43.81 & 39.49 & 37.02 & 35.94 & 35.94 & 37.02 & 39.49 & 43.81 & 50 \\
50 & 48.10 & 46.68 & 45.83 & 45.44 & 45.44 & 45.83 & 46.68 & 48.10 & 50 \\
50 & 51.90 & 53.32 & 54.17 & 54.56 & 54.56 & 54.17 & 53.32 & 51.90 & 50 \\
50 & 56.19 & 60.51 & 62.98 & 64.06 & 64.06 & 62.98 & 60.51 & 56.19 & 50 \\
50 & 62.34 & 69.55 & 73.18 & 74.66 & 74.66 & 73.18 & 69.55 & 62.34 & 50 \\
50 & 73.63 & 82.17 & 85.52 & 86.72 & 86.72 & 85.52 & 82.17 & 73.63 & 50 \\
50 & 100 & 100 & 100 & 100 & 100 & 100 & 100 & 100 & 50
\end{bmatrix}$$

C.5 Distributing the Data for Parallel Performance

In Compaq's implementation of HPF, FORALL and Fortran 90 array assignment statements by themselves are not enough to achieve parallel execution. In order to achieve any improvement in performance through parallel execution, these structures must operate on arrays that have been distributed using the DISTRIBUTE directive. In general, parallel performance cannot be achieved without explicit use of the DISTRIBUTE directive.

C.5.1 Deciding on a Distribution

Because communication of data is time-consuming, a distribution of data that minimizes communication between processors is absolutely critical for application performance.

In many calculations (such as the LU decomposition example considered in Appendix B), cyclic distribution proves to be preferable because of load-balancing considerations. However, in nearest-neighbor problems such as the heat-flow example, it is advantageous to keep nearby elements on the same processor as much as possible in order to minimize communication. Distributing the array in large blocks allows all four nearest-neighbor elements to be on the same processor in most instances. If this is done, only the elements along the perimeter of the blocks require communication between processors in order to obtain the values of neighboring elements.

Large blocks can be obtained with the keyword BLOCK. For two-dimensional nearest-neighbor problems, the most important options are (BLOCK, BLOCK) distribution and (*, BLOCK) distribution. In the current version of Compaq Fortran, (*, BLOCK) distribution is highly optimized (see Section C.5.2), and frequently produces superior results. Because distributions are so easy to

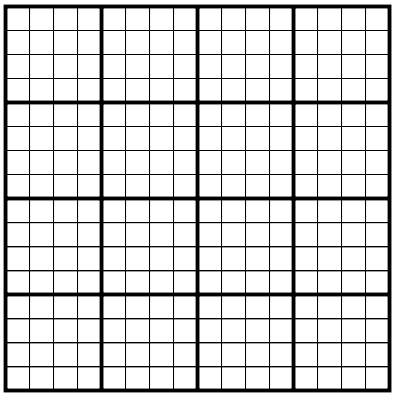
change in HPF, it can be worthwhile to time both options and choose whichever performs better.

Figures showing (∗, BLOCK) distribution and (BLOCK, BLOCK) distribution can be found in Appendix B (Figures B–1 and B–4) and in Chapter 5 (Figures 5–5, 5–6, 5–17, and 5–18).

For a fuller discussion of basic distribution options, see Section B.3.

C.5.2 Optimization of Nearest-Neighbor Problems

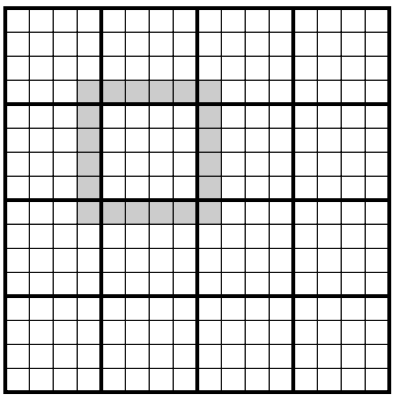
The Compaq Fortran compiler performs an optimization of nearest neighbor calculations to reduce communications. Each processor sends **shadow edges** to each processor that needs this data during the execution of the computation. For instance, if a 16 × 16 array *A* is allocated (BLOCK, BLOCK) on a machine with 16 processors, the array can be thought of as divided into 16 blocks, as follows:



MLO-011934

In addition to the area in each processor's memory allocated to the storage of that processor's block of the array, extra space is allocated for a surrounding shadow area which holds storage for those array elements in the neighboring processors which are needed for the computations. Within any one program unit, the compiler automatically determines the correct shadow-edge widths on an array-by-array, dimension-by-dimension basis. The shadow area for array *A* in the memory of one of the sixteen processors is shown shaded in Figure C–2.

