HPF/JA
Language Specification

JAHPF (Japan Association for High Performance Fortran)

January 31, 1999
Version 1.0
English Version 1.0
November 11, 1999

Contents

Chapter 1

Overview

This document specifies the HPF extended language specifications HPF/JA 1.0 defined by the Japan Association for High Performance Fortran (JAHPF) to more practically use the High Performance Fortran (HPF). The HPF/JA 1.0 is designed as a set of extensions and modifications to the HPF language specification defined by the High Performance Fortran Forum (HPFF). The version of the HPF language specification used as a base is the HPF2.0 language and its approved extension (High Performance Fortran Language Specification version 2.0, Jan. 31, 1997) at the present time (Jan. 1999).

The HPF/JA extension specifications are classified into the following two major purposes:

- The enhancement of description performance for program parallel processing and the enlargement of application range
- Compensate for insufficiency of compiler capability in the current stage by enabling the user to describe parallel processing and optimization in detail

The extensions fall into the following two categories.

1. Enlargement of description capability for parallel processing
    - Specification of REDUCTION kind
      ... Enlarges the application range of the REDUCTION clause.
2. Optimization of communication
    - Asynchronous transfer
      ... Overlaps communication between processors with computation processing.
    - Extension of SHADOW directive
      ... Enables the user to select full-shadow allocation with fast access speed.
    - REFLECT directive
      ... Explicitly set the value of the shadow area.
    - Extension of HOME clause in ON directive
      ... Enables the user to specify an active processor, taking into account a shadow area.
    - Extension of LOCAL clause or directive
      ... Specifies that communication is unnecessary for data access.
    - Reuse of communication schedule
      ... Efficiently processes communication repeated in the same pattern.

Chapter 2

Notation and Syntax

This chapter describes the notational conventions employed in this document and syntax of HPF/JA directives.

2.1 Notation

This document uses the same notation as the HPF2.0 specification and Fortran 95 standard, including particularly the syntax rules. The BNF description of the language features are defined in the same style as the HPF specification and Fortran standard. Each HPF/JA rule has an identification number of the form J*snn* to distinguish the HPF/JA syntax rules from the HPF and Fortran syntax rules. In J*snn*, *s* corresponds to a chapter number, and *nn* indicates a two-digit sequence number. Nonterminals not defined in this document are defined in the HPF2.0 specification or Fortran standard. The rule of having an identification number of the form H*snn* is defined in the HPF2.0 specification, and the rule of having an identification number of the form R*snn* in the Fortran standard. Some technical terms, for example "mapping" and "storage unit", are defined in the HPF2.0 specification or Fortran standard.

The HPF/JA syntax rule is an extension of one similar to the HPF2.0 syntax rule. In this case, the name of a nonterminal symbol is suffixed by *-ja*. When a nonterminal symbol such as *name* or *name-extended* in the HPF approved extension specification is redefined, it is therefore referred to as *name-ja* under the proviso that any reference to *name* or *name-extended* to be replaced by *name-ja* in the rest of the syntax rules.

> *Rationale.* Throughout this document, material explaining the rationale for including features, for choosing particular feature definitions, and for making other decisions, is set off in this format. Readers interested only in the language definition may wish to skip these sections, while readers interested in language design may want to read them more carefully. (*End of rationale.*)

> *Advice to users.* Throughout this document, material that is primarily of interest to users (including most examples of syntax and interpretation) is set off in this format. Readers interested only in technical material may wish to skip these sections, while readers wanting a more tutorial approach may want to read them more carefully. (*End of advice to users.*)

> *Advice to implementors.* Throughout this document, material that is primarily of interest to implementors is set off in this format. Readers interested only in the lan-

guage definition may wish to skip these sections, while readers interested in compiler implementation may want to read them more carefully. (*End of advice to implementors.*)

2.2 Syntax of Directives

The HPF/JA directives are consistent with the HPF2.0 directives and Fortran syntax in the following sense: if any HPF/JA directive were to be adopted as a part of the future HPF specification, the only change necessary to convert an HPF/JA program to an HPF program would be to replace the hpfja-directive-origin with `!HPF$`: and, if any HPF/JA directive were to be adopted as a part of the future Fortran standard, the only change necessary to convert an HPF/JA program to an Fortran program would be to replace the *hpfja-directive-origin* with blanks.

HPF/JA directives have the following general formats:

J201	*hpfja-directive-line*	**is**	*hpfja-directive-origin hpf-directive*
J202	*hpfja-directive-origin*	**is**	`!HPFJ`
		or	`CHPFJ`
		or	`*HPFJ`

To use the HPF/JA specification defined in the next chapter and afterward, each directive must begin with *hpfja-directive-origin.* HPF2.0 directives not based in the HPF/JA specification may also begin with *hpfja-directive-origin.* HFP2.0 directives may begin with the *directive-origin* of HPF2.0.

Advice to users. When using a system including an HPF2.0 compiler but not an HPF/JA compiler, begin each HPF2.0 directive with `!HPF$`. At least the HPF2.0 directives become valid when the HPF2.0 compiler is used, and the portability of the program is enhanced. When using only an HPF/JA compiler, use only `!HPFJ`. Thus, users need not check whether directives are included in the HPF2.0 specification.

HPF/JA directives are designed to function as a correct HPF2.0 program even if ignored. (*End of advice to users.*)

The rules related to character types (uppercase and lowercase letters), line formats, blanks, and continuation lines conform to the HPF2.0 directive syntax. However, an HPF/JA directive line must not be continued into an HPF2.0 directive line.

In the next chapter and afterward, some syntax rules are added and deleted for *specification-directive-extended* (H206), *executable-directive-extended* (H207), and *executable-construct-extended* (H208) defined in the HPF2.0 specification. This updated content is newly defined as follows:

J203	*specification-directive-ja*	**is**	*processors-directive*
		or	*subset-directive*
		or	*align-directive*
		or	*distribute-directive*
		or	*inherit-directive*
		or	*template-directive*
		or	*combined-directive*
		or	*sequence-directive*
		or	*dynamic-directive*
		or	*shadow-directive*
		or	*asyncid-directive*

J204	*executable-directive-ja*	**is**	*independent-directive-ja*
		or	*realign-directive-ja*
		or	*redistribute-directive-ja*
		or	*on-directive*
		or	*resident-directive*
		or	*asynchronous-directive*
		or	*asyncwait-directive*
		or	*reflect-directive*
		or	*local-directive*
		or	*index-reuse-directive*

J205	*executable-construct-ja*	**is**	*action-stmt*
		or	*case-construct*
		or	*do-construct*
		or	*if-construct*
		or	*where-construct*
		or	*on-construct*
		or	*resident-construct*
		or	*task-region-construct*
		or	*asynchronous-construct*
		or	*local-construct*

Chapter 3

HPF/JA Extension Related to Parallel Processing Specification

3.1 Specification of REDUCTION Kind

The purpose of this extension specification is to increase the flexibility of reduction description.

