Sun™ HPF 1.0 Guide

THE NETWORK IS THE COMPUTER™

Sun Microsystems Computer Company
A Sun Microsystems, Inc. Business
901 San Antonio Road
Palo Alto, CA 94303-4900 USA
650 960-1300 fax 650 969-9131

Part No.: 805-1558-10
Revision A, November 1997

Contents

Preface vii

1. Introduction 1-1

2. Getting Started with Sun HPF 2-1

2.1 Preliminary Notes 2-1

2.1.1 Parallel and Serial Arrays 2-1

2.1.2 Array Distribution 2-3

2.1.3 Distribute Directive Overview 2-4

2.2 Example of F77 to HPF Transition 2-8

2.3 Using the `DISTRIBUTE` Directive 2-11

3. Compiling and Linking Sun HPF Programs 3-1

3.1 `hpf` Command Basics 3-1

3.1.1 Command Syntax 3-1

3.1.2 File Name Extensions 3-2

3.1.3 Linking Libraries with `hpf` 3-2

3.2 `hpf` in Action 3-5

3.2.1 Compiling and Linking 3-7

3.2.2 Compiling CMF Programs into Sun HPF Executables 3-7

3.2.3 Using the `cpp` Preprocessor 3-8

3.2.4 Compiling Incrementally 3-8
3.2.5 Program Debugging and Profiling 3-8
3.2.6 Command-Line Arguments 3-9
3.3 Summary of Optional Switches 3-10

4. Program Development Tools 4-1

4.1 Timing a Program 4-1
4.1.1 Hints on Using the Timing Utility 4-4
4.1.2 Synchronization Considerations 4-4
4.2 The C Language Preprocessor 4-5
4.2.1 Used with `-D` 4-5

5. File Systems and File System I/O 5-1

5.1 Introduction 5-1
5.2 PFS File Path Names 5-2
5.3 Programming Examples 5-3

6. Performance Notes 6-1

6.1 Use Parallel Language Expressions 6-1
6.1.1 Array Assignments 6-1
6.1.2 Intrinsic Functions Applied to Arrays or Array Expressions 6-3
6.2 Minimize Communication 6-4
6.3 Be Explicit 6-4
6.4 Map Arrays Explicitly 6-5
6.5 Avoid Passing Array Sections 6-6
6.6 Operate on Whole Arrays 6-8
6.7 Ratio of Processes to Processors 6-9
6.8 Avoid Expensive Operations 6-9
6.9 Use S3L Functions 6-10
6.10 Use Simple Constructs 6-11
6.11 Avoid General Communications 6-11

6.12 Compiler Switches 6-11
6.13 Shared Memory Environment Variables 6-13
6.14 Performance Analysis 6-13
6.14.1 Do Repeated Timing Runs 6-13
6.14.2 Use `-Xlist` to Analyze Communication 6-13
6.14.3 Examine the `.f` File (for advanced users) 6-14
6.14.4 Profile the Code 6-14

7. Sun HPF Summary 7-1
7.1 Fortran 90 Features in Sun HPF 7-1
7.2 HPF Directives and Language Extensions 7-4
7.2.1 Summary of HPF 1.1 Subset 7-4
7.2.2 Sun HPF Restrictions 7-4
7.3 Additional Fortran 90 Features 7-5

8. F77_LOCAL Interface 8-1
8.1 Introduction 8-1
8.2 Processor Synchronization 8-2
8.3 Linking for F77_LOCAL 8-2
8.4 Debugging F77_LOCAL Code with Prism 8-2
8.5 Argument Passing 8-2
8.6 HPF-Style Utilities 8-4
8.7 Subgrid-Inquiry Utilities 8-5
8.7.1 `TMHPF_SUBGRID_INFO` 8-5
8.7.2 `F77_SUBGRID_INFO` 8-6
8.8 Programming Examples 8-6
8.8.1 Using `MAP_TO(F77_ARRAY)` 8-7
8.8.2 Using `MAP_TO(NO_CHANGE)` 8-9

9. HPF Intrinsic Functions and the HPF Library 9-1
9.1 HPF Intrinsic Functions 9-1

9.1.1 System Inquiry Intrinsic Functions 9-1
9.1.2 Elemental Intrinsics Function 9-2
9.1.3 Array Location Intrinsic Functions 9-2
9.2 The HPF Library 9-2
9.2.1 Bit Manipulation Functions 9-2
9.2.2 Mapping Inquiry Subroutines 9-2
9.2.3 Array Reduction Functions 9-3
9.2.4 Array Combining Scatter Functions 9-3
9.2.5 Array Prefix and Suffix Functions 9-3
9.2.6 Array Ranking Functions 9-4
9.2.7 Array Sorting Functions 9-4
9.3 HPF Library Exceptions and Other Notes 9-4

10. The HPF/CMF Utility Library 10-1

A. IOSTAT Message Summary A-1

Index Index-1

Preface

This manual is directed toward developers of High Performance Fortran (HPF) programs and explains how to use the Sun HPF compiler. It also describes the principal language features of Sun HPF, including an EXTRINSIC(F77_LOCAL) interface and a CMF back-compatibility mode.

Using UNIX Commands

This document does not describe basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See the following for this information:

- AnswerBook™ online documentation for the Solaris 2.x software environment.

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output.	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output.	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized. Command-line variable; replace with a real name or value.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be `root` to do this. To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	*machine_name*%
C shell superuser	*machine_name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Related Publications

Books

Among the documents included with Sun HPC Software, you may want to pay particular attention to these:

TABLE P-3 Related Documentation

Application	Title	Part Number
All	*Sun HPC Software 2.0 Release Notes*	805-2191-10
Prism	*Prism 5.0 User's Guide*	805-1552-10
Prism	*Prism 5.0 Reference Manual*	805-1553-10
S3L	*S3L 2.0 Guide*	805-1557-10
Sun MPI Programming	*Sun MPI 3.0 Guide*	805-1556-10

This book, which is not available from Sun, should be available at your local computer bookstore:

- *The High Performance Fortran Handbook.* Charles H. Koelbel et al. Cambridge, MA.: MIT Press, 1994.

On the Internet

- The *High Performance Fortran Language Specification*, Version 2.0, is available online via anonymous ftp:

  `ftp://ftp.cs.rice.edu/public/HPFF/draft/`

- The High Performance Fortran Forum maintains a web site:

  `http://www.crpc.rice.edu/HPFF/`

Ordering Sun Documents

SunDocs[SM] is a distribution program for Sun Microsystems technical documentation. Contact SunExpress for easy ordering and quick delivery. You can find a listing of available Sun documentation on the World Wide Web.

TABLE P-4 SunExpress Contact Information

Country	Telephone	Fax
Belgium	02-720-09-09	02-725-88-50
Canada	1-800-873-7869	1-800-944-0661
France	0800-90-61-57	0800-90-61-58
Germany	01-30-81-61-91	01-30-81-61-92
Holland	06-022-34-45	06-022-34-46
Japan	0120-33-9096	0120-33-9097
Luxembourg	32-2-720-09-09	32-2-725-88-50
Sweden	020-79-57-26	020-79-57-27
Switzerland	0800-55-19-26	0800-55-19-27
United Kingdom	0800-89-88-88	0800-89-88-87
United States	1-800-873-7869	1-800-944-0661
World Wide Web: `http://www.sun.com/sunexpress/`		

Sun Documentation on the Web

The `docs.sun.com` web site enables you to access Sun technical documentation on the World Wide Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email or fax your comments to us. Please include the part number of your document in the subject line of your email or fax message.

- Email: `smcc-docs@sun.com`
- Fax: SMCC Document Feedback
  1-650-786-6443

LSF Technical Support

LSF 3.0, a product of Platform Computing Corporation, is part of the Sun HPC Software 2.0 Foundation Package. As such, it is supported by Sun as part of Sun HPC Software 2.0.

Sun HPC Software includes LSF Base and LSF Batch. However, LSF JobScheduler and LSF MultiCluster are not included and, therefore, not supported by Sun.

Information Sources for PVM and PETSc

TABLE P-5 lists organizations and resources for information about the publicly available libraries PVM and PETSc. This information is subject to change.

TABLE P-5 Information Sources for PVM and PETSc

Product	Contact
PVM	Copyright holders: University of Tennessee, Oak Ridge National Laboratory, Emory University Electronic mail: `pvm@msr.epm.ornl.gov` Newsgroup: `comp.parallel.pvm` Web site: `http://www.epm.ornl.gov/pvm/pvm_home.html`
PETSc	Developed and supported by the Mathematics and Computer Science Division of the Argonne National Laboratory.

CHAPTER 1

Introduction

The Sun HPF compiler is the product of more than a decade of experience in developing advanced data parallel compilers and run-time systems for high-performance parallel architectures. It is a key component of the Sun Ultra HPC software environment.

Sun HPF 1.0 supports the following functionality:

- Subset HPF
- Fortran 90 free source form
- HPF library
- Various features from Fortran 90 not in Subset HPF
- Source-level debugging and program development with Prism
- An `EXTRINSIC(F77_LOCAL)` mechanism that supports local programming with MPI message passing
- Routines from the CM Fortran utility library (for back-compatibility when a CM Fortran routine is more efficient than its equivalent in HPF)

CHAPTER 2

Getting Started with Sun HPF

This chapter introduces some of the features that distinguish HPF from traditional Fortran. It is not intended to be a comprehensive overview of the HPF language, but simply an introduction. It uses a few basic HPF features to illustrate key HPF programming concepts.

The introduction to basic HPF features and programming concepts begins in Section 2.2, “Example of F77 to HPF Transition.” Before proceeding to that section, however, take time to read Section 2.1.1, “Parallel and Serial Arrays,” which provides an overview of how Sun HPF arrays can be distributed across processes.

Chapter 7 of this manual provides a summary of the set of HPF features that constitute Sun HPF, as well as an overview of extensions it includes. For a comprehensive description of the HPF language, refer to the *High Performance Fortran Language Specification*, Version 2.0. See the section “Related Publications” of the preface for information about accessing this document on the Internet.

2.1 Preliminary Notes

2.1.1 Parallel and Serial Arrays

The Sun HPF compiler implements each array in a program unit as either a parallel array or a serial array. This has the following consequences for data distribution:

- Parallel arrays are both *nonsequential* and *distributed*. This means the compiler assumes that the parallel array does not involve any sequence association. It also means the compiler will divide the array into blocks and distribute the blocks across multiple memory units.

- Serial arrays are both sequential and nondistributed. The compiler keeps a serial array intact and replicates it on every memory unit. Serial arrays can be used in Fortran 77-style sequence and storage association.

Parallel arrays are the cornerstone of HPF programming. Because they are distributed across multiple processes, operations on the array elements can be performed in parallel, effectively multiplying execution speed for that program block by the number of processes engaged in the parallel operation.

An array can be implemented as a parallel array either *implicitly* or *explicitly*.

An array is implicitly parallel when it is referenced in an array operation using F90 array syntax, as in the following examples:

```
A = B                 ! assignment
A(:,:) = C(:,:,5)     ! subscript triplet notation
A = cshift(B,1,1)     ! F90 intrinsic operation
WHERE (A > B) C = D   ! WHERE statement or construct
```

The most direct method for explicitly making an array parallel is to use the compiler directive `DISTRIBUTE`, which tells the compiler to distribute one or more of the array's dimensions onto a set of abstract processes.

There are other explicit directives that force an array to be parallel, including use of the `ALIGN` directive to align it with a parallel array.

Note – You are likely to find the `DISTRIBUTE` directive to be the most effective HPF feature for use in performance tuning your HPF applications. Consequently, it is the principal focus of this chapter.

If an array is neither implicitly nor explicitly treated as a parallel array, the Sun HPF compiler treats it as serial. Consequently, Sun HPF arrays are sequential by default. Note that this is contrary to the HPF recommendation.

Serial arrays are useful for those data operations where elements must be accessed sequentially or where every process needs to access the same data elements.

The `DISTRIBUTE` directive can be used to make an array explicitly serial by distributing it onto a scalar processor arrangement or by collapsing all its dimensions. To *collapse* an array dimension means to replicate the entire dimension on every participating process.

2.1.2 Array Distribution

Like all optimizing compilers, the Sun HPF compiler looks for clues in the source code that imply opportunities for generating more efficient output code.

In HPF programs, the most critical efficiency clues are implicitly provided by the use of parallel language constructions in parallel expressions. The F90 array syntax used to express parallel array operations tells the compiler that the referenced arrays can be distributed to multiple processes for parallel execution of the operations.

These parallel constructions are essential for parallelization to take place. That is, if a parallel language construction is *not* used to express an operation on an array, the compiler will treat the operation as serial and wil *not* divide the operations among the available processes.

These implicit clues can be augmented by explicit instructions to the compiler in the form of HPF compiler directives. These directives, which take the form of comments in the source code, give the compiler additional information about how the arrays should be mapped for optimal execution efficiency.

Sun HPF supports all the compiler directives included in the Subset HPF 1.1 specification. These are:

- `DISTRIBUTE` — Tells the compiler how each dimension of an array or template should be distributed across the participating processes. See Section 3.3 of the High Performance Fortran Language Specification, Version 2.0, for a detailed description of the `DISTRIBUTE` directive.
- `PROCESSORS` — Defines a rectilinear processor configuration that has a user-assigned name, some number of dimensions (its rank), and a specified length for each dimension. This abstract processor arrangement can then be named as the set of processors to which an array is distributed. See Section 3.6 of the High Performance Fortran Language Specification, Version 2.0, for a detailed description of the `PROCESSORS` directive.
- `ALIGN` — Tells the compiler how one or more arrays should be aligned with each other or with some predefined array template. Note that an array cannot be both an alignee and a distributee in the same program unit. See Section 3.4 of the High Performance Fortran Language Specification, Version 2.0, for a detailed description of the `ALIGN` directive.
- `TEMPLATE` — Defines an abstract array that has no content, but provides a set of indexed positions to which actual arrays can be aligned. The user specifies the template's name and its extent or range of indices for each dimension. See Section 3.7 of the High Performance Fortran Language Specification, Version 2.0, for a detailed description of the `TEMPLATE` directive.
- `SEQUENCE` — Declares an array to be explicitly sequential. A `NO SEQUENCE` directive is also provided to allow the user to explicitly specify an array to be nonsequential. Note that this is never needed if a mapping directive such as

DISTRIBUTE or ALIGN is given for the same array. See Section 3.8.2 of the High Performance Fortran Language Specification, Version 2.0, for a detailed description of the SEQUENCE directive.

- INDEPENDENT — Tells the compiler that the operations in the DO-loop or FORALL statement immediately following are independent of each other, for each value of the DO loop variable or each combination of the FORALL indices, and so can be performed in any order. See Section 5.1 of the High Performance Fortran Language Specification, Version 2.0, for a detailed description of the INDEPENDENT directive

---

Note – The Sun HPF compiler accepts the INDEPENDENT directive, but takes no action based on it.

The rest of this discussion will focus on the DISTRIBUTE directive, which is the simplest and most effective HPF directive you can use to help the compiler generate highly optimized code. See the section “Related Publications” of the preface for information about accessing the *High Performance Fortran Language Specification*, Version 2.0, on the Internet.

2.1.3 Distribute Directive Overview

The DISTRIBUTE directive allows you to specify how each dimension of an array should be mapped to a set of abstract processors. Any one of three types of distribution can be specified: block, block(N), or collapsed. The main features of these distribution types are summarized below.

Note – The HPF language also allows cyclic and cyclic(N) distribution. The current Sun HPF compiler converts these into block distribution.

2.1.3.1 Block Distribution

When the compiler does block-style distribution of an array dimension, it partitions that dimension into blocks of roughly equal number of indices and distributes the blocks across the corresponding dimension of an abstract rectilinear processor arrangement. This arrangement can be either explicitly declared by the user or formed at run time from the available processes. At most one abstract processor can be associated with each physical process.

This distribution produces a set of subarrays (called subgrids) residing on the participating processes. Each subgrid may be viewed as a rectilinear block of contiguous elements of the distributed array, resulting from partitioning it along one or more of its dimensions.

The run-time system will attempt to make the blocks as small as possible. In other words, it will try to distribute the array onto as many processes as possible to maximize the degree of parallelism and, therefore, the aggregate speed of execution.

The subgrids for an array distributed onto more than one process will be smaller versions of the distributed array. That is, they will have the same number of dimensions, but the block-distributed dimensions will typically be smaller (will contain fewer indices). If the run-time system cannot distribute the array evenly across all the processes, some subgrids will be smaller than their siblings.

FIGURE 2-1 shows examples of a 16-element 2D array (A) being distributed onto two processes and onto four processes. Another example of this directive in use is provided in Section 2.3, “Using the DISTRIBUTE Directive.”

2.1.3.2 Block(*N*) Distributions

This is a more detailed form of block distribution. It allows the programmer to specify the number of dimension subscripts in each (typical) block for a given dimension.

