SX-Aurora TSUBASA

SX-Aurora TSUBASA

NEC HPF User's Guide

Proprietary Notice

Trademarks and Copyrights

Preface

This document explains how to use the NEC HPF compiler for the Vector Engine.

The latest version of this document is available at the NEC Aurora Forum:

https://sxauroratsubasa.sakura.ne.jp/wiki/index.php?title=Special:WikiForum&forum=68

Currently, parallelization of large scale scientific programs that require super-computers is inevitable to obtain execution performance or use large amount of memory. However, development of distributed-memory parallel programs is very time-consuming because programmers have to explicitly assign data and computation to computation nodes and describe data transfer among them.

High Performance Fortran (HPF) is a set of extensions to Fortran 95 published by HPF Forum (HPFF), which was led by Ken Kennedy of Rice University. A goal of HPF is to enable programmers to parallelize programs for distributed-memory parallel computers easily.

The effort to standardize HPF began in 1991, and HPF 1.0 was published as early as in May 1993, which was revised to HPF 1.1 with minor improvements in November 1993. As a result of further discussions in HPFF2, HPF 2.0 was published in January 1997, in which features are reduced from HPF 1.0 to facilitate early development of HPF compilers. HPF 2.0 also defines HPF Approved Extensions to make up for functional insufficiency of the language.

In Japan, Japan Association for HPF (JAHPF), which consisted of domestic compiler manufacturers and supercomputer users, started in 1997 and published HPF/JA 1.0 specification, which defines features that enable more detail control of parallelization and data transfer in addition to main features of HPF 2.0 and HPF Approved Extensions, in January 1999.

Description of High Performance Fortran (HPF) Language in this document is based on the following documents published by HPF Forum.

- High Performance Fortran Language Specification, High Performance Fortran Forum, November 10, 1994 Version 1.1
- High Performance Fortran Language Specification, High Performance Fortran Forum, January 31, 1997 Version 2.0

Description of HPF/JA, which is an extension of HPF, in this document is based on the following document.

- HPF/JA Language Specification, JAHPF (Japan Association for High Performance Fortran), January 31, 1999 Version 1.0
  English Version 1.0 November 11,1999

Please refer to the documents at the following sites to learn the specifications of HPF and HPF/JA in more detail.

- http://hpff.rice.edu/versions/
- http://site.hpfpc.org/home/former_hpfpc/gengo-shiyou

(Note) The information above is as of July 2022.

The following is related documents for using NEC HPF.

- How to use the NEC Fortran compiler
  Fortran Compiler User's Guide (G2AF02E)
- How to use NEC MPI
  NEC MPI User's Guide (G2AM01E)
- How to use PROGINF and FTRACE
  PROGINF/FTRACE User's Guide (G2AT03E)
- How to use NQSV
  NEC Network Queuing System V (NQSV) User's Guide (G2AD03E)

Definitions and Abbreviations

Term	Description
Vector Engine (VE)	The core part of the SX-Aurora TSUBASA system, on which applications are executed. A VE is implemented as a PCI Express card and attached to a server called a vector host.
Vector Host (VH)	A Linux (x86) server to which VEs are attached, in other words, a host computer equipped with VEs.
Host	A VH or VE
NQSV	A job scheduler for the SX-Aurora TSUBASA.
NQSV request execution	Program execution using NQSV.
VE number	An identification number of a VE. VE numbers of VEs attached to a VH are consecutive integer values starting at 0.
VH name	The hostname of a VH, which is a host computer.
MPI	Abbreviation of Message Passing Interface. MPI is a standard specification for a communication library. It can be used together with OpenMP or automatic parallelization.

Contents

List of tables

List of figures

Chapter1 Getting Started

1.1 Introduction to HPF

1.1.1 Distributed-Memory Parallel Programming with HPF

Program development for distributed-memory parallel computers requires consideration of the following three points:

- Data Mapping

  It is necessary to decide which part of data should be allocated on which process, which is called data mapping.
  Access to data allocated on remote processes involves much higher overhead compared with that allocated on the local process. Therefore, data used in a series of processing should be allocated on the same process.

- Processing Assignment (Computation Mapping)

  It is necessary to decide which processing such as computations, assignments, and branches should be executed on which process. This is called computation mapping. Generally, to achieve N-times speed-up using N processes, it is necessary for processes to share the processing equally and processing on each process must be able to be executed simultaneously.

- Data Transfer

  When a process that performs some processing differs from a process on which data needed for the processing is allocated, the data has to be transferred from the latter process to the former process as shown in Figure 1 since the former process cannot directly access the data on the latter.

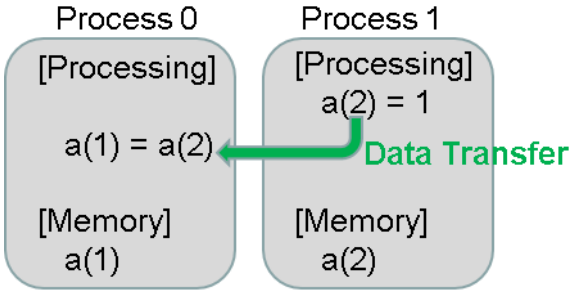

Figure 1 Data Transfer

Moreover, synchronization between the processes is also necessary before and after the data transfer.

It is really time-consuming and error-prone to develop high-performance parallel programs considering all these points as in parallel programming with MPI. The basic concept of HPF is that programmers only decide data mapping, and HPF compilers decide computation mapping and generate necessary data transfer and synchronization among processes automatically according to the data mapping. Therefore, programmers can develop parallel programs as if all processes could access all data on all processes without considering whether data is allocated on the local process or remote processes. This programming model is called global model in HPF.

HPF compilers parallelize programs by mainly assigning iterations of parallelizable loops to processes. HPF compilers decide computation mapping so that data can be accessed locally as much as possible. Therefore, the main task of programmers in HPF is to map arrays along the axis accessed by parallelizable loops as shown in Figure 2.

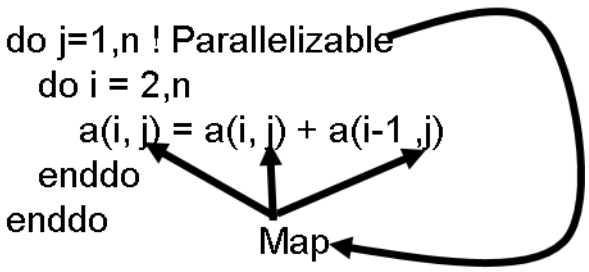

Figure 2 HPF Programming

1.1.2 HPF Program Examples

The following program assigns sum of two arrays a and b to the array c.

```
      real a(10), b(10), c(10)
        :
      do i=1, 10
        c(i) = a(i) + b(i)
      enddo
```

Figure 3 Fortran Program Example

It is possible to compile and execute this program with HPF compilers as it is. However, HPF compilers do not parallelize this program at all for the following reason. HPF compilers allocate whole arrays for which HPF directives are not specified on every process and assign processing on the arrays so that each process accesses only data on itself as much as possible. As a result, all processes execute all processing, and no speed-up is obtained no matter how many processes execute.

It is necessary to specify data mapping of the arrays to parallelize this program with HPF. The second line of Figure 4 is a DISTRIBUTE directive, the most basic HPF directive for data mapping. This directive specifies that the arrays a, b, and c should be distributed onto processes evenly.

```
      real a(10), b(10), c(10)
!HPF$ DISTRIBUTE (BLOCK) :: a, b, c
        :
      do i=1, 10
        c(i) = a(i) + b(i)
      enddo
```

Figure 4 HPF Program Example

When this program is compiled and linked with HPF compilers and executed on two processes,

process 0 and process 1, arrays are split evenly and the first half is allocated on process 0, and the second half on process 1. As for the processing, the first half of the loop is executed by process 0, and the second half by process 1 so that each process accesses only data on itself as much as possible. As a result, this program is parallelized well.

In this way, what programmers mainly have to do in HPF programming is to specify data mapping of arrays by inserting HPF directives into serial Fortran programs, which are treated as comment lines by Fortran compilers.

1.1.3 Overview of the HPF Specification

The HPF specification consists of the HPF 2.0 specification, HPF Approved Extensions, and HPF/JA Extensions. NEC HPF's extensions are also available. These are categorized in the following three features.

- Data Mapping Related Directives

  Directives to specify how to map arrays onto processes, which are the main feature of HPF.

- Computation Mapping and Data Transfer Related Directives

  HPF compilers sometimes fail to judge parallelizable loops as parallelizable or select optimal assignment of processing to processes, or generate unnecessary data transfer. In such cases, programmers can specify parallelizable loops (INDEPENDENT directive) or optimal assignment of processing to processes (ON-HOME directive), or that data transfer is not needed.

- Other Features

  HPF defines various other features including intrinsic procedures such as NUMBER_OF_PROCESSORS(), library procedures such as mapping inquiry procedures and array computation procedures, the EXTRINSIC procedure feature, which enables HPF procedures to call non-HPF procedures.

Usage of HPF directives is explained in Chapter4. Please refer to High Performance Fortran Language Specification and HPF/JA Language Specification for the accurate specifications of HPF directives.

1.2 Introduction to the NEC HPF compiler

1.2.1 Compilation and Link of HPF Programs

It is possible to compile and link HPF programs with the HPF compilation command ve-hpf. Execution of the command ve-hpf generates executable HPF programs parallelized from HPF source programs as shown in Figure 5. NEC Fortran compiler (version 3.0.7 or after) in NEC SDK and NEC MPI are required to use NEC HPF. Please refer to Chapter2 for details of compilation and link of HPF programs.

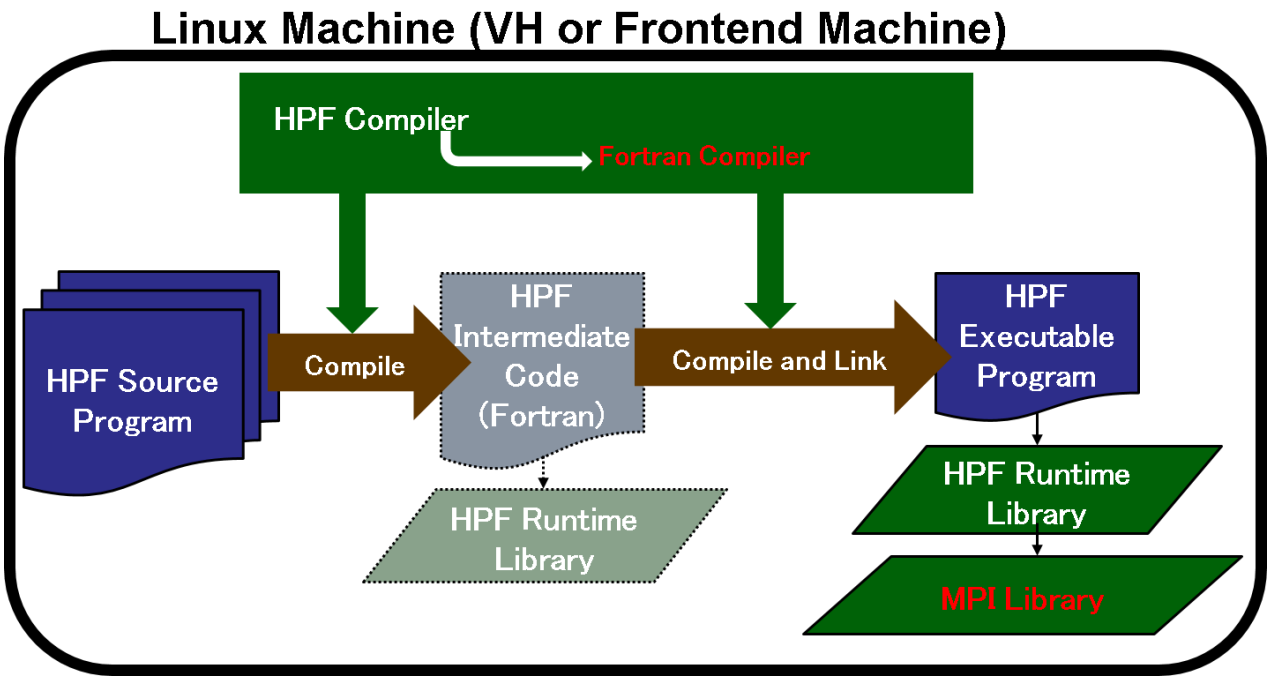

Figure 5 HPF compiler

1.2.2 Execution of HPF Programs

HPF executable programs are actually MPI executable programs with references to the MPI library. Therefore, it is possible to execute HPF executable programs with the command mpirun or mpiexec just like MPI executable programs. Refer to Chapter3 for details of

execution of HPF programs.

1.2.3 Notes and Restrictions

- The execution performance of formatted I/O is not fully tuned. Therefore, use unformatted I/O for reading or writing large size data if possible. Also, refer to the description that uses Figure 103, Figure 106, Figure 109, and Figure 113 in Subsection 5.1.4 for performance improvement of I/O.

- Derived types can be used only for declaring shadow areas. Therefore, only derived types whose only component is a one-dimensional array of type integer can be used. Also, derived types cannot be mapped.

- Derived type arrays cannot appear in DATA statements.

- Derived type constructors whose components include array constructors cannot appear in DATA statements.

- Characteristics of pointer dummy arrays cannot be used for declaring other variables. For example, pointer dummy arrays cannot be referenced as the argument of the intrinsic functions LBOUND, UBOUND, or SIZE as follows:

```
subroutine sub(p)
integer, pointer :: p(:,:)
integer, dimension(lbound(p,1):ubound(p,1), size(p,2)) :: a   ! Reference of p
```

- Named multi-dimensional array constants cannot appear in initialization expressions. Especially, they cannot appear in the following contexts:

  - Case expressions in CASE statements
  - Kind parameters in declaration statements
  - KIND arguments of intrinsic procedures
  - Initialization expressions in PARAMETER statements or declarations statements. For example, the following description is not allowed.

```
integer, parameter, dimension(2,2) :: x = reshape((/1,2,3,4/), (/2,2/))
integer, parameter :: y = x(1,2)     ! Named multi-dimensional array constant x
```

- Named array constants declared in modules cannot be referenced in initialization expressions using the use association. Especially, named arrays and derived types declared in modules cannot appear in the following context.

  - Case expressions in CASE statements
  - Kind parameters in declaration statements
  - KIND arguments of intrinsic procedures
  - Initialization expressions in PARAMETER statements or declarations statements.

Chapter2　Compilation and Link of HPF Programs

This chapter describes how to compile and link HPF programs.

2.1　Compilation and Link of HPF Programs

Firstly, execute the following command to read the MPI setup script each time you log in to a VH, in order to set up the MPI and Fortran compilation environment. The setting is available until you log out.

(In the case of bash)

%> **source /opt/nec/ve/mpi/***{version}***/bin/necmpivars.sh**

(In the case of csh)

%> **source /opt/nec/ve/mpi/***{version}***/bin/necmpivars.csh**

Here, *{version}* above is the directory name corresponding to the version of NEC MPI you use. For example, execute the following command to use NEC MPI version 2.5.0.

(In the case of NEC MPI version 2.5.0 and bash)

%> **source /opt/nec/ve/mpi/2.5.0/bin/necmpivars.sh**

Please refer to NEC MPI User's Guide for details.

After that, execute the HPF compilation command ve-hpf to compile and link HPF programs as follows

%> **ve-hpf**　[{options}]　{sourcefiles}　[{options}]

Here,

- {options} indicates compiler options. The compiler options are HPF compiler options, major NEC Fortran compiler options, and NEC MPI compiler options.
- {sourcefiles} indicates HPF source programs.

- Descriptions in [] are optional.

2.2 File Name Conventions

2.2.1 Input Files

The HPF compiler processes input files according to their suffixes as shown in Table 1.

Table 1 Suffixes of Input Files

Suffix	Process
.hpf	Compiles as a fixed form HPF source program.
.f	Compiles as a fixed form HPF source program.
.F	Preprocesses and compiles as a fixed form HPF source program.
.for	Compiles as a fixed form HPF source program.
.f90	Compiles as a free form HPF source program.
.F90	Preprocesses and compiles as a free form HPF source program.
.f95	Compiles as a free form HPF source program.
.F95	Preprocesses and compiles as a free form HPF source program.
.o	Links as an object file
.a	Links as a library of object files

2.2.2 Output Files

The HPF compiler outputs files with suffixes shown in Table 2 according to input files and HPF compiler options specified. It is possible to specify the name of the executable file the HPF compiler generates with the HPF compiler option –o, which defaults to a.out.

Table 2 Suffixes of Output Files

Suffix	Description
.d	Static data initialization file generated during compilation, which

	is saved with the HPF compiler option –Mkeepstatic.
.f	Fortran intermediate file with reference to HPF runtime library, which is saved with the HPF compiler option -Mftn or –Mkeepftn.
.mod	Module file generated for HPF source files with modules
.o	Object file

2.3 Compiler Options

This section describes compiler options available in the HPF compilation command ve-hpf. Table 3 shows common compiler options. The common compiler options control behaviors of the HPF compiler and Fortran compiler. Optional specifications in suboptions are enclosed in [] in the table.

Table 3 Common Compiler Options

Option	Suboption	Description
-c		Stops after compiling (The object file name is *filename.o*).
-D	*name*[**=***value*]	Defines a preprocessor macro *name,* with value *val* if specified.
-E		Displays a pre-processed HPF source file to the standard output without compilation and link.
-F		Saves a pre-processed HPF source file in *filename.f*.
-I	*directory*	Adds a directory *directory* to the search path for include files.
-L	*directory*	Adds a directory *directory* to the search path for library files. To use multiple directories for retrieval, this option can be specified multiple

		times in the order in which retrieval is to be performed.
-l	library	Loads the library *library*, in addition to the standard libraries. To retrieve multiple libraries, this option can be specified twice or more in the order in which retrieval is to be performed.
-O	Specifies the code optimization level (0 - 4). The default is 2.	
	0	The HPF compiler does not perform optimizations and passes the -O0 option to the back-end Fortran compiler.
	1	The HPF compiler does not perform optimizations and passes the –O1 option to the back-end Fortran compiler.
	2	The HPF compiler performs optimizations and passes the –O2 option to the back-end Fortran compiler.
	3	The HPF compiler performs optimizations and passes the –O3 option to the back-end Fortran compiler.
	4	The HPF compiler performs optimizations and passes the –O4 option to the back-end Fortran compiler.
-o	*filename*	Names the object file *filename*.
-U	*name*	Undefines a preprocessor macro *name*.
-V		Displays the HPF compiler version.
-v		Displays the HPF compiler, backend Fortran compiler and linker phase invocations.

Table 4 shows the HPF compiler options, which must be specified following –M and the

suboptions -Must be specified following the corresponding options without spaces.