The `REDUCTION` clause of HPF2.0 does not explicitly indicate a reduction kind. Instead, the reference format of a reduction variable is restricted to the format of a reduction statement (*reduction-stmt*), so that the compiler can identify each reduction kind. (Section 5.1.3 in HPF2.0 specification) The flexibility of reduction description is therefore limited. The `MAXLOC` and `MINLOC` computations frequently used by an application program are not included in the HPF2.0 reduction description. The user cannot therefore specify `INDEPENDENT` for a loop including those computations.

This extension specification defines a reduction kind in a `REDUCTION` clause so that a reduction variable can be referenced in any format and an `INDEPENDENT` loop can be described including the `MAXLOC` and `MINLOC` computations.

3.1.1 Syntax

The syntax rules of the `INDEPENDENT` directive (H501 and H503 in the HPF specification, Section 5.1) are modified as follows:

J301	*independent-directive-ja*	**is**	`INDEPENDENT` [, *new-clause*] [, *reduction-clause-ja-list*]
J302	*reduction-clause-ja*	**is**	`REDUCTION` ([*reduction-kind* :] *reduction-spec-list*)
J303	*reduction-kind*	**is**	*reduction-operator*
		or	*reduction-function*
		or	*maxmin-kind*

J304 *reduction-operator* **is** `+`
or `*`
or `.AND.`
or `.OR.`
or `.EQV.`
or `.NEQV.`

J305 *maxmin-kind* **is** `FIRSTMAX`
or `FIRSTMIN`
or `LASTMAX`
or `LASTMIN`

J306 *reduction-spec* **is** *reduction-variable* [/ *location-variable-list* /]

J307 *location-variable* **is** *scalar-variable-name*

reduction-function is defined in the HPF specification, Section 5.1.3.

The following constraints are added to Section 5.1 in the HPF specification:

Constraint: When *reduction-kind* is *maxmin-kind*, *reduction-spec* must have *location-variable-list*. When *reduction-kind* is not *maxmin-kind* or *reduction-kind* is omitted, *reduction-spec* must not have *location-variable-list*.

Constraint: When *reduction-kind* is *maxmin-kind*, *reduction-variable* in *reduction-spec* must be *scalar-variable-name*.

Constraint: The type of variable specified in *reduction-variable* must be defined for each *reduction-kind* value as follows:

Logical type for `.AND.`, `.OR.`, `.EQV.`, and `.NEQV.`

Integer type for `IAND`, `IOR`, and `IEOR`

Numeric type for `+` and `*`

Integer or real type for `MAX`, `MIN`, `FIRSTMAX`, `FIRSTMIN`, `LASTMAX`, and `LASTMIN`

Constraint: *reduction-variable* specified in *reduction-clause* without *reduction-kind* must be referenced in the reduction statement format in the loop defined in Section 5.1.3 of the HPF2.0 specification. (*reduction-variable* specified in *reduction-clause* with *reduction-kind* may be referenced in any format in a loop.)

In section 5.1 of the HPF2.0 specification, the fifth constraint is modified as follows:

Constraint: A variable specified as *reduction-variable* or *location-variable* must not be specified two or more times in the same *independent-directive*. It must not also be specified in *new-clause* and *reduction-clause* within the range of the succeeding *do-stmt*, *forall-stmt* and *forall-construct* (that is, loop body in the source program) to which the *independent-directive* applies.

3.1.2 Semantics

The INDEPENDENT directive asserts that the iterations of a DO loop do not mutually interfere. (Section 5.1 in HPF specification) The condition of this interference is relaxed in the REDUCTION clause with reduction kind as follows:

- The second exception in the first interference condition is modified as follows. The modified content is underlined.
  - Exception: If a variable appears in a REDUCTION clause <u>without reduction kind</u>, then assignments to it by reduction statements in the range of the DO loop *do not* interfere with assignments to it by other reduction statements in the same loop. The reason for this is explained in Section 5.1.3.

  The following exception is added:
  - Exception: If a variable appears as a reduction variable or a location variable in a REDUCTION clause with reduction kind, then assignments to it *do not* interfere with assignments to it in a different iteration of the DO loop. The DO loop must however compose a reduction computation with the reduction kind and the reduction variable corresponding to the variable.
- The second exception in the second interference condition is modified as follows. The modified content is underlined.
  - Exception: If a variable appears in a REDUCTION clause <u>without reduction kind</u>, then assignments to it by reduction statements in the range of the DO loop *do not* interfere with the allowed uses of it by reduction statements in the same loop. The reason for this is explained in Section 5.1.3.

  The following exception is added:
  - Exception: If a variable appears as a reduction variable or a location variable in a REDUCTION clause with reduction kind, then assignments to it *do not* interfere with uses of it in a different iteration of the DO loop. The DO loop must however compose a reduction computation with the reduction kind and the reduction variable corresponding to the variable.

Composing a *reduction computation* is defined as shown below. Considering a certain iteration of a DO loop as one block. Let the value of variable X at the entry of the iteration be X^{in} and let the value at the exit be X^{out}. X^{in} is a virtual value in the sense that it does not matter whether X can actually take the value or not. Depending on the value of X^{in}, X^{out} may not be defined.

- If there exist an associative operation f and a value c not depending on R^{in} and the following formula holds for any value of R^{in}, the iteration in a DO loop composes a reduction computation for a variable R

$$R^{out} = f(R^{in}, c)$$

  The value c is called a reduction element; f must be one of the operations defined in Table 3.1.

Reduction kind	$f(x, y)$
+	x + y
*	x * y
.AND.	x.AND.y
.OR.	x.OR.y
.EQV.	x.EQV.y
.NEQV.	x.NEQV.y
MAX	MAX(x, y)
MIN	MIN(x, y)
IAND	IAND(x, y)
IOR	IOR(x, y)
IEOR	IEOR(x, y)
FIRSTMAX	MAX(x, y)
FIRSTMIN	MIN(x, y)
LASTMAX	MAX(x, y)
LASTMIN	MIN(x, y)

Table 3.1: Computation for each reduction kind

- If all iterations in a DO loop compose a reduction computation with the same operation for a variable R, the DO loop composes a reduction computation for the variable R.