Figure C–2 Shadow Edges for Nearest-Neighbor Optimization



MLO-011933

The `-show hpf` command-line option indicates which statements were detected by the compiler as nearest-neighbor statements.

You can also use the SHADOW directive and the `-nearest_neighbor` command-line option to do the following:

- Manually set shadow-edge widths to preserve the shadow edges when nearest-neighbor arrays are passed as arguments.
- Change the maximum allowable shadow-edge width.
- Disable the nearest-neighbor optimization.

For More Information:

- On using `-nearest_neighbor` and the SHADOW directive to control the nearest-neighbor optimization, see Sections 5.5.7 and 6.1.1.5.
- On the conditions that allow the compiler to recognize a statement as a nearest-neighbor computation, see Section 7.6.
- On the `-show hpf` option, see Section 6.1.1.7.

C.6 Packaging the Code

Executable source code for the heat-flow problem presented in this appendix can be found in the file `/usr/examples/hpf/heat_example.f90`.

D

HPF Tutorial: Visualizing the Mandelbrot Set

This appendix describes the development of a program to visualize the Mandelbrot set. It provides:

- A brief, non-technical introduction to the mathematics of the Mandelbrot set (see Section D.1).
- Explanation of the Mandelbrot example program (see Section D.2), with discussion of many of the Fortran 90 and HPF features used in the program, including:
  - Fortran 90 syntactical features, such as entity-oriented declarations, long symbol names, and the DO WHILE looping construct
  - The PURE attribute (Section D.2.4)

Source code of the program on which this appendix is based can be found in the file `/usr/examples/hpf/mandelbrot.f90`.

D.1 What Is the Mandelbrot Set?

The Mandelbrot set is generated from the iteration of the function $z^2 + c$, where c is a complex constant. At the first iteration, z is given a value of zero. For each subsequent iteration, the result of the previous iteration is used for z. Example D–1 shows the iteration of this function expressed in Fortran.

If a sufficient number of iterations are done, one of two possible results will be seen for each value of c:

- The value of the function increases rapidly with each iteration, tending to infinity.
- The value of the function stays within some finite range, and does not tend to infinity.

The Mandelbrot set is defined as those values of c for which the value of the function does not tend to infinity.

D.1.1 How Is the Mandelbrot Set Visualized?

The Mandelbrot set is customarily plotted on a grid representing the complex plane. In practical terms, the way this is usually done is based on the way Benoit Mandelbrot first visualized the set back in 1979: Each pixel of a computer monitor represents a point on the grid. The pixel is colored in (white, or some other color) if that point is proven to be outside of the set. The pixels that remain black (not colored in) represent an approximation of the Mandelbrot set.

You may notice that the definition of the Mandelbrot set is phrased in negative terms: the set consists of points on the complex plane (that is, values of *c*) which do *not* tend to infinity. For many points, the value of the function will seem to vary within some range for a large number of iterations before escalating suddenly and diverging to infinity.

If the function does not tend to infinity after a given number of iterations, there is frequently no way to know whether it would diverge if additional iterations were performed. No matter how many iterations are performed, any visualization of the Mandelbrot set will inevitably include some points that would have diverged if still more iterations were performed. Therefore, all visualizations of the Mandelbrot set are approximations that overestimate the size of the set. As the number of iterations is increased, the image of the Mandelbrot set gradually shrinks toward a more accurate shape.

D.1.2 Electrostatic Potential of the Set

It turns out that testing whether a point is in the Mandelbrot set yields important information even if that point is found to be outside of the set. A great deal of information can be gained by studying the **electrostatic potential** the set creates in the region outside of the set.

In concrete terms, imagine a metal pipe of very large diameter standing up on end. Standing up in the middle of this pipe, imagine a very thin stick-like object with the same length as the pipe, having the unusual property that its cross-section is shaped like the Mandelbrot set. If the stick is given a potential of zero, and the pipe is given a high potential, an electrical field will be created in the region between the stick and the pipe.

When the diameter of the pipe is increased to infinity, then a plane cutting horizontally through this system will represent the complex plane with the Mandelbrot set at its center. The infinite region containing the electrical field is the complement of the Mandelbrot set.