Table 4　HPF Compiler Options

Option	Suboption	Description
allow_nfort_cncall		Allows the Fortran compiler directive cncall. Note that if data transfer occurs in procedures invoked in parallelized loops, the behavior of the program is not guaranteed.
allow_nfort_paralleldo		Allows the Fortran compiler directive parallel do. Note that if data transfer occurs in loops parallelized by the Fortran compiler, the behavior of the program is not guaranteed.
autodist	Specifies distribution of arrays. When the suboptions below are omitted, all arrays are distributed along the last axis with BLOCK distribution. ● Arrays that appear in both COMMON statements and NAMELIST statements must not be distributed. If distributed, the behavior of the program is not guaranteed. ● The following arrays are not distributed ・ Arrays that appear in DISTRIBUTE directives, ALIGN directives, INHERIT directives, or DYNAMIC directives ・ Arrays that appear in SEQUENCE directives ・ Arrays that appear in PARAMETER statements, EQUIVALENCE statements, or NAMELIST statements ・ Arrays of type character or derived type ・ Arrays with POINTER attribute or TARGET attribute ・ Assumed size arrays ・ Arrays in local procedures except for dummy array	

	arguments ● When an HPF program uses common blocks or interface blocks, the same option must be specified to all the files that constitutes the HPF program. ● Distribution specified to each array is displayed with the HPF compiler option -Minform=inform. ● You can use parallelization information files output with the HPF compiler option -Mlist2 to judge whether distributions of arrays are appropriate.	
	=all[**:***b*]	Distributes all arrays. The binary integer *b* corresponds one to one, from the least significant bit, to axes of arrays from the last axis to the first, and the axes that correspond to bit 1 are distributed with BLOCK distribution. If omitted, only the last axis is distributed with BLOCK distribution.
	=rank*?*[**:***b*]	Rank *?* arrays are distributed, where *?* is an integer. The binary integer *b* corresponds one to one, from the least significant bit, to axes of arrays from the last axis to the first, and the axes that correspond to bit 1 are distributed with BLOCK distribution. If omitted, only the last axis is distributed with BLOCK distribution.
backslash		Specifies that the backslash character in quoted strings is treated as a normal character rather than as an escape character.
nobackslash		Specifies that the backslash character in quoted strings is treated as an escape character. [Default]
chkhome		Specifies that arrays cannot be a home array of

		loops if they could be accessed out of bounds in the loops. This option prevents run-time errors with the message “invalid alignment”, but can cause performance degradation.
commonchk		Detects inconsistencies in declarations of COMMON block variables among procedures at run-time. This option has to be specified to all the procedures that constitute an executable program. The **-Mnoentry** or **-Mnoerrline** option cannot be used with this option. When any of these is also specified, only the option specified last is available.
cprop		Promote the constant propagation optimization. However, the back-end Fortran compiler may detect a compile-time error when this optimization makes a denominator of division zero.
nocprop		The constant propagation optimization is not performed on denominators of divisions. [Default]
dclchk		Specifies that all variables must have explicit declarations.
nodclchk		Specifies that variables do not have to be declared explicitly. [Default]
dintrin		The references to the following Fortran intrinsic procedures, HPF library procedures, and HPF local library procedures are treated as the references to the corresponding extended procedures whose results are of type 8-byte integer.

		● Fortran intrinsic procedures COUNT, LBOUND, MAXLOC, MINLOC, SHAPE, SIZE, or UBOUND ● HPF library procedures COUNT_PREFIX, COUNT_SCATTER, COUNT_SUFFIX, GRADE_DOWN, or GRADE_UP, ● HPF local library procedures GLOBAL_SHAPE, GLOBAL_SIZE, LOCAL_BLKCNT, LOCAL_LINDEX, or LOCAL_UINDEX
dlines		In fixed source form, the HPF compiler treats lines containing "*", "D", or "d" at column 1 as valid statements. In free source form, the compiler treats lines beginning with "!!" as valid statements.
nodlines		In fixed source form, the compiler treats lines containing "*", "D", or "d" at column 1 as comment statements. In free source form, the compiler treats lines beginning with "!!" as comment statements. [Default]
extend		The HPF compiler accepts 2048-column source code.
f90		Compiles program units as Fortran 90 procedures. This option does not affect procedures whose extrinsic kinds are explicitly specified.
fixed		Assumes source files are written in fixed form.
free		Assumes source files are written in fixed form.
nofree		Assumes source files are written in fixed form. [Default]

ftn		Stops after HPF compilation and keeps the intermediate output files.
fullref		The values of all shadow objects are always set with those of the corresponding data objects even if only part of the shadow objects is specified in partial REFLECT directives. It might be faster than transferring only part of shadow objects because of reuse of data transfer information in such cases that patterns of partial REFLECT directives change every time. Note that this option must be used consistently to all the procedures which constitute one executable program.
g		Enables the –Mkeepftn option and invokes the backend Fortran compiler with the –g option.
hpfout		Generates HPF source files with the suffix .hpf.src in which DISTRIBUTE directives specified with the –Mautodist option are inserted.
info		Outputs loop parallelization information into the standard output.
inform	Specifies the diagnostic message level.	
	=fatal	Outputs diagnostic messages with the fatal level.
	=severe	Outputs diagnostic messages with the severe and fatal level.

	=warn	Outputs diagnostic messages with the warning, severe and fatal level. [Default]
	=inform	Outputs diagnostic messages of all levels.
keepftn		Generates Fortran intermediate files with references to HPF runtime library, in addition to an HPF executable file.
keepstatic		Generates static data initialization files with the suffix .d, in addition to an HPF executable file.
list		Generates list files with the suffix .lst.
list2 **list3**		Generates parallelization information list files with the suffix .lst, which include parallelization and communication information. When the Fortran compiler option –report-format or –report-all is specified together, vectorization and shared-memory parallelization information is merged into the list. Note that line numbers in messages generated by the Fortran compiler do not correspond to those in HPF source programs in this case.
nolist		Generates parallelization information list files with the suffix .lst, which include parallelization and communication information and Fortran intermediate source images. When the Fortran compiler option –report-format or –report-all is specified together, vectorization and shared-memory parallelization information is merged into the list. Note that line numbers in messages generated by the Fortran compiler do not correspond to those in HPF source programs in this case.

		Does not generate list files. [Default]
local		Compiles all procedures as the LOCAL model except for those with explicit extrinsic kinds.
noentry		Does not generate information for runtime error messages. This option can improve the execution performance. Note that when runtime errors occur, the behavior of the program is not guaranteed. Therefore, specify this option only to programs that you have confirmed run properly. This option cannot be used with the option –Mcommonchk, -Mprof, or –Msubchk. If used, only the option specified last is effective.
noerrline		Does not generate line number information for runtime error messages. This option can improve the execution performance. Note that when a runtime error occurs, the line number that caused the error is not displayed. Therefore, specify this option only to programs that you have confirmed run properly. This option cannot be used with the option –Mcommonchk, -Mprof, or –Msubchk. If used, only the option specified last is effective.
nogenblock		Treats GEN_BLOCK distribution as BLOCK distribution.
noindependent		Disables INDEPENDENT directives and parallelizes programs based only on the HPF compiler's analysis.
nolocal		Disables LOCAL clauses.
nomapnew		Treats arrays that appear in INDEPENDENT loops, are not mapped, and are not reduction variables as NEW variables.

overlap	**=size:***n*	Sets the width of shadow areas, which are added to axes distributed with BLOCK distribution or GEN_BLOCK distribution, to *n.*
preprocess		Preprocesses HPF source files regardless of suffixes of them.
r8		Treats variables of type default real as type double precision and of type default complex as type double precision complex.
recursive		Allows recursive calls. This option may adversely affect performance. Please note that not all procedures can be made recursive. For example, procedures that modify variables with the SAVE attribute or COMMON block variables as well as procedures that perform I/O are generally not candidates for recursion.
res2local		Treats RESIDENT clauses as LOCAL clauses.
scalarnew		Treats all scalar variables that appear INDEPENDENT loops and are not reduction variables as NEW variables.
sequence		Specifies that all variables have the SEQUENCE attribute.
nosequence		Specifies that only assumed-size arrays and variables that appear in SEQUENCE directives or EQUIVALENCE statements have the SEQUENCE attribute. [Default]
serial		Compiles all procedures as the SERIAL model except for those with explicit extrinsic kinds.
subchk		Detects references of arrays out of declared bounds along each axis at run-time. This option does not check LOCAL procedures. The **-Mnoentry** or **-Mnoerrline** option cannot

		be used with this option. When any of these is also specified, only the option specified last is available.
upcase		Treats uppercase letters and their lowercase counterparts as different. Fortran keywords must be in lowercase.
noupcase		Treats uppercase letters and their lowercase counterparts as same. [Default]

2.3.1 NEC Fortran Compiler Directives

Major NEC Fortran compiler directives are available. The following directives are not supported:

cncall, forced_collapse, loop_count(n), option, outerloop_unroll(n), parallel do

Also, the directive vreg can be specified in the execution part or as the last line of the specification part.

Refer to NEC Fortran User's Guide for details of NEC Fortran compiler directives.

2.3.2 NEC Fortran Compiler Options

Major NEC Fortran compiler options are available in addition to the common compiler options. The following NEC Fortran compiler options are not available. Refer to NEC Fortran User's Guide for details of NEC Fortran compiler options.

-S, -cf, clear, -fsyntax-only, -x, @<file-name>, -fivdep, -floop-count=n, -fopenmp, -pthread, -fdefault-integer=n, -fdefault-double=n, -fdefault-real=n, -fextend-source, -ffree-form, -ffixed-from, fmax-continuation-lines=n, -realloc-lhs, -frealloc-lhs-array, -frealloc-lhs-scalar, -std=standard, -use, -w, -report-file, -report-append-mode, -[no-]report-cg, -[no-]report-diagnostics, -[no-]report-inline, -[no-]report-vector, -dD, -dl, -dM, -fpp, -nofpp, -fpp-name,

-dN, -E, -H, -I-, -M, -MD, -MF <filename>, -MP, -MT <target>, -fpp, -nofpp, -fpp-name, -isysroot, -isystem, -nostdinc, -P, -Wp, -Bdynamic, -Bstatic, -static, -shared, --sysroot, -B, -fintrinsic-modules-path, -module, -J, --help, -print-file-name, -print-prog-name, -noqueue, --version

When the NEC Fortran compiler option –report-all or –report-format is specified, the intermediate source parallelized by the HPF compiler is output in the format list and line numbers in the intermediate source are displayed in the Fortran compiler messages. When the HPF compiler option –Mlist2 or –Mlist3 is specified together, the vectorization and shared-memory parallelization information is merged into the parallelization information list generated by the HPF compiler.

2.3.3 NEC MPI Compiler Options

NEC MPI compiler options -mpiprof, -show, -ve, -static-mpi, and –shared-mpi are available. Refer to NEC MPI User's Guide for details of NEC MPI compiler options.

2.4 Environment Variables

This section describes environment variables available at compilation time.

- VE_HPF_COMPILER_PATH
  When you use the HPF compiler that is not at the standard path /opt/nec/ve/bin/ve-hpf, this environment variable enables the omission of specifying the path. For example, when you use the HPF compiler /opt/nec/ve/hpf/1.0.0/bin/ve-hpf, perform the following commands.

```
(For bash)
%> export VE_HPF_COMPILER_PATH=/opt/nec/ve/hpf/1.0.0
%> export PATH=${VE_HPF_COMPILER_PATH}/bin:$PATH
%> ve-hpf
```

Chapter3 Execution of HPF Programs

This chapter describes how to execute HPF programs.

3.1 Execution of HPF Programs

Firstly, execute the following command to read the MPI setup script each time you log in to a VH, in order to set up the MPI and Fortran compilation environment. The setting is available until you log out.

(In the case of bash)

%> **source /opt/nec/ve/mpi/***{version}***/bin/necmpivars.sh**

(In the case of csh)

%> **source /opt/nec/ve/mpi/***{version}***/bin/necmpivars.csh**

Here, *{version}* above is the directory name corresponding to the version of NEC MPI you use. For example, execute the following command to use NEC MPI version 2.5.0.

(In the case of NEC MPI version 2.5.0 and bash)

%> **source /opt/nec/ve/mpi/2.5.0/bin/necmpivars.sh**

It is possible to execute HPF executable programs with the MPI execution command mpirun or mpiexec as follows, as with MPI executable programs.

%> **mpirun** [{mpioptions}] {hpfexec> [{args}] [**-hpf** {hpfoptions}]

%> **mpiexec** [{mpioptions}] {hpfexec} [{args}] [**-hpf** {hpfoptions}]

Here,

- {mpioptions} means MPI runtime options.
- {hpfexec} means specification of program execution (HPF-execution specification). An HPF executable program or a shell script that executes an HPF executable program can be specified as {hpfexec}. Please note that only one {hpfexec} can appear in the MPI

execution command.

- {args} indicates argments to the HPF executable program.
- {hpfoptions} indicates HPF runtime options.
- Descriptions in [] above are optional.

3.2 Runtime Options

Table 5 shows HPF runtime options. The HPF runtime options must be specified after -hpf in the MPI execution command. The following example specifies the HPF runtime option –version at the execution of the HPF executable program a.out.

```
%> mpirun –np 2 ./a.out -hpf -version
```

The environment variable HPF_OPTS can be used to specify HPF runtime options as follows. The specification of HPF runtime options in the MPI execution command takes precedence over that with the environment variable.

```
%> setenv HPF_OPTS “-version”
```

Table 5 HPF Runtime Options

HPF Runtime Option	Environment Variable	Description
-commmsg	**HPF_COMMMSG**	Warning messages are output when data transfer occurs across procedure boundaries.
-maxxfer [*n*]	**HPF_MAXXFER** [*n*]	Specifies the maximum size of the buffer area used for data transfer in MB, which must be in the range from 16 to 1024. The default value is

		32.
-no_stop_message	**HPF_NO_STOP_MESSAGE**	Disables the default FORTRAN STOP message display when a STOP statement with no string is executed.
-subchk [**warn**|**fatal**]	**HPF_SUBCHK** [**warn**|**fatal**]	Specifies whether detection of access out of declared bounds of arrays terminates the execution immediately or not when the HPF compiler option **-Msubchk** is specified at compilation time. The optional arguments are as follows: **warn** The execution continues after outputting the warning message. [Default] **fatal** The execution is aborted with the error message.
-version	**HPF_V**	Outputs the version of the HPF runtime library.
-V	**HPF_VERSION**	Outputs the version of the HPF runtime library.
-zmem [yes|no]	**HPF_ZMEM** [**yes**|**no**]	Specifies whether dynamically allocated arrays such as allocatable arrays and

		mapped arrays are initialized with the value zero. The optional suboptions are as follows: **yes** Initialized with the value zero. **no** Not initialized. [Default]

3.2.1 NEC Fortran Compiler Runtime Environment Variables

Major runtime environment variables of the NEC Fortran compiler are available. The following environment variables have no effect.

VE_ERRCTL_ALLOCATE, VE_ERRCTL_DEALLOCATE, VE_FMTIO_OFFLOAD, VE_FMTIO_OFFLOAD_THRESHOLD, VE_FORT*n*, VE_FORT_DEFAULTFILE, VE_FORT_FILEINF, VE_FORT_FMT_NO_WRAP_MARGIN, VE_FORT_FMTBUF[n], VE_FORT_FOR_PRINT, VE_FORT_FOR_READ, VE_FORT_FOR_TYPE, VE_FORT_NML_DELIM_BLANK, VE_FORT_NML_REPEAT_FORM, VE_FORT_PAUSE, VE_FORT_RECORDBUF[n], VE_FORT_SETBUF[n], VE_FORT_SUBRCW, VE_FORT_UFMTADJUST[n], VE_FORT_UFMTENDIAN, VE_FORT_UFMTENDIAN_NOVEC

Refer to NEC Fortran User's Guide for details of the environment variables.

3.2.2 NEC MPI Runtime Options

Major NEC MPI runtime options of NEC MPI are available. The following options cannot be used because HPF programs can be executed only on VE.

-vh, -sh, -vpin, -vpinning, -pin_mode, -pin_reserve, -cpu_list, -pin_cpu, -veo, -cuda

Refer to NEC MPI User's Guide for details of the runtime options.

3.2.3 NEC MPI Environment Variables

All NEC MPI environment variables are available. Refer to NEC MPI User's Guide for details of the environment variables.

Chapter4 HPF Programming

This chapter explains how to parallelize Fortran programs with HPF. The HPF features are categorized into directives for data mapping, directives for computation mapping and data transfer, and other features.

The syntax rules in this chapter are described with the following conventions:

- Characters in Bold face are written literally as shown.
- Symbols enclosed in <> are replaced with particular symbols in actual directives.
- Characters in italics represent expressions or names of objects.
- Symbols enclosed in [] are optional.
- ,... represents optionally repeated item, separated with a comma.

4.1 Data Mapping

This section describes usage of directives for data mapping.

4.1.1 DISTRIBUTE Directive

Each process that executes an HPF programs is called an abstract processor. The number of the abstract processors is the same as that of processes that execute an HPF program.
It is possible to distribute axes of arrays onto abstract processors using DISTRIBUTE directives. The HPF compiler decides optimal computation mapping and generates necessary data transfer according to the data mapping and how arrays are accessed.
.

The syntax of the DISTRIBUTE directive is as follows:

In the case of specifying a processor arrangement (See subsection 4.1.3)

!HPF$ DISTRIBUTE *a* **(** **<**distribution-format>,⋯ **)** **ONTO** *p*

or

!HPF$ DISTRIBUTE (**<**distribution-format>,⋯ **)** **ONTO** *p* **::** *a*,⋯

- *a* indicates the name of an array or template
- *p* indicates the name of a processor arrangement
- **<**distribution-format> is *****, **BLOCK**[**(**<expression>**)**], **GEN_BLOCK(***map***)**, or **CYCLIC**[**(**<expression>**)**]
  - ***** specifies that the corresponding axis of the array or template is not distributed.
  - **BLOCK** specifies that the corresponding axis of the array or template is distributed evenly. The width of the distribution can be specified with the optional **(**<expression>**)**. The width is calculated as follows by default:

    (Extent along the corresponding axis of the array or template - 1)/(Extent of the corresponding axis of the processor arrangement)
  - **GEN_BLOCK** specifies that the corresponding axis of the array or template is distributed unevenly. **(***map***)** specifies the number of array elements distributed onto each element along the corresponding axis of the processor arrangement. The values of the one-dimensional array *map* must be defined in advance.
  - **CYCLIC** specifies that the corresponding axis of the array or template is distributed in a round-robin fashion. **(**<expression>**)** specifies the width of the distribution. When the width of the distribution is omitted, the width is 1.

In the case of not specifying a processor arrangement

!HPF$ DISTRIBUTE *a* **(** **<**distribution-format>,⋯ **)**

or

!HPF$ DISTRIBUTE (**<**distribution-format>,⋯ **)** **::** *a*,⋯

Figure 6　Syntax of DISTRIBUTE Directive

Figure 7 shows an example of the BLOCK distribution, which is the most common distribution.

```
        real a(11), b(11)
!HPF$ DISTRIBUTE (BLOCK) :: a, b
          :
        do i=1, 11
          b(i) = a(i) + 1
        enddo
```

Figure 7　Example of the DISTRIBUTE Directive

When this code is executed on four abstract processors p(1), p(2), p(3), and p(4), elements of the arrays are distributed onto the abstract processors as shown in Figure 8.

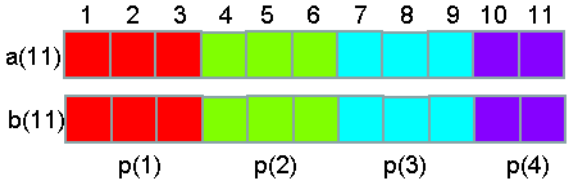

Figure 8　One-Dimensional Distribution onto Four Abstract Processors

Since the corresponding elements of the arrays a and b are distributed onto the same abstract processor, the HPF compiler assigns the computation evenly onto the abstract processors and it is executed without data transfer as shown in Figure 9.