A block(*N*) distribution for a dimension means that the array's dimension should be broken into chunks of size *N* (or smaller, at the end of the axis) and distributed in this fashion across the corresponding abstract processor dimension. For example,

```
!hpf$ distribute (block(M),block(N)) :: C
```

means that each typical subgrid of `C` will have shape *M* by *N*, although some may be smaller.

2.1.3.3 Collapsed Dimensions

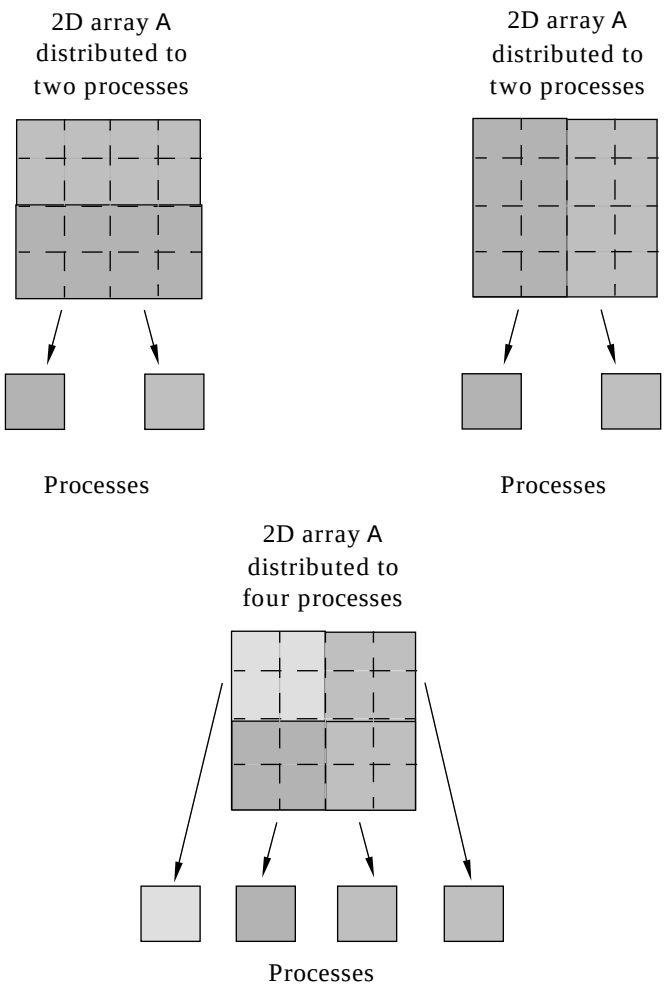
A collapsed dimension is one that is kept in a single block rather than being partitioned into smaller ones. It is considered a purely local (or serial) dimension rather than a distributed, parallel dimension, and it will not be mapped onto any abstract processor dimension.

In the `DISTRIBUTE` directive, collapsed dimensions are specified using an "*" rather than "block". For example:

```
real D(N,M), E(L1,L2,N)
!hpf$ distribute (block,*)     :: D
!hpf$ distribute (*,*,block)   :: E
```

means that `D` and `E` will be partitioned only along their sole block dimension of extent *N*, and also that they will be mapped onto 1D rather than 2D or 3D processor arrangements.

Note that for Sun HPF, it is also safe to assume that the parallel dimension of `D` or `E` is distributed like other block-distributed arrays in the same program having exactly one parallel dimension of extent *N*. In particular, array sections of the form `D(:,I)` and `E(J,K,:)` will be implicitly aligned.



NOTE: The following is an example of the `DISTRIBUTE` directive used to produce these block distributions.

```
!HPF$ distribute (block, block) :: A
```

Other examples of this directive are provided in Section 2.3, “Using the DISTRIBUTE Directive.”

FIGURE 2-1 Examples of Block Distribution

2.2 Example of F77 to HPF Transition

The introduction to HPF programming begins with the sample F77 program, `CONVOLVE_F77`, which is shown in FIGURE 2-2.

`CONVOLVE_F77` operates on two 2D arrays, `P` and `Q`, making use of elements of a 1D array, `F`. The 2D arrays are the same size and shape, 512 x 512 elements. The 1D array has 16 elements.

The computational tasks in `CONVOLVE_F77` are based on two sets of nested `DO` loops. One loop shifts the first dimension of `P`. Following the shift, the second loop completes the convolution, storing the result in `Q`.

The first change required for making the transition to an HPF program is to replace the F77-style code with array operations employing F90-based array syntax. To do this example, use array assignment operations and a `FORALL` statement to initialize and update `P` and `Q`. The array assignment statements cause the compiler to assume that `P` and `Q` (but not `F`) should be parallel arrays. The result of this transformation is shown in FIGURE 2-3.

Note – A `FORALL` statement is an HPF parallel array assignment. It expresses assignments to array elements in arbitrary order over a specified range of index variables.

```
program CONVOLVE_F77
      integer NX, NT, NFP
      parameter (NX=512, NT=512, NFP=16)
      integer I, IFP
      real F(NFP)
      real P(NT,NX), Q(NT,NX)

C  Initialize P and Q
      do I = 1, NX
         do J = 1, NT
            P(J,I) = J
            Q(J,I) = 0.0
         end do
      end do

C  Initialize F
      do I = 1, NFP
         F(I) = I
      end do

C  Shift array P by one over the first dimension
C  Compute Q after the shift on P

      do IFP = 1, NFP
         do I = 1, NX
            do J = NT-1, 1, -1
               P(J+1,I) = P(J,I)
            end do
            P(1,I) = 0.0
         end do

         do I = 1, NX
            do J = 1, NT
               Q(J,I) = Q(J,I) + P(J,I) * F(IFP)
            end do
         end do
      end do
```

FIGURE 2-2 F77 Program Example — CONVOLVE_F77

```
program CONVOLVE_F90
      integer NX, NT, NFP
      parameter (NX=512, NT=512, NFP=16)
      integer I, IFP
      real F(NFP)
      real P(NT,NX), Q(NT,NX)

C  Initialize P and Q
      FORALL(J=1:NT, I=1:NX) P(J,I) = J
      Q = 0.0

C  Initialize F
      do I = 1, NFP
         F(I) = I
      end do


C  Shift array P by one over the first dimension
C  Compute Q after the shift on P

      do IFP = 1, NFP
        P(2:NT,:) = P(1:NT-1,:)
        P(1,:) = 0.0
        Q = Q + P * F(IFP)
      end do
```

FIGURE 2-3 CONVOLVE_F77 Example becomes CONVOLVE_F90

These changes allow the compiler to implement CONVOLVE_F90 using parallel operations, which are spread out across the available processes.

However, although this conversion from serial to parallel execution will yield significant performance gains by itself, you can also use the DISTRIBUTE directive to further assist the compiler in optimizing the program. This technique is described in the next section.

2.3 Using the DISTRIBUTE Directive

Assume that CONVOLVE_F90 will be compiled to run on eight processes. By default, the Sun HPF run-time system would distribute P, Q, and F across eight processes in the following manner:

Array, dimension	Distribution	Block Size
P, first	block	256
P, second	block	128
Q, first	block	256
Q, second	block	128
F	collapsed	Full array is replicated on all processes.

Without instructions to do otherwise, the compiler would distribute P and Q in (block, block) fashion onto a 2x4 process grid.

However, since the array assignment statement

```
P(2:NT,:) = P(1:NT-1,:)
```

in CONVOLVE_F90 shifts data along the first dimension of P while keeping the second dimension unchanged (e.g., copies each Ith row P(I,:) into the I+1st row P(I+1,:), and since the array assignment

```
P(1,:) = 0.0
```

operates only on the first row of P, it would be more efficient to distribute only the second axis of P across processes, while keeping the first axis of P local by giving it a collapsed distribution.

Thus, a (*,block) distribution would be more efficient for these operations than a (block, block) distribution. You can think of a (*,block)-distributed 2D array P as a collection of rows P(I,:), each of which is distributed in the same fashion across processes and each of which may be accessed independently of the others. Their resulting alignment allows corresponding elements to be copied between rows with maximal efficiency, like whole array operations on 1D arrays of the same shape and distribution.

This would make all operations on those elements local to the respective processes, thereby avoiding interprocess communication.

To tell the compiler that you want the first dimension of P to be collapsed, add the following directive to the declaration section of CONVOLVE_F90.

```
!HPF$ distribute (*, block) :: P
```

In this context, the * tells the compiler to collapse the first dimension of P, replicating it on every process. The block entry specifies block distribution for the second dimension of P.

Since you will also want P and Q to have the same distribution, add Q to the list of arrays to receive the (*,block) distribution. These additions are shown in FIGURE 2-4.

```
!hpf$ distribute (*, block) :: P,Q
```

Note – See Chapter 6 for additional discussion of the use of DISTRIBUTE, as well as other notes on enhancing the performance HPF programs.

```
 program CONVOLVE_HPF
      integer NX, NT, NFP
      parameter (NX=512, NT=512, NFP=16)
      integer I, IFP
      real F(NFP)
      real P(NT,NX), Q(NT,NX)
!hpf$ distribute (*, block) :: P,Q

C  Initialize P and Q using F90 assignments
      FORALL(J=1:NT, I=1:NX) P(J,I) = J
      Q = 0.0

C  Initialize F
      do IFP = 1, NFP
         F(I) = I
       end do

C  Shift array P by one over the first dimension
C  Compute Q after the shift on P

      do IFP = 1, NFP
        P(2:NT,:) = P(1:NT-1,:)
        P(1,:) = 0.0
        Q = Q + P * F(IFP)
      end do
```

FIGURE 2-4 Adding the DISTRIBUTE Directive to Produce CONVOLVE_HPF

CHAPTER 3

Compiling and Linking Sun HPF Programs

This chapter explains how to create Sun HPF executable (`a.out`) files using the compile/link command `hpf`.

The `hpf` command operates in much the same manner as the Sun command `f77`. Experienced users of `f77` will find many familiar features in `hpf`.

3.1 `hpf` Command Basics

3.1.1 Command Syntax

The `hpf` command syntax is:

`hpf [`*switches*`]` *filename* `...`

One or more optional Solaris-style switches can be specified to control `hpf` behavior. These switches are described in the `hpf` man page. Examples of their use are presented throughout this chapter. Each *filename* argument specifies a source or object file to be compiled or linked.

Switches and file names are separated by blank spaces. The command line

```
% hpf -tmprofile -o mill mill.hpf
```

passes the source file `mill.hpf` to the Sun HPF compiler and causes it to generate an executable file named `mill`, while instrumenting the executable file for use by the Prism performance analysis tool. If the `-o` switch were not used, the executable would be given the default name, `a.out`.

3.1.2 File Name Extensions

The source file's file name extension indicates the type of file it is, which tells `hpf` how to handle it. TABLE 3-1 identifies the file name extensions that `hpf` recognizes and summarizes the action taken for each.

TABLE 3-1 Sun HPF File Name Extensions

`.hpf`	Subset HPF source — invokes the Sun HPF compiler.
`.HPF`	Subset HPF source for `cpp` — invokes the preprocessor `cpp`, followed by the Sun HPF compiler.
`.fcm`	CM Fortran source — invokes the Sun HPF compiler (in CMF compatibility mode).
`.FCM`	CM Fortran source for `cpp` — invokes the preprocessor `cpp`, followed by the Sun HPF compiler.
`.s`	Sun assembler source — passes the source file directly to the `fbe` assembler.
`.S`	Sun assembler source for `cpp` — invokes the preprocessor `cpp` and then passes the `cpp` output to the `fbe` assembler.
`.f`	Sun Fortran source — invokes the `f77` compiler.
`.for`	Sun Fortran source — same as `.f`.
`.F`	Sun Fortran source for `cpp` — invokes the preprocessor `cpp`, followed by the `f77` compiler.
`.FOR`	Sun Fortran source for `cpp` — same as `.F`.
`.o`	Object file — passed to linker. These files may be from a previous invocation of `hpf`. They may also be the output of the `f77` compiler.
`.so`	Shared object file — passed to the linker.
`.a`	Object library file — passed to the linker.

3.1.3 Linking Libraries with `hpf`

At link time, `hpf` links into the executable all files with a `.o`, `.so`, or `.a` extension. The various libraries that may be linked into a Sun HPF program are listed below and discussed in the following subsections.

- HPF library
- `F77_LOCAL` interface
- S3L
- CM Fortran utilities

Note – The header files for these utilities are in `/opt/SUNWhpc/include`.

3.1.3.1 HPF Library

The HPF library extends F90 array intrinsics with support for several classes of parallel operations. It requires the entry `-lhpf` on the `hpf` link line. For example:

```
% hpf -o mill mill.hpf -lhpf
```

Program units that call HPF library procedures must either include header files for every HPF library function called by the program unit or include the HPF library's top-level include file, `hpflib.h`. For example, if a program unit has calls to the `iall` and `iany` reduction functions, it must include either

```
INCLUDE 'tmc/iall.h'
INCLUDE 'tmc/iany.h'
```

or

```
INCLUDE 'tmc/hpflib.h'
```

Note – The top-level HPF library include file (`hpflib.h`) is very large and will greatly increase compilation time. If possible, you should avoid including the full library header file.

TABLE 3-1 provides a complete list of the individual HPF include files.

3.1.3.2 F77_LOCAL Library

The `F77_LOCAL` subroutines provide an interface between Sun HPF applications and local `f77` message-passing code. To link the `F77_LOCAL` interface library you need to include `-lf77local` on the `hpf` link line. For example:

```
% hpf -o mill mill.hpf -lf77local
```

Sun HPF program units that use the `F77_LOCAL` interface must include the header file `tmhpflib.h`. Use the relative path:

```
INCLUDE 'tmc/tmhpflib.h'
```

```
! HPF Inquiry functions
    INCLUDE 'tmc/hpf_alignment.h'    INCLUDE 'tmc/hpf_distribuition.h'
    INCLUDE 'tmc/hpf_template.h'

! HPF Reduction functions
    INCLUDE 'tmc/iall.h'             INCLUDE 'tmc/iparity.h'
    INCLUDE 'tmc/iany.h'             INCLUDE 'tmc/parity.h'

! HPF Sort functions
    INCLUDE 'tmc/grade_up.h'         INCLUDE 'tmc/sort_up.h'
    INCLUDE 'tmc/grade_down.h'       INCLUDE 'tmc/sort_down.h'

! HPF Scatter functions
    INCLUDE 'tmc/all_scatter.h'      INCLUDE 'tmc/iparity_scatter.h'
    INCLUDE 'tmc/any_scatter.h'      INCLUDE 'tmc/maxval_scatter.h'
    INCLUDE 'tmc/copy_scatter.h'     INCLUDE 'tmc/minval_scatter.h'
    INCLUDE 'tmc/count_scatter.h'    INCLUDE 'tmc/parity_scatter.h'
    INCLUDE 'tmc/iall_scatter.h'     INCLUDE 'tmc/product_scatter.h'
    INCLUDE 'tmc/iany_scatter.h'     INCLUDE 'tmc/sum_scatter.h'

! HPF Prefix functions
    INCLUDE 'tmc/all_prefix.h'       INCLUDE 'tmc/iparity_prefix.h'
    INCLUDE 'tmc/any_prefix.h'       INCLUDE 'tmc/maxval_prefix.h'
    INCLUDE 'tmc/copy_prefix.h'      INCLUDE 'tmc/minval_prefix.h'
    INCLUDE 'tmc/count_prefix.h'     INCLUDE 'tmc/parity_prefix.h'
    INCLUDE 'tmc/iall_prefix.h'      INCLUDE 'tmc/product_prefix.h'
    INCLUDE 'tmc/iany_prefix.h'      INCLUDE 'tmc/sum_prefix.h'

! HPF Suffix functions
    INCLUDE 'tmc/all_suffix.h'       INCLUDE 'tmc/iparity_suffix.h'
    INCLUDE 'tmc/any_suffix.h'       INCLUDE 'tmc/maxval_suffix.h'
    INCLUDE 'tmc/copy_suffix.h'      INCLUDE 'tmc/minval_suffix.h'
    INCLUDE 'tmc/count_suffix.h'     INCLUDE 'tmc/parity_suffix.h'
    INCLUDE 'tmc/iall_suffix.h'      INCLUDE 'tmc/product_suffix.h'
    INCLUDE 'tmc/iany_suffix.h'      INCLUDE 'tmc/sum_suffix.h'
```

FIGURE 3-1 HPF Library Include Files.

3.1.3.3 S3L

S3L is a library of highly optimized mathematical functions that are used to manipulate and perform computations on parallel arrays. To use these functions, you need to include the entry `-ls3l` on the `hpf` link line. For example:

```
% hpf -o mill mill.hpf -ls3l
```

Program units that call S3L routines must include its header file, `s3l-hpf.h`. Use the relative path:

```
INCLUDE 's3l/s3l-hpf.h'
```

3.1.3.4 CM Fortran Library

`hpf` automatically links the CM Fortran (CMF) utility library when it is needed to provide back compatibility to CM Fortran code. Program units that call procedures from this library must include its header file, `CMF_defs.h`. Use the relative path:

```
INCLUDE 'cm/CMF_defs.h'
```

Note – Sun HPF does not support the use of CM Fortran Library's `CMF_FILE_` routines to perform serial file operations.