Equipotential lines, which are lines connecting points of equal potential, can be drawn in the Mandelbrot complement region of this horizontal plane. These lines form a series of concentric rings, which are near-perfect circles at great distances from the origin, and increasingly distorted and twisted closer to the Mandelbrot set region. These equipotential lines, and the **field lines** that cross them at right angles, give a large amount of information about the shape and other characteristics of the Mandelbrot set.

A remarkable mathematical property of this system is that the potential of any point in the Mandelbrot complement set is a simple function of its **escape time**. Escape time is defined as the number of iterations needed for the value of the Mandelbrot function to escape beyond a circle of some (arbitrary) large radius centered at the origin. Since the entire Mandelbrot set lies inside the circle of radius 2, any radius greater than or equal to 2 can be used. However, the larger the radius, the more accurate the approximation of the Mandelbrot complement set.

Put simply, the potential of a point in the Mandelbrot complement set is measured by how quickly the value diverges toward infinity.

D.2 Mandelbrot Example Program

At this point, you might want to compile and run the Mandelbrot example program provided with your software. Simple instructions can be found in the file `/usr/examples/hpf/README.mandelbrot`.

When the example program first starts, a window is displayed showing the Mandelbrot set in black, surrounded by the Mandelbrot complement set shown in multiple colors representing various ranges of potential according to the electrostatic model explained in Section D.1.2.

The window and image are sized so that the center of the window represents the origin of the complex plane. The axes of the plane intersect the edges of the window at a distance of 2 from the origin. The point representing -2, the point in the Mandelbrot set most distant from the origin, is located in the middle of the left side of the window. The size of the display area is 625×625. The mouse buttons are used to zoom in or zoom out, creating a new image of different scale with each click, as explained in the file `/usr/examples/hpf/README.mandelbrot`.

D.2.1 Developing the Algorithm

Example D–1 shows the iteration of the function that determines whether a given complex number is in the Mandelbrot set.

Example D–1 Iteration of the Function $z^2 + c$

```
COMPLEX            :: z, c
INTEGER            :: n, esc_time=0
INTEGER, PARAMETER :: n_max=1000      ! Arbitrary maximum # of iterations
INTEGER, PARAMETER :: escape_radius=400 ! Arbitrary criterion for escape
LOGICAL            :: in_the_mandel_set
z=0
n = 0
DO WHILE (ABS(z) < escape_radius .AND. (n <= n_max) )
  z = z**2 + c
  n = n + 1
END DO

esc_time = n
IF (n <= n_max) THEN
  in_the_mandel_set = .TRUE.
ELSE
  in_the_mandel_set = .FALSE.
END IF
```

Some of the Fortran 90 features used in Example D–1 are:

- The **::** notation — Used in the **entity-oriented declaration** form in which all attributes of an entity may be grouped in a single statement
- Long symbol names using the underscore (_) character
- DO WHILE looping construct

Example D–1 tests whether any given value of `c` is in the Mandelbrot set. The loop condition uses the ABS intrinsic function, because complex numbers can only be compared by absolute value, which is defined as the distance from the origin in the complex plane. If `n_max` iterations are performed without the absolute value of `z` exceeding the escape radius, the given value of `c` is presumed (although not proven) to be part of the Mandelbrot set. If the the loop is exited before `n_max` iterations have been completed, the given value of `c` has been proven to lie outside of the Mandelbrot set.

For points proven to be outside of the set, the value of *n* when the loop is exited is the **escape time**. The escape time can be used to plot equipotential lines and to color in regions of varying potential.

Any value greater than or equal to 2 could have been chosen for the escape radius, because the Mandelbrot set is entirely contained within a circle of radius 2. However, a substantially larger value (400) was chosen because it will cause the equipotential lines in the Mandelbrot complement set to be considerably more accurate.

Even though it makes the complement set more accurate, using a larger escape radius causes a very slight degradation in the accuracy of the shape of the Mandelbrot set itself. However, the effect of this degradation is barely noticable in visual terms, because values tend to escalate very rapidly once their absolute value exceeds 2.

D.2.2 Computing the Entire Grid

The image of the Mandelbrot set is plotted on a grid, with each pixel of a window of a computer monitor representing one point on the grid. The escape time is calculated for each point proven to lie outside of the Mandelbrot set, with all points having the same escape time assigned the same color in the image. Points not proven to lie outside of the set are left black. Example D–1, which calculates the escape time for a single point, can be expanded to generate the entire grid simply by putting nested DO loops around the calculation. See Example D–2.