If the reduction kind is "+" or "*", a computation error may occur depending on the computation sequence in an actual computer. The value of c may be unstabilized depending on the value of R^{in}. Taking into account this point, when the reduction kind is "+" or "*" and there is a sequence of values not depending on R^{in}, $c_1, c_2, \cdots, c_n$ $(n \geq 0)$ satisfying the below formula below instead of the above formula, a reduction computation is assumed to be composed.

$$R^{out} = f(\cdots f(f(R^{in}, c_1), c_2) \cdots, c_n)$$

3.1.3 Constraints

In this section, a reduction variable for reduction computation is written as R and a location variable as $L_1, \cdots, L_m$ $(m \geq 0)$. If the value and the status of each variable at the end of a certain iteration of a DO loop do not depend on the value of $R^{in}, {L_1}^{in}, \cdots, {L_m}^{in}$, the value and the status are defined to be *invariant* for the reduction computation in the iteration.

If a value and a status are invariant for reduction computation in all iterations of a DO loop, they are defined to be invariant for reduction computation in a DO loop.

1. A DO loop specified by INDEPENDENT having a REDUCTION clause with reduction kind must compose reduction computation for the reduction variable. The correspondence between the reduction kind and reduction computation must conform to the combination permitted in Table 3.1.

2. When the reduction kind is *maxmin-kind*, all iterations in the DO loop must satisfy the following conditions for all location variables L_k corresponding to reduction variable R:

   - When R^{in} is within an R update range, ${L_k}^{out}$ must be defined, and its value must be invariant for the reduction computation.
   - When R^{in} is within an R non-update range, ${L_k}^{out}$ must be undefined if ${L_k}^{in}$ is undefined and also must have the same value as ${L_k}^{in}$ if ${L_k}^{in}$ is defined.

   The following table lists the R update range and R non-update range depending on reduction kinds. In this table, c_i indicates a reduction element for iteration i.

Reduction kind	R update range	R non-update range
FIRSTMAX	$R^{in} < c_i$	$R^{in} \geq c_i$
FIRSTMIN	$R^{in} > c_i$	$R^{in} \leq c_i$
LASTMAX	$R^{in} \leq c_i$	$R^{in} > c_i$
LASTMIN	$R^{in} \geq c_i$	$R^{in} < c_i$

3. The values and attributes of all data objects (excluding a reduction, location, or NEW variable) and the file and unit status (presence or absence, contents of records, file position and other characteristics inquired by the INQUIRE statement) must be invariant for all reduction computations composed by the DO loop.

4. A reduction variable specified in a REDUCTION clause with reduction kind must be invariant for all reduction computations excluding those composed by the variable itself.

5. A location variable specified in a REDUCTION clause with reduction kind must be invariant for all reduction computations excluding those composed by the reduction variable specified by the same *reduction-spec*.

   Rationale. Basis of constraint 1

   A reduction variable without reduction kind can be accessed only when a reduction statement is used. (Section 5.1.3 in HPF2.0 specification) A reduction variable with reduction kind can be accessed freely; however, it must compose a reduction computation as the whole. REDUCTION without reduction kind is restricted in the syntax; one with reduction kind in the semantics of computation. For example, sum has an add iteration as a condition regardless of the way of description. (*End of rationale.*)

   Advice to users. Program errors related to a combination of reduction kind and reduction computation cannot be completely detected by a compiler. The user must perform appropriate programming, understanding the semantics of reduction computation. (*End of advice to users.*)

Example. Example of constraint 1

```
DO I=1,100
  X = X+A(I)
  IF (I.EQ.3) X = X+B
END DO
```

When the reduction kind is "+", reduction element c for each iteration is obtained by the following formula regardless of the value of X.

- At $\mathtt{I} \neq 3$, $X^{out} = X^{in} + \mathtt{A(I)}$
  That is, $c = \mathtt{A(I)}$
- At $\mathtt{I} = 3$, $X^{out} = X^{in} + \mathtt{A(3)} + \mathtt{B}$
  That is, $c = \mathtt{A(3)} + \mathtt{B}$
  (More strictly, $c_1 = \mathtt{A(3)}$, $c_2 = \mathtt{B}$, taking into account an error depending on the computation sequence.)

Therefore, this DO loop satisfies constraint 1. for describing a REDUCTION clause

```
REDUCTION(+:X)
```

Example. Example of constraint 2.

```
!HPFJ INDEPENDENT, REDUCTION(FIRSTMAX:AMAX/ILOC/)
      DO I=1,N
        IF (AMAX.LT.A(I)) THEN
          AMAX = A(I)
          ILOC = I
        END IF
      END DO
```

When the value of $\mathtt{AMAX}^{in}$ is changed, $\mathtt{AMAX}^{out}$ and $\mathtt{ILOC}^{out}$ change as shown below, considering the program content.

$\mathtt{AMAX}^{in}$	$\mathtt{AMAX}^{out}$	$\mathtt{ILOC}^{out}$
-HUGE	A(I)	I
$\vdots$	A(I)	I
A(I)	$\mathtt{A(I)=AMAX}^{in}$	$\mathtt{ILOC}^{in}$
$\vdots$	$\mathtt{AMAX}^{in}$	$\mathtt{ILOC}^{in}$
HUGE	$\mathtt{AMAX}^{in}$	$\mathtt{ILOC}^{in}$

From this table, reduction element c is assumed to be A(I). When $\mathtt{AMAX}^{in} < \mathtt{A(I)}$, $\mathtt{ILOC}^{out} = \mathtt{I}$ holds. When $\mathtt{AMAX}^{in} \geq \mathtt{A(I)}$, $\mathtt{ILOC}^{out} = \mathtt{ILOC}^{in}$ holds. Taking into account these results, constraint 2. is assumed to be satisfied.

If a conditional clause of an IF statement changes to (AMAX.LE.A(I)), the table changes as follows:

$\mathtt{AMAX}^{in}$	$\mathtt{AMAX}^{out}$	$\mathtt{ILOC}^{out}$
-HUGE	A(I)	I
$\vdots$	A(I)	I
A(I)	$\mathtt{A(I)=AMAX}^{in}$	I
$\vdots$	$\mathtt{AMAX}^{in}$	$\mathtt{ILOC}^{in}$
HUGE	$\mathtt{AMAX}^{in}$	$\mathtt{ILOC}^{in}$

In this case, when $\texttt{AMAX}^{in}$ is equal to A(I), $\texttt{ILOC}^{out}$ is not necessary equal to $\texttt{ILOC}^{in}$. As a result, constraint 2. is not satisfied.