3.1.3.5 CM and TM Timer Routines

`hpf` supports calls to both `TM_timer_*` and `CM_timer_*` routines. Program units that implement these timing interfaces must include one of the `timer-fort.h` header files.

```
INCLUDE 'tmc/timer-fort.h'
```

or

```
INCLUDE 'cm/timer-fort.h'
```

The `TM_timer_*` and `CM_timer_*` routines are discussed more fully in Chapter 4.

3.2 `hpf` in Action

This section illustrates some of the most commonly used features of the `hpf` command. For experienced F77 users, most of these operations will be well known. For many, in fact, the chief lesson to be learned in the following pages is how closely the `hpf` command adheres to tradition.

TABLE 3-2 shows the main stages of an `hpf` compile/link sequence. The actual start and end points in the sequence depend on the source file's file name extension and on whether certain optional switches are used. The diagram includes a summary of its main points. These will also be covered in the following discussions.

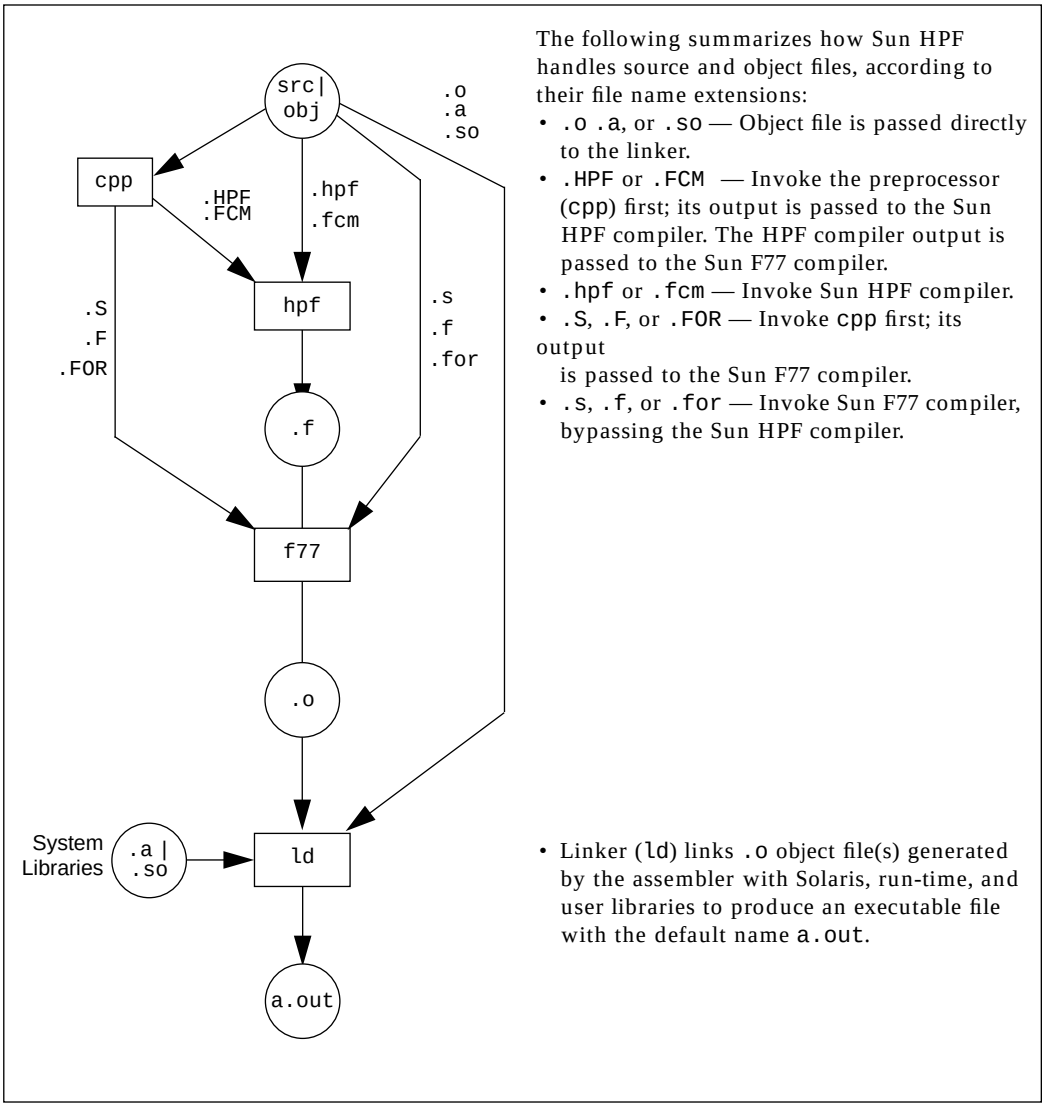


FIGURE 3-2 The `hpf` Compile/Link Sequence

3.2.1 Compiling and Linking

This section demonstrates compiling and linking under `hpf` with a simple example that uses two optional switches:

`-o` *filename*	Renames the executable output to *filename*.
`-l`*library*	Links the library *library*.

In the following example, two source files, `mill.hpf` and `grist.hpf`, are compiled and linked with the Sun Scientific Subroutine Library (S3L) to generate an executable output file named `flour`.

```
% hpf -o flour mill.hpf grist.hpf -ls3l
```

Because `hpf` is being passed `.hpf` source files, it invokes the Sun HPF compiler directly (without going through the `cpp` preprocessor). This compiler generates `mill.f` and `grist.f`, which are passed to the native `f77` compiler, which produces `mill.o` and `grist.o`.

In the final stage, the object files are linked with S3L, as well as with various other libraries, to produce an executable output file named `flour`. If the optional `-o` switch had not been used, the executable would have been given the default name `a.out`.

Use the `-temp=`*dir* switch to specify where the driver should create temporary intermediate files. This switch is useful if the default location `/tmp` has insufficient space.

3.2.2 Compiling CMF Programs into Sun HPF Executables

`hpf` accepts source files with `.fcm` and `.FCM` extensions, which means applications written in CM Fortran (CMF) can be compiled directly by `hpf`.

```
% hpf -o flour mill.fcm grist.fcm -ls3l
```

As can be seen in this example, `hpf` does not require any special command-line switches to handle a `.fcm` or `.FCM` source file. Simply specify the CM Fortran source file as you would a Sun HPF source file; `hpf` will understand from the file name extension how to proceed.

Note – When `hpf` is passed `.hpf` or `.HPF` source files that contain CM Fortran code, use the optional switch `-cmf_compatible`. This switch is on by default when compiling `.fcm` or `.FCM` source files.

3.2.3 Using the cpp Preprocessor

The hpf command driver accepts files with (uppercase) extensions of .HPF, .FCM, .F, .FOR, and .S and invokes the C language preprocessor cpp on each file before passing it on to the appropriate compiler. For files with a .S extension, the f77 compiler immediately invokes the assembler.

3.2.4 Compiling Incrementally

Four optional switches permit control over when the compilation sequence terminates.

-F	Preprocessor — Output from the preprocessor (cpp) is placed in a .hpf file and the sequence terminates. This switch has meaning only for source files with uppercase file name extensions (.HPF, .FCM, .F, or .FOR).
-F77	Fortran 77 — The Sun HPF compiler generates an f77 output file and terminates the compilation sequence.
-S	Assembler — The compilation sequence terminates after the f77 compiler generates a .s output file.
-c	Compilation only — An object file is generated for each source file, and then the compilation sequence terminates.

The -F77 switch permits developers of Sun HPF programs to quickly check for syntactic and semantic errors without waiting for a full compilation pass to complete. The -c switch is useful in development of large projects that involve many source files. The -F and -S switches are provided for advanced users who want the option of evaluating every intermediate stage of the compilation sequence.

3.2.5 Program Debugging and Profiling

hpf provides two debugging switches (-g and -gf77) and a profiling switch (-tmprofile).

Each debugging switch produces additional symbol table information for use by the Prism debugger. The -g switch supplies information at the HPF source level, while -gf77 provides information about the generated f77 source.

The -g switch suppresses certain optimization steps usually performed by the hpf compiler. Ordinarily, the compiler fuses multiple parallel statements together to increase execution speed. The -g switch puts each parallel statement in its own parallel code block so that debugger output can relate to specific lines of source code

more effectively. The `-g` switch is, of course, intended for use only during application development, since suppressing optimization steps can degrade program performance significantly,

---

Note – If you suspect a program error exists that the Prism debugger is unable to locate, try compiling with both `-g` and the run-time safety switch set to the highest safety level, `-safety=10`. Since run-time safety checking degrades performance even further, programmers usually avoid using it on the unoptimized code produced by `-g`. However, the combination of switches is sometimes useful in finding especially subtle bugs.

The `-tmprofile` switch causes performance analysis data to be generated, characterizing program behavior at multiple levels—across the entire program, procedure-by-procedure, and at individual source lines. The programmer can then examine this information within the Prism environment, looking for bottleneck conditions or imbalances in resource use.

Note – The `-g`, `-gf77`, and `-tmprofile` options are all incompatible with the optimization switches, `-O`*n* and `-xO`*n*.

3.2.6 Command-Line Arguments

You can write a routine to get command-line arguments by using operating system interfaces from the Sun F77 library—namely, the function `IARGC()` and subroutine `GETARG(I, NAME)`. FIGURE 3-3 shows an example of such a routine.

```
PROGRAM LOOP
IMPLICIT NONE
INTEGER N1, N2, N3, N4, IARGC
CHARACTER*80 ARGUMENT

IF (4 .NE. IARGC()) THEN
    PRINT *, 'Usage : LOOP N1 N2 N3 N4'
    STOP
END IF

CALL GETARG(1, ARGUMENT)
READ (UNIT=ARGUMENT, FMT='(i6)') N1

CALL GETARG(2, ARGUMENT)
READ (UNIT=ARGUMENT, FMT='(i6)') N2

CALL GETARG(3, ARGUMENT)
READ (UNIT=ARGUMENT, FMT='(i6)') N3

CALL GETARG(4, ARGUMENT)
READ (UNIT=ARGUMENT, FMT='(i6)') N4

PRINT *, IARGC(), N1, N2, N3, N4

STOP
END
```

FIGURE 3-3 Sample Command-Line Argument Routine

3.3 Summary of Optional Switches

TABLE 3-2 lists the switches that are available to control `hpf` behavior. Switches are applied from left to right. In most cases, if a particular switch is specified more than once on the command line, the rightmost setting or value is used.

Note – Abbreviation of compiler switches is not currently supported.

See the hpf man page for a more detailed description of the compiler command.

TABLE 3-2 Summary of hpf Switches.

–B*x*	Prefer dynamic or require static library linking. The default is dynamic.
–c	Compile only; do not make executable file.
–cmf_compatible	Interpret source code using CMF syntax rules (that is, accept CMF code file) and ignore any HPF directives.
–cmf_directives	Accept CMF directives embedded in .hpf or .HPF source code.
–common_initialized	Use when program units contain distributed arrays that are initialized via BLOCK_DATA.
continuations[=*number*]	Specify the maximum number of continuation lines in a statement.
–dalign	Double-align; allow double-word load/store.
–dbl_align_all=y	Force alignment of all data on 8-byte boundaries. Note: This switch is available only with f77 Version 4.2.
–depend	Analyze loops for data dependencies.
–d_lines	Accept lines beginning with D in column 1 as ordinary statements.
–double_precision	Interpret REAL declarations as DOUBLE PRECISION.
–dryrun	Show commands built by the driver but do not execute (debugging switch).
–e	Extend source line maximum length to 132 characters.
–F	Invoke the source file preprocessor but do not compile.
–f	Align on 8-byte boundaries.
–F77	Stop compilation after generating an intermediate .f file for each source file.
–fast	Select the combination of options that optimizes for speed.
–fixed	Expect fixed form source.
–flags	Synonym for –help.
–fnonstd	Initialize floating-point hardware to nonstandard preferences.
–fns	Select nonstandard floating point.
–free	Expect free form source.

TABLE 3-2 Summary of `hpf` Switches.

`–fround=r`	Select the IEEE rounding mode in effect at startup. Choices are: `nearest`, `tozero`, `negative`, `positive`.
`–fsimple[=`*n*`]`	Select floating-point optimization preferences.
`–ftrap=`*t*	Set floating-point trapping mode.
`–G`	Build a dynamic shared library.
`–g`	Support debugging in Prism at the HPF source level.
`–gf77`	Support debugging in Prism at the `f77` source level.
`–h` *name*	Specify the name of the generated dynamic shared library.
`–help`	Show a list of command-line options.
`–I`*dir*	Add *dir* to the include file search path.
`–implicit_none`	Inhibit implicit typing of variable, constant, and function names.
`–inline=`*rl*	Request inlining of the specified user-written routines, named in the list *rl*.
`–Kpic`	Synonym for `–pic`.
`–KPIC`	Synonym for `–PIC`.
`–L`*dir*	Add *dir* to the list of directories to search for libraries.
`–l`*x*	Link with the library `lib`*x*`.a` (or `lib`*x*`.so`).
`–libmil`	Inline selected `libm` math library for optimization.
`–misalign`	Allow for misaligned data in memory.
`–mt`	Link with multithread libraries.
`–native`	Optimize for the host system.
`–nodepend`	Turn off loop-dependence analysis.
`–nodir_warnings`	Suppress warnings about HPF or CMF directives.
`–nof77localsync`	Disable control synchronization around an `f77` local call.
`–nolib`	Do not link with system libraries.
`–nolibmil`	Cancel `–libmil` on the command line.
`–noqueue`	Disable license queueing.
`–norunpath`	Do not build a run-time library search path into the executable.
`–O[`*n*`]`	Specify optimization level.
`–o` *name*	Specify the name of the executable file to be written.

TABLE 3-2 Summary of `hpf` Switches.

`-p`	Compile for profiling with `prof`.
`-pg`	Compile for profiling with `gprof`.
`-pic`	Compile position-independent code for shared library.
`-PIC`	Similar to `-pic`, but with 32-bit addresses.
`-qp`	Synonym for `-p`.
`-Qoption` *pr ls*	Pass option list *ls* to the compilation phase *pr*.
`-R`*list*	Build library search paths into executable.
`-S`	Compile and only generate assembly code.
`-s`	Strip the symbol table from the executable file.
`-safety=`*number*	Generate code that performs some level of run-time checking.
`-silent`	Suppress compiler messages.
`-temp=`*dir*	Define directory for temporary files.
`-time`	Show execution time for each compilation phase.
`-tmprofile`	Build an executable that can be used with Prism to profile the code.
`-V`	Show name and version of each compilation phase.
`-v`	Verbose mode; show compilation details.
`-xarch=`*a*	Specify the target architecture instruction set.
`-xcache=`*c*	Define cache for optimizer.
`-xchip=`*c*	Specify target processor for optimizer.
`-xdepend`	Synonym for `-depend`.
`-xhelp=flags`	Show options summary.
`-xinline=`*rl*	Synonym for `-inline=`*rl*.
`-xlibmil`	Synonym for `-libmil`.
`-xlibmopt`	Use library of optimized math routines.
`-xlicinfo`	Show license server user IDs.
`-Xlist`	Produce listings and do global program checking.
`-XlistI`	Same as `-Xlist`, plus checks include files as well.
`-XlistL`	Listings only (no `xref`).
`-XlistX`	`xref` only (no listings).

TABLE 3-2 Summary of `hpf` Switches.

`–xnolib`	Synonym for `–nolib`.
`–xnolibmil`	Synonym for `–nolibmil`.
`–xnolibmopt`	Cancel `–xlibmopt`.
`–xO[`*n*`]`	Synonym for `–O[`*n*`]`.
`–xpg`	Synonym for `–pg`.
`–xregs=`*r*	Specify register usage.
`–xs`	Allow debugging by `dbx` without `.o` files.
`–xsafe=`*mem*	Assume no memory-based traps.
`–xspace`	Do not increase code size.
`–xtarget=`*t*	Specify system for optimization.
`–xtime`	Synonym for `–time`.
`–ztext`	Make no library with relocations.

CHAPTER 4

Program Development Tools

This chapter describes the support Sun HPF provides for timing program activity and for invoking the C language preprocessor, `cpp`.

`hpf` is also closely integrated with Sun Microsystems graphical programming and debugging environment, Prism. When `hpf` is invoked with `-g` (debugging switch) or `-tmprofile` (performance analysis switch), the resulting executable may be processed with Prism. See the Prism documentation set for information about

- Debugging
- Performance analysis
- Data visualization

In addition, various othe HPF utilities are managed with compiler switches; these are described in the `hpf` man page:

- Run-time safety checking (using the switch `-safety`)
- Listing the kinds of communication being generated (using the switch `-Xlist`)
- Locating symbols and line labels (using the switch `-XlistX`)

4.1 Timing a Program

A timing facility provided for use by Sun HPF programs allows you to determine how much time any part of a program takes to execute. This facility has the following features:

- A timer calculates total time the processing nodes were active.
- Multiple timers can be active at the same time.
- Timers can be nested. This allows you, for example, to start one timer that will time the entire program, while using other timers to determine how different parts of the program contribute to the overall time.