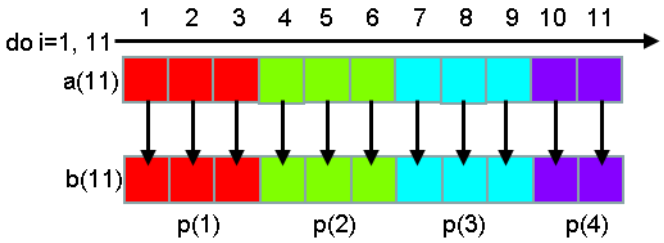

Figure 9　Parallel Execution of the Loop by Four Abstract Processors

The DISTRIBUTE directive in Figure 10 specifies that two-dimensional array a is distributed with the BLOCK distribution along the second axis.

```
      real a(11,11)
!HPF$ DISTRIBUTE (*, BLOCK) :: a
```

Figure 10　One-Dimensional BLOCK Distribution of Two-Dimensional Array

When four abstract processors p(1), p(2), p(3), and p(4) execute the code in parallel, elements of the array a are distributed onto the abstract processors as shown in Figure 11.

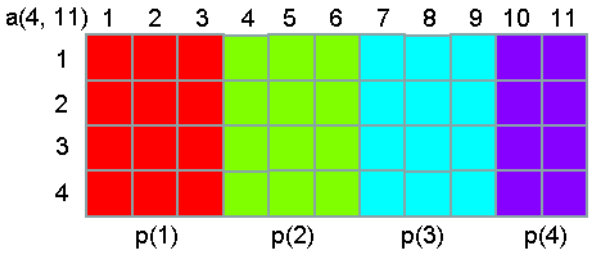

Figure 11　One-Dimensional Distribution of Two-Dimensional Array onto Four Abstract Processors

The width of the BLOCK distribution can be specified explicitly as shown in Figure 12.

```
      real a(11)
!HPF$ DISTRIBUTE (BLOCK(3)) :: a
```

Figure 12　Explicit Width of the BLOCK Distribution

Note that any element of arrays must be distributed onto at least one abstract processor. For example, the code in Figure 12 cannot be executed on three abstract processors because the array elements a(10) and a(11) are not distributed onto any abstract processors.

Array elements can be distributed in a round-robin fashion with the CYCLIC distribution as shown in Figure 13.

```
      real a(11)
!HPF$ DISTRIBUTE (CYCLIC) :: a
```

Figure 13　CYCLIC Distribution

When the code is executed on four abstract processors p(1), p(2), p(3), and p(4), elements of the array a are distributed as shown in Figure 16.

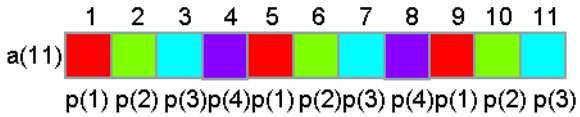

Figure 14　CYCLIC Distribution onto Four Abstract Processors

The width of the CYCLIC distribution can be specified explicitly as shown in Figure 15.

```
      real a(11)
!HPF$ DISTRIBUTE (CYCLIC(2)) :: a
```

Figure 15　Explicit Width of the CYCLIC Distribution

When the code is executed on four abstract processors p(1), p(2), p(3), and p(4), elements of the array a are distributed as shown in Figure 16.

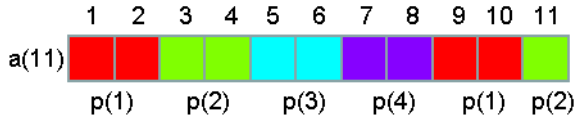

Figure 16　CYCLIC(2) Distribution onto Four Abstract Processors

Array elements can be distributed unevenly using the GEN_BLOCK distribution, generalized BLOCK distribution, as shown in Figure 17.

```
      real a(13)
      integer map(4)
      data map/6,3,2,2/
!HPF$ DISTRIBUTE (GEN_BLOCK(map)) :: a
```

Figure 17　GEN_BLOCK Distribution

Here, the one-dimensional integer array map specified in parentheses after the keyword GEN_BLOCK is called a mapping array.　The size of the mapping array must be equal to or larger than the extent along the corresponding axis of the processor arrangement and the sum of the values of the mapping array elements must be the same as the extent along the corresponding axis of the distributed array.

When the code is executed on four abstract processors p(1), p(2), p(3), and p(4), elements of the array a are distributed as shown in Figure 18.

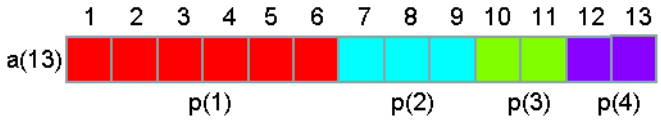

Figure 18　GEN_BLOCK Distribution onto Four Abstract Processors

The CYCLIC distribution and GEN_BLOCK distribution are useful for balancing the load. For example, the code in Figure 19 calculates the sum of two triangular matrices.

```
      real a(8,8), b(8,8), c(8,8)
        :
      do j=1, 13
        do i=1,j
          a(i,j) = b(i,j) + c(i,j)
        enddo
      endo
```

Figure 19　Sum of Triangular Matrices

If the second axis of the arrays are distributed with BLOCK distribution onto four abstract processors as shown in Figure 20, the load will be unbalanced as the abstract processors p(1), p(2), p(3), and p(4) execute 3, 7, 11, and 15 assignment statements, respectively as shown in Figure 21.

```
      real a(8,8), b(8,8), c(8,8)
!HPF$ DISTRIBUTE (*, BLOCK) :: a, b, c
```

Figure 20　BLOCK Distribution along the Second Axis

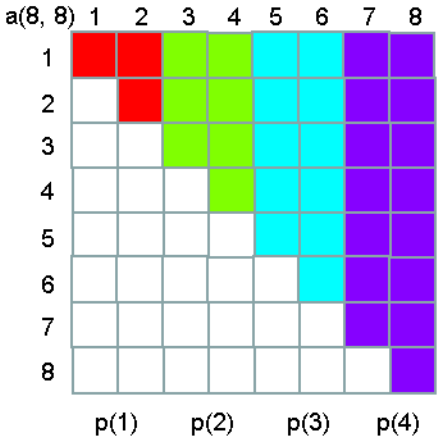

Figure 21　Unbalanced Loads between Abstract Processors

The load balance is improved by distributing arrays along the second axis with the GEN_BLOCK distribution onto four abstract processors as shown in Figure 22, as the abstract processors p(1), p(2), p(3), and p(4) execute 10, 11, 7, and 8 assignment statements, respectively as shown in Figure 23.

```
      real a(8,8), b(8,8), c(8,8)
      integer map(4)
      data map/4,2,1,1/
!HPF$ DISTRIBUTE (*, GEN_BLOCK(map)) :: a, b, c
```

Figure 22　GEN_BLOCK Distribution along the Second Axis

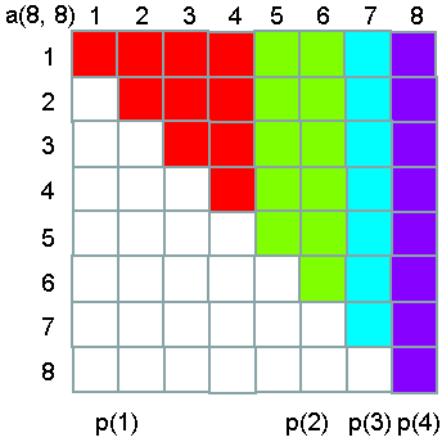

Figure 23　Sum of the Triangular Matrices with the GEN_BLOCK Distribution

4.1.2 Selection of Distribution Format

Appropriate distribution format depends on the access pattern of arrays. In many cases, when the amount of computation on each array element is approximately equal, BLOCK distribution is suitable. Otherwise, GEN_BLOCK distribution is suitable. It is easier for the HPF compiler to parallelize loops that access arrays distributed with these distribution formats efficiently because the granularity of parallelization tends to be large and consecutive array elements are allocated on the same abstract processor. Moreover, it is easier to achieve high performance because efficient data transfer patterns such as shift transfer described later are applicable.

4.1.3 PROCESSORS Directive

It is possible to declare arrangements of abstract processors (processor arrangements) with the PROCESSORS directive. Processor arrangements declared as arrays called processor arrays. The size of processor arrays must be the same as the number of processes.

The syntax of the PROCESSORS directive is as follows:

!HPF$ PROCESSORS *p* **(** <>,⋯ **)**

or

!HPF$ PROCESSORS (<>,⋯ **) ::** *p*,⋯

- *p* indicates the name of a processor arrangement
- <> indicates bounds along each axis of a processor array. For example, in the following PROCESSORS directive:

```
!HPF$ PROCESSORS p(n1,n2)
```

The number of abstract processers is the same as the size of the processor array p, n1*n2, and the rank of the processors array, 2, is equal to the number of distributed axes of arrays.

Figure 24　Syntax of PROCESSORS Directive

The shapes of processor arrays can be chosen freely according to programming convenience. The ranks of processor arrays correspond to how many axes of arrays are distributed onto processes and how many loop nests are parallelized.

```
      real a(100,100)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE a(*, BLOCK) ONTO p
```

Figure 25　One-Dimensional Distribution onto a Rank-One Processor Array

```
      real a(100,100)
!HPF$ PROCESSORS p(2,2)
!HPF$ DISTRIBUTE a(BLOCK,BLOCK) ONTO p
```

Figure 26　Two-Dimensional Distribution onto a Rank-Two Processor Array

Please note that the total number of parallelization is always the same as the number of processes. In many cases, one-dimensional parallelization using rank-one processor arrays is suitable for inhibiting overhead for parallelization.

When you would like to decide the number of abstract processors at runtime, the intrinsic

function NUMBER_OF_PROCESSORS(), which returns the number of processes, is useful.

```
      real a(100,100)
!HPF$ PROCESSORS p(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE a(*, BLOCK) ONTO p
```

Figure 27　Use of the Intrinsic Function NUMBER_OF_PROCESSORS()

Actually, when the declaration of processor arrangements is omitted, arrays are automatically distributed onto a processor array whose size is the same as the number of processes. Therefore, the DISTRIBUTE directive in Figure 28 has the same meaning as that in Figure 27. In particular, the declaration of processor arrangements is not necessary for one-dimensional distribution.

```
      real a(100,100)
!HPF$ DISTRIBUTE a(*, BLOCK)
```

Figure 28　Omission of PROCESSORS Directives

It is possible to declare processor arrays with different shapes as shown in Figure 29 as long as their sizes are identical. However, use of processor arrays with different shapes can lead unnecessary data transfers, because it is more difficult for the HPF compiler to judge whether elements of arrays distributed on processor arrays with different shapes are on the same abstract processor or not.

```
      real a(100,100), b(100,100)
!HPF$ PROCESSORS p1(4), p2(2,2)
!HPF$ DISTRIBUTE a(*, BLOCK) ONTO p1
!HPF$ DISTRIBUTE b(BLOCK, BLOCK) ONTO p2
```

Figure 29　Processor Arrays with Different Shapes

4.1.4 ALIGN Directive

When necessary data sizes are decided at runtime and sizes of arrays are declared larger than needed, some abstract processors can have no data targeted for computation since the BLOCK distribution distributes arrays evenly. For example, when the value of the variable n is six and four abstract processors p(1), p(2), p(3), and p(4) execute the code in Figure 30, the array elements targeted for the computation are distributed only onto the abstract processors p(1) and p(2), and the abstract processors p(3) and p(4) will be idle.

```
      real a(11), b(11)
!HPF$ DISTRIBUTE (BLOCK) :: a, b
      read(*,*)n
      do i=1, n
        b(i) = a(i) + 1
      enddo
```

Figure 30　Necessary Data Size is Determined at Runtime

When necessary data sizes are determined at runtime, it is better to allocate arrays with needed sizes using allocatable arrays or automatic arrays as shown in Figure 31.

```
      real, allocatable :: a(:)          ! Allocatable array
!HPF$ DISTRIBUTE (BLOCK) :: a
      read(*,*)n
      allocate(a(n))
          :
      call sub(a,n)
          :
      end

      subroutine sub(a,n)
      real a(n)                          ! Automatic array
!HPF$ DISTRIBUTE (BLOCK) :: a
```

Figure 31　Distribution Using Allocatable Arrays and Automatic Arrays

Note that it is not determined until runtime which element of an allocatable array is distributed onto which abstract processor since bounds of the array are decided at runtime. When the bounds of the one-dimensional array a and b, which are distributed onto the processor array p with the BLOCK distribution, are a(1:10) and b(1:11), respectively as shown in the code Figure 32, the array elements a(6) and b(6) are allocated on the abstract processor p(2) and p(1), respectively as shown in Figure 33. Therefore, execution of the assignment statement a(6)=b(6) requires the data transfer. As this example shows, when declared bounds of arrays are unknown at compilation time, data mapping only with DISTRIBUTE directives can lead inefficient executable programs even if the bounds are actually the same.

```
      real, allocatable :: a(:), b(:)        ! Allocatable arrays
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE (BLOCK) ONTO p :: a, b
      read(*,*)n1, n2
      allocate(a(n1), b(n2))
      do i=1,10
        a(i) = b(i)
      enddo
```

Figure 32　BLOCK Distribution of Allocatable Arrays

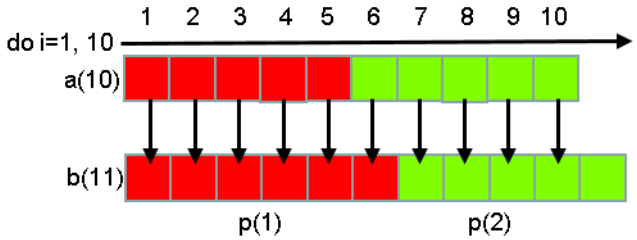

Figure 33　BLOCK Distribution Leads Data Transfer

The ALIGN directive is effective for such cases. The ALIGN directive specifies the relative location of multiple arrays (alignment). The syntax of the ALIGN directive is as follows:

!HPF$ ALIGN *a* **(<***i***>,···) WITH** *t***(<***f(i)***>,···)**

or

!HPF$ ALIGN (<*i***>,···) WITH** *t***(<***f(i)***>,···) ::** *a*,···

- *a* indicates the name of an array
- *t* indicates the name of an array or template
- <*i*> indicates an integer scalar variable or *. * specifies the axis is not aligned.
- **<***f(i)***>** indicates a linear expression *s**<*i*>**+***o*, or *, where *s* and *o* are integer expressions.
  - When <*f(i)*> is a linear expression *s**<*i*>**+***o*, the element <*i*> of array *a* is aligned with the element *s**<*i*>**+***o* of the align-target *t*.
  - When <*f(i)*> is *, the whole array *a* is replicated along the axis of the processor array to which the axis of the align-target *t* to which * is specified corresponds.

Figure 34 Syntax of ALIGN Directive

The ALIGN directive in Figure 35 specifies that the array element a(i) is mapped onto the same abstract processor as the array element b(i) is mapped onto as shown in Figure 36. The base array b of the ALIGN directive is called an align target. The data mapping of the array a is automatically determined by the relative position with the align target b, when the array b is distributed with a DISTRIBUTE directive. It is known at compilation time that the array elements b(i) and a(i) are always allocated on the same abstract processor by the correspondence between subscripts of the arrays, though the bounds of the arrays are unknown until runtime. Therefore, the HPF compiler can generate efficient parallel code because it can judge that no data transfer is needed.

```
      real, allocatable :: a(:), b(:)          ! Allocatable arrays
!HPF$ PROCESSORS p(2)
!HPF$ ALIGN a(i) WITH b(i)
!HPF$ DISTRIBUTE (BLOCK) ONTO p :: b
      read(*,*)n1, n2
      allocate(a(n1), b(n2))
      do i=1,10
        a(i) = b(i)
      enddo
```

Figure 35　Data Mapping with the ALIGN Directive

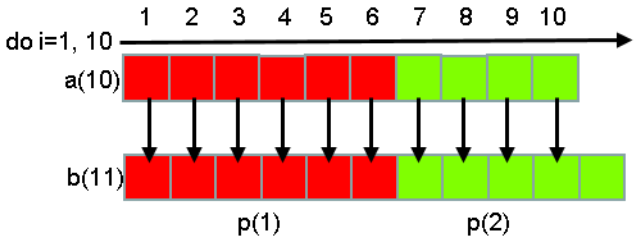

Figure 36　Effect of the ALIGN Directive

The ALIGN directive is also effective for assumed-shape arrays and automatic arrays whose bounds are declared using different variables as shown in Figure 37.

```
       :
      call sub(a,100, 100)
      end

      subroutine sub(a,n,m)
      real :: a(:)          ! Assumed shape arrays
      real :: b(n), c(m)    ! Automatic arrays
```

Figure 37　Assumed-Shape Arrays and Automatic Arrays

The ALIGN directive is also effective for the case that arrays with different bounds are accessed in a loop as shown in Figure 38. If arrays with different bounds are distributed with

the BLOCK distribution, the ranges of the array sections allocated on each abstract processor also become different as shown in Figure 39. This causes data transfer at runtime of the loop. The data transfer can be inhibited with the ALIGN directive as shown in Figure 40 since the array elements a(i) and b(i) are mapped onto the same abstract processors as the array element c(i), which is accessed in the same iteration of the loop, is mapped onto.

```
real a(0:9), b(10), c(0:10)

do i=1,9
  c(i) = a(i) + b(i)
enddo
```

Figure 38　Arrays with Different Declared Bounds

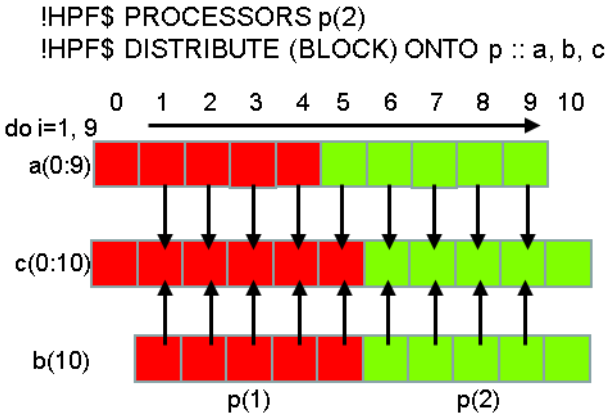

Figure 39　BLOCK Distribution of Arrays with Different Declared Bounds

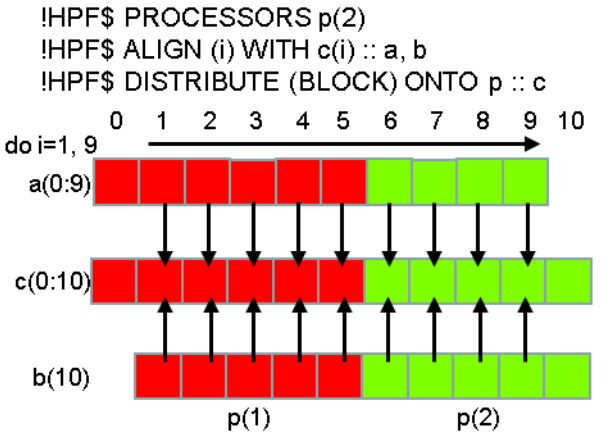

Figure 40　Alignment of Arrays with Different Declared Bounds

Note that the declared bounds of the align target c ((0:10) in this case) must include the declared bounds along the corresponding axis of the aligned arrays a and b ((0:9) and (1:10), respectively in this case). This is because if any elements of aligned arrays run out of the declared bounds along the corresponding axis of the align target, the elements are not mapped onto any abstract processor, which causes errors at compilation time or runtime.