3.1.4 Examples

```
!HPFJ INDEPENDENT, REDUCTION(MIN:AMIN), REDUCTION(+:S1,S2), NEW(TMP)
      DO I = 1,N
        IF(A(I).LT.AMIN) AMIN=A(I)
        TMP = S1+B(I)
        S1 = TMP+C(I)
        S2 = ADD(S2,D(I))
      END DO
```

ADD(x,y) is assumed to be a user-defined function only for computing x + y.
An example using FIRSTMAX follows:

```
!HPFJ INDEPENDENT, NEW(I), REDUCTION(FIRSTMAX:AMAX/ILOC,JLOC/)
      DO J=1,N
        DO I=1,M
          IF(AMAX.LT.A(I,J)) THEN
            AMAX=A(I,J)
            ILOC=I
            JLOC=J
          END IF
        END DO
      END DO
```

Chapter 4

HPF/JA Extension for Communication Optimization

4.1 Asynchronous Transfer Function

The asynchronous transfer function performs data transfer between processors in parallel with execution of another executable statement.

This function is defined by a combination of an executable directive for instructing the start of data transfer with one for waiting for the end. These directives correspond with the same ID.

4.1.1 ASYNCID declaration directive

The ASYNCID declaration directive declares one ID each to correspond with statements for starting and ending the asynchronous transfer, respectively.

4.1.1.1 Syntax

Add *asyncid-directive* to *specification-directive-extended* (H206).

J401	*asyncid-directive*	**is**	`ASYNCID` *async-id-list*
J402	*async-id*	**is**	*async-id-name*

Add `ASYNCID` and `SAVE` to *combined-attribute-extended* (H801).

Constraint: When `SAVE` is defined in *combined-directive*, `ASYNCID` must also be defined.

Example.

```
    ASYNCID ID1,ID2
    ASYNCID :: X
    ASYNCID,SAVE :: S,T,U
```

4.1.1.2 Semantics

ASYNCID directive Declares that *async-id* is an asynchronous identifier. To use an asynchronous identifier, be sure to declare this statement.

The asynchronous identifier has the following features:

- A local entity belonging to class (1). (See Section 14.1.2 in the Fortran standard.) Therefore, its name is valid only in a scoping unit (procedure and so on), and it must not be the same as another local entity[1] belonging to class (1) in the same scoping unit.
- It can be associated regardless of a scoping unit by using the use association (declared in a module and referenced in multiple scoping units) or host association (declared in a host procedure and shared by a host-slave procedure).
- The asynchronous identifier has either the *enabled* state or *disabled* state. The initial state is the disabled state. When an asynchronous identifier is referred to in the ASYNCHRONOUS directive (Section 4.1.2), it is placed in the enabled state. When an asynchronous identifier is referred to in the ASYNCWAIT directive (Section4.1.2), it is placed in the disabled state again.
- The asynchronous identifier can have a SAVE attribute. An asynchronous identifier having the SAVE attribute holds the association, allocation, and enabled/disabled states after the RETURN or END statement is executed.

SAVE Declares that an asynchronous identifier has a SAVE attribute.

Rationale. Reason why the asynchronous identifier is regarded as a new local entity There is a proposal in which the asynchronous identifier is assumed to be an integer-type variable (the name has a meaning) or integer expression (the value has a meaning). However, for the following reason the asynchronous identifier is assumed to be a new local entity.

- Clear syntax
  - The user and language processor can recognize clearly that the name is used as an asynchronous identifier.
    Program readability is improved. The language processor has more opportunities to detect an error. Optimization in the language processor is promoted.
  - The asynchronous identifier is defined only in directives.
    The program can be modified by adding directives (without correcting Fortran statements). If a Fortran variable or expression is used as an identifier, variables used only for declaration may be defined at compilation with sequential interpretation.
- The language processor can be implemented naturally and easily.
  The runtime data structure can be prepared statistically using the declaration of an identifier as a trigger. If the value of a variable or expression is used as an identifier, a wasteful structure may be generated, and implementation may be

[1]Includes a named variable, statement function, built-in procedure, and so on. In addition, a processor and template are also local entities in HPF.

difficult depending on architecture. (For example, when the basic integer type is 32 bits and an address space is 64 bits in length, the value of the basic integer type is too small to save address data.)

(*End of rationale.*)

4.1.1.3 Example

See the example shown in Section 4.1.2.3. An example requiring the SAVE declaration is shown in Section 4.1.5.

4.1.2 ASYNCHRONOUS directives

Consisting of a simple directive and specification syntax, the ASYNCHRONOUS directive specifies the start of asynchronous transfer. The ASYNCWAIT directive is used to wait for the completion.

4.1.2.1 Syntax

Add *asynchronous-directive* and *asyncwait-directive* to *executable-directive-extended*(H207). Also add *asynchronous-construct* to *executable-construct-extended*(H208).

Simple ASYNCHRONOUS directive

J403	*asynchronous-directive*	**is**	`ASYNCHRONOUS` *asynchronous-stuff*
J404	*asynchronous-stuff*	**is**	`( [ ID = ]` *async-id* `)` [, *nobuffer-clause*]

Example.

```
ASYNCHRONOUS (ID=ID1)
ASYNCHRONOUS(ZZ)
```

ASYNCHRONOUS directive construct

J405	*asynchronous-construct*	**is**	*hpfja-directive-origin block-asynchronous-directive* *block* *hpfja-directive-origin end-asynchronous-directive*
J406	*block-asynchronous-directive*	**is**	`ASYNCHRONOUS` *asynchronous-stuff* `BEGIN`
J407	*end-asynchronous-directive*	**is**	`END ASYNCHRONOUS`

Example.

```
!HPFJ ASYNCHRONOUS(ID1) BEGIN
      A(:)=B(1:100)
      FORALL(I=1:M,J=1:N) S(I,J)=T(J,I)
!HPFJ END ASYNCHRONOUS
```

ASYNCWAIT directive

J408 *asyncwait-directive* **is** `ASYNCWAIT ( [ID =]` *async-id* `)`

Example.

```
ASYNCWAIT(ID=ID1)
```

4.1.2.2 Semantics

The following executable statements and directives are called *asynchronously executable statements.*

(1) Built-in assignment statement[2]

(2) Simple FORALL statement whose body is a built-in assignment statement

(3) REDISTRIBUTE directive

(4) REALIGN directive

(5) REFLECT directive (HPF/JA extension)

For details of *nobuffer-clause*, see Section 4.1.3.

Simple ASYNCHRONOUS directive Instructs the system that, for the immediately succeeding asynchronously executable statement, it is possible to start the subsequent processing without waiting for the completion of data transfer resulting from execution of the statement.

The transfer identifier *async-id* is placed into the enabled state by executing the simple ASYNCHRONOUS directive.

ASYNCHRONOUS directive construct Instructs the system that, for all asynchronously executable statements included in block, it is possible to start the subsequent processing without waiting for the completion of data transfer resulting from the execution of the statements.

The transfer identifier *async-id* is placed into the enabled state by executing the ASYNCHRONOUS construct.