Example D–2 Using a DO Loop to Compute the Grid

```
COMPLEX            :: z, c
INTEGER            :: n, esc_time=0, target(grid_height, grid_width)
INTEGER, PARAMETER :: n_max=1000        ! Arbitrary maximum # of iterations
INTEGER, PARAMETER :: escape_radius=400 ! Arbitrary criterion for escape
INTEGER, PARAMETER :: grid_height=625, grid_width=625
DO x = 1, grid_width
   DO y = 1, grid_height
      c = CMPLX(x, y)
      z=0
      n = 0
      DO WHILE (ABS(z) < escape_radius .AND. (n <= n_max) )
        z = z**2 + c
        n = n + 1
      END DO
      esc_time = n
      target(x, y) = esc_time
   END DO
END DO
```

As a simplification, Example D–2 assumes the origin of the complex plane is in the lower left-hand corner of the image.

D.2.3 Converting to HPF

DO loops prescribe that calculations be done in a certain order. Therefore, Example D–2 prescribes the order in which the grid points are calculated. However, careful examination of Example D–2 reveals that the computation for each grid point is completely independent and unrelated to the computation for any other point on the grid. Thus, the order of the calculation has no effect on the result of the program. The same result would be produced if the grid points were calculated in the opposite order, or even in random order. This means that this routine is an excellent candidate for parallelizing with HPF. When the routine is converted to HPF, several grid points will be calculated simultaneously, depending upon the number of processors available. Generating Mandelbrot visualizations is a completely (or "embarrassingly") parallel computation.

To allow parallel execution of this routine, the target array must be distributed across processors using the DISTRIBUTE directive, and the two outer DO loops must either be replaced with a FORALL structure or marked with the INDEPENDENT directive.

Replacing DO loops with a FORALL structure presents a problem, however: FORALL is not a loop, but an assignment statement. An assignment statement cannot contain the assignments to multiple variables and flow control constructs (such as DO WHILE) that occur in Example D–2. A FORALL structure is limited to assigning values to elements of a single array.

The solution to this problem is to package the bulk of the calculation into a user-defined function. Function calls inside assignment statements are permitted, and in this way the entire routine can be parallelized. Example D–3 shows the FORALL structure containing a call to the user-defined function `escape_time`, and Example D–4 shows the function, which contains the calculation for a single grid point.

Example D–3 Using a FORALL Structure to Compute the Grid

```
INTEGER           :: target(grid_height, grid_width)
INTEGER, PARAMETER :: n_max=1000     ! Arbitrary maximum # of iterations
INTEGER, PARAMETER :: grid_height=625, grid_width=625
FORALL(x=1:grid_width, y=1:grid_height)
   target(x, y) = escape_time( CMPLX(x, y), n_max )
END FORALL
```

Example D–4 PURE Function escape_time

```
PURE FUNCTION escape_time(c, n_max)
  COMPLEX, INTENT(IN) :: c
  INTEGER, INTENT(IN) :: nmax
  INTEGER             :: n
  COMPLEX             :: z
  n = 0
  z = c
  DO WHILE (ABS(z) < 2.0 .AND. (n < n_max))
    z = z * z + c
    n = n + 1
  END DO
  IF (n >= n_max) THEN
    escape_time = nmax
  ELSE
    escape_time = n
  END IF
END FUNCTION escape_time
```

D.2.4 PURE Attribute

The escape_time function is given the PURE attribute. The PURE attribute is an assertion that a function has no side effects and makes no reference to mapped variables other than its actual argument. A PURE function's only effect on the state of the program is to return a value.

User-defined functions may be called inside a FORALL structure only if they are PURE functions. The reason for this rule is that iterations of a FORALL structure occur in an indeterminate order. Therefore, allowing functions that have side effects (such as modifying the value of a global variable) to be called from within a FORALL structure could lead to indeterminate program results.

For details about PURE and side effects, see Section 5.6.4, PURE Attribute and the High Performance Fortran Language Specification.

E

HPF Tutorial: Simulating Network Striped Files

This appendix explains how to optimize temporary input/output (I/O) through the use of network striped file simulation. Network striped files are useful for programs that use checkpointing or whenever temporary I/O needs to be done.