4.1.5 TEMPLATE Directive

In Figure 41, it seems good to align a(i) and b(i) which are accessed in the same iteration of the loop. However, since the declared bounds of the arrays a and b are different, alignment of either of them with the other causes some elements to run out of the align target. To declare an array whose bounds include the bounds of both arrays will resolve the problem as shown in Figure 38, but this wastes memory just for specifying the data mapping.

```
real a(0:9), b(10)

do i=1,9
  b(i) = a(i) + 1.0
enddo
```

Figure 41　Arrays with Different Declared Bounds

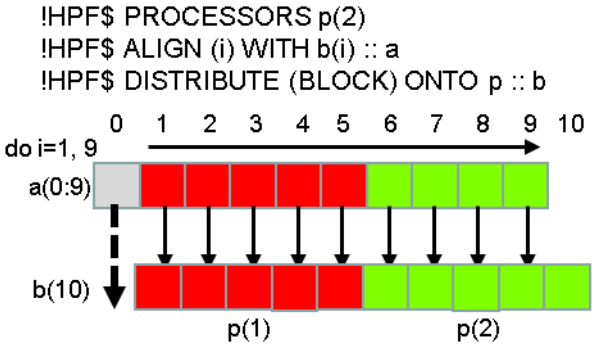

Figure 42　Alignment in which Aligned Arrays run out of the Align Target

The TEMPLATE directive enables the declaration of templates, virtual arrays that do not use memory, and is useful for such cases. The syntax of the TEMPLATE directive is as follows:

!HPF$ TEMPLATE *t* **(<>,⋯)**

or

!HPF$ TEMPLATE (<>,⋯) :: *t*,⋯

- *t* indicates a template
- <> indicates bounds along each axis of templates

Figure 43　Syntax of TEMPLATE Directive

By declaring the template t whose bounds include the bounds along the corresponding axes of arrays a and b, aligning the arrays with the template, and distributing the template as shown in Figure 44, it is possible to map the corresponding elements of the arrays a and b onto the same abstract processor so that no data transfer occurs at the execution of the loop in Figure 41.

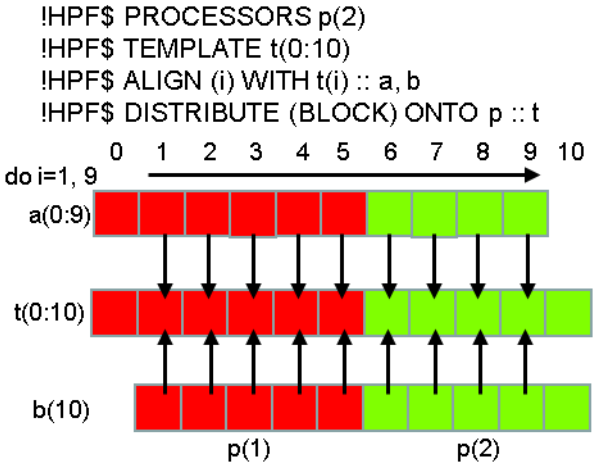

Figure 44　Data Mapping Using a Template

The subscripts of the arrays a and b are shifted by one in the loop in Figure 45 though the declared bounds of them are identical. Therefore, data mapping only with the DISTRIBUTE directive can also cause data transfer. Alignment of a(i+1) and b(i) with the ALIGN directive as shown in Figure 46 enables execution of the loop without data transfer in such cases. Also in this example, the template t whose bounds include the bounds of the arrays a and b is used as the align target since the direct alignment of a(i+1) and b(i) causes the run-out alignment.

```
real a(10), b(10)

do i=1,9
  b(i) = a(i+1) + 1.0
enddo
```

Figure 45　Loop where Arrays are Accessed with Different Subscripts

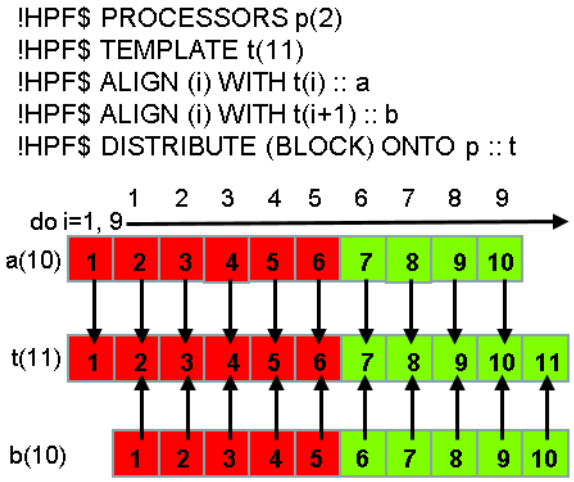

Figure 46　Alignment of Arrays Accessed with Different Subscripts

4.1.6　Summary of Data Mapping in HPF

Data mapping of arrays can be specified with the DISTRIBUTE directive and ALIGN directive in HPF. In general, by specifying alignment of arrays with a base array or template with ALIGN directives and distributing the base array or template with a DISTRIBUTE directive, data mapping of all arrays is determined as shown in Figure 47.

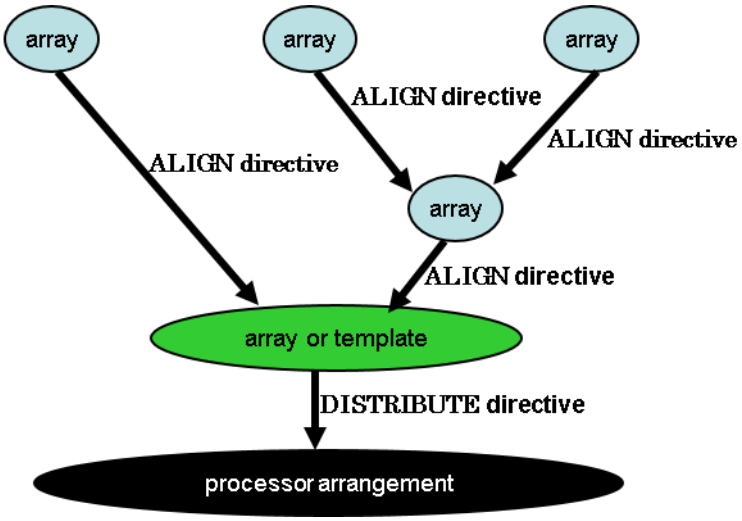

Figure 47　Data Mapping in HPF

Arrays that do not appear in a DISTRIBUTE directive nor ALIGN directive and scalar variables are replicated on all abstract processors. The replication is suitable for variables that are only read because all abstract processors can read them without data transfer.

4.1.7　Variables That Cannot Be Mapped

When an actual argument and dummy argument whose shapes are different are associated based on the Fortran sequence association or when variables whose shapes are different are associated via COMMON blocks and EQUIVALENCE statements based on the Fortran storage association, they must appear in the SEQUENCE directive in the specification part of the scoping unit. Variables specified in the SEQUENCE directive cannot be mapped. The syntax of the SEQUENCE directive is shown in Figure 48. The NOSEQUENCE directive can be used for variables you want to map when the HPF compiler option –Msequence is specified.

!HPF$ [NO] SEQUENCE　[[**::**] *s***,**⋯]

- *s* is the name of an array or **/**common block name**/**. When *s***,**⋯ is omitted in the SEQUENCE directive, it is treated as if it contained all common block and variables that are not mapped explicitly. When *s***,**⋯ is omitted in the NOSEQUENCE directive, it is treated as if it contained all common block and variables.

Figure 48　Syntax of SEQUENCE Directive

4.2 Computation Mapping and Data Transfer

This section explains how to use directives for improving computation mapping and data transfer.

4.2.1 INDEPENDENT Directive

The INDEPENDENT directive enables programmers to teach the HPF compiler that loops are parallelizable as shown in Figure 49. Loops are parallelizable when they do not have loop-carried dependencies. The loops which immediately follow INDEPENDENT directives are called INDEPENDENT loops.

```
!HPF$ INDEPENDENT
      do i=1,n
        a(i) = i
      enddo
```

Figure 49 INDEPENDENT Loop

Figure 50 shows examples of the loop-carried dependencies and these loops cannot be parallelized. In short, loops that define data in an iteration which is defined or referenced in other iterations, or loops that has branches out of the loops are not parallelizable.

```
! True Dependency: The array element a(i) is referenced after definition
      do i=1,n
        a(i) = a(i) + a(i-1)
      enddo

! Anti Dependency: The array element a(i) is defined after reference
      do i=1,n
        a(i) = a(i) + a(i+1)
      enddo

! Output Dependency: The scalar variable s is defined after definition
      do i=1,n
        if(a(i) > 0.0) s = a(i)
      enddo

! Control Dependency: Execution of the loop can terminate in the middle of the iterations
      do i=1,n
        if(a(i) > 0.0)goto 99
      enddo
99    continue
```

Figure 50　Loop-Carried Dependency

The syntax of the INDEPENDENT directive is as follows:

Perfectly Parallelizable Loops

!HPF$ INDEPENDENT [**, NEW(** *v*,⋯ **)**]

- *v* indicates the name of a variable (NEW variable)

Parallelizable Loops with Reduction

!HPF$ INDEPENDENT [**, NEW(** *v*,⋯ **)**]**,** <REDUCTION clause>,⋯

- *v* indicates the name of a variable (NEW variable)
- <REDUCTION clause> is

  REDUCTION([<reduction-kind1> **:**] *r*,⋯ **)**

  or

  REDUCTION([<reduction-kind2> **:**] *r* **/***p*,⋯**/**,⋯ **)**

  - <reduction-kind1> is **+, *, .AND.**, **.OR.**, **.EQV.**, **.NEQV.**, **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**
  - *r* indicates the name of a reduction-variable
  - <reduction-kind2> is **FIRSTMAX**, **FIRSTMIN**, **LASTMAX**, or **LASTMIN**
  - *p* indicates the name of a position variable
  - When <reduction-kind1> **:** is omitted, reduction assignments must be described any of the following forms.

    r **=** *r* <op> <expr> or *r* **=** <expr> <op> *r*

    or

    r **=** <f**(***r***,** <expr>**)**> or *r* **=** <f**(**<expr>**,** *r***)**>

    - *r* indicates the name of a reduction-variable
    - <op> indicates a reduction operator *****, **/**, **+**, **-**, **.AND.**, **.OR.**, **.EQV.**, or **.NEQV.**
    - <expr> indicates an expression that does not include the reduction variables and is estimated before the operation <op>.
    - <f**()**> indicates a reference to the function **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**

Figure 51　Syntax of the INDEPENDENT Directive

The HPF compiler parallelizes loops automatically without INDEPENDENT directives as if they were INDEPENDENT loops when it can judge that the loops are parallelizable. But the HPF

compiler sometimes fails to judge parallelizable loops as parallelizable depending on access patterns of arrays in them. When the parallelization information list or diagnostic messages show that parallelizable loops are not parallelized or unnecessary data transfers are generated, insertion of INDEPENDENT directives may improve execution performance of HPF programs significantly.

The loop nest in Figure 52 can be parallelized only with the data transfer between neighboring abstract processors (shift transfer) by assigning each iteration of the loop nest to the abstract processor that has the left hand side g(j,inew) of the assignment statement, because the left hand side g(:,inew) and the right hand side g(:,iold) never overlap in the loop nest. However, the HPF compiler currently cannot find that the values of the variables iold and inew are always different, and fails to parallelize the loop nest.

```
      subroutine sub(n, ncycles, g)
      real g(n+2,2)
!HPF$ DISTRIBUTE g(BLOCK,*)

       iold=1
       inew=2
       do it=1, ncycles
          do j = 2, n+1
             g(j,inew) = g(j-1,iold) + g(j+1,iold) + g(j,iold)
          enddo
       enddo
       iold = 3 - iold
       inew = 3 - inew
       enddo
```

Figure 52　Example of a Loop Nest Not Parallelized Automatically

Compiling the code with the HPF compiler option –Minfo displays the following diagnostic messages.

```
7, Invariant assignments hoisted out of loop
8, Distributing inner loop; 2 new loops
   expensive communication: scalar communication (get_scalar)
   expensive communication: scalar communication (get_scalar)
```

The two diagnostic messages “expensive communication: scalar communication (get_scalar)”

show that high-overhead data transfers are generated. You can find the automatic parallelization is not successful since the necessary data transfers are only the low-overhead shift transfer. In such cases, insert the INDEPENDENT directive immediately before the do j loop as shown in Figure 53.

```
      subroutine sub(n, ncycles, g)
      real g(n+2,2)
!HPF$ DISTRIBUTE g(BLOCK,*)

       iold=1
       inew=2
       do it=1, ncycles
!HPF$    INDEPENDENT
         do j = 2, n+1
            g(j,new) = g(j-1,iold) + g(j+1,iold) + g(j,iold)
         enddo
       enddo
       iold = 3 - iold
       inew = 3 - inew
       enddo
```

Figure 53　Insertion of the INDEPENDENT Directive

Then compiling the code with the HPF compiler option –Minfo displays the following diagnostic messages, which show the loop nest is parallelized well without high-overhead data transfers.

```
7, Invariant communication calls hoisted out of loop
9, Independent loop parallelized
```

4.2.2 NEW Clause

The loop in Figure 54 has a loop-carried dependency and is not parallelizable because the scalar variable s is defined and referenced in multiple iterations of the loop.

```
      do i=1,n
        s = sqrt(a(i)**2 + b(i)**2)
        c(i) = s
      enddo
```

Figure 54　Loop with a Work Variable

However, the INDEPENDENT directive with the NEW clause that specifies the variable s as shown in Figure 55 can be used to specify that the loop can be parallelized by using distinct memory areas for the variable s in distinct iterations of the loop. Variables specified in NEW clauses are called NEW variables.

```
!HPF$ INDEPENDENT, NEW(s)
      do i=1,n
        s = sqrt(a(i)**2 + b(i)**2)
        c(i) = s
      enddo
```

Figure 55　INDEPENDENT Directive with the NEW Clause

Note that values of NEW variables become undefined after execution of the INDEPENDENT loops. Therefore, if the NEW variables (s in the example above) are referenced without defining them after execution of the INDEPENDENT loop, the result of execution is not guaranteed.

The HPF compiler usually detects scalar work variables and treats them as NEW variables automatically. As for array work variables, users have to insert INDEPENDENT directives with NEW clauses that specify them, since the HPF compiler cannot detect them automatically.
For example, the arrays u and flux are used as array work variables and defined and referenced in each iteration of the loop nest in Figure 56. The do k loop, which corresponds to the distributed axis of the array f, can be parallelized without data transfers using distinct memory areas for these arrays in distinct iterations of the do k loop.

```
      subroutine rhs(f, u, n1, n2, n3)
      common /com/c1, c2, q
      dimension flux(2,n1), u(2,n1)
      dimension f(2, n1, n2, n3)
!HPF$ DISTRIBUTE F(*,*,*,BLOCK)

      do k=2, n3-1
        do j=2,n2-1
          do i=1,n1
            do m=1,2
              u(m, i) = c1 –c2
            enddo
            flux(1, i) = q * u(1,i)
            flux(2, i) = q * u(2,i)
          enddo
          do i=2, n1-1
            f(1,i,j,k) = f(1,i,j,k) * (flux(1,i+1) – flux(1, i-1))
            f(2,i,j,k) = f(2,i,j,k) * (flux(2,i+1) – flux(2, i-1))
          enddo
        enddo
      enddo
```

Figure 56　Loop Nest with Array Work Variable

Compiling the code with the HPF compiler option –Minfo displays the following diagnostic messages.

```
    9, Distributing loop; 2 new loops
         1 FORALL   generated
         2 FORALLs generated
         no parallelism: replicated array, u
         no parallelism: replicated array, flux
         no parallelism: replicated array, flux
   10, Independent loop
   16, Independent loop
         2 FORALLs generated
```

The diagnostic messages “10, Independent loop” and “16, Independent loop” show that the do m loop in line 10 and do i loop in line 16 are parallelizable. However, the most important loop do k is not detected as parallelizable. Then insert the INDEPENDENT directive with the NEW clause that specifies the arrays u and flux, and the do variables of the inner do loops j, i, and m immediately before the do k loop as shown in Figure 57.

```
      subroutine rhs(f, u, n1, n2, n3)
      common /com/c1, c2, q
      dimension flux(2,n1), u(2,n1)
      dimension f(2, n1, n2, n3)
!HPF$ DISTRIBUTE F(*,*,*,BLOCK)

!HPF$ INDEPENDENT, NEW(u, flux, j, i, m)
      do k=2, n3-1
        do j=2,n2-1
          do i=1,n1
            do m=1,2
              u(m, i) = c1 –c2
            enddo
            flux(1, i) = q * u(1,i)
            flux(2, i) = q * u(2,i)
          enddo
          do i=2, n1-1
            f(1,i,j,k) = f(1,i,j,k) * (flux(1,i+1) – flux(1, i-1))
            f(2,i,j,k) = f(2,i,j,k) * (flux(2,i+1) – flux(2, i-1))
          enddo
        enddo
      enddo
```

Figure 57　INDEPENDENT Directive with Array NEW Variables

Compiling the code with the HPF compiler option –Minfo displays the following diagnostic messages, which show the do k loop in line 8 is parallelized as an INDEPENDENT loop.

8, Independent loop parallelized

11, Independent loo

17, Independent loop

When INDEPENDENT loops are nested, the INDEPENDENT loop to which a NEW clause must be specified is the innermost one that defines the NEW variable. In the example Figure 58, the NEW clause that specifies the work variable s, which is defined in the loops do i and do j, must be specified in the INDEPENDENT directive to the innermost do j loop.

```
!HPF$ INDEPENDENT, NEW(j)
      do i=1, n
!HPF$   INDEPENDENT, NEW(s)
        do j=1,n
          s = sqrt(a(i,j)**2 + b(i,j)**2)
          c(i,j) = s
        enddo
      enddo
```

Figure 58　NEW Variable Defined in Multiple INDEPENDENT Loops

When all scalar variables defined in loops in a program are NEW variables except for reduction variables, which are explained in the next subsection, explicit NEW clauses for the scalar variables can be omitted using the HPF compiler option –Mscalarnew. Also, when all arrays that are not mapped and defined in loops in a program are NEW variables except for reduction variables, explicit NEW clauses for the arrays can be omitted using the HPF compiler option –Mnomapnew.

In the code Figure 57, since all scalar variables k, i, j, and m, and all non-mapped arrays u and flux that are defined in the loop nest, are NEW variables, the HPF compiler treats these variables as NEW variables by inserting the INDEPENDENT directive without the NEW clause and compiling the code with the HPF compiler options –Mscalarnew and –Mnomapnew.