Note – The Sun HPF timing facility supports the CM timer call syntax to simplify the porting of CM Fortran programs to the Sun Ultra HPC System environment. While the interface to these routines has not changed, some CM timer behavior has, as described in this section.

You can have up to 64 timers running in a program. Individual timers are referenced by unsigned integers (from 0 to 63, inclusive) used as arguments to the timing instructions. Timing instructions affect only those timers whose numbers are used as arguments to the instruction.

To start timer 0, for example, put a call to the `TM_timer_start` routine in your program, with 0 as an argument:

```
TM_timer_start(0);
```

To stop a timer, call `TM_timer_stop` with the number of the timer specified. For example, to stop timer 0 enter

```
TM_timer_stop(0);
```

This function stops the timer and updates the values for total elapsed time and total node idle time being held by the timer. You can then restart timer 0 at a later point by calling `TM_timer_start` again. Timing starts at the values currently held in the timer. This is useful for measuring how much time is spent in a frequently called subroutine. The timer keeps track of the number of times it has been restarted.

You can start or stop other timers while timer 0 is running; each timer runs independently.

To get the results from any timer, call the `TM_timer_print` routine after you have called `TM_timer_stop`. For example, to print timer 0 results, enter

```
TM_timer_print(0);
```

`TM_timer_print` reports timing information to `stdout` using Solaris parameters, such as `user` and `system`. An example of this output is

```
TMRTS timer 0:  State is Stopped, 1 starts.
User time:           0.901456198
System time:         0.000267869
Trap time:           0.000492248
Elapsed time:        3.850932740
```

The various components of the `TM_timer_print` report are explained below.

`User time`	The time accounted to the user process in user mode between the last start/stop pair.
`System time`	The additional time accounted to system operations called via `syscall` on behalf of the user process between the last start/stop pair.
`Trap time`	The additional time accounted to other system operations (such as breakpoint traps) on behalf of the user process between the last start/stop pair.
`Elapsed time`	The amount of wallclock time elapsed between the last start/stop pair.

These routines return specific information from the timer for use in a program:

- `TM_timer_read_starts` returns an integer that represents the number of times the specified timer has been started.
- `TM_timer_read_elapsed` returns a double-precision value that represents the total elapsed time (in seconds) for the specified timer. *Elapsed time* refers to process time, not wallclock time.
- `TM_timer_read_run_state` returns 1 if and only if the specified timer is running. Otherwise, the routine returns 0.

If you use any of these `TM_timer_read_`*xxx* routines, include the file `tmc/timer-fort.h`.

In addition, `TM_timer_set_starts` takes a timer number and an integer as arguments. It sets the number of starts for the specified timer to the specified value. This is useful if you want to write a function that can query a running timer without changing the number of starts. Not changing the number of starts is important if you want to know how many times a large chunk of code was called, but you also want to get sub-timings within that block.

To clear the values maintained by a timer, call `TM_timer_clear`. For example, to clear the value maintained by timer 0, put this call in your program:

```
TM_timer_clear(0);
```

This zeroes the total elapsed time, the total node idle time, and the number of starts for this timer.

Note – CM Fortran users, versions of these timing routines, which begin with a `CM_` prefix, are available for back compatibility with CM Fortran. Include the header file `cm/timer-fort.h` at the beginning of any program unit that uses `CM_` routines.

4.1.1 Hints on Using the Timing Utility

The elapsed time reported by a timer includes time when the process is swapped out on a node. The more processes that are running, the more distorted this figure will be. Therefore, we recommend that you use nodes that are as unloaded as possible.

If you can't guarantee that you will have exclusive use of the nodes, try to run the process several times; the minimum elapsed time reported will be the most accurate.

In addition, we recommend that you avoid stopping a process that is being timed.

Note that the inclusion of calls to the timer functions can change the generated code somewhat and, therefore, can itself affect performance.

Finally, note that if you are using Prism to analyze the performance of a program that includes timer calls, Prism performance data will include the overhead assigned to these calls; thus, the elapsed time reported by Prism will be somewhat greater than the elapsed time reported by the timing routines.

Note – The `dbx` debugger is incompatible with programs that call `tm_timer`.

4.1.2 Synchronization Considerations

Some timer calls, such as `TM_timer_(clear,start,stop)`, do not synchronize the local processor with other processors in the partition—that is, they only take effect locally. This means they can be used more freely without concern for undesired nonlocal consequences. It also means, however, that application developers should include processor synchronization in their code at the beginning of a timing run. An example of this is shown below.

```
! clear the timers
 call TM_timer_clear(0)
 call TM_timer_clear(1)
! do some preliminary IO
 write(6,*)'data array:', (x(i,i=1,n)
! start timing
 dummy = TM_timer_read_elapsed(0)
 call TM_timer_start(0)
! loop
 do iter = 1, niter
! communications
 call TM_timer_start(1)
 call communications()
 call TM_timer_stop(1)
```

```
! computations
 call computations()
end do
! stop timing
 call TM_timer_stop(0)
! report timing
 write(6,*) 'overall time:'
 call TM_timer_print(0)
 write(6,*) 'computation time only:'
 call TM_timer_print(1)
```

Note that a dummy `TM_timer_read_elapsed()` call was introduced just before the timing was started. The results of this call are not of interest—only its synchronizing side effect is. If this call were not introduced, many of the processors could start their timers even though the partition as a whole would not be ready to start the run. In such a case, one of the processors would still be busy with the preliminary I/O. However, no unneeded synchronization is introduced within the loop because it would perturb the timing results.

4.2 The C Language Preprocessor

The Sun HPF command driver accepts files with (uppercase) extensions of `.HPF`, `.FCM`, `.F`, `.FOR`, and `.S` and invokes the C language preprocessor `cpp` on each file before passing it on to the appropriate compiler. For files with a `.S` extension, the f77 compiler immediately invokes the assembler.

The online man page for `cpp` describes the program switches and preprocessor command lines in detail, including a facility for defining macros with arguments.

4.2.1 Used with `-D`

The C preprocessor can provide a useful conditional compilation facility for Sun HPF or CM Fortran source code when used with the `hpf` command line switch `-D`, which is described in the `hpf` man page. For example, the following program contains preprocessor control lines that conditionally define a parameter `N`, which is used in the declaration of a matrix `A`.

```
      PROGRAM CPP
#if ASIZE > 0 && ASIZE < 10
      PARAMETER (N = ASIZE)
#else
      PARAMETER (N = 9)
```

```
#endif
      CHARACTER*10 FMT
      INTEGER A(N,N)
      A = 0
      FORALL (I = 1:N, J=1:N) A(I,J) = I*10 + J
      WRITE (FMT, 10) N
10    FORMAT( "(1X,", I2.2, "I3)" )
      PRINT FMT, TRANSPOSE(A)
      END
```

In this example, the preprocessor control lines (those beginning with the character #) test whether the value of the symbol ASIZE is in the range 1 to 9 and, if so, select the first PARAMETER statement for compilation, otherwise the second. The control lines themselves are filtered from the file that is passed to the compiler, along with the unselected PARAMETER statement. The value for ASIZE is substituted for all occurrences of the symbol ASIZE in the program; the value of symbol ASIZE can be defined in the source code, on the command line, or it can be left undefined (in which case it assumes the value zero). If the program is in the file cpp.HPF, then the command line

```
% hpf -DASIZE=7 cpp.HPF
```

causes the matrix A to be declared as a 7x7 array.

The following example, on the other hand, does not supply a value to the symbol ASIZE.

```
% hpf -DASIZE cpp.HPF
```

In this case, the value of ASIZE is assumed to be 1, causing the matrix to be declared as a 1x1 array. If no definition of ASIZE is supplied on the command line or in the source file, its value is taken as zero, and so the second PARAMETER statement is compiled, making it equal to the default value of 9.

CHAPTER 5

File Systems and File System I/O

5.1 Introduction

All Sun HPF I/O operations—both serial and parallel—are carried out through intrinsic Fortran I/O statements. Unformatted I/O statements are routed through the MPI I/O library.

The MPI I/O layer passes serial I/O requests to the Solaris operating environment and parallel I/O requests to PFS, Sun HPC's parallel file system facility. See the *Sun MPI User's Guide* for more information about the MPI I/O facility.

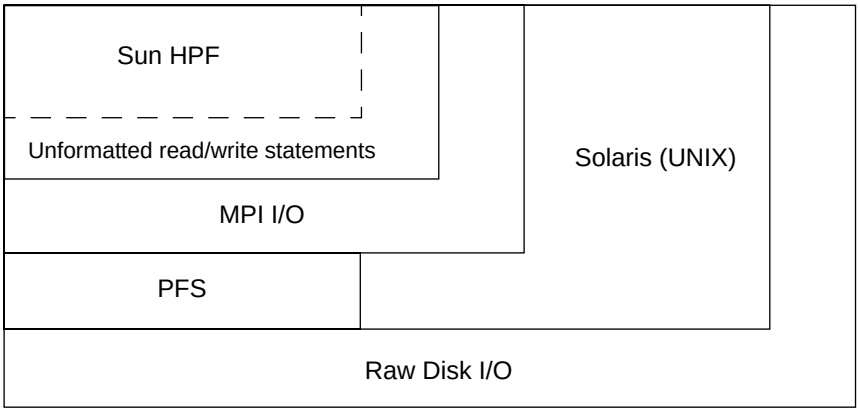


FIGURE 5-1 Sun HPF I/O Facilities.

PFS combines multiple disks and multiple I/O servers into a single, unified file system. Each PFS file is subdivided into blocks, and the blocks are distributed across all the disks in the file system. This arrangement provides multiple, independent I/O channels, which allow each PFS file to be read and written in multiple parallel streams.

From the programmer's perspective, PFS closely resembles Solaris file systems. It uses a conventional inverted-tree hierarchy, with a `root` directory at the top and subdirectories and files branching down from there. The fact that individual PFS files are distributed across multiple disks managed by multiple I/O servers is transparent to the programmer. How PFS files are actually mapped files to the physical storage facilities is implementation dependent and is based on file system configuration entries in the run-time environment (RTE) database.

5.2 PFS File Path Names

The most obvious difference between PFS and UNIX file systems is in file path name construction. Where a full UNIX path name begins with `/` (root), a PFS path name takes the form `pfs:`*filesystem*`:/`*pathname*.

- `pfs:` identifies this as a parallel file system—this prefix is required.
- *filesystem*: is the name of the file system—this is an arbitrary ASCII string assigned by the system administrator when configuring the file system. It is terminated by a `:` (colon). If there is only one Parallel File System or when your current working directory is contained in *filesystem*, this path name component is optional. If there are multiple Parallel File Systems and your current working directory is not on *filesystem*, the *filesystem*`:`component is required.
- The third term, */pathname*, is the file's Solaris-style path name.

For example, the PFS file `/users/clovis/paris` in the file system `cities` would be named `pfs:cities:/users/clovis/paris`.

Serial files follow Solaris naming conventions. If you wish, you can add the prefix `ufs:` to a serial file name as an explicit indicator of its serial nature. For example, `ufs:/payroll/march. ufs:` stands for UNIX file sSystem. The `ufs:` prefix is optional.

5.3 Programming Examples

Use standard `hpf` unformatted I/O statements to open, close, read, and write files. The only difference between parallel and serial operations is the form of the name specified in the OPEN statement. As described in Section 5.2, "PFS File Path Names," parallel file names include a `pfs:` prefix. If there are multiple parallel file systems, they also include the file system name. Serial file names may include the optional `ufs:` prefix.

A simple program example is presented below, illustrating the various I/O operations that might be requested by a Sun HPF program. Both serial and parallel file operations are represented.

```
      integer ap, as
      dimension ap(10), as(10)
!hpf$ distribute as(*)
!hpf$ distribute ap(:)

      forall (i=1:10) ap(i) = i
      do i=1,10
        as(i) = i
      end do

      ! This unit can be used for any unformatted I/O.
      ! All I/O on this unit will be serialized -- that is,
      ! it will be performed by a single process.
      open(1,file='filename',form='unformatted')

      ! This unit can be used for any formatted I/O.
      ! All I/O on this unit will be serialized, that is,
      ! will be performed by a single process.
      open(2,file='filename')

      ! This unit can only be used for unformatted I/O.
      ! I/O on this unit will be executed in parallel.
      open(3,file='pfs:filename',form='unformatted')
```

```
! All participating nodes send data to the
! appropriate positions of the file.
write(3) AP

! The data in AS is distributed across the physical
! processes upon which the HPF program is executing.
! The resulting parallel array is written to the
! PFS file as each participating node independently
! sends its data to the appropriate positions of the
! file.
write(3) AS

! This creates a distributed array consisting of one
! element. This case is included for illustration
! only. It is *not* a practical example and should
! not be implemented.
write(3) AS(1)
write(3) AP(1)

! This will serialize the data in AP -- that is, it
! will copy the entire array to sequential locations
! in the memory of a single processor, which
! processor will perform the formatting and the
! I/O.
write(2,*)AP

! This will serialize the data in AP -- that is, copy
! the entire array to sequential locations in the
! memory of a single processor, which processor will
! perform the unformatted I/O.
write(1,*)AP

! This should fail at run time, aborting the
! program. Currently, PFS files cannot be opened in
! formatted mode.
open(4,file='pfs:filename')
```

```
      ! This is illegal; a run–time error will occur,
      ! aborting the program. Formatted I/O cannot be
      ! performed on a unit that was opened for
      ! unformatted I/O.
      write(3,*)AP

C not reached

      close(1)
      close(2)
      close(3)
      close(4)
      stop

      end
```

CHAPTER 6

Performance Notes

This chapter contains advice on how to improve the performance of your Sun HPF applications. It also offers suggestions for better performance analysis—tips on how to gain better insight into your program's behavior and how to determine which areas of the code are the best candidates for improving performance.

6.1 Use Parallel Language Expressions

Parallel operations must be expressed in parallel syntax. Otherwise the compiler will treat them as serial operations, computing only one value at a time. Examples of using parallel syntax for array assignment statements and for intrinsic functions applied to arrays or array expressions.

6.1.1 Array Assignments

When you want an operation to assign values to multiple array elements in parallel, use the HPF language's array assignment constructs rather than a `DO` loop. For example, Sun HPF would implement the following serially, even if it were preceded by an `INDEPENDENT` directive.

```
do i=2,N1
   do j=2,N1
      mask(i,j)=.TRUE.
   endo
endo
```

To indicate that this operation should be parallelized, use either of the following expressions instead:

```
mask(2:N1,2:N1)=.TRUE.
```

or

```
forall (i=2:N1, j=2:N1) mask(i,j) = .TRUE.
```

The former expression will always be parallelized; the latter will be parallelized as long as MASK is a distributed array.

The Sun HPF compiler currently assumes that DO loops are reserved for operations that, for whatever reason, are best performed in a serial loop, one step at a time, rather than aggregated in a single parallel operation. Of course, array operations such as the parallel assignments to array MASK shown above, can themselves be included as steps in a serial loop.

Similarly, the following code

```
real a(N1,N1), P1
logical mask(N1,N1)
do i=1,N1
   do j=1,N1
     if (mask(i,j)) a(i,j) = P1 * a(i,j)
   end do
end do
```

can be parallelized if the array assignment is expressed in either of the following forms:

```
where (mask)  a = P1*a
```

or

```
forall (i=1:N1,j=1:N1, mask(i, j)) a(i,j) = P1 * a(i,j)
```

Note – Since explicit index variables are not necessary to describe the array assignments shown above, the first (simpler) replacements are somewhat preferable.

6.1.2 Intrinsic Functions Applied to Arrays or Array Expressions

All standard Fortran 77 numeric intrinsic functions can be applied in parallel to array expressions. These *elemental functions* act pointwise on corresponding elements of same-shape array or scalar arguments to produce an array of results of the same shape as any array arguments. For example, the following code

```
real a(N1,N1), b(N1,N1)
do i = 1,N1
   do j = 1,N1
      b(i,j) = cos(a(i,j))
   end do
end do
```

would be performed one element at a time, regardless of the number of processes available. The equivalent HPF array operation

```
b = cos(a)
```

would, however, be carried out in parallel on the parallel instances of `a` and `b`.

Other intrinsic functions applied to arrays or array expressions can be parallelized. For example, the following serial code for computing the maximum value of an array `a`,

```
M = a(1,1)
do i = 1,N1
   do j = 1,N1
      if (a(i,j) > M)  M = a(i,j)
   end do
end do
```

can be parallelized if it is rewritten in the following form

```
M = maxval(a)
```

6.2 Minimize Communication

The previous section advises the use of parallel language constructs wherever possible so as much of your program will be executed in parallel as possible. While the steps described in Section 6.1, “Use Parallel Language Expressions” are essential to realizing the benefits of parallel execution, by its nature, parallelism introduces the potential for interprocess communication.