The ASYNCHRONOUS directive construct is equivalent to a representation using multiple simple ASYNCHRONOUS directives.

[2] Ordinary assignment statement, excluding user-defined and pointer assignment statements

```
!HPFJ ASYNCHRONOUS(ID=id)
      BEGIN
      statement-1
      statement-2
      ...
      statement-n
!HPFJ END ASYNCHRONOUS
      ...
!HPFJ ASYNCWAIT(ID=id)
```

⇔

```
!HPFJ ASYNCHRONOUS(ID=id-1)
      statement-1
!HPFJ ASYNCHRONOUS(ID=id-2)
      statement-2
      ...
!HPFJ ASYNCHRONOUS(ID=id-n)
      statement-n
      ...
!HPFJ ASYNCWAIT(ID=id-1)
!HPFJ ASYNCWAIT(ID=id-2)
      ...
!HPFJ ASYNCWAIT(ID=id-n)
```

ASYNCWAIT directive Instructs the system to wait for the completion of asynchronous transfer started by the simple ASYNCHRONOUS directive or ASYNCHRONOUS directive construct having the same *async-id*.

The transfer identifier *async-id* is placed into the disabled state by executing the ASYNCWAIT directive.

4.1.2.3 Example

```
      REAL A(N),S(M,N),T(N,M)
!HPFJ ASYNCID ID1,ID2                     ! async-id
!HPF$ DISTRIBUTE A(BLOCK)
!HPF$ DISTRIBUTE (*,BLOCK) :: S,T
      ...
!HPFJ ASYNCHRONOUS (ID=ID1)
      FORALL(I=1,M) T(:,I)=A(:)*10.0      ! Start of transfer to T
      ...                                 ! T non-access processing
!HPFJ ASYNCWAIT (ID=ID1)                  ! End of transfer to T
!HPFJ ASYNCHRONOUS (ID2) BEGIN
!HPF$ REDISTRIBUTE A(BLOCK)
      FORALL(I=1:M,J=1,N) S(I,J)=T(J,I)
!HPFJ END ASYNCHRONOUS                    ! Start of transfer to A and S
      ...                                 ! A and S non-access processing
!HPFJ ASYNCWAIT(ID2)                      ! End of transfer to A and S
      ...
```

4.1.2.4 Constraints

Basic constraints

1. The executable statements and directives to be processed by the simple ASYNCHRONOUS directive and ASYNCHRONOUS directive construct must be asynchronously executable statements (see Section 4.1.2.2).

2. At execution of the simple ASYNCHRONOUS directive or ASYNCHRONOUS directive construct, the asynchronous identifier must not be in the enabled state. At

execution of the ASYNCWAIT directive, the transfer identifier must be in the enabled state.

Constraints related to object variable In respect to the executable statements and directives to be processed by the ASYNCHRONOUS directive, the following variables are called *asynchronous objects*.

Object statement	Asynchronous object
Built-in assignment statement	Left-hand side
Simple FORALL statement	Left-hand side of assignment statements in the body
REDISTRIBUTE directive	Distributee and all data objects ultimately aligned to it
REALIGN directive	Alignee
REFLECT directive	*reflect-object*

1. The statements executed in a period from the execution of the ASYNCHRONOUS statement to the execution of the corresponding ASYNCWAIT statement must not include the reference of an asynchronous object. However, the following reference is allowed:

   - Reference for inquiring the attributes (type, shape, allocation state, and so on) of an asynchronous object
   - Reference for referencing mapping (reference in the HOME clause of the ON statement, and so on). Not allowed for the asynchronous objects of the REDISTRIBUTE and REALIGN directives, however.

Example.

```
      REAL A(M,N),B(M,N)
!HPFJ ASYNCID :: ID1
!HPF$ DISTRIBUTE B(BLOCK,*)
!HPF$ ALIGN A(:,:) WITH B(:,:)
!HPF$ DYNAMIC A,B
      ...
!HPFJ ASYNCHRONOUS(ID=ID1)
!HPF$ REDISTRIBUTE B(*,BLOCK)    ! The asynchronous variables are A
      ...                        !        and B.
      C(I)=A(I)                  ! Prohibited: A cannot be referenced.
      B=D+E                      ! Prohibited: B cannot be defined.
      CALL SUB(B)                ! Prohibited: B cannot be referrenced
                                 !        as an actual parameter.
      DEALLOCATE(A)              ! Prohibited: A cannot be placed in
                                 !        the undefined state.
!HPF$ REALIGN A(:,:) WITH T(:,:) ! Prohibited: A cannot be realigned.
      ...
!HPFJ ASYNCWAIT(ID=ID1)
```

Rationale. Reason why reference is prohibited as an actual argument
This is because the value of an actual argument may be overwritten by the value of a dummy argument after it is transferred by the asynchronous transfer. The compiler may perform the argument passing by value association (method of copying an actual argument to a local variable at the entry of a subprogram and returning the local variable to the original actual argument). The compiler may also perform the automatic redistribution at the entry and exit of a subprogram. (*End of rationale.*)

2. In the ASYNCHRONOUS directive construct, a variable defined as an asynchronous object must not be referenced again after the asynchronously executable statement.

Example. The underlined variables are asynchronous objects.

```
!HPFJ ASYNCHRONOUS(ID=ND) BEGIN
      A(1:N)=B(1:N)
      C(:)=A(:)+D(:)                !(a) Not allowed.
      P(:)=D(:)                     !(b) Allowed.
!HPF$ REALIGN B(:) WITH T(:)        !(c) Allowed.
      A(N+1:NN)=E(N+1:NN)           !(d) Allowed.
      FORALL(I=1:9) G(I+1)=G(I)     !(e) Allowed.
!HPFJ END ASYNCHRONOUS
```

(a) Since the array section of `A` is an asynchronous object, it cannot be referenced.
(b) `D` is referenced multiple times; this is allowed since `D` is not an asynchronous object.
(c) Since `B` is first defined as an asynchronous object, it is allowed.
(d) The array section of `A` is an asynchronous object, but it is not overlapped. `A` is therefore allowed.
(e) An asynchronous object can be referenced with the same statement.

Constraints of asynchronous realignment In case of a REALIGN directive constraints are as follows:

1. The following processing must not be performed directly or indirectly for the ultimately aligned target in the REALIGN directive (variable `C` in this example) within a period from the start of asynchronous transfer by the ASYNCHRONOUS directive to the execution of the corresponding ASYNCWAIT directive.