4.2.3 REDUCTION Clause

The loop in Figure 59 executes the same operation (addition) repeatedly and accumulates the result value on a variable (r, in this case). The INDEPENDENT directive cannot be specified to this loop because of the loop-carried dependency on the variable r. However, since addition is associative and commutative, abstract processors can execute the loop almost in parallel by storing the sum of the elements of the array a that are mapped on each abstract processor in a temporal area allocated on itself (local reduction) and then adding up the values of the temporal areas on all abstract processors while transferring them (global reduction). This kind of computation is called reduction computation and the result variable of the reduction computation (r, in this case) is called a reduction variable.

```
      real a(10)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE a(BLOCK) ONTO p

      r=0
      do i=1,10
        r = r + a(i)
      enddo
```

Figure 59 Reduction Loop

The INDEPENDENT directive with the REDUCTION clause that specifies the reduction variables as shown in Figure 60 can be specified to loops that perform reduction computation. It is not correct to specify reduction variables in a NEW clause, since the values of reduction variables have to be accumulated across iterations of loops.

```
      real a(10)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE a(BLOCK) ONTO p

      r=0
!HPF$ INDEPENDENT, REDUCTION(r)
      do i=1,10
        r = r + A(I)
      enddo
```

Figure 60　INDEPENDENT Directive with REDUCTION Clause

When INDEPENDENT loops that perform reduction computation are nested, the REDUCTION clause must be specified to the outermost INDEPENDENT loop. For example, since both the loops do i and do j perform reduction computation on the variable s in the code Figure 61, the REDUCTION clause for the variable s must be specified in the INDEPENDENT directive to the outermost INDEPENDENT loop do i.

```
!HPF$ INDEPENDENT, NEW(j),REDUCTION(s)
      do i=1,n
!HPF$   INDEPENDENT
        do j=1,n
          s = s + a(i,j)
        enddo
      enddo
```

Figure 61　Where to Specify the REDUCTION Clause

4.2.4 Parallelization of Loops with Reference to Procedures

Loops that contain references to procedures as shown in Figure 62 cannot be parallelized automatically since it is not possible to analyze at compilation time whether the loops are

parallelizable.

```
      integer a(100,100)
!HPF$ DISTRIBUTE A(*,BLOCK)
          :
      do i=1,100
        call sub(a(:,i))
      enddo
          :
      end
      subroutine sub(a)
      integer a(100)
      do j=1,100
        a(j) = j
      enddo
      end
```

Figure 62　Loop with a Reference to a Procedure

Each iteration of the loop do i in Figure 62 invokes the subroutine sub and a column of the two-dimensional array a passed as the argument is defined in it. Since variables except for the argument are not defined and no I/O is performed, the loop is actually parallelizable. In such cases, the EXTRINSIC procedure feature that enables HPF procedures to invoke Fortran procedures can be used to parallelize the loop. With the EXTRINSIC procedure feature, HPF procedures can invoke procedures that are not HPF or global model by declaring an EXTRINSIC prefix as shown in Figure 63 at the beginning of the PROGRAM statement, FUNCTION statement, SUBROUTINE statement, or MODULE statement. Local model and serial model are available in addition to global model. The local model procedures are executed by each abstract processer independently like MPI procedures. The serial model procedures are executed only by one abstract processor.

EXTRINSIC (<lang> **,** <model> **)**

or

EXTRINSIC (<extrinsic-kind-keyword> **)**

- <lang> is **”HPF”** or **“Fortran”**
- <model> is **”GLOBAL”**, **“LOCAL”**, or **”SERIAL”**. **”GLOBAL”**, **“LOCAL”**, and **”SERIAL”** indicate global model, local model, and serial model, respectively.
- <extrinsic-kind-keyward> is **HPF**, **HPF_LOCAL**, **HPF_SERIAL**, **Fortran_LOCAL**, or **Fortran_SERIAL**, which indicate global model HPF, local model HPF, serial model HPF, local model Fortran, and serial model Fortran, respectively.

Figure 63　EXTRINSIC Prefix

The following description as shown in Figure 64 makes it possible to parallelize the loop.

- Declare the procedure referenced in the loop as EXTRINSIC(Fortran_LOCAL) in the explicit interface (interface block).
- Declare EXTRINSIC(Fortran_LOCAL) at the beginning of the SUBROUTINE statement of the referenced procedure.
- Specify the INDEPENDENT directive to the loop.

The HPF compiler parallelizes the loop assigning each iteration of the loop to the abstract processor that has the elements of the array a passed as the argument in the iteration.

Note that the HPF compiler parallelizes the loop assuming that data transfers are not needed in local model procedures. Therefore, when data transfers are needed for global variables or dummy arguments in the local model procedures, the behavior of the program is not guaranteed.

```
      integer a(100,100)
!HPF$ DISTRIBUTE A(*,BLOCK)
      interface
        EXTRINSIC(Fortran_LOCAL) subroutine sub(a)
        integer a(100)
        intent(out) :: a
        end subroutine
      end interface
         :
!HPF$ INDEPENDENT
      do i=1,100
        call sub(a(:,i))
      enddo
         :
      end
      EXTRINSIC(Fortran_LOCAL) subroutine sub(a)
      integer a(100)
      intent(out) :: a
      do i=1,100
        a(i) = i
      enddo
      end
```

Figure 64　Fortran_LOCAL Procedure Invoked in the INDEPENDENT Loop

4.2.5 ON-HOME-LOCAL Directive Construct and Directive

When the HPF compiler parallelizes a loop nest, it selects one mapped array as the base array for the parallelization and assigns iterations of the loops so each abstract processor accesses only the base array elements mapped on itself. The base array is called a home array. When the home array selected by the HPF compiler is not appropriate, unnecessary data transfers can occur. In the example Figure 65, since the do variable i corresponds to the non-mapped axis of the array a, all abstract processors execute the whole loop redundantly. On the other hand, the subscript along the mapped axis of the array a is always one. Since the array

elements accessed in the loop are mapped only on the first abstract processor, data transfers are needed.

```
      real a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
        :
      do i=1,99
        a(i,1) = a(i,1) + a(i+1,1)
      enddo
```

Figure 65　Boundary Processing Loop

In such cases, it is possible to improve the execution performance by inserting the ON-HOME-LOCAL directive construct as shown in Figure 66, which specifies that no data transfers are needed when the whole loop is executed only by the abstract processor onto which the array section a(:,1) is mapped.

```
      real a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
        :
!HPF$ ON HOME(a(:,1)), LOCAL BEGIN
      do i=1,99
        a(i,1) = a(i,1) + a(i+1,1)
      enddo
!HPF$ END ON
```

Figure 66　ON-HOME-LOCAL Directive Construct That Encloses the Whole Loop

The example Figure 67 shows the loop nest that performs a matrix-vector product for a sparse matrix a in the Compressed Row Storage (CRS) format. The arrays are distributed as shown in Figure 68 so that no data transfers are needed when four abstract processors execute the loop nest. However, it is currently difficult for the HPF compiler to judge that no data transfers are needed when arrays distributed with the BLOCK distribution and those with the GEN_BLOCK distribution are accessed in the same loop nest.

```
      real a(5), v(4), r(4)
      integer rst(5), cidx(5)
      integer, parameter :: m(4) = (/1,1,2,1/)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE r(BLOCK) ONTO p
!HPF$ DISTRIBUTE (GEN_BLOCK(m)) ONTO p :: a, cidx
          :
      do i=1,4
        r(i) = 0.0
        do j = rst(i), rst(i+1)-1
          r(i) = r(i) + a(j) * v(cidx(j))
        enddo
      enddo
```

Figure 67　Matrix-Vector Product in CRS Format

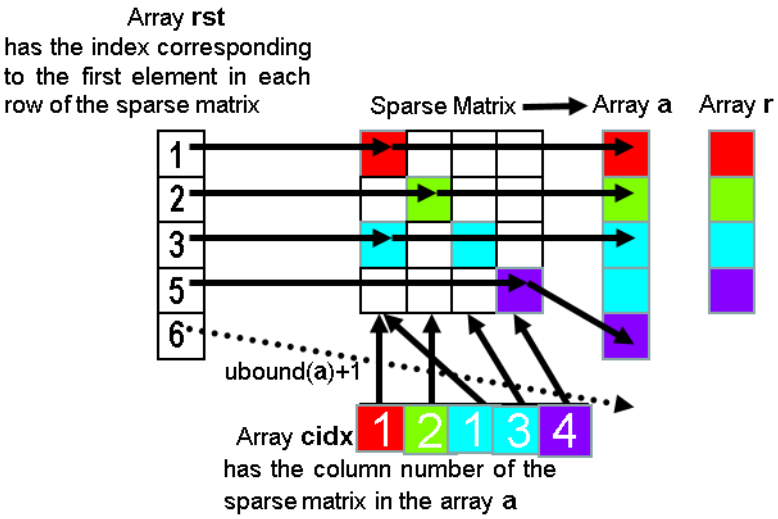

Figure 68　Mapping of Arrays in CRS Format

Then it is possible to parallelize the loop nest efficiently by specifying that data transfers for the arrays a and cidx are not needed when each iteration of the loop do i is assigned to the abstract processor that has the home array r(i) as shown in Figure 69.

```
      real a(5), v(4), r(4)
      integer rst(5), cidx(5)
      integer, parameter :: m(4) = (/1,1,2,1/)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE r(BLOCK) ONTO p
!HPF$ DISTRIBUTE (GEN_BLOCK(m)) ONTO p :: a, cidx
            :
      do i=1,4
!HPF$   ON HOME(r(i)), LOCAL(a, cidx) BEGIN
         r(i) = 0.0
         do j = rst(i), rst(i+1)-1
           r(i) = r(i) + a(j) * v(cidx(j))
         enddo
!HPF$   END ON
      enddo
```

Figure 69　ON-HOME-LOCAL Directive Construct to a Matrix-Vector Product

When the target of the ON-HOME-LOCAL directive construct is one executable statement or construct, the ON-HOME-LOCAL directive, in which keywords BEGIN and END ON are omitted, can also be used. The syntax of the ON-HOME-LOCAL directive construct and ON-HOME-LOCAL directive is as follows:

ON-HOME-LOCAL directive construct

!HPF$ ON HOME(<array section**>)** [**, LOCAL**[**(** *v*,⋯ **)**]] **BEGIN**

Sequence of <executable statement or construct>

!HPF$ END ON

- The abstract processors onto which <array section> is mapped execute the sequence of <executable statement or construct>.
- v indicates the name of a variable for which data transfers are not needed. When **(** *v*,⋯ **)** is omitted, data transfers are not needed for all variables that appear in the sequence of <executable statement or construct>.

ON-HOME-LOCAL directive

!HPF$ ON HOME(<array section**>)** [**, LOCAL**[**(** *v*,⋯ **)**]]

- The abstract processors onto which <array section> is mapped execute the immediately following executable statement or construct.

Figure 70　ON-HOME-LOCAL Directive Construct and Directive

4.2.6 SHADOW Directive and REFLECT Directive

When the do i loop in Figure 71 is parallelized selecting the left hand side b(i) as the home array, the data transfer between adjacent abstract processors is necessary because the computation references the array elements mapped on the adjacent abstract processors as shown in Figure 72.

```
      real a(12), b(12)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO p :: a, b
        :
      do i=2,11
        b(i) = a(i-1) + a(i) + a(i+1)
      enddo
```

Figure 71　Loop with References to Adjacent Elements

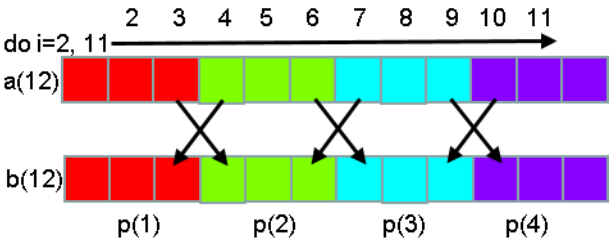

Figure 72　References between Adjacent Abstract Processors

The data transfer can be performed efficiently by allocating buffer areas to store the data received from adjacent abstract processors in advance as shown in Figure 73. The loop itself can also be executed efficiently without data transfers during the execution by referencing the values of the buffer areas as shown in Figure 74. The buffer areas are called the shadow area, and the data transfers between adjacent abstract processors are called the shift transfer.

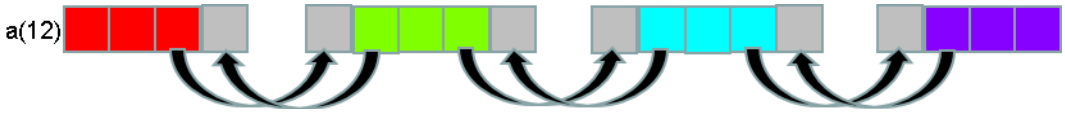

Figure 73　Shift Transfer

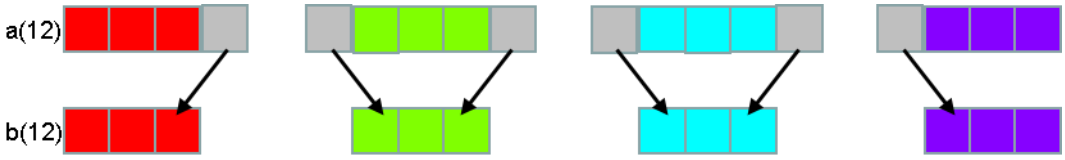

Figure 74　Parallel Execution Referencing the Shadow Area

The HPF compiler allocates the shadow area with width four along the axes distributed with the BLOCK distribution or GEN_BLOCK distribution by default and generates the shift transfer automatically.

However, it is not possible to generate the shift transfer in the example Figure 75, because it is unknown at compilation time whether the shadow area includes the adjacent reference of width n.

```
      subroutine sub(a,b,n)
      real a(100), b(100)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE (BLOCK) ONTO p :: a, b
         :
      do i=2,99
        b(i) = a(i) + a(i+n)
      enddo
```

Figure 75　Width of Adjacent References are Determined at Runtime

If a programmer knows that the value of the variable n is -1 or 1, the HPF directives as shown in the example Figure 76 make it possible to parallelize the loop only with the efficient shift transfer.

1. Declare the shadow area explicitly with the SHADOW directive.
2. Perform the shift transfer before the loop with the REFLECT directive.
3. Specify with the ON-HOME-LOCAL directive that the loop can be executed without data transfers by selecting the array reference b(i) as the home array.

```
      subroutine sub(a,b,n)
      real a(100), b(100)
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE (BLOCK) ONTO p :: a, b
!HPF$ SHADOW (1) :: a
          :
!HPF$ REFLECT a

      do i=2,99
!HPF$   ON HOME(b(i)), LOCAL
        b(i) = a(i) + a(i+n)
      enddo
```

Figure 76　SHADOW Directive, REFLECT Directive, and ON-HOME-LOCAL Directive

The syntax of the SHADOW directive is shown in Figure 77. Note that when a dummy argument appears in the SHADOW directive, the same SHADOW directive should be specified to the corresponding actual argument. This is because when the shadow widths of a dummy argument and the corresponding actual argument are different, copy between them occurs to make the shadow widths match up.

!HPF$ SHADOW *a***(** <shadow width>,⋯ **)**

or

!HPF$ SHADOW (<shadow width>,⋯ **) ::** *a*,⋯

- *a* indicates the name of an array
- <shadow width> is *n* or *l* **:** *u*, where *n* is equivalent to *n* **:** *n*, which indicates the lower shadow width and upper shadow width, respectively. The shadow width must be a constant.

Figure 77　Syntax of the SHADOW Directive

The syntax of the REFLECT directive is as follows:

!HPF$ REFLECT [**(**<shadow width>,⋯**)**] [**::**] *a,*⋯

- *a* indicates the name of an array, which must appear in the SHADOW directive in the specification part of the scoping unit.
- When <shadow width>,⋯ is specified, the shift transfer is performed only on the specified part of the shadow area, which is called the partial REFLECT directive.

Figure 78 Syntax of the REFLECT Directive

4.3 Extended Intrinsic Procedures

This section describes extended intrinsic procedures supported by the HPF compiler.

4.3.1 Timing Procedures

- **HPF_LOCAL_WCLOCK(ATIME)**
  - Description.

    Each abstract processor returns the value of the wall-clock time on itself without synchronization. The values on different abstract processors are generally different. It can be used to know the load balance for a specific computation segment.
  - Class.

    subroutine.
  - Argument.

    ATIME

    must be an array of type double precision. It is an INTENT(OUT) argument. It must appear in the DISTRIBUTE directive that specifies the BLOCK distribution along all axes. The shape of it must be the same as that of the processor array onto which it is distributed.

    Each element of the array **ATIME** is assigned the current time on the corresponding abstract processor in seconds. The values are non-negative.

- Example.

```
      double precision t1(2), t2(2)
      integer a(100)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE (BLOCK) ONTO p :: t1, t2
            :
      call HPF_LOCAL_WCLOCK(t1)
      do i=1,100
        a(i) = i
      enddo
      call HPF_LOCAL_WCLOCK(t2)
```

The values of t2(1) – t1(1) and t2(2) - t1(2) are the elapsed times required for executing the loop on the abstract processors p(1) and p(2), respectively.

- **HPF_WCLOCK(TIME)**
  - Description

    It returns the wall-clock time. A representative abstract processor measures the wall-clock time after synchronization among all abstract processors and broadcasts the value to all abstract processors. The value is the same on all abstract processors, but the overhead for the synchronization and broadcast is involved. Therefore, it is suitable for measurement of relatively large computation segments.

  - Class.

    subroutine.

  - Argument

    TIME

    must be a scalar variable of type double precision. It is an INTENT(OUT) argument. The current wall-clock time is set in seconds. The value is non-negative.

➢ Example

```
double precision t1, t2
integer a(100)
      :
call HPF_WCLOCK(t1)
call sub()
call HPF_WCLOCK(t2)
```

The value t2 – t1 indicates the elapsed time in seconds required for executing the subroutine sub.

4.4 Clean up of Fortran Code

Fortran 95 programs can basically be compiled with the HPF compiler as they are as HPF is an extension of Fortran 95. However, when the following old Fortran features are used, code modifications are required.

- Multiple Variables share the same memory via EQUIVALENCE statements and COMMON statements (storage association).
- Order of array elements is assumed (sequence association). For example, the order of the elements of an array a with the shape (2,3) is a(1,1), a(2,1), a(1,2), a(2,2), a(1,3), and a(2,3).

These characteristics cannot be kept in HPF since arrays are divided and parts of them are mapped onto the distributed-memory separately. Mapped arrays are subject to the following constraints.

- Mapped arrays cannot appear in the EQUIVALENCE statement.
- Every COMMON block variable must have the same attributes such as the shape, type, and data mapping in all occurrences in a program in principle.
- The shapes of each actual argument and corresponding dummy argument must be the same in principle.

- When an actual argument is an array element (for example, a(1,2)), the corresponding dummy argument must not be an array (address passing. refer to Figure 79): That is, when an actual argument is an array element, the corresponding dummy argument must be a scalar variable.
- Assumed-size arrays, whose upper bound along the last axis is * like a(n,*), cannot be mapped.

```
real a(n,n)
do i=1,n
  call sub(a(1,i),n)
enddo
end

subroutine sub(a,n)
real a(n)
```

Figure 79 Address Passing (Not Allowed in HPF)

Before parallelizing existing Fortran programs, modify these descriptions as follows, and then insert HPF directives.

- Delete EQUIVLAENCE statements to mapped arrays. When only part of a large array is used, declare the array with the shape and type actually used using the features to determine shapes of arrays at runtime such as allocatable arrays or automatic arrays.
- Declare every common block variable so that it has the same attributes including data mapping in all occurrences in a program. It is helpful to declare each common block in an include file or module to prevent omission or error in the declaration.
- Declare actual arguments and corresponding dummy arguments so that they have the same shapes. The following modification can be required.
  - Each address passing as shown in Figure 79 must be modified into an array section actual argument as shown in Figure 80 to explicitly specify that an array is passed as an actual argument.

- Assumed-size arrays as shown in Figure 81 must be modified into explicit shape arrays as shown in Figure 82 or assumed-shape arrays.

```
real a(n,n)
do i=1,n
  call sub(a(:,i),n)
enddo
end

subroutine sub(a,n)
real a(n)
```

Figure 80　Array Section Actual Argument

```
real a(n,n)
call sub(a,n)
end

subroutine sub(a,n)
real a(n,*)
```

Figure 81　Assumed-Size Array

```
real a(n,n)
call sub(a,n)
end

subroutine sub(a,n)
real a(n,n)
```

Figure 82　Explicit Shape Array

It can sometimes be difficult or very troublesome to parallelize existing Fortran programs with HPF due to constrains described above. However, procedures that are not needed to

parallelize can be compiled with the HPF compiler without modifications using any of the following methods.