Interprocess communication is often the largest barrier to high perfomance in a parallel application. This is particularly true for communication across a network. Interprocess communication within an SMP, while costly, is less expensive than communication over a network.

This means that minimizing communication should be a primary goal in developing parallel applications. When communication cannot be avoided, a secondary goal should be to keep the communication within SMPs as much as possible.

Note – If you compile with the `–Xlist` option, the compiler generates a list file that identifies, among othere things, which source lines invoke communication routines. This option is a key tool for understanding which parts of a program are most communication-intensive.

Subsequent sections discuss a few specific techniques for achieving this goal, such as using HPF data mapping directives to guide the compiler in its optimization task.

6.3 Be Explicit

Give the compiler as much explicit information about how to optimize your program as possible.

Like all optimizing compilers, Sun HPF follows a set of built-in rules for generating efficient code. Hard-coded rules cannot, however, be expected to generate optimal code for every possible code construction. It is therefore a good practice to supply the compiler with as much explicit guidance as possible. For example,

- Use the `DISTRIBUTE` directive to tell the compiler how to map parallel arrays.
- Avoid assumed-shape declarations.
- Avoid transcriptive mappings.
- Whenever it is practical, use parameters rather than variables.

- Use the compile-line switches to specify the desired optimization behavior.

Note – Refer to the *High Performance Fortran Language Specification*, Version 2.0, for detailed information on the language features referred to in this chapter. See the section “Related Publications” of the preface for information about accessing this document on the Internet.

6.4 Map Arrays Explicitly

One of the most important factors affecting the performance of data parallel programs is how efficiently a program’s arrays are mapped. In this context, *efficiency* means avoiding costly interprocess communication during array operations.

You can usually improve the performance of your HPF programs by explicitly mapping arrays in a way that maximizes data locality—that is, in a way that locates operationally related array elements in the same process.

Since default mapping choices made by the compiler may not be consistent across program units, it is preferable to use explicit directives to indicate how array variables should be distributed across processes. Generally this is done by using either the `DISTRIBUTE` directive (for parallel arrays) or the `SEQUENCE` directive (for non-parallel) arrays.

Note – The compile-time switch `-safety=n`, with *n*=`1` or higher, will generate run-time warnings when mappings are inconsistent across program units.

The default mapping for a parallel array chosen by the compiler will be block-distribution along all dimensions, which results in a minimal allocation of array elements per process. This is generally the preferred distribution for all whole array operations.

However, many special uses of array sections can be optimized with the `DISTRIBUTE` directive by making one or more (but not usually all) of the array's dimensions collapsed. The goal in these cases would be to improve locality with respect to other arrays or other sections of the same array. After initial parallelization, this is often the single most important optimization you can perform. Examples of how such mappings can improve efficiency are presented later in this section.

Other ways to customize mappings include controlling the shapes of abstract processor arrangements onto which you want arrays to be mapped. This is done using the `PROCESSORS` directive and an `ONTO` clause in a `DISTRIBUTE` directive.

Additionally, you can use `BLOCK(N)` for distributed dimensions. This more detailed version of `BLOCK` offers full control of what sets of elements are mapped to each process. However, it is often better to defer this more fine-grained control until you are tuning an application to run on a particular partition of a specific size or with known characteristics.

6.5 Avoid Passing Array Sections

It is possible to pass arbitrary array sections to subprograms as array arguments, but it is generally inefficient to do so unless the data can be passed in place. In the following example, Sun HPF would, by default, copy the data from each array section `A(1:2,:)`, `B(201:400)`, and `C(:,3)` into temporary arrays with default distributions before passing control to the subroutine being called.

```
real A(100,200), B(400), C(300,3)
call foo(A(1:2,:))
call bar(B(201:400))
call baz(C(:,3))
end
```

After returning from the subroutine, preparation would also be made for copying changed data back from the temporary array into the parent array (`A`, `B`, or `C`) in case the argument was altered by the subroutine. Thus, two potentially expensive communication operations will be generated for each call.

This extra communication results from Fortran arguments being interpreted by default as both input and output arguments, or `INTENT(INOUT)` arguments, in Fortran 90 language. One technique to reduce such communication is to inform the calling program when an argument is being used as an `INTENT(IN)` (input only) or an `INTENT(OUT)` (output only) argument. This is done with the aid of an explicit interface block for each routine, analogous to function prototypes in C.

For example:

```
real A(100,200), B(400), C(200,3)

interface
subroutine foo(X)
real, intent(in), dimension(200) :: X
end subroutine foo
end interface

interface
subroutine bar(X)
real, intent(out), dimension(200) :: X
end subroutine bar
end interface

interface
subroutine baz(X)
real, intent(inout), dimension(200) :: X
end subroutine baz
end interface

call foo(A(1:2,:))
call bar(B(201:400))
call baz(C(:,3))
end
```

With the explicit interfaces shown above, the array `A(1:2,:)` is being passed in as an input argument only, so no *copy out* communication operation will be generated by the compiler—that is, only one copy operation will be needed.

Similarly, since `B(201:400)` is being used exclusively as an output argument, no initial communication of data from that array section to a temporary array is required, and a *copy in* communication operation can be avoided.

However, as the array argument `C(:,3)` is used for both input and output, no reduction in communication results from the third interface block, and two communication operations will be generated for the call to `baz`.

In certain special cases, Sun HPF can pass an array section in place, with no communication required either on entry or exit. This occurs only when the array section is allocated as a contiguous block of memory across all processes, with all the block-distributed dimensions of the parent array passed in full.

These conditions can be met if only the right-most collapsed axes are restricted to single subscripts, or if one collapsed axis has a limited range of subscripts and all those to the right are further restricted to a single value.

In the following examples, all of the conditions for passing an array section in place are met. Note that all the collapsed dimensions must lie to the right of the block-distributed dimensions for this particular optimization.

```
    real C(200,3), D(200,100),E(40,50,10,3,3)
!hpf$ distribute C(block,*)
!hpf$ distribute D(block,*)
!hpf$ distribute E(block,block,*,*,*)
    call sub1(C(:,3))
    call sub2(D(:,1:50))
    call sub3(E(:,:,:,1:2,1))
    end
```

6.6 Operate on Whole Arrays

For distributed arrays, it is usually more efficient to operate on the entire array, rather than on array sections.

References to individual array elements, such as `p(3,1,1)`, implies dereferencing that is more complicated than in F77. It may also require communication, since all processes may not have copies of that element.

References to array sections may also be inefficient. This is because operations are performed on the full array even when only a portion of an array is referenced. For example, `p(2:4)= 0` will reference the entire array, even though only three elements are actually written.

However, array sections that restrict the full range of the array only along collapsed axes can typically be accessed without operating on the full array. Thus, operations such as the following:

```
C(:,K,:) = X
D(2:N1:2, :) = D(1:N1:2, :)
E(:,J,K1:K2) = F(:)
```

can be maximally efficient if the arrays `C`, `D`, `E`, and `F` are distributed as follows:

```
!hpf$ distribute C(block,*,block)
!hpf$ distribute D(*,block)
!hpf$ distribute E(block,*,*)
!hpf$ distribute F(block)
```

6.7 Ratio of Processes to Processors

When running Sun HPF programs, the number of processes should be less than or equal to the number of processors in the partition where you are running.

It is possible to run a program where the number of processes exceeds the number of available processors. This is done by invoking `tmrun` or `tmsub` with the `–W` switch, which causes the processes to *wrap*. When processes are allowed to wrap, more than one process may run on a given processor, essentially time-sharing that processor, with reduction of performance as a likely consequence.

There may be some circumstances where the performance losses due to process wrapping are acceptable, particularly when developing and debugging your program.

6.8 Avoid Expensive Operations

Note – The tips described in this section are not specific to Sun HPF programming. They apply in general to the performance-tuning of any Fortran program.

Some expensive operations can be replaced by more functional equivalents that are more efficient. Prime examples of this include

- *Square roots* – Replace square root operations when possible. For example, assuming `X` is not negative, replace

```
IF (X .GT. SQRT(Y)) THEN
```

  with

```
IF (X*X .GT. Y) THEN
```

- *Divides* – Replace divide-by operations when possible. For example, replace

```
XARRAY(:) = XARRAY(:) / SCALAR
```

with

```
XARRAY(:) = XARRAY(:) * (1 / SCALAR)
```

- *Modulo operations* – Replace modulo operations when possible. For example, replace

```
IX_EAST = MOD(IX + 1, NX)
```

with

```
IX_EAST = IX + 1; IF (IX_EAST .EQ. NX) IX_EAST = 0
```

6.9 Use S3L Functions

S3L is a thread-safe, parallel library of scalable routines that are widely used in scientific and engineering computing. These routines are optimized for execution on Sun HPC Systems and should be called from any Sun HPF program that involves

- Dense-matrix operations
- LU-factorization and LU-solve routines
- 1D, 2D, and 3D FFTs
- Parallel random number generators
- Matrix inversion
- Parallel sorting
- Parallel transpose
- Array copying

6.10 Use Simple Constructs

Sometimes a general construct can be replaced with a simpler one that may be more efficient. For example, replace

```
FORALL (I=1:M,J=1:N) X(I,J) = 0
```

with

```
X = 0
```

6.11 Avoid General Communications

Where possible, replace statements that indirectly imply communications with more specific constructs. For example, replace

```
FORALL (I=1:N,J=1:N) Y(I,J) = X(J,I)
```

with

```
Y = TRANSPOSE(X)
```

6.12 Compiler Switches

The compiler provides several switches that control its optimization behavior in various ways. The following combination is recommended for compiling most Sun HPF applications.

```
–fast –fsimple=2
```

Note – If no other architecture is specified via `-xtarget` or `-xarch`, then `-xarch=v8plusa` will be used by default.

These recommended performance options are discussed briefly below.

`-fast`	Optimize compilation using a selection of options. Select the combination of options that optimizes for speed of execution without excessive compilation time. This option provides close to the maximum performance for many realistic applications. For some critical routines, it may be better to try for more optimization with the `-fast -xO5` combination. If you do not specify an optimization level with `-fast`, the default is `-xO4`. This is a convenience option. and it chooses: - The `-native` hardware target. - The `-xO4` optimization level.. - The `-fsimple=1` option. - The `-dalign` option. - The `-xlibmopt` option. - The `-depend` option. - The `-fns` option. - The `-ftrap=%none` option.
`-fsimple[=`*n*`]`	Select floating-point optimization preferences. Allow the optimizer to make simplifying assumptions concerning floating-point arithmetic. If *n* is present, it must be `0`, `1`, or `2`. Set *n* to `2` to permit aggressive floating-point optimizations.
`-xarch=`*a*	Specify the target architecture instruction set. For Sun Ultra HPC Systems, this *must* be set to `xarch=v8plusa`.
`-xO[`*n*`]`	Use this switch to specify an optimization level for compilation. See the `hpf` man page for details. If `-xO` is used without specifying an optimization level *n* and `-fast` is not specified, level 3 optimization is selected by default.

6.13 Shared Memory Environment Variables

Communication between Sun HPF processes uses the Sun MPI message-passing library and a shared-memory facility. If your HPF program involves significant communication, its level of performance will depend to some extent on how much shared memory is allocated for its message passing requirements.

See Section 3.5.1 in the Sun MPI User's Guide for more information on allocation of MPI shared memory buffers.

6.14 Performance Analysis

This section offers tips on how to analyze your HPF program to find opportunities for improving performance.

6.14.1 Do Repeated Timing Runs

When timing sections of your program, run the program for many passes. The set of data points this will generate will be much more useful than the information you would have after a single run. There are many chance-related factors that can skew the results of a single run. Multiple timing runs provide more reliable results.

Note – Section 4.1, "Timing a Program" describes the Sun HPF timing facility.

6.14.2 Use `–Xlist` to Analyze Communication

The `–Xlist` switch causes the compiler to generate a list file with a `.lis` extension. This file contains, for each program unit, a section labeled `COMMUNICATION ROUTINES`, which identifies by source-line number where communication operations are invoked. This information can help you decide how to revise the program to reduce the communication overhead.

For example, if the file `foo.hpf` contains the following code,

```
      subroutine sub(p, q)
      real p(:), q(:)
!hpf$ distribute (block) :: p, q
      p = q
      end
```

executing the command

```
% hpf -c -Xlist foo.hpf
```

will produce the list file `foo.lis`, which will contain

```
COMMUNICATION ROUTINES
    Name                  Line Number (number of times)
    BLOCKMOVE             4
```

This shows that the array assignment `p = q` involves a communication operation. Because the arrays were made assumed-shape, the compiler had less explicit information about the arrays than it needed to generate maximally efficient code. The `BLOCKMOVE` operation could have been avoided by giving the arrays `p` and `q` the same explicit shapes.

6.14.3 Examine the `.f` File (for advanced users)

Since the compiler effectively generates F77 nodal code with message-passing calls, advanced users may wish to examine the F77 source code by compiling with `–F77`. This shows how the compiler is handling your Sun HPF source code.

This intermediate output can be difficult to read. For this reason you may want to isolate the HPF code fragments of interest and compile only those isolated subroutines with `–F77`.

6.14.4 Profile the Code

The Prism graphical programming and debugging environment provides a powerful array of tools for profiling and debugging HPF programs. To use Prism's profiling capabilities, compile with the `–tmprofile` switch. This instruments the code with

timer operations. To generate profiling information, you must then execute the code under Prism with performance data collection turned on. For example, load Prism and your executable code as follows:,

```
% prism -np n a.out
```

In the resulting graphical user interface generated by Prism, pull down the Performance menu and click on Collection to turn on the collection tool. This enables Prism to collect profiling information as `a.out` executes. Then click on Run on the Execute menu to run `a.out`.

You can then use Prism's various graphical display and analysis tools to study the profiling data. For example:

- Select Display Data on the Performance menu.
- In the resulting performance window, select Sort By —> Time from the Options menu.

This will sort the performance data, listing the most time-consuming resources and program units first.

Note – Optimized code cannot be profiled.

Refer to the *Prism 5.0 User's Guide* and *Prism 5.0 Reference Manual* for complete instructions on using Prism.

CHAPTER 7

Sun HPF Summary

Sun HPF 1.0 compiler is an implementation of the Subset HPF language.

Sun HPF supports CM Fortran (CMF) backward compatibility with a switchable CMF mode. This mode allows the processing of both CMF layout directives and CMF-specific syntax, such as `DO TIMES` and nested `WHERE`s.

Sun HPF also provides an `EXTRINSIC(F77_LOCAL)` interface that allows a global Sun HPF program to contain node-level programming sections.

Sun HPF supports the Prism debugging and data visualization tool. In addition, the output code can be instrumented for performance analysis with Prism.

Note – Subset HPF is defined in Annex C of the High Performance Fortran Language Specification, Version 2.0. For convenience, the relevant portion of the HPF 2.0 document is reproduced (almost verbatim) in Section 7.1, “Fortran 90 Features in Sun HPF.” and Section 7.1, “Fortran 90 Features in Sun HPF.” See the section “Related Publications” of the preface for information about accessing this document on the Internet.

7.1 Fortran 90 Features in Sun HPF

The F90 features that are contained in the Subset HPF language are listed below. For cross-referencing convenience, the section numbers from the Fortran 90 standard are given along with the related syntax rule numbers.

- All Fortran 77 standard conforming features, except for storage and sequence association.
- The Fortran 90 definitions of MIL-STD-1753 features:
  - `DO WHILE` statement (8.1.4.1.1 / R821)

- END DO statement (8.1.4.1.1 / R825)
- IMPLICIT NONE statement (5.3 / R540)
- INCLUDE line (3.4)
- Scalar bit manipulation intrinsic procedures: IOR, IAND, NOT IEOR, ISHFT, ISHFTC, BTEST, IBSET, IBCLR, IBITS, MVBITS (13.13)
- Binary, octal and hexadecimal constants for use in DATA statements (4.3.1.1 / R407 and 5.2.9 / R533)

- Arithmetic and logical array features:
  - Array sections (6.2.2.3 / R618–621)

    - Subscript triplet notation (6.2.2.3.1)

    - Vector-valued subscripts (6.2.2.3.2)
  - Array constructors limited to one level of implied DO (4.5 / R431)
  - Arithmetic and logical operations on whole arrays and array sections (2.4.5 and 7.1)
  - Array assignment (2.4.5, 7.5, 7.5.1.4, and 7.5.1.5)
  - Masked array assignment (7.5.3)

    - WHERE statement (7.5.3 / R738)

    - Block WHERE . . . ELSEWHERE construct (7.5.3 / R739)
  - Array-valued external functions (12.5.2.2)
  - Automatic arrays (5.1.2.4.1)
  - ALLOCATABLE arrays and the ALLOCATE and DEALLOCATE statements (5.1.2.4.3, 6.3.1 / R622 and 6.3.3 / R631)
  - Assumed-shape arrays (5.1.2.4.2 / R516)
- Intrinsic procedures:

  The list of intrinsic functions and subroutines presented below is a combination of two types of routines: those that are entirely new to Fortran and routines that have always been part of Fortran, but have now been extended to new argument and result types.