E.1 Why Simulate Network Striped Files?

In HPF, all I/O operations are serialized through a single processor. For example, when output must be done, all of the data being written is copied to a temporary buffer on Peer 0 and the print function is then performed by Peer 0. This communication of data is necessary to produce meaningful output, because only a fragment of a distributed data object is normally available on any given processor. In order for distributed data to be output in a usable form, the data must first be gathered onto a single processor.

However, checkpointing is a special case. In checkpointing, the state of a program is preserved through I/O to allow restarting in case of software or hardware failure. Although ordinary serialized I/O through Peer 0 accomplishes this goal, the performance cost of this approach is very great in many cases. Because the only purpose of checkpointing is to preserve the state of the program, the relevant data stored on each peer can be output directly without regard to how it fits together with data stored on other peers.

This method of parallel output is known as **simulating a network striped file**. As long as the relevant data stored on every peer is output, checkpointing can be accomplished without first gathering the data onto one peer. This technique eliminates the need for moving the data to Peer 0. Because all movement of data between processors is eliminated, network striped file simulation is a much more efficient checkpointing technique than ordinary I/O.

For More Information:

- On I/O in HPF, see Section 7.11.

E.1.1 Constructing a Module for Parallel Temporary Files

This appendix will show how to simulate network striped files using a module containing EXTRINSIC routines that simulate Fortran I/O statements such as READ and WRITE. EXTRINSIC routines allow code based upon non-HPF computing models to be incorporated into an HPF program.

The data parallel model uses a single logical thread of control and views the distributed processing environment as a single logical computer. This is the HPF computing model.

Simulating a network striped file requires diverging from the HPF computing model. A programming model that views the entire cluster as a single computer would not allow us to specify that each node should write its data to its own local device.

A more appropriate programming model for simulating a network striped file is explicit single program/multiple data (SPMD) programming. Explicit SPMD programming lacks the global addressing that is available in HPF. In explicit SPMD programming, a separate copy of the same program — parameterized by processor number — is executed by each processor. Unlike an HPF routine, in which a distributed array is addressed as a single entity, distributed arrays have no direct representation in explicit SPMD routines. Each processor addresses its own slice of such an array as if it were a separate local array. The global array that is the sum of the parts stored by each local processor exists only in the mind of the programmer.

HPF lets you mix programming models on a procedure basis:

- EXTRINSIC(HPF) procedures are those using the HPF data parallel model. This is the default model.
- EXTRINSIC(HPF_LOCAL) procedures are those using an explicit SPMD model

Using EXTRINSIC(HPF_LOCAL), explicit SPMD code to simulate a network striped file can be incorporated into an HPF program. In network striped file simulation, a set of files (one file for each peer) is treated as a single logical file.

The module should define subroutines to be parallel versions of OPEN, CLOSE, READ, WRITE, and REWIND. The module uses two private variables defining the range of logical unit numbers to be used for temporary files, in this case 90 to 99:

```
EXTRINSIC(HPF_LOCAL) MODULE parallel_temporary_files
  INTEGER, PRIVATE :: highest_unit_number = 90
  INTEGER, PRIVATE :: maximum_unit_number = 99
CONTAINS
  EXTRINSIC(HPF_LOCAL) SUBROUTINE parallel_open(unit_number, ok)
   . . .
  END SUBROUTINE parallel_open
   . . .
END MODULE parallel_temporary_files
```

E.2 Subroutine parallel_open

The subroutine parallel_open assigns the next available logical unit number in the range highest_unit_number to maximum_unit_number. A unique eight-character file name is generated by concatenating the letters "tmp" with the logical unit number and the peer number. A new file with this name is opened in the current directory and the logical unit number is returned as an OUT parameter.

Assume the example is compiled for four peers. The first time parallel_open is called, it opens four files, tmp90000, tmp90001, tmp90002, and tmp90003, one on each peer. It returns the single scalar value 90 as the value of unit_number and .true. as the value of ok.

```
EXTRINSIC(hpf_local) SUBROUTINE parallel_open(unit_number, ok)
  INTEGER, INTENT(OUT)  :: unit_number
  LOGICAL, INTENT(OUT)  :: ok
  CHARACTER*8           :: file_name
  IF (highest_unit_number <= maximum_unit_number) THEN
    unit_number = highest_unit_number
    highest_unit_number = highest_unit_number + 1
    WRITE(unit=file_name, fmt='(a3,i2.2,i3.3)')    &
      'tmp', unit_number, my_processor()
  ELSE  ! Too many temporary files
    ok = .false.
    RETURN
  END IF
  OPEN(UNIT=unit_number,   STATUS='new',         &
       FORM='unformatted', FILE=file_name,       &
       IOSTAT=jj,          ERR=100)
  ok = .true.
  RETURN
  100 ok = .false.
END SUBROUTINE parallel_open
```