- Procedures that do not have mapped arrays and I/O can be compiled with the HPF compiler as they are.
- The EXTRINSIC procedure feature enables compilation of procedures as Fortran. Describe explicit interfaces such as interface blocks to declare the EXTRINSIC prefix to specify that Fortran procedures are referenced. Refer to subsection 4.2.4 for the EXTRINSIC feature.
- Create object files or archive files with the Fortran compiler and link HPF programs with them with the HPF compiler. This method is also available to call existing Fortran libraries from HPF programs.

Chapter5 Tuning and Debug

This chapter explains how to tune and debug HPF programs.

5.1 Tuning

5.1.1 Parallelization Information List

Parallelization information lists, which display parallelization and data transfer information by the HPF compiler, are generated with the HPF compiler option -Mlist2. The suffix of the parallelization information lists is .lst.

Figure 84 shows an example of the parallelization information list for the HPF program in Figure 83. The meanings of the marks in the parallelization information list is shown in Table 6.

```
        real :: a(100,100) = 0
!HPF$ DISTRIBUTE a(*,block)

        do i=1,99
          do j=1,100
            a(j,i) = a(j,i) + a(j,i-1)
          enddo
        enddo

        do j = 1,100
           do i = 1,100
              x = max(x,a(i,j))
           end do
        end do

        write(*,*)x
```

Figure 83 Example of an HPF Program

```
(     1)                          real :: a(100,100) = 0
(     2)                    !HPF$ DISTRIBUTE a(*,block)
(     3)
(     4) <S>-------------        do i=1,99
          COMM: SFT [a] [LINO: 5 in src.hpf]
(     5) <N>-------------          do j=1,100
(     6)  |                          a(j,i) = a(j,i) + a(j,i-1)
(     7)  +--------------          enddo
(     8)                       enddo
(     9)
          COMM: RED [x] [LINO: 10 in src.hpf]
          HOME: a(:,j)
(    10) <P>-------------        do j = 1,100
(    11)  |<I>-----------          do i = 1,100
(    12)  |                           x = max(x,a(i,j))
(    13)  |                      end do
(    14)  +--------------        end do
(    15)
(    16)                       write(*,*)x
(    17)                       end
```

Figure 84　Example of the Parallelization Information List

Table 6　Marks in the Parallelization Information List

Mark	Description
(　1)	Line number in the HPF source file
COMM: SFT **[**a**] [LINO:** 5 **in** src.hpf**]**	Data transfer is generated by the HPF compiler. The format is as follows: **COMM:** *Kind* **[***Variable name***] [LINO:** *Line number***]**

	, where *Kind* is one of the following: **RED** : Reduction **SFT** : Shift **CPY** : Copy of an array **G/S** : Gather/Scatter **SCL** : Data transfer for a scalar variable It is possible to obtain the list of data transfers by extracting the lines that contain the mark "**COMM:**" as follows, and check whether redundant data transfers are generated. %>grep "COMM:" src.lst Of the data transfers above, the marks **RED** and **SFT** are usually not problems, but elimination of the mark **CPY** can improve execution performance if possible. The marks **G/S** and **SCL** indicate very high-overhead data transfers in many cases, and should be eliminated by tuning the HPF program.
<S>	Loop is judged as non-parallelizable. It is possible to obtain the list of loops that are judged as non-parallelizable by extracting lines that contain the mark <S> as follows: %>grep "<S>" src.lst When a parallelizable loop is judged as non-parallelizable, inserting an INDEPENDENT directive can lead to the parallelization of the loop.
<N>	Loop is judged as parallelizable, but not parallelized. When no data transfer is generated for the loop, it

	is not a problem. Changing data mappings of arrays that appear in the loop can lead to the parallelization of the loop. It is possible to obtain the list of loops that are judges as parallelizable, but not parallelized by extracting the lines that contain the mark <N> as follows: %>grep "<N>" src.lst
<P>	Loop is parallelized by the HPF compiler.
<I>	Loop is judged as parallelizable. When the loop has reduction computation, the mark <R> is displayed instead of the mark <I>.
HOME: a(:,j)	Home array (base array for parallelization) for the immediately following loop nest.

Detailed Parallelization information lists, which display intermediate code by the HPF compiler in addition to parallelization and data transfer information, are generated with the HPF compiler option –Mlist3.

Figure 85 shows an example of the detailed parallelization information list for the HPF program in Figure 83.

```
(     9)
          COMM: RED [x] [LINO: 10 in src.hpf]
          HOME: a(:,j)
(    10) <P>------------          do j = 1,100
(    11)  |<I>-----------            do i = 1,100
(    12)  |                              x = max(x,a(i,j))
(    13)  |                           end do
(    14)  +--------------         end do
       .
       .        x$ind = x
       .        j$indl = a$sd(84)
       .        j$indu = a$sd(85)
       .        pghpf_saved_local_mode = pghpf_local_mode
       .        pghpf_local_mode = 1
       .!NEC$nosync
       .!NEC$shortloop
       .        do j = j$indl, j$indu
       .!NEC$nosync
       .            do i = 1, 100
       .                x$ind = max(x$ind,a(i,j))
       .            enddo
       .        enddo
       .        pghpf_local_mode = pghpf_saved_local_mode
       .        call pghpf_global_maxval(x$ind,a,125_8,pghpf_type(27),a$sd,
       .       +pghpf_type(26))
       .!       call .reduce_maxval(x$ind,a,125_8)
       .        x = x$ind
       .
```

Figure 85　Example of the Detailed Parallelization Information List

Please note that parallelization information marks are not displayed for array assignment statements.

Loop optimization information by the NEC Fortran compiler is also displayed at the right of parallelization information by the HPF compiler as shown Figure 86 by specifying the NEC Fortran compiler option –report-format or –report-all and HPF compiler option -Mlist2 or -Mlist3 at the same time, if the HPF compiler option –Mftn is not specified.

```
(     1)                              real :: a(100,100) = 0
(     2)                        !hpf$ distribute a(*,block)
(     3)
(     4) <S>+------------          do i=1,99
           COMM: SFT [a] [LINO: 5 in src.hpf]
(     5) <N>V------------            do j=1,100
(     6)   |                              a(j,i) = a(j,i) + a(j,i-1)
(     7)   +--------------           enddo
(     8)                               enddo
(     9)
           COMM: RED [x] [LINO: 10 in src.hpf]
           HOME: a(:,j)
(    10) <P>P------------          do j = 1,100
(    11)   |<I>V----------             do i = 1,100
(    12)   |                                x = max(x,a(i,j))
(    13)   |                            end do
(    14)   +--------------         end do
(    15)
(    16)                               write(*,*)x
(    17)                               end
```

Figure 86　Parallelization Information List with Loop Optimization Information

The meanings of the loop optimization information marks are the same as those in the NEC Fortran compiler format list. For example, the mark P indicates a shared-memory parallelized loop and the mark V a vectorized loop. Moreover, the marks including I (Inline expansion) and S (partial vectorization) are displayed at the left of the first column of source code lines. Refer to “Fortran Compiler User’s Guide” for details.

When a loop is divided into multiple loops by the HPF compiler, and the loops are optimized in various ways by the NEC Fortran compiler, the mark M is displayed.

5.1.2 Diagnostic Messages

Diagnostic messages are displayed with the HPF compiler option –Minfo. The diagnostic messages you should pay attention to are as follows:

- expensive communication
  High-Overhead data transfer is generated

- Array "*array name*" not aligned with home array; array copied
  The array "*array name*" is copied into a temporary area, which usually involves data transfer, because the data mapping of it does not match that of the base array of loop parallelization (home array).

- communication is generated: array copy
  An array is copied into a temporary area, which usually involves data transfer.

5.1.3 Use of the FTRACE Region Feature

Use of the FTRACE region feature in HPF requires the explicit interface that declares the subroutines FTRACE_REGION_BEGIN and FTRACE_REGION_END to be Fortran_LOCAL extrinsic procedures as shown in Figure 87.

```
interface
   extrinsic(Fortran_LOCAL) subroutine ftrace_region_begin(label)
   character(*) label
   end subroutine
   extrinsic(Fortran_LOCAL) subroutine ftrace_region_end(label)
   character(*) label
   end subroutine
end interface
```

Figure 87 Explicit Interface to Use the FTRACE Region Feature

Refer to "PROGINF/FTRACE User's Guide" for details of the performance profiler FTRACE.

5.1.4 Examples of Tuning of HPF Programs

This subsection describes typical tuning examples of HPF programs.

- Parallelization of a Loop Nest that Contains a Work Array

  The loop nest in Figure 88 is not automatically parallelized because the work array tmp is defined in multiple iterations of the loop do k.

```
      integer tmp(100),a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
        :
      do k = 2, nz - 1
        do j = 2, ny - 1
          do i = 1, 100
            tmp(i) = i
          enddo
          a(j,k) = tmp(i) + tmp(i+1)
        enddo
      enddo
      write(*,*)a
      end
```

Figure 88　Loop Nest that Contains a Work Array

Inserting the INDEPENDENT directive with the NEW clause for the work array tmp as shown in Figure 89 enables parallelization of the loop do k.

```
      integer tmp(100),a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
        :
!HPF$ INDEPENDENT, NEW(tmp,i,j)
      do k = 2, nz - 1
        do j = 2, ny - 1
          do i = 1, 100
            tmp(i) = i
          enddo
          a(j,k) = tmp(i) + tmp(i+1)
        enddo
      enddo
      write(*,*)a
      end
```

Figure 89　INDEPENDENT Directive with a NEW Clause for a Work Array

- Loop Fission

  In the loop nest in Figure 90, data transfer is needed for the array a or b, because the subscripts along the distributed axis of the left hand side of the assignment statements are different. Data transfer for a defined array involves higher overhead than that for a referenced array because allocation of a temporary area, copy of the value of the corresponding array to that of the temporary, and copy back from the temporary to the corresponding array are required.

```
      real a(10,10), b(10,10), c(10,10)
!HPF$ DISTRIBUTE (*,BLOCK) :: a, b, c
        :
      do j=1,9
        do i=1,99
          a(i+1,j) = -c(i+1,j+1)
          b(i,j+1) = c(i+1,j+1)
        enddo
      enddo
```

Figure 90　Subscripts along the Distributed Axis are Different.

Then, the loop fission as shown in Figure 91 to use only one subscript along the distributed axis of the left hand side enables efficient parallelization only with the shift transfer for the right hand side array c for the first loop nest.

```
      real a(10,10), b(10,10), c(10,10)
!HPF$ DISTRIBUTE (*,BLOCK) :: a, b, c
        :
      do j=1,9
        do i=1,9
          a(i+1,j) = -c(i+1,j+1)
        enddo
      enddo
      do j=1,9
        do i=1,9
          b(i,j+1) = c(i+1,j+1)
        enddo
      enddo
```

Figure 91　Loop Fission

- Inhibition of Data Transfers for Boundary Processing Loops

  Without HPF directives, inefficient data transfers are generated for boundary processing

loops as shown in Figure 92 that access only the elements at the end of a distributed axis of arrays, because all abstract processors take part in the execution.

```
      double precision a(100,100)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE a(*,BLOCK) ONTO p

      do i=1,100
        a(i,1) = a(i,2)
        a(i,100) = a(i,99)
      enddo
```

Figure 92　Boundary Processing Loop

The data transfers can be inhibited by inserting the ON-HOME-LOCAL directives as shown in Figure 93 to specify that only the abstract processers onto which the elements at the end of the distributed axis of arrays are mapped execute the statements

```
      double precision a(100,100)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE a(*,BLOCK) ONTO p

      do i=1,100
!HPF$   ON HOME(a(:,1)), LOCAL
        a(i,1) = a(i,2)
!HPF$   ON HOME(a(:,100)), LOCAL
        a(i,100) = a(i,99)
      enddo
```

Figure 93　ON-HOME-LOCAL Directive to Boundary Processing

- Loop Peeling for boundary Processing

  The boundary processing under the IF construct in the loop nest as shown in Figure 94 can inhibit parallelization of the loop nest or lead to inefficient data transfers.

```
      parameter(n=100)
       real a(n,n),b(n,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: a,b

       do j=1,n
         if(j.eq.n)then
           do i=1,n
             a(i,j) = 0.9
           enddo
         else
           do i=1,n
             a(i,j) = b(i,j) + b(i,j+1)
           enddo
         endif
       enddo
```

Figure 94　Loop that Contains Boundary Processing

Efficient parallel execution is possible by splitting the boundary processing as a distinct loop and inserting the ON-HOME-LOCAL directive construct to it as shown in Figure 95.

```
      parameter(n=100)
      real a(n,n),b(n,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: a,b

       do j=1,n-1
         do i=1,n
           a(i,j) = b(i,j) + b(i,j+1)
         enddo
       enddo

       j=n
!HPF$ ON HOME(a(:,j)), NEW(i), LOCAL(a) BEGIN
       do i=1,n   ! Boundary processing loop
         a(i,j) = 0.9
       enddo
!HPF$ END ON
```

Figure 95　Loop Peeling of Boundary Processing

- Subscripts in Boundary Processing

  When the subscript in the distributed axis of arrays is constant as shown in Figure 96, inefficient data transfers can occur because the subscript does not correspond to the DO variable.

```
      parameter(n=100)
      real a(n,n),b(n,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: a,c

      do j=1,n
        if(j.eq.2)then
          do i=1,n
            a(i,1) = a(i,1) - b(i)*c(i,1)
          enddo
        endif
      enddo
```

Figure 96　Constant Subscript in the Distributed Axis

Rewrite the subscript along the distributed axis using a linear expression of the DO variable as shown in Figure 97.

```
!HPF$ DISTRIBUTE (*,BLOCK) :: a,c
      do j=1,n
        if(j.eq.2)then
          do i=1,n
            a(i,j-1) = a(i,j-1) - b(i)*c(i,j-1)
          enddo
        endif
      enddo
```

Figure 97　Subscript Using a Linear Expression of the DO Variable

- Actual Arguments with Different Data Mappings

  When a procedure is invoked with actual arguments with different data mappings as shown in Figure 98, data transfers occur in some invocations of the procedure, which can lead to poor performance.

```
      double precision a(100,100),b(100,100)
!HPF$DISTRIBUTE a(*,BLOCK)
      call sub(a)
      call sub(b)
      end
```

Figure 98　Actual Arguments with Different Data Mappings

In such cases, it is possible to improve the performance by making copies of the procedure so that the dummy arguments of each procedure have the same data mappings as the corresponding actual arguments has as shown in Figure 99. This kind of optimization is called procedure cloning.

```
      double precision a(100,100),b(100,100)
!HPF$DISTRIBUTE a(*,BLOCK)
      call sub1(a)
      call sub2(b)
      end

      subroutine sub1(a)
      double precision a(100,100)
!HPF$DISTRIBUTE a(*,BLOCK)
        :
      end

      subroutine sub2(b)
      double precision b(100,100)
        :
      end
```

Figure 99　Copies of a Procedure Corresponding to Data Mappings of the Argument

- Data Mapping of Dummy Arguments

  In the example Figure 100, the data transfer to match the data mapping of the actual argument with that of the corresponding dummy argument occurs when the subroutine sub is invoked.

```
      program main
      real a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
      call sub(a)
      end

      subroutine sub(a)
      real a(100,100) ! Not Mapped
```

Figure 100　Data Mappings of the Actual Argument and Dummy Argument Differ

It is possible to check whether data transfers at invocations of procedures occur by executing with the HPF runtime option -hpf –commmsg, as the warning message like the following is output for data transfers across procedure boundaries.

"a": Communication occurs at procedure boundary PROG=sub ELN=7 Called from main ELN=4

Execution performance is improved by matching the data mappings of dummy arguments with those of the corresponding actual arguments as shown in Figure 101.

```
      program main
      real a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
      call sub(a)
      end

      subroutine sub(a)
      real a(100,100)
!HPF$ DISTRIBUTE a(*,BLOCK)
```

Figure 101　Explicit Data Mapping of the Dummy Argument

- I/O

  Element by element I/O as shown in Figure 102 is not efficient.

```
      write(13,*) (a(i), b(i), i=1, n)
```

Figure 102　Element by Element I/O

Read or write whole arrays using unformatted I/O as shown in Figure 103 especially when sizes of arrays read or written are large.

```
      write(13,*) a, b
```

Figure 103　I/O of Whole Arrays

- Performance Improvement of Inputs Utilizing Fortran

  When inputting data in HPF, only one process inputs data, and then transfers it to the processes onto which it is mapped. Therefore, if an array that is not mapped is read element by element as shown in Figure 104, the read and data transfer of each element are performed repeatedly, which results in significant performance degradation.

```
real a(n,n,n)
  :
open(10,file=' data' )
do j=1,n
  do i=1,n
    read(10,*) (a(i,j,k), k=1,n)
  enddo
enddo
close(10)
```

Figure 104 Inefficient Element-Wise Input of Unmapped Data

In this case, performance improvement by elimination of the data transfer is possible by making the input part an independent Fortran subroutine as shown in Figure 105 and referencing it from HPF as a Fortran_LOCAL extrinsic procedure (Figure 106), which every process invokes, because every process performs the input without the data transfer.

```
subroutine read_all_fortran(a,n,n,n)
real a(n,n,n)
integer n,n,n
open(10,file=' data' )
do j=1,n
  do i=1,n
    read(10,*) (a(i,j,k), k=1,n)
  enddo
enddo
close(10)
end subroutine
```

Figure 105 Fortran Subroutine That Consists of the Input Part

```
real a(n,n,n)
interface
   extrinsic(Fortran_LOCAL) subroutine read_all_fortran(a,n,n,n)
   read a(n,n,n)
   integer n,n,n
   end subroutine
end interface
  :
call read_all_fortran(a,n,n,n)
```

Figure 106 Input Performance Improvement Using Fortran_LOCAL Extrinsic

On the other hand, when the input data is mapped as shown Figure 107, data transfer is inevitable. However, performance improvement is still possible by making the input part an independent Fortran subroutine as shown in Figure 108 and referencing it from HPF as a Fortran_SERIAL extrinsic procedure (Figure 109), which only one process invokes, because the data transfer is performed in a collective manner after the input.

```
      real a(n,n,n)
!HPF$ DISTRIBUTE a(*,*,BLOCK)
        :
      open(10,file=' data' )
      do j=1,n
        do i=1,n
          read(10,*) (a(i,j,k), k=1,n)
        enddo
      enddo
      close(10)
```

Figure 107 Inefficient Element-Wise Input of Mapped Data

```
subroutine read_one_fortran(a,n,n,n)
real a(n,n,n)
integer n,n,n
open(10,file=' data' )
do j=1,n
  do i=1,n
    read(10,*) (a(i,j,k), k=1,n)
  enddo
enddo
close(10)
end subroutine
```

Figure 108 Fortran Subroutine That Consists of the Input Part

```
      real a(n,n,n)
!HPF$ DISTRIBUTE a(*,*,BLOCK)
      interface
         extrinsic(Fortran_SERIAL) subroutine read_one_fortran(a,n,n,n)
         read a(n,n,n)
         integer n,n,n
         end subroutine
      end interface
         :
      call read_one_fortran(a,n,n,n)
```

Figure 109 Input Performance Improvement Using Fortran_SERIAL Extrinsic

To use Fortran extrinsic procedures, compile Fortran procedures with the Fortran compiler with the -c option first to generate Fortran object files, and then link the Fortran object files at the compilation of HPF procedures as shown in Figure 110.

```
%> nfort -c read_all_fortran.f
%> ve-hpf hpfsourcefile.hpf read_all_fortran.o -o hpfexecfile
```