  The new or extended definitions of these routines are part of the subset. If a Fortran 77 routine is not included in this list, only the original Fortran 77 definition is part of the subset.

  For all intrinsics that have the optional argument DIM, only actual argument expressions for DIM that are initialization expressions and therefore deliver a known shape at compile time are part of the subset. Intrinsics with this constraint are marked with ** in the list below.
  - The argument presence inquiry function: PRESENT (13.10.1)

- All the numeric elemental functions: ABS, AIMAG, AINT, ANINT, CEILING, CMPLX, CONJG, DBLE, DIM, DPROD, FLOOR, INT, MAX, MIN, MOD, MODULO, NINT, REAL, SIGN (13.10.2)
- All mathematical elemental functions: ACOS, ASIN, ATAN, ATAN2, COS, COSH, EXP, LOG, LOG10, SIN, SINH, SQRT, TAN, TANH (13.10.3)
- All bit manipulation elemental functions: BTEST, IAND, IBCLR, IBITS, IBSET, IEOR, IOR, ISHFT, ISHFTC, NOT (13.10.10)
- All vector and matrix multiply functions: DOT_PRODUCT, MATMUL (13.10.13)
- All array reduction functions: ALL**, ANY**, COUNT**, MAXVAL**, MINVAL**, PRODUCT**, SUM** (13.10.14)
- All array inquiry functions: ALLOCATED, LBOUND**, SHAPE, SIZE**, UBOUND** (13.10.15)
- All array construction functions: MERGE, PACK, SPREAD**, UNPACK (13.10.16)
- The array reshape function: RESHAPE (13.10.17)
- All array manipulation functions: CSHIFT**, EOSHIFT**, TRANSPOSE (13.10.18)
- All array location functions: MAXLOC**, MINLOC** (13.10.19)
- All intrinsic subroutines: DATE_AND_TIME, MVBITS, RANDOM_NUMBER, RANDOM_SEED, SYSTEM_CLOCK (3.11)

- Declarations:
  - Type declaration statements, with all forms of *type–spec* except: *kind–selector* and TYPE(*type–name*), and all forms of *attr–spec* except *access–spec*, TARGET, and POINTER. (5.1 / R501-503, R510)
  - Attribute specification statements: ALLOCATABLE, INTENT, OPTIONAL, PARAMETER, SAVE (5.2)
- Procedure features:
  - INTERFACE blocks with no *generic–spec* or *module–procedure–stmt* (12.3.2.1)
  - Optional arguments (5.2.2)
  - Keyword argument passing (12.4.1 /R1212)
- Syntax improvements:
  - Long (31 character) names (3.2.2)
  - Lowercase letters (3.1.7)
  - Use of _ in names (3.1.3)
  - ! initiated comments, both full line and trailing (3.3.2.1)

7.2 HPF Directives and Language Extensions

7.2.1 Summary of HPF 1.1 Subset

The following directives and language extensions to Fortran 90 were incorporated into the High Performance Fortran, Version 1.1 Subset. The Sun HPF compiler recognized and parsed by the Sun HPF compiler:

- The static data distribution and alignment directives: `ALIGN`, `DISTRIBUTE`, `PROCESSORS`, and `TEMPLATE`
- The *forall–statement* (but not the *forall–construct*)
- The `INDEPENDENT` directive
- The `SEQUENCE` and `NO SEQUENCE` directives
- The system inquiry intrinsic functions `NUMBER _OF_PROCESSORS` and `PROCESSORS_SHAPE`
- The computational intrinsic functions `ILEN`, and the HPF extended Fortran intrinsics `MAXLOC` and `MINLOC`, with the restriction that any actual argument expression corresponding to an optional *DIM* argument must be an initialization expression

7.2.2 Sun HPF Restrictions

Note that, while the Sun HPF compiler treats all these HPF 1.1 Subset features as valid source-file code elements, it does not implement the following functionality:

- Only `BLOCK` and collapsed distributions are implemented, not cyclic or block-cyclic distributions. The compiler converts these other distributions to an appropriate `BLOCK` distribution.
- Complex alignments (transposed, collapsed, or replicated axes, and arbitrary linear alignments).
- Prescriptively mapped dummy arguments without an explicit interface. If mapping of a dummy argument (either prescriptive or descriptive) in a subprogram differs from that of the actual argument, the Sun HPF programmer must supply an explicit interface for the data to be correctly remapped at the procedure boundary. Since such dummy arguments are typically declared as assumed-shape arrays, Fortran 90 rules require an explicit interface in most, but not all, such cases.

(This is a limitation relative to HPF 1.1 only; Sun HPF is consistent with HPF 2.0 on this issue.)

- The array location functions, `MINLOC` and `MAXLOC`, do not support the optional *DIM* argument.

7.3 Additional Fortran 90 Features

Sun HPF supports the Fortran 90 features from Subset HPF. The following additional standard Fortran 90 features are supported as well. (Fortran 90 references are included.)

- `CYCLE`, `EXIT` (8.1.4.4 / R834,R835)
- The construct for integer cases, `SELECT CASE` ... `CASE` ... `END SELECT` (8.1.3.1 / R808-811)
- Construct names for:
  - `IF` (8.1.2.1 / R802-R806)
  - `SELECT CASE` (8.1.3.1 / R808-811)
  - `DO`, `EXIT`, `CYCLE` (8.1.4.1–8.1.4.4.4 / R819-825)
- Named `END` statements:
  - `END PROGRAM` *program-name* (11.1 / R1103)
  - `END FUNCTION` *function-name* (12.5.2.2 / R1218)
  - `END SUBROUTINE` *subroutine-name* (12.5.2.3 / R1222)
- List-directed I/O to internal files
- `KIND` support, `KIND=` in declarations and typed constants for real and integer values (4.3.1.1–2, 5.1)
- The functions `SELECTED_INT_KIND` and `SELECTED_REAL_KIND` (13.10.6).
- Numeric inquiry functions: `DIGITS`, `EPSILON`, `HUGE`, `MAXEXPONENT`, `MINEXPONENT`, `PRECISION`, `RADIX`, `RANGE`, `TINY` (13.10.8)
- The function `BIT_SIZE` (13.13.16)
- The following specifiers on `OPEN` and `INQUIRE: ACTION=`, `POSITION=`, `DELM=`, `PAD=` (9.3.4.*x*, 9.6.1.*x*)
- Free source form
- `NAMELIST` I/O

Note, however, that Sun HPF does not support modules or derived types.

Sun HPF supports CMF features that are not part of standard Fortran 90 in back-compatibility mode:

- `DO (`*N*`) TIMES`
- Nested `WHERE` statements
- The intrinsic functions `DIAGONAL`, `REPLICATE`, `FIRSTLOC`, `LASTLOC`, `PROJECT`
- CMF-type array constructors (repeat factors and triplet-style sequences of integers)

Note – Triplet-style array constructors are also incorporated into the default (that is, HPF) mode.

Sun HPF provides two compile-time switches for back compatibility with CMF code:

- `-cmf_compatible` enables the compiler to accept CMF syntax, including CMF directives.
- `-cmf_directives` enables the compiler to accept CMF directives in place of HPF directives so that these may be used together with standard HPF syntax for nondirectives. Note that CMF and HPF directives may not be mixed in the same program unit.

Sun HPF supports some extensions to standard Fortran 90 that are in common use:

- The `DOUBLE COMPLEX` data type, and the `DCMPLX` type conversion function.
- The alternate method indicating `KIND` via the `*N` type suffix, as in `REAL*8`.
- The floating-point type-checking function `ISNAN`

CHAPTER 8

F77_LOCAL Interface

8.1 Introduction

F77_LOCAL is a global/local interface that allows global HPF programs to call local `f77` subroutines. It can be used to perform operations such as system calls on just a subset of the processors involved in a computation, or to combine message-passing code with HPF data-parallel code.

The F77_LOCAL interface follows the same HPF rules that apply to all `EXTRINSIC` subroutine calls. In particular, local subroutines must be declared as `EXTRINSIC (F77_LOCAL)` subroutines in HPF `INTERFACE` blocks. They must also follow other HPF guidelines for local subroutines—such as those prescribed for control flow and data layout.

Message passing via Sun MPI is supported for `F77_LOCAL` subroutines. However, no `MPI_Init()` or `MPI_Finalize()` calls should appear in `F77_LOCAL` subroutines.

All data sharing between global Sun HPF and F77_LOCAL program units is performed through argument passing. No `COMMON` blocks may be shared between global and local routines. Each instance of an F77 subroutine sees a local slice of the parallel data passed through the F77_LOCAL interface.

8.2 Processor Synchronization

Any HPF call to extrinsic, local subroutines should behave as though all processors were synchronized before entry to and after exit from the extrinsic procedure. The `–nof77localsync` compilation switch disables explicit synchronization before and after an `f77` local call. Synchronization around these calls is enabled by default.

8.3 Linking for F77_LOCAL

If your Sun HPF program uses the F77_LOCAL interface, link with `–lf77local`. See Chapter 3 for a discussion of compile/link switches.

8.4 Debugging F77_LOCAL Code with Prism

Code that uses F77_LOCAL subroutines may still be debugged with Prism, which will switch between data parallel mode and message-passing mode as appropriate when crossing the F77_LOCAL boundary. When F77 subroutines are compiled using Sun's F77 compiler directly, rather than using the HPF driver, the `–xs` flag must be used in conjunction with `-g` to make sure that the local code is debuggable by Prism. Compiling local code by using theSun HPF driver is recommended.

8.5 Argument Passing

Nondistributed arguments, such as `SEQUENCE` arrays, arrays with only collapsed axes, and scalars, are always passed by reference.

To determine how a distributed-array argument will be passed to an F77_LOCAL subroutine it is necessary to give the argument a `MAP_TO()` attribute, by using a `!TMHPF$` directive in the subroutine's interface block. The syntax for this attribute resembles the syntax used in `DISTRIBUTE`, `ALIGN`, etc. It may appear only in `EXTRINSIC(F77_LOCAL)` declarations in `INTERFACE` blocks. For example:

```
      INTERFACE
          EXTRINSIC (F77_LOCAL) SUBROUTINE SUB(X,Y,Z,W)
          REAL, DIMENSION(:) :: X, Y, Z, W
!HPF$ DISTRIBUTE (BLOCK) :: X, Y, Z, W
!TMHPF$ MAP_TO X(       F77_ARRAY)
!TMHPF$ MAP_TO Y(LAYOUT=NO_CHANGE)
!TMHPF$ MAP_TO  (       HPF_ARRAY) :: Z, W
          END SUBROUTINE SUB
      END INTERFACE
```

The three ways of passing distributed-array arguments are:

- `MAP_TO([LAYOUT=]F77_ARRAY)`

  In the HPF model for local subroutines, each physical processor contains a subset of array elements that can be locally arranged in a rectangular configuration. If `MAP_TO(F77_ARRAY)` is used, this rectangular configuration of these elements—*and these elements only*—is locally arranged to have Fortran 77 sequence association, and this data is passed by reference to the local subroutine. If no `MAP_TO()` attribute is specified, then `MAP_TO(F77_ARRAY)` is assumed.

  In short, `MAP_TO(F77_ARRAY)` causes the local subroutine to see the local slice of the HPF array as if it were an F77 array of the same rank.

- `MAP_TO([LAYOUT=]NO_CHANGE)`

  In this case, the local data corresponding to the global array is passed by reference to the local Fortran 77 subroutine. There is no local rearrangement of the data. The local programmer must know how the global HPF compiler stores array elements in memory.

  In particular, the subgrid enquiry utilities described in Section 8.7, “Subgrid-Inquiry Utilities,” must be used by the local subroutine to find the local data.

- `MAP_TO([LAYOUT=]HPF_ARRAY)`

  In this case, an array descriptor for the global HPF array is passed by reference to the local subroutine. The local programmer may not directly manipulate the array descriptor or the array data via this mechanism but may only pass the descriptor on to HPF-style utilities described in Section 8.6, “HPF-Style Utilities.” In order to access array data and to use HPF-style utilities, the programmer must pass the array in question to the `F77_LOCAL` subroutine twice: once using `MAP_TO(HPF_ARRAY)`, and once using either `MAP_TO(F77_ARRAY)` or `MAP_TO(NO_CHANGE)`.

8.6 HPF-Style Utilities

A global HPF program can always call standard HPF array-inquiry routines and intrinsics. At the local level, Fortran 77 versions of particular HPF intrinsics and `HPF_LOCAL_LIBRARY` routines are supported. Since these routines must be called from Fortran 77 and not HPF, the following distinctions between the two approaches must be understood:

- The prefix `F77_` is prepended to the utility name.
- All arguments are required. None are optional. The effect of omitting an optional `DIM` or `PROC` argument may be attained by specifying a value of –1 for that argument.
- Arguments are Fortran 77 sequence associated.
- The utilities are subroutines—none are functions. When the HPF analogs are functions, the `F77_` counterparts return values in a new argument that is prepended to the argument list.
- Arguments corresponding to global arrays should use dummy arguments passed in from the HPF caller by `MAP_TO(HPF_ARRAY)`.

Otherwise, these routines should behave the same as their counterparts, without the `F77_` prefaces, would in `HPF`. The set of locally supported Fortran 77 HPF-style routines consists of the following:

```
F77_GLOBAL_ALIGNMENT, F77_GLOBAL_DISTRIBUTION, F77_GLOBAL_TEMPLATE

F77_ABSTRACT_TO_PHYSICAL, F77_PHYSICAL_TO_ABSTRACT,
F77_LOCAL_TO_GLOBAL, F77_GLOBAL_TO_LOCAL

F77_LOCAL_BLKCNT, F77_LOCAL_LINDEX, F77_LOCAL_UINDEX

F77_GLOBAL_SHAPE, F77_GLOBAL_SIZE

F77_SHAPE, F77_SIZE

F77_MY_PROCESSOR
```

See Annex G, “The FORTRAN 77 Local Library,” in the *High Performance Fortran Language Specification*, Version 2.0, for detailed descriptions of these subroutines. See the section “Related Publications” of the preface for information about accessing this document on the Internet.

8.7 Subgrid-Inquiry Utilities

The subgrid-inquiry utilities provide information about each processor's subgrid of a distributed array. This information may be obtained in the HPF code using TMHPF_SUBGRID_INFO and passed to the F77_LOCAL code, or obtained in the F77_LOCAL code using F77_SUBGRID_INFO. The former approach makes it easier to declare the subgrids as arrays in the local code, while the latter approach keeps the HPF code from being cluttered with details of F77_LOCAL programming.To use the subgrid-inquiry utility, include the file tmc/tmhpflib.h.

8.7.1 TMHPF_SUBGRID_INFO

The format of the HPF subgrid-inquiry utility is

TMHPF_SUBGRID_INFO

(*ARRAY*,*IERR*,*DIM*,*LB*,*UB*,*STRIDE*,*LB_EMBED*,*UB_EMBED*,*AXIS_MAP*)

The arguments to TMHPF_SUBGRID_INFO are described in TABLE 8-1.

TABLE 8-1 TMHPF_SUBGRID_INFO Argument Descriptions

ARRAY	A distributed array of any type, size, or shape.
IERR	An error-status return variable, which is zero upon successful return and nonzero otherwise.
DIM	An optional argument indicating the axis along which the parameters are desired — if no axis is specified, parameters are returned for all axes.
LB, *UB*, *STRIDE*, *LB_EMBED*, *UB_EMBED*, *AXIS_MAP*	All optional, INTENT (OUT), integer arrays. 1. If a *DIM* argument is given, they are rank-one arrays of size *NUMBER_OF_PROCESSORS*() and BLOCK distribution. 2. If no *DIM* argument is given, each of them has a second, collapsed axis whose extent is at least the rank of the global array.

On any processor, *LB* and *UB* return the lower bound(s) and upper bound(s) of the array section that is mapped to the processor. Bounds are described in terms of the global indices of the HPF array, assuming such indices are one-based.

If MAP_TO(NO_CHANGE) is used to pass a distributed array to an F77_LOCAL subroutine, then the local programmer must know how the global compiler stores array elements in local memory. One model for local storage is that the local array

section is “embedded” in an array that is sequence associated in local memory. If *LB_EMBED* and *UB_EMBED* are specified, then they return the lower and upper bounds of the embedding array.

The *IERR* argument returns a nonzero value if an error is encountered. For example, it is an error if the *DIM* argument exceeds the rank of *ARRAY*, or if *LB_EMBED* or *UB_EMBED* are specified when local storage of the subgrid cannot be expressed in terms of an embedding array.

Note – The *STRIDE* and *AXIS_MAP* arguments return information that is not interesting for the range of distributions supported by this release of the compiler. They are included for possible future use.