   - Deallocation and making allocation undefined
   - Reference as an actual argument for a procedure call
   - Remapping (including asynchronous one)

Example.

```
!HPFJ ASYNCID ID1                   ! async-id
      REAL A(100,200)
      REAL B1(100,200),C1(100)
```

```
!HPF$ DISTRIBUTE C1(BLOCK)
!HPF$ ALIGN B1(:,*) WITH C1(:)
      REAL B2(100,200),C2(200)
!HPF$ DISTRIBUTE C2(BLOCK)
!HPF$ ALIGN B2(*,:) WITH C2(:)
!HPF$ ALIGN A(:,:) WITH B1(:,:)       ! A is first aligned to B1.
!HPF$ DYNAMIC A,B1,B2,C1,C2
      ...
!HPFJ ASYNCHRONOUS(ID=ID1)
!HPF$ REALIGN A(:,:) WITH B2(:,:)     ! Start of asynchronous
      ...                              !  realignment of A
!HPF$ REDISTRIBUTE C2(BLOCK,:)        ! Prohibited.
      CALL FOO(C2)                    ! Prohibited.
      DEALLOCATE C2                   ! Prohibited.
      ...
!HPFJ ASYNCWAIT(ID=ID1)               ! End of asynchronous
      ...                              !  realignment of A
```

Constraint of active processor The ASYNCHRONOUS directive and corresponding ASYNCWAIT directive must be executed on the same set of active processors.

Example.

```
!HPF$ ON (P(1:4)) BEGIN      ! The active processors are P(1:4).
!HPFJ ASYNCHRONOUS(ID=ID1)
      ...
!HPFJ ASYNCHRONOUS(ID=ID2)
      ...
!HPFJ ASYNCHRONOUS(ID=ID3)
      ...
!HPF$ END ON
!
!HPFJ ASYNCWAIT(ID=ID1)      ! Not allowed.  All processors are active.

!HPF$ ON (P(5:8)) BEGIN
!HPFJ ASYNCWAIT(ID=ID2)    ! Not allowed.  The active processors are P(5:8).
!HPF$ END ON

!HPF$ ON (P(1:4)) BEGIN
!HPFJ ASYNCWAIT(ID=ID3)      ! Allowed.  The active processors are P(1:4).
!HPF$ END ON
```

4.1.3 NOBUFFER clause in ASYNCHRONOUS directive

A NOBUFFER clause is supplied to efficiently perform the asynchronous transfer with an assignment statement and FORALL statement.

4.1.3.1 Syntax

The NOBUFFER clause can be specified optionally in *asynchronous-directive* and *block-asynchronous-directive* (Section 4.1.2.1).

J409 *nobuffer-clause* **is** `NOBUFFER`

Example.

```
ASYNCHRONOUS (ID=ID1), NOBUFFER
ASYNCHRONOUS(ZZ),NOBUFFER
```

Example.

```
!HPFJ ASYNCHRONOUS(ID=Z), NOBUFFER BEGIN
      A(:)=B(:)
      FORALL(I=1:N) S(:,I)=T(I,:)
!HPFJ END ASYNCHRONOUS
```

4.1.3.2 Semantics

The following statements are called *asynchronously executable statements without buffer.*

(1) Assignment statement whose right-hand side consists of only one variable (whole array, array section, array element, or scalar variable)

(2) FORALL statement whose assignment statement is according to item (1)

The NOBUFFER clause declares that, for the right-hand side of an asynchronously executable statement without buffer, the following processing is not performed directly or indirectly within a period from the start of asynchronous transfer by the ASYNCHRONOUS directive to the execution of the ASYNCWAIT directive.

- Value definition and making a value undefined (Value reference is allowed.)
- Deallocation and making allocation undefined
- Reference as an actual argument for procedure calling
- Remapping (including asynchronous one)
- Reference for referencing mapping (HOME clause in ON directive, and so on)

Rationale. The NOBUFFER clause does not force the compiler to perform the asynchronous transfer without buffer; it is used to report to the compiler that the conditions for enabling the asynchronous transfer without buffer are satisfied. (*End of rationale.*)

Advice to implementors. The ASYNCHRONOUS directive having the NOBUFFER clause should be executed by a transfer method without buffer if efficient. However, this method is not mandatory. Select an efficient method depending on the type and architecture of a described assignment statement. (*End of advice to implementors.*)

4.1.3.3 Example

```
      REAL A(1000),B(1000)
      REAL C(100,100),D(100,100)
      INTEGER E(200),F(100,200,300)
      REAL S(500,20),T(800,20)
      INTEGER IX1(N),IX2(N)
!HPFJ ASYNCID :: DD
      ...
!HPFJ ASYNCHRONOUS(ID=DD), NOBUFFER BEGIN
      A=B                                ! Transfer from whole array
                                         !  to whole array
      E=F(J,:,K+1)                       ! Transfer from array section
                                         !  to whole array
      FORALL(I=1,N) C(:,I)=D(I,:)        ! transpose transfer between
                                         !  array sections
      S(IX1,:)=T(IX2,:)                  ! Transfer with vector subscript
!HPFJ END ASYNCHRONOUS
      ...                   ! Here, A, E, C, and S are not accessed;
      ...                   ! B, F, D, and T are not accessed,
                           !  excluding the reference of their values.
!HPFJ ASYNCWAIT(DD)
```

4.1.3.4 Constraints

1. The executable statement and execution statement to be processed by the simple ASYNCHRONOUS directive and ASYNCHRONOUS directive construct having the NOBUFFER clause must be asynchronously executable statements without buffer (see Section 3.2).

2. In an ASYNCHRONOUS construct having the NOBUFFER clause, a variable that appears in the right-hand side of an asynchronously executable statement without buffer must not appear in the construct as an asynchronous object.

Example. The underlined variable is defined in the right-hand side of an asynchronously executable statement without buffer.

```
!HPFJ ASYNCHRONOUS(ID=ND), NOBUFFER BEGIN
      A(1:N)=B(1:N)
      B(:)=C(:)                          !(a) Not allowed.
      D(:)=C(:)                          !(b) Allowed.
      FORALL(I=1:9) G(I+1)=G(I)          !(c) Not allowed.
      S(1:100)=T(1:100)
      T(101:200)=U(1:100)                !(d) Allowed.
!HPFJ END ASYNCHRONOUS
```

(a) The range of `B(1:N)` is referenced in the right-hand side.
(b) `C(:)` is defined many times; however, it is allowed because it is not defined as an asynchronous object.
(c) `G` is overlapped. Overlapping is not allowed even in the same statement.
(d) Any array elements of `T` are not overlapped in the asynchronous object and right-hand side.

4.1.4 ASYNC prefix

The asynchronous execution of the REDISTRIBUTE, REALIGN, and REFLECT directives can be described by combining with an ASYNCHRONOUS directive. To more easily represent asynchronous execution, an ASYNC prefix is supplied.