Figure 110 Compilation of an HPF Program That Invokes Fortran Procedures

- Performance Improvement of Outputs Utilizing Fortran

When outputting data in HPF, the data is transferred to one process, and then the process output it. Therefore, if a mapped array is written element by element as shown in Figure 111, the data transfer and output of each element are performed repeatedly, which results in significant performance degradation.

```
      real a(n,n,n)
!HPF$ DISTRIBUTE a(*,*,BLOCK)
         :
      open(10,file=' data' )
      do j=1,n
        do i=1,n
          write(10,*) (a(i,j,k), k=1,n)
        enddo
      enddo
      close(10)
```

Figure 111 Inefficient Element-Wise Output of Mapped Data

In this case, performance improvement is possible by making the output part an independent Fortran subroutine as shown in Figure 112 and referencing it from HPF as a Fortran_SERIAL extrinsic procedure (Figure 113), which only one process invokes, because the data transfer is performed in a collective manner before the output.

```
subroutine write_one_fortran(a,n,n,n)
real a(n,n,n)
integer n,n,n
open(10,file=' data' )
do j=1,n
  do i=1,n
    write(10,*) (a(i,j,k), k=1,n)
  enddo
enddo
close(10)
end subroutine
```

Figure 112 Fortran Subroutine That Consists of the Output Part

```
      real a(n,n,n)
!HPF$ DISTRIBUTE a(*,*,BLOCK)
      interface
        extrinsic(Fortran_SERIAL) subroutine write_one_fortran(a,n,n,n)
        read a(n,n,n)
        integer n,n,n
        end subroutine
      end interface
         :
      call write_one_fortran(a,n,n,n)
```

Figure 113 Output Performance Improvement Using Fortran_SERIAL Extrinsic

- Nesting Order of Loops that perform reduction computation

  The do k loop in Figure 114 performs reduction computation on the array a.

```
      double precision w(100,100,100),a(100,100)
!HPF$ DISTRIBUTE w(*,*,block)

      do k=1,100
        do j=1,100
          do i=1,100
            a(i,j) = a(i,j) + w(i,j,k)
          enddo
        enddo
      enddo
```

Figure 114　Loop Nest that Performs Reduction Computation

When you use the shared-memory parallelization by the NEC Fortran compiler with the compiler option –mparallel in addition to the distributed-memory parallelization by the HPF compiler, the outermost loop should be the perfectly parallel loop without reduction dependencies as shown in Figure 115 for efficient shared-memory parallelization. In this case, the HPF compiler distributed-memory-parallelizes the do k loop that corresponds to the distributed axis of the array w, and the NEC Fortran compiler shared-memory-parallelizes the outermost do j loop.

```
      double precision w(100,100,100),a(100,100)
!hpf$ DISTRIBUTE w(*,*,BLOCK)

      do j=1,100
        do k=1,100
          do i=1,100
            a(i,j) = a(i,j) + w(i,j,k)
          enddo
        enddo
      enddo
```

Figure 115　The Outermost Loop Should be Perfectly Parallelizable

5.2 An Easy and Simple Way of Developing HPF Programs

The HPF compiler option –Mautodist makes it possible to compile serial Fortran programs as HPF programs in which all arrays are distributed along the last axis with the BLOCK distribution. Also, the suboptions **=all**[**:***b*] and **=rank***?*[**:***b*] enable more detailed specification of data mappings of arrays. The HPF compiler option –Mlist2 generates the parallelization information lists for the HPF programs in which the data mappings are specified and you can check whether each loop is parallelized and where and what data transfers are generated.
This section explains how to parallelize the Fortran program “sample.F” shown in Figure 116, Figure 117, and Figure 118 with HPF using these HPF compiler options.

```
module param
parameter(n=1023,maxiter=10)
end module
```

Figure 116 Sample Program: Module

```
      program sample
      use param
      double precision a(n,n),b(n,n),c(n,n),sum,ap
      integer idxx(n),idxy(n),ix,iy,i,j,iter
      data ap/0.0d0/

      do i=1,n
        idxx(i) = n - i + 1
        idxy(i) = n - i + 1
      enddo
      do j=2,n-1
        do i=1,n
          b(i,j) = 1.0d0
          c(i,j) = 1.0d0
        enddo
      enddo
      call bound(b)
      call bound(c)

      do iter=1,maxiter
! main loop
        do j=2,n-1
          do i=2,n-1
            ix = idxx(i)
            iy = idxy(j)
            a(i,j)=(b(i,j)+b(i-1,j)+b(i+1,j)                                    &
     &    +b(i,j-1)+b(i,j+1))*0.2d0*c(ix,iy)+ap
          enddo
        enddo
        do i=1,n
          a(1,i) = a(2,i)
          a(n,i) = a(n-1,i)
        enddo
        call bound(a)
        do j=1,n
          do i=1,n
            ix = idxx(i)
            b(ix,j)=a(i,j)*c(i,j)
            ap = ap + a(i,j)
          enddo
        enddo
      enddo

      write(*,*)ap
      end
```

Figure 117　Sample Program: Main Program

```
      subroutine bound(dummy)
      use param
      double precision dummy(n,n)
      do i=1,n
        dummy(i,1) = dummy(i,2)
        dummy(i,n) = dummy(i,n-1)
      enddo
      end
```

Figure 118　Sample Program: Subroutine Bound

First of all, compile the program with the HPF compiler options -Mautodist and –Mlist. Then the parallelization information list “sample.lst” is generated for the HPF program in which all the arrays are distributed along the last axis with the BLOCK distribution.

Figure 119 shows the parallelization information list for the main program. Focusing on the mark “COMM:”, which indicates data transfer is generated, you can find that a lot of data transfers are generated for lines 26 and 40 and the program is inefficiently parallelized. You must not execute the program as it is because execution performance of an inefficient distributed-memory parallel program can be hundreds or thousands times slower than the original serial program. The following describes how to improve the program.

```
(    11) <I>------------         do i=1,n
(    12)                            idxx(i) = n - i + 1
(    13)                            idxy(i) = n - i + 1
(    14)                          enddo
(    15) <I>------------         do j=2,n-1
(    16) <I>------------           do i=1,n
(    17)                              b(i,j) = 1.0d0
(    18)                              c(i,j) = 1.0d0
(    19)                           enddo
(    20)                          enddo
(    21)                          call bound(b)
(    22)                          call bound(c)
(    23)
(    24) <S>------------         do iter=1,maxiter
(    25)                     ! main loop
         COMM: SFT [b] [LINO: 26 in sample.F]
         COMM: CPY [idxx] [LINO: 26 in sample.F]
         COMM: G/S [c] [LINO: 26 in sample.F]
         HOME: idxy(j)
(    26) <P>------------           do j=2,n-1
(    27)  |<I>----------             do i=2,n-1
(    28)  |                            ix = idxx(i)
(    29)  |                            iy = idxy(j)
(    30)  |                            a(i,j)=(b(i,j)+b(i-1,j)+b(i+1,j)
(    31)  |                       &    +b(i,j-1)+b(i,j+1))*0.2d0*c(ix,iy)+ap
(    32)  |                            enddo
(    33)  +--------------           enddo
         HOME: a(:,i)
(    34) <P>------------           do i=1,n
(    35)  |                            a(1,i) = a(2,i)
(    36)  |                            a(n,i) = a(n-1,i)
(    37)  +--------------           enddo
(    38)                            call bound(a)
(    39) <S>------------           do j=1,n
         COMM: CPY [idxx] [LINO: 40 in sample.F]
         COMM: CPY [a] [LINO: 40 in sample.F]
         COMM: SCL [c] [LINO: 40 in sample.F]
         COMM: SCL [a] [LINO: 40 in sample.F]
(    40) <S>------------             do i=1,n
         COMM: RED [ap] [LINO: 41 in sample.F]
(    41)                                ix = idxx(i)
(    42)                                b(ix,j)=a(i,j)*c(i,j)
(    43)                                ap = ap + a(i,j)
(    44)                              enddo
(    45)                            enddo
(    46)                          enddo
(    47)
(    48)                          write(*,*)ap
(    49)                          end
```

Figure 119　Parallelization Information List: Main Program

Figure 120 shows the data transfers for line 26, in which the marks “HOME: idxy(j)” and “<P>” indicate that the do j loop is parallelized based on the home array idxy(j), which is distributed along the last axis. (The do i loop in line 27 is not parallelized though the HPF compiler has judged it as parallelizable as the mark “<I>” shows.)

```
        COMM: SFT [b] [LINO: 26 in sample.F]
        COMM: CPY [idxx] [LINO: 26 in sample.F]
        COMM: G/S [c] [LINO: 26 in sample.F]
        HOME: idxy(j)
(    26) <P>-------------          do j=2,n-1
(    27)  |<I>-----------            do i=2,n-1
(    28)  |                           ix = idxx(i)
(    29)  |                           iy = idxy(j)
(    30)  |                           a(i,j)=(b(i,j)+b(i-1,j)+b(i+1,j)
(    31)  |                    &    +b(i,j-1)+b(i,j+1))*0.2d0*c(ix,iy)+ap
```

Figure 120　Data Transfers for Line 26

Of the three data transfers, the first one marked with “COMM: SFT [b]” is relatively efficient shift transfer, which is usually not a problem. The second one marked with “COMM: CPY [idxx]” is generated because the array idxx, which is distributed along the last axis, is accessed with the subscript i (idxx(i)), which does not correspond to the parallelized loop do j. Insert the DISTRIBUTE directive not to distribute the array idxx as shown in Figure 121 because the axis which is accessed with the subscript that does not use a DO variable of a parallelized do loop should not be distributed.

```
!HPF$ DISTRIBUTE (*) :: idxx
```

Figure 121　DISTRIBUTE Directive Not to Distribute the Rank One Array IDXX

The third one marked with ”COMM: G/S [c]” is generated because the array c, which is distributed along the last axis, is accessed with indirect subscripts ix and iy (c(ix,iy)) in the

parallelized loop do j. Insert the DISTRIBUTE directive not to distribute the array c as shown in Figure 122 because the subscripts of the array c do not use the DO variable of the parallelized loop do j.

```
!HPF$ DISTRIBUTE (*,*) :: c
```

Figure 122　DISTRIBUTE Directive Not to Distribute the Rank Two Array C

Figure 123 shows the data transfers for line 40, which are generated between the loops do j and do i, which are not parallelized as the mark “<S>” shows.

```
(    39) <S>-------------          do j=1,n
         COMM: CPY [idxx] [LINO: 40 in sample.F]
         COMM: CPY [a] [LINO: 40 in sample.F]
         COMM: SCL [c] [LINO: 40 in sample.F]
         COMM: SCL [a] [LINO: 40 in sample.F]
(    40) <S>-------------            do i=1,n
         COMM: RED [ap] [LINO: 41 in sample.F]
(    41)                               ix = idxx(i)
(    42)                               b(ix,j)=a(i,j)*c(i,j)
(    43)                               ap = ap + a(i,j)
```

Figure 123　Data Transfers for Line 40

The loop do j, which performs the reduction computation (sum) on the scalar variable ap, is actually parallelizable, but the HPF compiler cannot judge it as parallelizable automatically. Therefore, insert the INDEPENDENT directive with the REDUCTION clause for the variable ap as shown in Figure 124. The NEW clause for the work variable ix and DO variable for the inner do loop i should also be specified.

```
!HPF$ INDEPENDENT, NEW(ix,i), REDUCTION(ap)
        do j=1,n
          do i=1,n
            ix = idxx(i)
            b(ix,j)=a(i,j)*c(i,j)
            ap = ap + a(i,j)
          enddo
        enddo
```

Figure 124　INDEPENDENT Directive with a REDUCTION Clause

At this point, compile the program with the HPF compiler options –Mautodist and –Mlist2 again. Figure 125 shows the parallelization information list “sample.lst” for the main program. You can find that the main loop nests in the program are parallelized as the mark “<P>” shows only with efficient shift transfer marked with “COMM: SFT [b]” and reduction transfer “COMM: RED [ap]”.

```
(    10)                    !HPF$ DISTRIBUTE (*) :: idxx
(    11)                    !HPF$ DISTRIBUTE (*,*) :: c
(    12)
(    13) <I>-------------         do i=1,n
(    14)                             idxx(i) = n - i + 1
(    15)                             idxy(i) = n - i + 1
(    16)                          enddo
(    17) <I>-------------         do j=2,n-1
(    18) <I>-------------           do i=1,n
(    19)                               b(i,j) = 1.0d0
(    20)                               c(i,j) = 1.0d0
(    21)                            enddo
(    22)                          enddo
(    23)                          call bound(b)
(    24)                          call bound(c)
(    25)
(    26) <S>-------------         do iter=1,maxiter
(    27)                       ! main loop
         COMM: SFT [b] [LINO: 28 in sample.F]
         HOME: idxy(j)
(    28) <P>-------------           do j=2,n-1
(    29)  |<I>-----------             do i=2,n-1
(    30)  |                             ix = idxx(i)
(    31)  |                             iy = idxy(j)
(    32)  |                             a(i,j)=(b(i,j)+b(i-1,j)+b(i+1,j)
(    33)  |                        &    +b(i,j-1)+b(i,j+1))*0.2d0*c(ix,iy)+ap
(    34)  |                             enddo
(    35)  +--------------          enddo
         HOME: a(:,i)
(    36) <P>-------------           do i=1,n
(    37)  |                             a(1,i) = a(2,i)
(    38)  |                             a(n,i) = a(n-1,i)
(    39)  +--------------          enddo
(    40)                         call bound(a)
(    41)                !HPF$ INDEPENDENT, NEW(i,ix), REDUCTION(ap)
         COMM: RED [ap] [LINO: 42 in sample.F]
         HOME: b(:,j)
(    42) <P>-------------           do j=1,n
(    43)  |<S>-----------             do i=1,n
(    44)  |                             ix = idxx(i)
(    45)  |                             b(ix,j)=a(i,j)*c(i,j)
(    46)  |                             ap = ap + a(i,j)
(    47)  |                           enddo
(    48)  +--------------          enddo
(    49)                          enddo
(    50)
(    51)                          write(*,*)ap
(    52)                          end
```

Figure 125　Parallelization Information List after Insertion of HPF Directives

Then check data transfers at procedure invocations, which are not displayed in the parallelization information list. The actual array arguments b, c, and a are passed to the procedure bound, which is referenced three times in the main program. The arrays a and b are distributed along the last axis with the HPF compiler option –Mautodist, whereas the array c is not distributed because of the explicit DISTRIBUTE directive. Therefore, data transfer occurs in any of the invocations of the procedure bound. To prevent the data transfers at procedure invocations, copy the procedure as shown in Figure 126 (procedure cloning) so that the actual arguments and corresponding dummy arguments always have the same data mapping.

```
      subroutine bound(dummy)
      use param
      double precision dummy(n,n) ! Distribute with the option –Mautodist
      do i=1,n
        dummy(i,1) = dummy(i,2)
        dummy(i,n) = dummy(i,n-1)
      enddo
      end

      subroutine bound_nodist(dummy)
      use param
      double precision dummy(n,n)
!HPF$ DISTRIBUTE (*,*) :: dummy    ! Not distribute
      do i=1,n
        dummy(i,1) = dummy(i,2)
        dummy(i,n) = dummy(i,n-1)
      enddo
      end
```

Figure 126　Copy of a Procedure (Procedure Cloning)

Then replace the reference of the procedure bound that has the non-mapped actual argument with that of the copied procedure bound_nodist.

```
call bound(c)
```

↓

```
call bound_nobound(c)
```

At this point, compile the program with the HPF compiler options –Mautodist and –Mlist2. Figure 127 and Figure 128 show the parallelization information list for the procedure bound and bound_nodist, respectively.

```
(    54)                    subroutine bound(dummy)
(    55)                    use param
(    56)                    double precision dummy(n,n)
         COMM: SFT [dummy] [LINO: 57 in sample.F]
         COMM: SFT [dummy] [LINO: 57 in sample.F]
(    57) <N>-------------       do i=1,n
(    58)  |                      dummy(i,1) = dummy(i,2)
(    59)  |                      dummy(i,n) = dummy(i,n-1)
(    60)  +--------------       enddo
(    61)                    end
```

Figure 127　Parallelization Information List: Subroutine Bound

```
(    62)                      subroutine bound_nodist(dummy)
(    63)                      use param
(    64)                      double precision dummy(n,n)
(    65)                    !HPF$ DISTRIBUTE (*,*) :: dummy
(    66) <N>----------          do i=1,n
(    67)  |                     dummy(i,1) = dummy(i,2)
(    68)  |                     dummy(i,n) = dummy(i,n-1)
(    69)  +----------          enddo
(    70)                     end
```

Figure 128　Parallelization Information List: Subroutine Bound_nodist

The parallelization will not be inefficient as it is because data transfers generated for these procedures are only efficient shift transfers for line 57 of the subroutine bound. However, these data transfers can be eliminated by inserting the ON-HOME-LOCAL directives as shown in Figure 129 so that only abstract processors onto which the elements at both ends of the array dummy are mapped execute the assignment statements because the loop do i performs the boundary processing along the second axis of the array dummy, which is distributed with the HPF compiler option -Mautodist.

```
      subroutine bound(dummy)
      use param
      double precision dummy(n,n) ! Distribute with the option -Mautodist
      do i=1,n
!HPF$  ON HOME(dummy(:,1)), LOCAL
        dummy(i,1) = dummy(i,2)
!HPF$  ON HOME(dummy(:,n)), LOCAL
        dummy(i,n) = dummy(i,n-1)
      enddo
      end
```

Figure 129　ON-HOME-LOCAL Directives to Boundary Processing

Finally, the HPF program “sample.hpf.src” is generated by compiling the program with the

HPF compiler options -Mautodist and –Mhpfout.

5.3 Debug

This section describes bugs that frequently appear in HPF programs and how to detect and fix them.

It is possible to execute HPF programs as serial Fortran programs by compiling them using the NEC Fortran compiler. Therefore, you should confirm that the programs run without problems before executing them as HPF programs.

The following subsections describe possible causes of problems when HPF programs do not run though they run as serial Fortran programs.

5.3.1 Inconsistency between Actual and Dummy Arguments

The shapes and types of actual arguments and corresponding dummy arguments must be the same in principle in HPF. Therefore the following descriptions that often appear in old FORTRAN programs are not allowed.

- Array Element Actual Arguments Associated with Dummy Array Arguments

```
real a(100,100),b(100,100)
do i=1,100
  call sub(a(1,i),b(1,i))   !   Array element actual arguments
enddo
end

subroutine sub(a,b)
real a(100),b(:)              ! Dummy array arguments
```

Figure 130 Array Element Actual Arguments and Dummy Array Arguments

The arguments as shown in Figure 130 cause runtime errors with the following error messages and abnormal termination of the programs.

- When a dummy argument is not an assumed-shape array

  "a": Nonsequential dummy array is associated with array element or scalar actual. PROG=sub ELN=8

- When a dummy argument is an assumed-shape array.

  "b": Assumed-shape dummy array is associated with array element or scalar actual. PROG=sub ELN=8

When you want to pass part of arrays as actual arguments, use array sections as shown in Figure 131.

```
real(10) a(100,100),b(100,100)
do i=1,100
  call sub(a(:,i),b(:,i)) ! Array section actual arguments
enddo
end

subroutine sub(a,b)
real a(100),b(:)
```

Figure 131　Array Section Actual Argument

- Mismatch in Shapes of Actual Arguments and Corresponding Dummy Arguments

  The shape of an actual argument must be the same as that of the corresponding dummy argument in HPF.

```
real a(10000),b(10000)
n = 100
call sub(a,b,n)
end

subroutine sub(a,b,n)
real a(n,n),b(n)
```

Figure 132　Shapes of Actual Arguments and Dummy Arguments Differ

The arguments as shown in Figure 132 cause runtime errors with the following error messages and abnormal termination of the programs.