8.7.2 F77_SUBGRID_INFO

The locally callable version of subgrid-inquiry routine is

F77_SUBGRID_INFO
(*ARRAY*,*IERR1*,*IERR2*,*DIM*,*LB*,*UB*,*STRIDE*,*LB_EMBED*,*UB_EMBED*,
AXIS_MAP)

This routine follows the conventions of the other F77_ routines except that two error arguments appear instead of the one that appears in the global version. At the local level, *IERR1* indicates the error status for *LB* and *UB* (and *STRIDE*) while *IERR2* indicates the error status for *LB_EMBED* and *UB_EMBED* (and *AXIS_MAP*).

8.8 Programming Examples

This section provides examples of two implementations of the F77_LOCAL interface in a Sun HPF program.

Each of these can be compiled and linked using

```
hpf –c global.hpf
hpf –c local.f
hpf global.o local.o –lf77local
```

8.8.1 Using MAP_TO(F77_ARRAY)

The first example uses the default MAP_TO(F77_ARRAY) attribute. It also shows the MAP_TO(HPF_ARRAY)attribute used for inquiry routines at the local level. We show the interface block separately in the file interface.hpf. The two local subroutines are declared EXTRINSIC(F77_LOCAL) and the MAP_TO attribute is used to describe how the arguments are to be passed. In the interface to local_sum we see that the default method of MAP_TO(F77_ARRAY) is also used.

The global program includes the interface block, sums the array in the usual HPF style, determines the extents of the local subgrids, and calls the two local subroutines.

interface.hpf

```
      interface
      extrinsic(f77_local) subroutine local_init
     &       (lb1, ub1, lb2, ub2, x, descrx)
        integer, dimension(:) :: lb1, ub1, lb2, ub2
        real x(:,:), descrx(:,:)
!hpf$   distribute(block) :: lb1, ub1, lb2, ub2
!hpf$   distribute(block,block) :: x, descrx
!tmhpf$ map_to(f77_array) :: x    ! not needed (default)
!tmhpf$ map_to(hpf_array) :: descrx
        end
      extrinsic(f77_local) subroutine local_sum(n,x,r)
        integer n(:)
        real x(:,:), r(:)
!hpf$   distribute n(block)
!hpf$   distribute x(block,block)
!hpf$   distribute r(block)
        end
      end interface
```

The first local subroutine, local_init, uses the distributed array's descriptor to get the global size of the first dimension, which is then used in the computation of the array elements. Notice that the array is declared as a two-dimensional local f77 array, and indexed as such.

The second subroutine, local_sum, treats the local array as a one-dimensional array, as is allowed with f77 arrays.

global.hpf	

```
      program global
      call example1()
      end program global

      subroutine example1()
      include 'tmc/tmhpflib.h'
! declare the data array and a verification copy
      integer, parameter :: nx = 24, ny = 24
      real, dimension(nx,ny) :: x, y
!hpf$ distribute(block,block) :: x, y
      real partial_sum(number_of_processors())
!hpf$ distribute partial_sum(block)

! local subgrid parameters are declared per processor
! for a rank–two array
      integer, dimension(number_of_processors(),2) ::
     & lb, ub, number
!hpf$ distribute(block,*) :: lb, ub, number

! define interfaces
      include 'interface.hpf'

! determine result using only global HPF
      ! initialize values
      forall (i=1:nx,j=1:ny) x(i,j) = i + (j–1) * nx
      ! determine and report global sum
      print *, 'global HPF result: ',sum(x)

! determine result using local subroutines
      ! initialize values
      call TMHPF_subgrid_info( y, ierr, lb=lb, ub=ub )
      if (ierr.ne.0) stop 'error!'
      call local_init(lb(:,1), ub(:,1), lb(:,2), ub(:,2),
     &               y, y)
      ! determine and report global sum
      number = ub – lb + 1
      call local_sum( number(:,1)*number(:,2),
     &               y, partial_sum)
      print *, 'F77_LOCAL result #1 : ',sum(partial_sum)
      end subroutine test1
```

local.f : local_init()	subroutine local_init(lb1, ub1, lb2, ub2, x, descrx) integer lb1, ub1, lb2, ub2 real x (lb1 : ub1, lb2 : ub2) integer descrx (*) ! get the global extent of the first axis ! HPF_LOCAL's GLOBAL_SIZE with an "F77_" prefix call F77_global_size (nx, descrx, 1) ! initialize elements of the array do j = lb2, ub2 do i = lb1, ub1 x(i,j) = i + (j–1) * nx end do end do end
local.f : local_sum()	subroutine local_sum(n, x, r) ! correspondence to the global indices is not important ! only the total size of the subgrid is passed in real x(n) r = 0. do i = 1, n r = r + x(i) end do end

8.8.2 Using MAP_TO(NO_CHANGE)

The second example illustrates an alternate way to perform the initialization part of the first example. It demonstrates use of the MAP_TO(NO_CHANGE) attribute as well as the addressing of data with respect to *embedding arrays*.

Notice that this time the call to HPF_SUBGRID_INFO obtains embedding array information, which is also passed to the local procedure, local_embedded. This information is used to declare the array in the local subroutine, but the actual extents are used for subgrid looping.

interface.hpf	```
	      interface
	      extrinsic(f77_local) subroutine local_embedded(
	     & lb1, ub1, lb_embed1, ub_embed1,
	     & lb2, ub2, lb_embed2, ub_embed2, x, descx)
	        integer, dimension(:) ::
	     & lb1, ub1, lb_embed1, ub_embed1,
	     & lb2, ub2, lb_embed2, ub_embed2
	        real, dimension(:,:) :: x, descx
	!hpf$ distribute (block) :: lb1, ub1, lb_embed1, ub_embed1
	!hpf$ distribute (block) :: lb2, ub2, lb_embed2, ub_embed2
	!hpf$ distribute (block,block) :: x, descx
	!tmhpf$ map_to x(no_change)
	!tmhpf$ map_to descx(hpf_array)
	        end
	      end interface
	``

global.hpf	

```
      program global
      call example2()
      end program global

      subroutine example2()
! This example performs only the initialization part of the
! first example. It illustrates use of the MAP_TO(NO_CHANGE)
! attribute and the addressing of data in terms of "embedding
! arrays."
      include 'tmc/tmhpflib.h'
      integer, parameter :: nx = 24, ny = 24
      real, dimension(nx,ny) :: y
!hpf$ distribute(block,block) :: y
! local subgrid parameters are declared per processor
! for a rank–two array
      integer, dimension(number_of_processors(),2) ::
     & lb, ub, lb_embed, ub_embed
!hpf$ distribute(block,*) :: lb, ub, lb_embed, ub_embed
! define interfaces
      include 'interface.hpf'
! initialize values
      call TMHPF_subgrid_info( y, ierr,
     &  lb=lb, lb_embed=lb_embed,
     &  ub=ub, ub_embed=ub_embed)
      if (ierr.ne.0) stop 'error!'
      call local_embedded(
     &  lb(:,1), ub(:,1), lb_embed(:,1), ub_embed(:,1),
     &  lb(:,2), ub(:,2), lb_embed(:,2), ub_embed(:,2),
     &  y, y )
      end subroutine example2
```

local.f	

```
      subroutine local_embedded(
     &  lb1, ub1, lb_embed1, ub_embed1,
     &  lb2, ub2, lb_embed2, ub_embed2,
     &  x, descx )
      integer lb1, ub1, lb_embed1, ub_embed1
      integer lb2, ub2, lb_embed2, ub_embed2
! the subgrid has been passed in its "embedded" form
      real x (lb_embed1 : ub_embed1, lb_embed2 : ub_embed2)
! we have also passed its descriptor
      integer descx(*)
! get the global extent of the first axis
! HPF_LOCAL's GLOBAL_SIZE with an "F77_"prefix
      call F77_global_size(nx,descx,1)
! otherwise, initialize elements of the array
! loop only over actual array elements
      do j = lb2, ub2
          do i = lb1, ub1
              x(i,j) = i + (j–1) * nx
          end do
      end do
      end
```

interface.hpf	

```
      interface
      extrinsic(f77_local) subroutine local_init
     &       (lb1, ub1, lb2, ub2, x, descrx)
        integer, dimension(:) :: lb1, ub1, lb2, ub2
        real x(:,:), descrx(:,:)
!hpf$   distribute(block) :: lb1, ub1, lb2, ub2
!hpf$   distribute(block,block) :: x, descrx
!tmhpf$ map_to(f77_array) :: x    ! not needed (default)
!tmhpf$ map_to(hpf_array) :: descrx
        end
      extrinsic(f77_local) subroutine local_sum(n,x,r)
        integer n(:)
        real x(:,:), r(:)
!hpf$   distribute n(block)
!hpf$   distribute x(block,block)
!hpf$   distribute r(block)
        end
      end interface
```

CHAPTER 9

HPF Intrinsic Functions and the HPF Library

9.1 HPF Intrinsic Functions

9.1.1 System Inquiry Intrinsic Functions

9.1.1.1 Integer Function PROCESSORS_SHAPE( )

This function generally returns a rank-one array of implementation-dependent size. Current implementation returns an array of one element equal to the value returned by NUMBER_OF_PROCESSORS (see below).

9.1.1.2 Integer Function NUMBER_OF_PROCESSORS(DIM)

The *DIM* argument is optional.The result will be the same as returned by CMF_NUMBER_OF_PROCESSORS. Also, any value of *DIM* other than 1 will be an error, since PROCESSORS_SHAPE( ) has extent one.

9.1.2 Elemental Intrinsics Function

`ILEN(`*I*`)`

Takes an integer (scalar or array) argument *I*, and returns a (scalar or conformable array) integer result representing the minimum number of bits to represent the given value as a 2's complement binary number.

9.1.3 Array Location Intrinsic Functions

`MAXLOC(`*ARRAY*`, `*DIM*`, `*MASK*`)`
`MINLOC(`*ARRAY*`, `*DIM*`, `*MASK*`)`

9.2 The HPF Library

Sun HPF supports the full HPF Library, which consists of the following functions and subroutines.

9.2.1 Bit Manipulation Functions

`LEADZ(`*I*`)`, `POPCNT(`*I*`)`, `POPPAR(`*I*`)`

9.2.2 Mapping Inquiry Subroutines

`HPF_ALIGNMENT(`*ALIGNEE, LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES*`)`

`HPF_TEMPLATE(`*ALIGNEE, TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED, DYNAMIC*`)`

`HPF_DISTRIBUTION(`*DISTRIBUTEE, AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE*`)`

All arguments but the first are optional.

9.2.3 Array Reduction Functions

In the following functions, *ARRAY* is an array of integer type; the arguments *DIM* and *MASK* are optional.

ALL(*ARRAY*,*DIM*,*MASK*) Bitwise AND reduction applied to integer data.
IANY(*ARRAY*,*DIM*,*MAS*) Bitwise OR reduction applied to integer data.
IPARITY(*ARRAY*,*DIM*,*MASK*) Bitwise EOR reduction applied to integer data.

In the following function, *MASK* is an array of logical type; the argument *DIM* is optional.

PARITY(*MASK*,*DIM*) Logical EOR reduction.

9.2.4 Array Combining Scatter Functions

In the following functions, all arguments are required.

FUNCTION *XXX*_SCATTTER(*MASK*, *BASE*, *INDX1*, ..., *INDXn*)
where *XXX* = ALL, ANY, COUNT, PARITY.

In the following functions, the argument *MASK* is optional.

FUNCTION *YYY*_SCATTTER(*ARRAY*, *BASE*, *INDX1*, ..., *INDXn*, *MASK*)
where *YYY* = COPY, IALL, IANY, IPARITY, MAXVAL, MINVAL, PRODUCT, or SUM.

These functions return arrays of the same data type, kind, and shape as input argument *BASE*. The number *n* of *INDX* arrays must equal the rank of array *BASE* and the result.

9.2.5 Array Prefix and Suffix Functions

In the following functions, the arguments *DIM* and *SEGMENT* are optional. They both return arrays of the same data type, kind, and shape as input argument *ARRAY*.

FUNCTION *XXX*_PREFIX/SUFFIX(*ARRAY*, *DIM*, *SEGMENT*)
where *XXX* = COPY.

In the following functions, the arguments *DIM*, *MASK*, *SEGMENT*, and *EXCLUSIVE* are optional. All return arrays of the same data type, kind, and shape as input argument *ARRAY*.

FUNCTION *YYY*_PREFIX/SUFFIX(*ARRAY*, *DIM*, *MASK*, *SEGMENT*, *EXCLUSIVE*)
where *YYY* = IALL, IANY, IPARITY, MAXVAL, MINVAL, PRODUCT, SUM.

In the following functions, the arguments *DIM*, *SEGMENT*, and *EXCLUSIVE* are optional. Except for *COUNT*, which returns an array of type `INTEGER`, these functions return arrays of type `LOGICAL`.

`FUNCTION` *ZZZ*`_PREFIX/SUFFIX(`*MASK*, *DIM*, *SEGMENT*, *EXCLUSIVE*`)`
where *ZZZ* = `ALL`, `ANY`, `COUNT`, or `PARITY`.

9.2.6 Array Ranking Functions

In the following functions, *ARRAY* must be of integer, real, or double-precision type; the argument *DIM* is optional.

`FUNCTION GRADE_UP(`*ARRAY*, *DIM*`)`
`FUNCTION GRADE_DOWN(`*ARRAY*, *DIM*`)`

When *DIM* is present, the result array is of the same type and shape as the argument *ARRAY*.

When *DIM* is not present, the result array is of the same type as the argument *ARRAY*, but is two-dimensional, with shape (`/`*r*`,`*s*`/`), where *r* is the rank of *ARRAY*, and *s* is the number of elements of *ARRAY*.

9.2.7 Array Sorting Functions

In the following functions, *ARRAY* must be of integer, real, or double-precision type; the argument *DIM* is optional.

`FUNCTION SORT_UP(`*ARRAY*, *DIM*`)`
`FUNCTION SORT_DOWN(`*ARRAY*, *DIM*`)`

These functions return an array containing the same elements as *ARRAY*, but reordered into increasing or decreasing numerical sequence either:

- Within sections parallel to the *DIM* axis
- Throughout the array, following array element order

The result array is of same type and same shape as the argument *ARRAY*.

9.3 HPF Library Exceptions and Other Notes

This section identifies a few Sun HPF exceptions to the HPF Library. It also indicates some related notes of interest.

- The HPF Library supports arrays of up to rank 7, the maximum supported by the HPF standard.
- The library subroutine HPF_TEMPLATE has three optional arguments for which we provide only default results: *LB*, *UB*, and *NUMBER_ALIGNED*. A warning will be given at run time if any of these outputs are requested.
- All of the routines currently expect array arguments of arbitrary size to be distributed parallel arrays rather than serial ones. The following exceptions currently apply to rank one array arguments that need to have at most seven elements for the mapping inquiry functions:
  - HPF_ALIGNMENT:

    Optional arguments *LB*, *UB*, *STRIDE*, and *AXIS_MAP* must be serial arrays.
  - HPF_DISTRIBUTION:

    Optional arguments *AXIS_TYPE*, *AXIS_INFO*, and *PROCESSORS_SHAPE* must be serial arrays.
  - HPF_TEMPLATE:

    Optional arguments *LB*, *UB*, *AXIS_TYPE*, and *AXIS_INFO* must be serial arrays.

  Additionally, the following routines require the indicated arguments to be distributed parallel arrays, where the current HPF standard allows any array or scalar arguments.
  - HPF_ALIGNMENT

    ALIGNEE must be a parallel array.
  - HPF_DISTRIBUTION

    DISTRIBUTEE must be a parallel array.
  - HPF_TEMPLATE

    ALIGNEE must be a parallel array.

For the {ALL/ANY/COPY/COUNT/IALL/IANY/IPARITY/MAXVAL/MINVAL/PARITY/PRODUCT/SUM}_SCATTER routines, the *INDX*(*I*) arguments must be parallel arrays.

CHAPTER 10

The HPF/CMF Utility Library

A supplementary library of compatible CM Fortran utility routines is supplied with the Sun HPF compiler as part of the compiler's support for CMF-legacy programs. The routines contained in this supplementary library are listed in TABLE 10-1.

You are encouraged to replace calls to these routines with equivalent Sun HPF code wherever possible. For example replace a call to `CMF_FE_ARRAY_TO_CM(A,B)` with the array assignment statement `A = B` for a parallel array `A` and a serial array `B`. In many cases, HPF library routines offer equivalent functionality to those in the CMF utility library.

TABLE 10-1 CMF Utility Library Routines Supported by Sun HPF

CMF Utility Library Routines	Equivalent Sun HPF Routine or Code
`cmf_allocate_table`	Table lookup utilities should be replaced with `FORALL` loops (see note).
`cmf_architecture`	Use a site-dependent substitute.
`cmf_aref_1d`	Table lookup utilities should be replaced with `FORALL` loops (see note).
`cmf_aset_1d`	Table lookup utilities should be replaced with `FORALL` loops (see note).
`cmf_cm_array_from_file`	Use Fortran unformatted reads and writes.
`cmf_cm_array_from_file_so`	Use Fortran unformatted reads and writes.
`cmf_cm_array_to_file`	Use Fortran unformatted reads and writes.
`cmf_cm_array_to_file_so`	Use Fortran unformatted reads and writes.