E.3 Subroutine parallel_write

When the subroutine parallel_write is called, it receives as parameters the scalar logical unit number and the section of the array actual argument on a particular peer. Each peer process only receives the part of the array actual argument, if any, that is mapped to that peer. The following HPF_LOCAL routine writes its part of the array actual argument to its local file.

```
EXTRINSIC(HPF_LOCAL) &
SUBROUTINE parallel_write(unit_number, A, ok)
  INTEGER, INTENT(IN)                          :: unit_number
  INTEGER, DIMENSION(:), INTENT(IN)            :: A
    !HPF$ DISTRIBUTE (BLOCK) :: A
  LOGICAL, INTENT(OUT) :: ok
  WRITE(unit_number, ERR=100, IOSTAT=jj) A
  ok = .true.
  RETURN
  100 ok = .false.
END SUBROUTINE parallel_write
```

E.3.1 Passing Data Through the Interface

It is good programming practice always to provide an explicit interface to subroutines and functions with mapped dummy arguments (such as A in the previous example). In quite a large number of cases, the HPF language requires an **explicit interface** for such routines.

An explicit interface consists of one of the following:

- The calling routine may contain an interface block describing the called routine. The interface block must contain dummy variable declarations and mapping directives that are in the routine it describes.
- The calling routine may contain a USE statement referring to a module that contains an interface block for the called routine.
- The calling routine may (using a CONTAINS statement) contain the called routine in its entirety.

Even when an explicit interface is not required (roughly speaking, when the dummy can get the contents of the actual without inter-processor communication), Compaq recommends that you do so anyway. Explicit interfaces cut down on programming errors, and give more information to the compiler, providing more opportunities for the compiler to optimize your program.

In the test program in Example E–1, program main contains a USE statement referring to the parallel_temporary_files module, which contains the interface block for parallel_write.

For More Information:

- On explicit interfaces, see Section 5.6.2.
- For an easy way to provide explicit interfaces, see Section 5.6.3.

E.4 Subroutines parallel_read, parallel_close, and parallel_rewind

These three subroutines follow the same structure as the subroutine `parallel_write` and can be found in the file `/usr/examples/hpf/io_example.f90`.

E.5 Module parallel_temporary_files

The complete source code for the module `parallel_temporary_files` (containing the subroutines `parallel_open`, `parallel_write`, `parallel_read`, `parallel_rewind`, and `parallel_close`) can be found in the file `/usr/examples/hpf/io_example.f90`.

The test program `main`, shown in Example E–1, can be found in the same location. Example E–1 is a test program for the module `parallel_temporary_files`. Notice that `parallel_read` is called only if the return status in the variable `ok` is true.

Example E–1 Test Program for Parallel Temporary Files

```
PROGRAM main
  USE parallel_temporary_files
  INTEGER, PARAMETER    :: n=12
  INTEGER               :: temp_unit
  LOGICAL               :: ok
  INTEGER, DIMENSION(n) :: DATA
    !HPF$ DISTRIBUTE data(BLOCK)
  INTEGER, DIMENSION(n) :: b
    !HPF$ ALIGN b(:) WITH data(:)

  FORALL (i=1:n) data(i) = i

  DO i=1,2
    b = 0
    CALL parallel_open(temp_unit, ok)
    PRINT *, "in main:", "open", temp_unit
    CALL parallel_write(temp_unit, data, ok)
    PRINT *, "in main:", "write", temp_unit, ok
    CALL parallel_rewind(temp_unit, ok)
    PRINT *, "in main:", "rewind", temp_unit, ok
    IF (ok) CALL parallel_read(temp_unit, b)
    CALL parallel_close(temp_unit)
    IF(ANY(b /= (/ (i, i=1,n ) /))) THEN
      PRINT *, 'Error'
    ELSE
      PRINT *, 'Ok!!!'
    ENDIF
  ENDDO

END PROGRAM main
```

Index

D

I

J

L

R

S