- When ranks of actual arguments and corresponding dummy arguments differ

  "a": Dummy argument rank differs from actual. PROG=sub ELN=7

- When extents along an axis differ between actual arguments and corresponding dummy arguments

  "b": Dummy array shape differs from actual in dim 1. PROG=sub ELN=7

When you want to determine sizes of arrays at runtime, use allocatable arrays as shown in Figure 133.

```
real, allocatable :: a(:,:) ! Allocatable array

n = 100
allocate(a(n,n))
```

Figure 133　Allocatable Array

Automatic arrays as shown in Figure 134 are also useful for data used within a procedure.

```
      subroutine sub(n)
      real :: a(n,n)      ! Automatic array
!HPF$ DISTRIBUTE (*,BLOCK) :: a
```

Figure 134　Automatic Array

When you want to determine sizes of arrays declared in a procedure at the first invocation and use the data areas thereafter, declare allocatable arrays with the SAVE attribute as shown in Figure 135 and allocate them at the first invocation.

```
      subroutine sub(n)
      integer :: iflag = 0
      real, save, allocatable :: a(:,:) ! Allocatalbe array witht the SAVE attribute
!HPF$ DISTRIBUTE a(*,BLOCK)
      if(iflag.eq.0)then
        allocate(a(n,n))
        iflag = 1
      endif
```

Figure 135　Allocation at the First Invocation

5.3.2 Inconsistency in Common Variables

The number of variables, and type, shape, and data mapping of each variable in every common block must be identical in an HPF program in principle.

The following descriptions are not allowed.

- The number of variables in a common block differs across procedures

```
        subroutine sub1()
        common /com/a(100,100),b(100,100)
!HPF$ DISTRIBUTE (*, BLOCK) :: a,b
            :
        end

        subroutine sub2()
        common /com/a(100,100)   !   Array b is not declared.
!HPF$ DISTRIBUTE (*,BLOCK) :: a
```

Figure 136　The Number of Common Block Variables Differs

- Data Mappings of Common Block Variables Differ across Procedures

```
        subroutine sub1()
        common /com/a(100,100)
!HPF$ DISTRIBUTE (*,BLOCK) :: a
            :
        end

        subroutine sub2()
        common /com/a(100,100)     ! No data mapping
```

Figure 137　Data Mapping of a Common Block Variable Differs

It is possible to detect these errors at runtime by compiling HPF programs with the HPF compiler option –Mcommonchk. When inconsistencies in common blocks in an HPF program are detected, the following error messages are output and the program terminates abnormally.

- Inconsistency in the number of common block variables

  Inconsistency detected in the number of components of common block between sub1 and sub2 : /com/ PROG=sub2

- Inconsistency in data mappings of common block variables

  Inconsistency detected in the number of explicitly mapped arrays of common block between sub1 and sub2 : /com/ PROG=sub2

Note that this option must be specified to all procedures that constitute an HPF executable program. Also, this option cannot be used with the HPF compiler option -Mnoentry or –Mnoerrline. When used, only the option specified last is effective.

5.3.3 Accesses out of Declared Bounds

Accesses out of declared bounds of arrays as shown in Figure 138 are not allowed in HPF programs.

```
      program main
      real a(100,100)
!HPF$ DISTRIBUTE a(BLOCK, *)
      do i=1,10000
        a(i,1) = i
      enddo
```

Figure 138 Accesses out of the Declared Bounds of an Array

It is possible to detect the accesses out of bounds at runtime by compiling HPF programs with the HPF compiler option –Msubchk. When the accesses out of bounds are detected, the following error message is output.

"a" is accessed out of declared bounds along 1st dim. PROG=main ELN=5

The code to detect the accesses out of bounds is generated so that vectorization and parallelization are not inhibited as much as possible, but can still cause performance degradation.
Note that this option cannot be used with the HPF compiler option -Mnoentry or –Mnoerrline. When used, only the option specified last is effective.

5.3.4 Wrong INDEPENDENT Directives

The loop nest in Figure 139 has the loop-carried dependency and is not parallelizable because the value of the variable l that is defined in the previous iteration is referenced.

```
        l=0
!HPF$   INDEPENDENT,NEW(I,J)   ! Wrong
        do i=1,n
          do j=1,n
            l = l+1
            a(j,i) = l
          enddo
        enddo
```

Figure 139 INDEPENDENT Directive to a Non-parallelizable Loop

The HPF compiler ignores all INDEPENDENT directives and performs only automatic parallelization by specifying the HPF compiler option –Mnoindependent. When this option enables correct program execution, the program can contain wrong INDEPENDENT directives.

It is useful to check loops that are not judged as parallelizable automatically referring to parallelization information lists for finding wrong INDEPENDENT directives.

Appendix A Syntax of HPF Directives

A.1 Directives in the Specification Part

A.1.1 DISTRIBUTE Directive

In the case of specifying a processor arrangement

!HPF$ DISTRIBUTE *a* **(** **<**distribution-format>,⋯ **)** **ONTO** *p*

or

!HPF$ DISTRIBUTE (**<**distribution-format>,⋯ **)** **ONTO** *p* **::** *a*,⋯

- *a* indicates the name of an array or template
- *p* indicates the name of a processor arrangement
- **<**distribution-format> is *****, **BLOCK**[**(**<expression>**)**], **GEN_BLOCK(***map***)**, or **CYCLIC**[**(**<expression>**)**]
  - ***** specifies that the corresponding axis of the array or template is not distributed.
  - **BLOCK** specifies that the corresponding axis of the array or template is distributed evenly. The width of the distribution can be specified with the optional **(**<expression>**)**. The width is calculated as follows by default:

    (Extent along the corresponding axis of the array or template - 1)/(Extent of the corresponding axis of the processor arrangement)
  - **GEN_BLOCK** specifies that the corresponding axis of the array or template is distributed unevenly. **(***map***)** specifies the number of array elements distributed onto each element along the corresponding axis of the processor arrangement. The values of the one-dimensional array *map* must be defined in advance.
  - **CYCLIC** specifies that the corresponding axis of the array or template is distributed in a round-robin fashion. **(**<expression>**)** specifies the width of the distribution. When the width of the distribution is omitted, the width is 1.

In the case of not specifying a processor arrangement

!HPF$ DISTRIBUTE *a* **(** **<**distribution-format>,⋯ **)**

or

!HPF$ DISTRIBUTE (**<**distribution-format>,⋯ **)** **::** *a*,⋯

A.1.2 TEMPLATE Directive

!HPF$ TEMPLATE *t* **(** **<**>,··· **)**

or

!HPF$ TEMPLATE (**<**>,··· **) ::** *t*,···

- *t* indicats a template
- <> indicates bounds along each axis of templates

A.1.3 PROCESSORS Directive

!HPF$ PROCESSORS *p* **(** <>,··· **)**

or

!HPF$ PROCESSORS (<>,··· **) ::** *p*,···

- *p* indicates the name of a processor arrangement
- **<**> indicates bounds along each axis of a processor array. For example, in the following PROCESSORS directive:

  !HPF$ PROCESSORS p(n1,n2)

  The number of abstract processers is the same as the size of the processor array p, n1*n2, and the rank of the processors array, 2, is equal to the number of distributed axes of arrays.

A.1.4 ALIGN Directive

!HPF$ ALIGN *a* **(** **<***i*>,⋯ **) WITH** *t***(** **<***f(i)*>,⋯ **)**

or

!HPF$ ALIGN (**<***i*>,⋯ **) WITH** *t***(** **<***f(i)*>,⋯ **) ::** *a*,⋯

- *a* indicates the name of an array
- *t* indicates the name of an array or template
- <*i*> indicates an integer scalar variable or *. * specifies the axis is not aligned.
- **<***f(i)***>** indicates a linear expression of <*i*> *s**<*i*>**+***o*, or *, where *s* and *o* are integer expressions.
  - When <*f(i)*> is a linear expression of <*i*> *s**<*i*>**+***o*, the element of array *a* <*i*> is aligned with the element of the align-target *t* *s**<*i*>**+***o*.
- When <*f(i)*> is *, the whole array *a* is replicated along the axis of the processor array to which the axis of the align-target *t* to which * is specified corresponds.
  -

A.1.5 SHADOW Directive

!HPF$ SHADOW *a* **(** <shadow width>,⋯ **)**

or

!HPF$ SHADOW (<shadow width>,⋯ **) ::** *a*,⋯

- *a* indicates the name of an array
- <shadow width> is *n* or *l* **:** *u*, where *n* is equivalent to *n* **:** *n*, which indicates the lower shadow width and upper shadow width, respectively. The shadow width must be a constant.

A.1.6 SEQUENCE Directive

!HPF$ [NO] SEQUENCE [[**::**] *s***,**⋯]

- *s* is the name of an array or **/**common block name**/**. When *s***,**⋯ is omitted in the SEQUENCE directive, it is treated as if it contained all common block and variables that are not mapped explicitly. When *s***,**⋯ is omitted in the NOSEQUENCE directive, it is treated as if it contained all common blocks and variables.

A.2 Directives in the Execution Part

A.2.1 INDEPENDENT Directive

Perfectly Parallelizable Loops

!HPF$ INDEPENDENT [**, NEW(** *v*,⋯ **)**]

- *v* indicates the name of a variable (NEW variable)

Parallelizable Loops with Reduction

!HPF$ INDEPENDENT [**, NEW(** *v*,⋯ **)**]**,** <REDUCTION clause>,⋯

- *v* indicates the name of a variable (NEW variable)
- <REDUCTION clause> is

  REDUCTION([<reduction-kind1> **:**] *r*,⋯ **)**

  or

  REDUCTION([<reduction-kind2> **:**] *r* **/***p*,⋯**/**,⋯ **)**
  - <reduction-kind1> is **+, *, .AND.**, **.OR.**, **.EQV.**, **.NEQV.**, **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**
  - *r* indicates the name of a reduction-variable
  - <reduction-kind2> is **FIRSTMAX**, **FIRSTMIN**, **LASTMAX**, or **LASTMIN**
  - *p* indicates the name of a position variable
  - When <reduction-kind1> **:** is omitted, reduction assignments must be described any of the following forms.

    r **=** *r* <op> <expr> or *r* **=** <expr> <op> *r*

    or

    r **=** <f**(***r***,** <expr>**)**> or *r* **=** <f**(**<expr>**,** *r***)**>
    - *r* indicates the name of a reduction-variable
    - <op> indicates a reduction operator *****, **/**, **+**, **-**, **.AND.**, **.OR.**, **.EQV.**, or **.NEQV.**
    - <expr> indicates an expression that does not include the reduction variables and is estimated before the operation <op>.
    - <f**()**> indicates a reference to the function **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**

A.2.2 ON-HOME-LOCAL Directive Construct and Directive

ON-HOME-LOCAL directive construct

!HPF$ ON HOME(<array section**>)** [**, LOCAL**[**(** *v*,⋯ **)**]] **BEGIN**

Sequence of <executable statement or construct>

!HPF$ END ON

- The abstract processors onto which <array section> is mapped execute the sequence of <executable statement or construct>.
- v indicates the name of a variable for which data transfers are not needed. When **(** *v*,⋯ **)** is omitted, data transfers are not needed for all variables that appear in the sequence of <executable statement or construct>.

ON-HOME-LOCAL directive

!HPF$ ON HOME(<array section**>)** [**, LOCAL**[**(** *v*,⋯ **)**]]

- The abstract processors onto which <array section> is mapped execute the immediately following executable statement or construct.

A.2.3 REFLECT Directive

!HPF$ REFLECT [**(**<shadow width>,⋯**)**] [**::**] *a,*⋯

- *a* indicates the name of an array, which must appear in the SHADOW directive in the specification part of the scoping unit.
- When <shadow width>,⋯ is specified, the shift transfer is performed only on the specified part of the shadow area, which is called the partial REFLECT directive.

A.3 Other Features

A.3.1 EXTRINSIC Prefix

EXTRINSIC (<lang> **,** <model> **)**

or

EXTRINSIC (<extrinsic-kind-keyword> **)**

- <lang> is **"HPF"** or **"Fortran"**
- <model> is **"GLOBAL"**, **"LOCAL"**, or **"SERIAL"**. **"GLOBAL"**, **"LOCAL"**, and **"SERIAL"** indicate global model, local model, and serial model, respectively.
- <extrinsic-kind-keyward> is **HPF**, **HPF_LOCAL**, **HPF_SERIAL**, **Fortran_LOCAL**, or **Fortran_SERIAL**, which indicate global model HPF, local model HPF, serial model HPF, local model Fortran, and serial model Fortran, respectively.

Appendix B Frequently Asked Questions

A.1 Data Mapping

- How are variables that do not appear in the DISTRIBUTE directive nor ALIGN directive mapped?
  - They are replicated on all abstract processors.

A.2 Data Transfer

- Redundant data transfers occur for allocatable arrays and assumed-shape arrays.
  - Map allocatable arrays and assumed-shape arrays using ALIGN directives. Refer to subsection 4.1.4 for details.

A.3 Execution Performance and Memory Usage

- Memory usage at runtime is too large.
  - Possible causes are as follows:
    - ✧ The shadow areas of width four are automatically allocated along the axes distributed with the BLOCK distribution or GEN_BLOCK distribution by default for efficient shift transfer. If your HPF program does not need the shift transfer, memory usage can be reduced by specifying the shadow width as zero. Specify the shadow width with the SHADOW directive or the HPF compiler option –Moverlap=size:*n* as follows.

```
%> ve-hpf –Moverlap=size:0 source.hpf
```

    - ✧ When large arrays are initialized in the specification part or using DATA statements, the memory area for the whole arrays is allocated on each abstract processor. In such cases, the memory usage can be reduced by initializing them at the beginning of runtime.

    - ✧ When data transfers occur for execution of loops or invocations of procedures, the memory area for the whole arrays targeted for the data transfers can be

allocated on each abstract processor. In such cases, the memory usage can be reduced by specifying that the loops are parallelizable or no data transfers are needed. You can find where data transfers occur referring to the parallelization information lists or diagnostic messages.

- Specify that loops are parallelizable with the INDEPENDENT directives (+ REDUCTION clauses).
- Specify that no data transfers are needed with the ON-HOME-LOCAL directive constructs.
- Modify data mappings of arrays or description of loops so that necessary data transfers are reduced.
- Specify data mappings so that the data mappings of actual arguments and corresponding dummy arguments are the same.

➢ Arrays that do not appear in the DISTRIBUTE directive nor ALIGN directive are replicated on all abstract processors. Map large arrays if possible.

➢ In the hybrid parallelization, where both the distributed-memory parallelization and shared-memory parallelization are performed, local variables are allocated on each thread. Therefore, when large local arrays are used, memory usage for the shared-memory parallelization becomes large. In such cases, the memory usage can be reduced by changing local arrays into global arrays, because the memory area for global arrays is shared by all threads by default.

- The execution performance significantly drops when the Fortran compiler option –mparallel is used.
  ➢ When the Fortran compiler option –mparallel is used, both the distributed-memory parallelization by the HPF compiler and shared-memory parallelization by the Fortran compiler are performed. The number of parallelization is the product of the number of abstract processors in HPF and number of threads for the shared-memory parallelization. When the number of parallelization on each VE node exceeds the number of cores on the VE node, the execution performance significantly drops because of the conflict. Specify the number of threads with the runtime environment variable OMP_NUM_THREADS or VE_OMP_NUM_THREAD, and the number of processes so that the number of parallelization does not exceed the number of cores

on every VE node.

- The execution performance is not good though major loops are parallelized and inefficient data transfers are not generated in the parallelization information lists and diagnostic messages.
  - Possible causes are as follows:
    - ✧ When data mappings with DISTRIBUTE directives, ALIGN directives, and SHADOW directives of actual arguments and corresponding dummy arguments differ, data transfers occur at the invocation of and return from the procedures. Check whether data transfers at procedure boundaries occur with the HPF runtime option –hpf –commmsg, because such data transfers cannot be detected at compilation time.

```
%> mpirun –np 4 ./a.out –hpf -commmsg
```

The following warning message at runtime shows that data transfer between an actual argument and the corresponding dummy argument occurs. Modify the HPF program so that the data mapping of the actual argument is the same as that of the corresponding dummy argument referring to the name of the procedure and dummy argument in the warning message.

"Dummy-argument name": Communication occurs at procedure boundary PROG="Procedure name" ELN="Line number"

    - ✧ The numbers of iterations of loops parallelized by the HPF compiler become smaller, and initial parameters and terminal parameters of loops become variables. As a result, the loops targeted for vectorization can be changed from the serial execution, and the performance can drop because of shorter vector length. When the FTRACE information shows that the vector length is much shorter than the serial execution, check whether loops which are parallelized and whose lengths become shorter are vectorized. Then change the loops targeted for vectorization using the NEC Fortran directives such as novector.

✧ The inline expansion of procedures invoked many times can be inhibited because of the parallelization by the HPF compiler, which can cause performance degradation.

- When procedures which are inline expanded in the serial execution do not have array dummy arguments, the inline expansion may be performed also in parallel execution with the HPF compiler option -Mnoentry.
- When procedures which are inline expanded in the serial execution have array dummy arguments, the inline expansion should be performed manually.

A.4 Miscellaneous

- INDEPENDENT directives cause incorrect execution results.
  - Possible causes are as follows. Also, refer to subsection 5.3.4 for how to detect wrong INDEPENDENT directives.

✧ INDEPENDENT directives to non-parallelizable loops result in incorrect execution results. In the following example, the INDEPENDENT directive cannot be specified because the value of the variable l that is defined in the previous iteration is referenced.

```
        l=0
!HPF$   INDEPENDENT,NEW(I,J)   ! Wrong
        do i=1,n
          do j=1,n
            l = l+1
            a(j,i) = l
          enddo
        enddo
```

The following modification makes the loop parallelizable, and the INDEPENDENT directive can be specified.

```
!HPF$   INDEPENDENT,NEW(I,J)
        do i=1,n
          do j=1,n
            a(j,i) = 1+n*(i-1)*(j-1)
          enddo
        enddo
```

✧ INDEPENDENT directives without REDUCTION clauses to the loops that perform reduction computation result in wrong execution results. In the following example, the INDEPENDENT directive without REDUCTION clause cannot be specified because the loop performs the sum-reduction computation on the array a.

```
!HPF$   INDEPENDENT,NEW(i)   ! Wrong
        do j=1,n
          do i=1,n
            a(i) = a(i) + b(i,j)
          enddo
        enddo
```

The correct execution result can be obtained by specifying the REDUCTION clause to the array a as follows, or deleting the INDEPENDENT directive.

```
!HPF$   INDEPENDENT,NEW(i),REDUCTION(a)
        do j=1,n
          do i=1,n
            a(j) = a(j) + b(i,j)l
          enddo
        enddo
```

Appendix CHistory

History table

September, 2020	1st edition
July, 2022	2nd edition

Change History

1st

2nd

NEC Fortran Compiler Options (2.3.2)

NEC Fortran Compiler Runtime Environment Variables (3.2.1)

How to use the FTRACE region feature is added (5.1.3).

Description about performance improvement of I/O is added (5.1.4).

Index

I

L

M

N

O

P

R

S

T

V

W

SX-Aurora TSUBASA System Software

SX-Aurora TSUBASA

NEC HPF User's Guide

2nd Edition July 2022

NEC Corporation