4.1.4.1 Syntax

Modify *redistribute-directive*(H802) and *realign-directive*(H803) as follows:

J410	*redistribute-directive-ja*	**is**	[*async-prefix*] *redistribute-directive*
J411	*realign-directive-ja*	**is**	[*async-prefix*] *realign-directive*
J412	*async-prefix*	**is**	`ASYNC ( [ ID = ]` *async-id* `)`

Rationale. The syntax of *reflect-directive* is defined in Section 4.4. (*End of rationale.*)

Example.

```
ASYNC(ID=Z) REDISTRIBUTE D(BLOCK,*) ONTO PROC
ASYNC (ID) REDISTRIBUTE (CYCLIC) ONTO P :: T1,T2
ASYNC(ID2) REALIGN A(:,:) WITH B(:,:)
ASYNC(ID=Y) REALIGN (*,I) WITH T(I+1) :: A,B,C
ASYNC(ID=MM) REFLECT A
```

4.1.4.2 Semantics

An execution statement (REDISTRIBUTE, REALIGN, or REFLECT directive only) having *async-prefix* is equivalent to the following combination with the ASYNCHRONOUS directive.

`!HPFJ ASYNC(ID=`*id*`)` *executable-directive*	⇔	`!HPFJ ASYNCHRONOUS(ID=`*id*`)` `!HPFJ` *executable-directive*

4.1.4.3 Example

```
!HPFJ ASYNCID ID1                            ! async-id
      REAL A(100,100),D(100,100)
!HPF$ ALIGN A(I,J) WITH D(I,J)
!HPF$ DISTRIBUTE D(*,BLOCK)
!HPF$ DYNAMIC A,D
      ...
```

```
!HPFJ ASYNC(ID1) REDISTRIBUTE D(BLOCK,*)  ! Start of redistribution
                                          !   for A and D
      ...                                 ! A and D non-access processing
!HPFJ ASYNCWAIT(ID1)                      ! Completion of redistribution
                                          !   for A and D
      ...                    ! A and D can be accessed with new mapping.
```

4.1.5 Notes on scoping unit

The constraints of the ASYNCHRONOUS and ASYNCWAIT directives are described in Section 4.1.2.4. This section provides notes concerning programming to meet those constraints.

4.1.5.1 Asynchronous transfer crossing scoping units

When ASYNCHRONOUS and ASYNCWAIT directives are in different scoping units, define the program carefully so that the allocation of an asynchronous identifier and object is not made undefined until the ASYNCWAIT directive is executed. To prevent this problem, globally declare the asynchronous identifier and object by one of the following methods:

- Declare the asynchronous identifier and object in a module referenced commonly in those scoping units.
- When those scoping units are defined by a relationship between a host and internal procedures or between internal procedures having a common host procedure, they must be declared in the host procedure.

Example. Use a module to define the asynchronous transfer crossing procedures.

- Module

```
      MODULE MOO
!HPFJ ASYNCID Z
      REAL A(100),D(100)
!HPF$ ALIGN A(:) WITH D(:)
!HPF$ DISTRIBUTE D(BLOCK)
!HPF$ DYNAMIC A,D
      END
```

- Caller program

```
      PROGRAM MAIN
      USE MOO
      ...
      CALL ASYNC_SUB
      ...
      CALL ASYNCWAIT_SUB
      ...
      END
```

- Subroutine starting the transfer

```
      SUBROUTINE ASYNC_SUB
      USE MOO
!HPFJ ASYNC(Z) REDISTRIBUTE D(CYCLIC)
      END SUBROUTINE
```

- Subroutine waiting for the transfer

```
      SUBROUTINE ASYNCWAIT_SUB
      USE MOO
!HPFJ ASYNCWAIT(Z)
      END SUBROUTINE
```

Example. Use a host association to define the same content.

```
      PROGRAM MAIN
!HPFJ ASYNCID Z
      REAL A(100),D(100)
!HPF$ ALIGN A(:) WITH D(:)
!HPF$ DISTRIBUTE D(BLOCK)
!HPF$ DYNAMIC A,D
      ...
      CALL ASYNC_SUB
      ...
      CALL ASYNCWAIT_SUB
      ...
      CONTAINS
      SUBROUTINE ASYNC_SUB
!HPFJ ASYNC(Z) REDISTRIBUTE D(CYCLIC)
      END SUBROUTINE
      SUBROUTINE ASYNCWAIT_SUB
!HPFJ ASYNCWAIT(Z)
      END SUBROUTINE
      END
```

An asynchronous identifier and object cannot be passed between procedures via argument association.

Rationale. Since the asynchronous identifier is not a data object, it cannot be passed between procedures by an argument. The asynchronous object variable cannot be referenced as an actual argument. (See Section 4.1.2.4.)

Since a dummy argument is made undefined by ending the execution of the procedure, the asynchronous transfer using a dummy argument as an asynchronous object must be made to wait in the same procedure. The user cannot write a program that waits for transfer to an actual argument after returning from a procedure because the dummy and actual arguments are not guaranteed to be located in the same storage. (*End of rationale.*)

Example. Example of incorrect program

- Module

```
      MODULE MOO
!HPFJ ASYNCID ID1
      REAL B(50,100)
      END
```

- Caller program

```
      USE MOO
      REAL A(100,100)
      ...
      CALL FOO(A(1:50,:))
!HPFJ ASYNCWAIT (ID1)        ! Waits for asynchronous transfer to A.
      ...
```

- Subroutine

```
      SUBROUTINE FOO(X)
      REAL X(50,100)
      ...
!HPFJ ASYNCHRONOUS (ID1)
      X=B
      RETURN              ! Prohibited:  The allocation of dummy
                          !   argument X is made undefined here.
      END
```

4.1.5.2 Asynchronous transfer between different calls for the same subprogram

When the ASYNCHRONOUS and ASYNCWAIT directives are in the same subprogram and are not executed in the same instance (for example, the program starts the asynchronous transfer by the first call and waits for the end of asynchronous transfer by the next call), the allocation of an asynchronous identifier and object must not be made undefined during asynchronous transfer. In this case, as described in Section 4.1.5.1, globally declare an asynchronous identifier and object or declare them with the SAVE attribute.

Example. In the following case, since an asynchronous object A and asynchronous identifier ID must not be made undefined after the subroutine ends, define a SAVE declaration.