TABLE 10-1 CMF Utility Library Routines Supported by Sun HPF *(Continued) (Continued)*

CMF Utility Library Routines	Equivalent Sun HPF Routine or Code
`cmf_deallocate_table`	Table lookup utilities should be replaced with `FORALL` loops (see note).
`cmf_deposit_grid_coordinate`	First remove send addresses from the code and then use array indices.
`cmf_describe_array`	Use `HPF_DISTRIBUTION`, if possible.
`cmf_describe_array_geometry`	Use `HPF_DISTRIBUTION`, if possible.
`cmf_fe_array_from_cm`	`B = A`, where `B` is serial.
`cmf_fe_array_to_cm`	`B = A`, where `A` is serial.
`cmf_file_close`	Use Fortran `CLOSE`.
`cmf_file_lseek`	No known substitute.
`cmf_file_open`	Use Fortran `OPEN`, perhaps with `FORM=PHYSICAL`.
`cmf_file_open_readonly`	Use Fortran `OPEN`, perhaps with `FORM=PHYSICAL`.
`cmf_file_rewind`	Use Fortran `REWIND`.
`cmf_file_truncate`	No known substitute. Note: When truncating to nonzero, smaller `size may give error.`
`cmf_file_unlink`	Use Fortran `CLOSE` with `STATUS=DELETE`.
`cmf_lookup_in_table`	Table lookup utilities should be replaced with `FORALL` loops and replicated table (see note).
`cmf_make_send_address`	First remove send addresses from the code and then use array indices.
`cmf_my_send_address`	First remove send addresses from the code and then use array indices.
`cmf_order`	Use `GRADE_UP`, if possible.
`cmf_random`	Use `RANDOM_NUMBER` or S3L random number generator functions.

TABLE 10-1 CMF Utility Library Routines Supported by Sun HPF *(Continued) (Continued)*

CMF Utility Library Routines	Equivalent Sun HPF Routine or Code
`cmf_randomize`	Use `RANDOM_SEED` or S3L random number generator functions.
`cmf_rank`	No known substitute.
`cmf_scan_add`	Use `SUM_PREFIX` or `SUM_SUFFIX` if possible.
`cmf_scan_copy`	Use `COPY_PREFIX` or `COPY_SUFFIX` if possible.
`cmf_scan_iand`	Use `IALL_PREFIX` or `IALL_SUFFIX` if possible.
`cmf_scan_ieor`	Use `IPARITY_PREFIX` or `IPARITY_SUFFIX` if possible.
`cmf_scan_ior`	Use `IANY_PREFIX` or `IANY_SUFFIX` if possible.
`cmf_scan_max`	Use `MAXVAL_PREFIX` or `MAXVAL_SUFFIX` if possible.
`cmf_scan_min`	Use `MINVAL_PREFIX` or `MINVAL_SUFFIX` if possible.
`cmf_send_add`	First remove send addresses from the code and then use `SUM_SCATTER`.
`cmf_send_iand`	First remove send addresses from the code and then use `IALL_SCATTER`.
`cmf_send_ieor`	First remove send addresses from the code and then use `IPARITY_SCATTER`.
`cmf_send_ior`	First remove send addresses from the code and then use `IANY_SCATTER`.
`cmf_send_max`	First remove send addresses from the code and then use `MAXVAL_SCATTER`.

TABLE 10-1 CMF Utility Library Routines Supported by Sun HPF *(Continued) (Continued)*

CMF Utility Library Routines	Equivalent Sun HPF Routine or Code
`cmf_send_min`	First remove send addresses from the code and then use `MINVAL_SCATTER`.
`cmf_send_overwrite`	First remove send addresses from the code and then use `COPY_SCATTER`.
`cmf_sort`	Use S3L sort routine, if possible. Or `GRADE_UP` with code to permute the result, if possible.

Note – Replacing table lookup utilities with `FORALL` loops may not improve performance significantly, but does contribute to making the CMF program conform more closely to HPF specifications.

APPENDIX **A**

IOSTAT Message Summary

TABLE A-1 lists the Fortran I/O status messages.

TABLE A-1 Fortran I/O Status Message Summary

Error Number	Error Code	Decription
-1	FI_IOSTAT_ENDFIL	end of file
00	FI_IOSTAT_NOTERR	no error
01	FI_IOSTAT_NOTFORSPE	not FORTRAN-specific error
02	FI_IOSTAT_NOTIMP	not implemented
03	FI_IOSTAT_IGNORED	ignored requested disposition
04	FI_IOSTAT_IGNNOTDEL	ignored requested disposition, file not deleted.
17	FI_IOSTAT_SYNERRNAM	syntax error in NAMELIST input.
18	FI_IOSTAT_TOOMANVAL	too many values for NAMELIST variable
19	FI_IOSTAT_INVREFVAR	invalid reference to variable
20	FI_IOSTAT_REWERR	REWIND error
21	FI_IOSTAT_DUPFILSPE	duplicate file specifications
22	FI_IOSTAT_INPRECTOO	input record too long
23	FI_IOSTAT_BACERR	BACKSPACE error
24	FI_IOSTAT_ENDDURREA	end-of-file during read
25	FI_IOSTAT_RECNUMOUT	record number outside range
26	FI_IOSTAT_OPEDEFREQ	OPEN or DEFINE FILE required

TABLE A-1 Fortran I/O Status Message Summary *(Continued)*

27	FI_IOSTAT_TOOMANREC	too many records in I/O statement
28	FI_IOSTAT_CLOERR	close error
29	FI_IOSTAT_FILNOTFOU	file not found
30	FI_IOSTAT_OPEFAI	open failure
31	FI_IOSTAT_MIXFILACC	mixed file access modes
32	FI_IOSTAT_INVLOGUNI	invalid logical unit number
33	FI_IOSTAT_ENDFILERR	ENDFILE error
34	FI_IOSTAT_UNIALROPE	unit already open
35	FI_IOSTAT_SEGRECFOR	segmented record format error
36	FI_IOSTAT_ATTACCNON	attempt to access non-existent record
37	FI_IOSTAT_INCRECLEN	inconsistent record length
38	FI_IOSTAT_ERRDURWRI	error during write
39	FI_IOSTAT_ERRDURREA	error during read
40	FI_IOSTAT_RECIO_OPE	recursive I/O operation
41	FI_IOSTAT_INSVIRMEM	insufficient virtual memory
42	FI_IOSTAT_NO_SUCDEV	no such device
43	FI_IOSTAT_FILNAMSPE	file name specification error
44	FI_IOSTAT_INCRECTYP	inconsistent record type
45	FI_IOSTAT_KEYVALERR	keyword value error
46	FI_IOSTAT_INCOPECLO	inconsistent OPEN/CLOSE parameters
47	FIO_DEF(FI_IOSTAT_WRIREAFIL	write to READONLY file
48	FIO_DEF(FI_IOSTAT_INVARGFOR	invalid argument for IO library
49	FIO_DEF(FI_IOSTAT_INVKEYSPE	invalid key specification
50	FIO_DEF(FI_IOSTAT_INCKEYCHG	inconsistent or duplicate key
51	FIO_DEF(FI_IOSTAT_INCFILORG	inconsistent file organization
52	FIO_DEF(FI_IOSTAT_SPERECLOC	specified record locked
53	FIO_DEF(FI_IOSTAT_NO_CURREC	no current record
54	FIO_DEF(FI_IOSTAT_REWRITERR	REWRITE error

TABLE A-1 Fortran I/O Status Message Summary *(Continued)*

55	FIO_DEF(FI_IOSTAT_DELERR	DELETE error
56	FIO_DEF(FI_IOSTAT_UNLERR	UNLOCK error
57	FIO_DEF(FI_IOSTAT_FINERR	FIND error
59	FIO_DEF(FI_IOSTAT_LISIO_SYN	list-directed I/O syntax error
60	FIO_DEF(FI_IOSTAT_INFFORLOO	infinite format loop
61	FIO_DEF(FI_IOSTAT_FORVARMIS	format/variable-type mismatch
62	FIO_DEF(FI_IOSTAT_SYNERRFOR	syntax error in format
63	FIO_DEF(FI_IOSTAT_OUTCONERR	output conversion error
64	FIO_DEF(FI_IOSTAT_INPCONERR	input conversion error
66	FIO_DEF(FI_IOSTAT_OUTSTAOVE	output statement overflows record
67	FIO_DEF(FI_IOSTAT_INPSTAREQ	input statement requires too much data
68	FIO_DEF(FI_IOSTAT_VFEVALERR	variable format expression value error
70	FIO_DEF(FI_IOSTAT_INTOVF	integer overflow
71	FI_IOSTAT_INTDIV	integer divide by zero
72	FI_IOSTAT_FLTOVF	floating overflow
73	FI_IOSTAT_FLTDIV	floating/decimal divide by zero
74	FI_IOSTAT_FLTUND	floating underflow
77	FI_IOSTAT_SUBRNG	subscript out of range
80	FI_IOSTAT_WRONUMARG	wrong number of arguments
81	FI_IOSTAT_INVARGMAT	invalid argument to math library
82	FI_IOSTAT_UNDEXP	undefined exponentiation
83	FI_IOSTAT_LOGZERNEGF	logarithm of zero or negative value
84	FI_IOSTAT_SQUROONEG	SQUROONEG square root of negative value
87	FI_IOSTAT_SIGLOSMAT	significance lost in math library
88	FI_IOSTAT_FLOOVEMAT	floating overflow in math library

TABLE A-1 Fortran I/O Status Message Summary *(Continued)*

89	FI_IOSTAT_FLOUNDMAT	floating underflow in math library
93	FI_IOSTAT_ADJARRDIM	adjustable array dimension error
94	FI_IOSTAT_NEGVEC	negative vector length in array math function
95	FI_IOSTAT_DOMERR	invalid argument to array math function
96	FI_IOSTAT_OVEEXE	floating point overflow in array math function
97	FI_IOSTAT_SIGLOS	loss of significance in array math function
98	FI_IOSTAT_DENNUM	denormalized floating pointnumber detected in array math function
99	FI_IOSTAT_NOTCM	Formatted IO is not supported on PFS files

Index

A

ALIGN directive, 2-3
array combining scatter functions
  XXX_SCATTER, 9-3
array location intrinsic functions
  MAXLOC, 9-2
  MINLOC, 9-2
array prefix and suffix functions
  XXX_PREFIX, 9-3
  XXX_SUFFIX, 9-3
array ranking functions
  GRADE_DOWN, 9-4
  GRADE_UP, 9-4
array reduction functions
  ALL, IANY, IPARITY, PARITY, 9-3
array syntax, 2-3
arrays
  default treatment, 2-2
  distributed, 2-1
  nondistributed, 2-2
  nonsequential, 2-1
  sequential, 2-2

B

bit manipulation functions
  LEADZ, 9-2
  POPCNT, 9-2
  POPPAR, 9-2

C

C preprocessor
  used with -D switch, 4-5
CM Fortran library
  header files, 3-5
  linking, 3-5
CMF extensions not in F90
  DO N TIMES, 7-6
CMF extensions to F90, 7-6
  DCMPLX type conversion, 7-6
  DOUBLE COMPLEX data type, 7-6
  KIND *N support, 7-6
  type checking function ISNAN, 7-6
CMF extensions to Subset HPF
  CMF-style array constructors, 7-6
  intrinsic functions DIAGONAL, REPLICATE, FIRSTLOC, LASTLOC, PROJECT, 7-6
  layout directives in CMF mode, 7-6
  nested WHERE statements, 7-6
CMF utility library, 10-1, A-1
collapsed dimensions, 2-5
compiler directives
  Subset HPF 1.1 list, 2-3
compiling
  incrementally, 3-8
  simple example, 3-7
  using the cpp preprocessor, 3-8, 4-5
compiling CMF sources
  directly to HPF, 3-7

D

debugging, 3-8
  F77 souce level, 3-8
  HPF source level, 3-8
directives. See compiler directives
DISTRIBUTE directive
  block distribution, 2-4
  block(N) distribution, 2-5
  brief definition, 2-3
  collapsed dimensions, 2-5
  types of data mapping available, 2-4
  use in F77 to HPF transformation example, 2-11

E

elemental intrinsics function
  ILEN, 9-2

F

F77 to F90 transformation example, 2-8 to 2-10
F77 to HPF transformation example, 2-8 to 2-12
F77_LOCAL interface, 8-1
  argument passing, 8-2
  array-inquiry utilities, 8-4
  MAP_TO([LAYOUT=]F77_ARRAY, 8-3
  MAP_TO([LAYOUT=]HPF_ARRAY), 8-3
  MAP_TO([LAYOUT=]NO_CHANGE), 8-3
  MAP_TO(F77_ARRAY) program example, 8-7
  MAP_TO(NO_CHANGE) program example, 8-9
  processor synchronization, 8-2
  program example, 8-6
  subgrid-inquiry utilities, 8-5
F77_LOCAL library
  linking, 3-3
F90 additions to Subset HPF, 7-5
  construct name support, 7-5
  CYCLE, EXIT, 7-5
  free source form, 7-5
  function BIT_SIZE, 7-5
  functions SELECTED_INT_KIND, 7-5
  functions SELECTED_REAL_KIND, 7-5
  integer case construct, 7-5
  KIND support, 7-5
  listdirected I/O to internal files, 7-5
  named END statements, 7-5
  NAMELIST I/O, 7-5
  numeric inquiry functions, 7-5
F90 feature summary
  arithmetic and logical features, 7-2
  declarations, 7-3
  F77 support, 7-1
  intrinsic procedures, 7-2
  MILSTD1753 support, 7-1
  procedure features, 7-3
  syntax improvements, 7-3
filename extensions, list, 3-2

H

header file directory, 3-3
header files
  CM Fortran library, 3-5
  HPF library, 3-3
  S3L, 3-5
  timer routines, 3-5
hpf
  command syntax, 3-1
  source file filename extensions, 3-2
HPF intrinsic functions, 9-1
  system inquiry, 9-1
HPF library
  header files, 3-3
  how to link, 3-3
HPF library exceptions
  default HPF_TEMPLATE arguments, 9-5
  exceptions to arbitrary size array arguments, 9-5
HPF library support, 9-2
  array combining scatter functions, 9-3
  array prefix and suffix functions, 9-3
  array ranking functions, 9-4
  array reduction functions, 9-3
  bit manipulation functions, 9-2
  mapping inquiry subroutines, 9-2
HPF library, linking, 3-3

I

INDEPENDENT directive, 2-4

L

-lhpf switch, 3-3
linking
  CM Fortran library, 3-5
  F77_LOCAL library, 3-3
  HPF library, 3-3
  simple example, 3-7

M

mapping inquiry subroutines
  HPF_ALIGNMENT, 9-2
  HPF_DISTRIBUTING, 9-2
  HPF_TEMPLATE, 9-2
maximum array rank, 9-5

O

optimization
  explicit guides for compiler, 2-3
  implicit clues for compiler, 2-3
optimization tips
  avoid array sections, 6-8
  avoid expensive operations, 6-9
  avoid general communications, 6-11
  map arrays explicitly, 6-5
  minimize communication, 6-4
  provide compiler with explicit directions, 6-4
  use optimization switches, 6-11
  use parallel language constructions, 6-1
  use S3L functions, 6-10
optimiztion tips
  use simple constructs, 6-11

P

parallel arrays
  block distribution of dimensions, 2-4
  collapsed dimensions, 2-5
  explicitly parallel, 2-2
  implicitly parallel, 2-2
  nonsequential and distributed, 2-1
  role in Sun HPF, 2-2
  using DISTRIBUTE directive on, 2-2
parallel file systems. See PFS
parallel language constructions, 2-3
performance analysis tips
  analyze Fortran output, 6-14
  repeat timing runs, 6-13
  use .lis file, 6-13
  use -tmprofile switch, 6-14
PFS
  brief definition, 5-2
  file path names, 5-2
  programming example, 5-3
printing timer results, 4-2
processor synchronization, 4-4
PROCESSORS directive, 2-3

S

S3L
  header files, 3-5
SEQUENCE directive, 2-3
serial arrays
  sequential and nondistributed, 2-2
Sun HPF I/O, 5-1
Sun HPF restrictions
  complex alignments, 7-4
  cyclic and blockcyclic distributions, 7-4
  prescriptive mapping, 7-4
switches, 3-10
system inquiry intrinsic functions, 9-1
  NUMBER_OF_PROCESSORS, 9-1
  PROCESSORS_SHAPE, 9-1

T

TEMPLATE directive, 2-3
timer routines, 3-5
  header files, 3-5
timer-fort.h, 4-3
timing utility, 4-1
  getting results, 4-2
  starting timers, 4-2
  stopping timers, 4-2
  synchronization issues, 4-4
  tips on using, 4-4