Caller program

Call a subroutine N times.

```
      DO I=1,N
        CALL PIPELINETRANS(I,N)
        ...
```

```
      END
```

Subroutine

Obtain variable ATMP from variable A and return the value from ATMP to A by the asynchronous transfer until the next call.

```
      SUBROUTINE PIPELINETRANS(NTIMES,NEND)
      REAL A(1000),ATMP(1000)
      SAVE A
!HPFJ ASYNCID,SAVE :: Z
! ---------------- Waits at the second and subsequent calls.
      IF(NTIMES>1) THEN
!HPFJ   ASYNCWAIT(Z)
      END IF
! ---------------- Obtains ATMP from A.
      DO I=2,999
        ATMP(I)=0.25*(A(I-1)+2*A(I)+A(I+1))
      END DO
! ---------------- Starts the transfer at call excluding the last.
      IF(NTIMES<NEND) THEN
!HPFJ   ASYNCHRONOUS(Z)
        A(2:999)=ATMP(2:999)
      END IF
! ---------------- Returns to call while executing the
!                                                asynchronous transfer.
      RETURN
      END
```

4.1.5.3 Asynchronous transfer in recursive procedure

In a recursive procedure, the asynchronous transfer is performed by two methods: in each instance (see (a) in Figure 4.1) and crossing instances (see (b) in Figure4.1). In the former case, an asynchronous identifier and object must be declared in the procedure without SAVE attribute. In the latter case, an asynchronous identifier and object must be declared in the procedure with SAVE attribute or globally in the module.

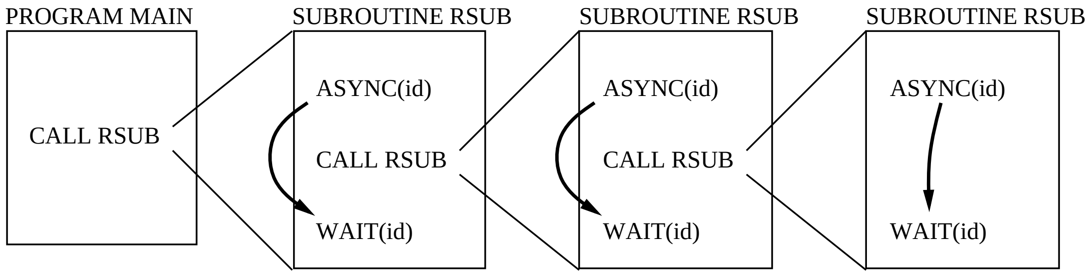
4.1.5.4 Notes on asynchronous remapping

For asynchronous redistribution, not only the distributee itself but also variables aligned with the distributee are regarded as asynchronous objects. Note that all asynchronous objects may not be made undefined during asynchronous transfer.

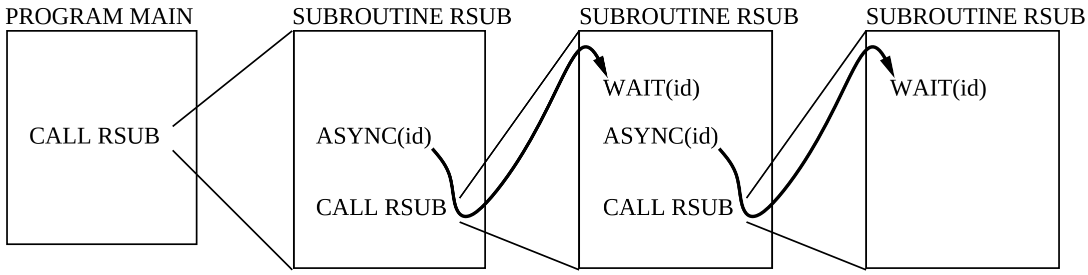
Also note that the ultimate align target may not be made undefined for asynchronous realignment during asynchronous transfer.

Example. Example of incorrect program

Since variable A aligned to D in a subroutine is made undefined during asynchronous transfer, the language processor cannot assure the operation. In this case, move the declaration related to A to a module.



(a) Asynchronous transfer closed for each instance



(b) Asynchronous transfer crossing instances

Figure 4.1: Asynchronous transfer in recursive procedure

- Module

```
      MODULE MODD
      REAL D(1000)
!HPF$ DISTRIBUTE(BLOCK),DYNAMIC :: D
!HPFJ ASYNCID :: ZZ
      END MODULE
```

- Caller program

```
      USE MODD
      ...
      CALL MISDIST
!HPFJ ASYNCWAIT(ID=ZZ)                ! Waits for the redistribution
      ...                             !                         of D.
```

- Subroutine

```
      SUBROUTINE MISDIST
      USE MODD
      REAL A(1000)
!HPF$ ALIGN(:) WITH D(:), DYNAMIC :: A      ! Aligns A to D.
      ...
```

```
      !HPFJ ASYNCHRONOUS(ID=ZZ)
      !HPF$ REDISTRIBUTE(CYCLIC) :: D      ! A and D are object variables.
            RETURN                         ! Prohibited: Local variable A
            END                            !           is made undefined.
```

4.2 Extension of SHADOW Directive

This section explains the extension of SHADOW directive.

In the case of the following example;

Example.

```
      REAL A(4,4)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO P
```

Many HPF language processors allocate only a local part of the whole declared array area onto each processor. (See Figure 4.2)

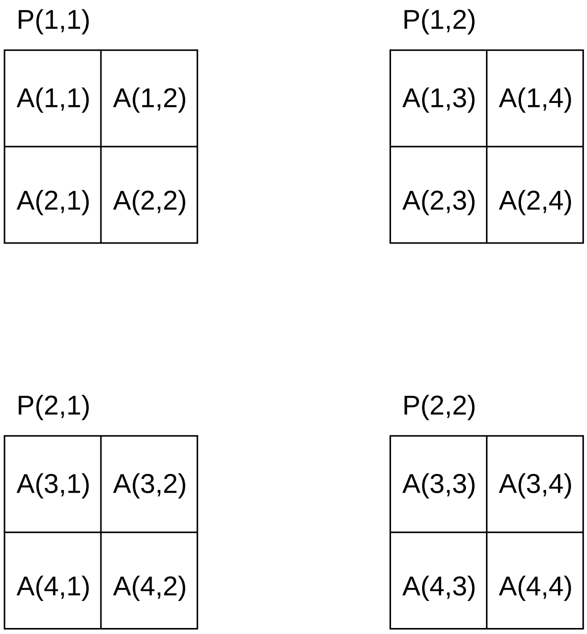


Figure 4.2: Normal allocation

On the other hand, an HPF language processor can allocate the whole declared array area onto each processor by extending the `SHADOW` directive so that an asterisk * can be specified in each dimension of *shadow-target*.(See Figure 4.3)

Example.

```
      REAL A(4,4)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO P
!HPFJ SHADOW A(*,*)                        ! Extended SHADOW directive.
```

The SHADOW attribute of the object for which SHADOW directive is specified in this format is specially called the *full* SHADOW attribute.

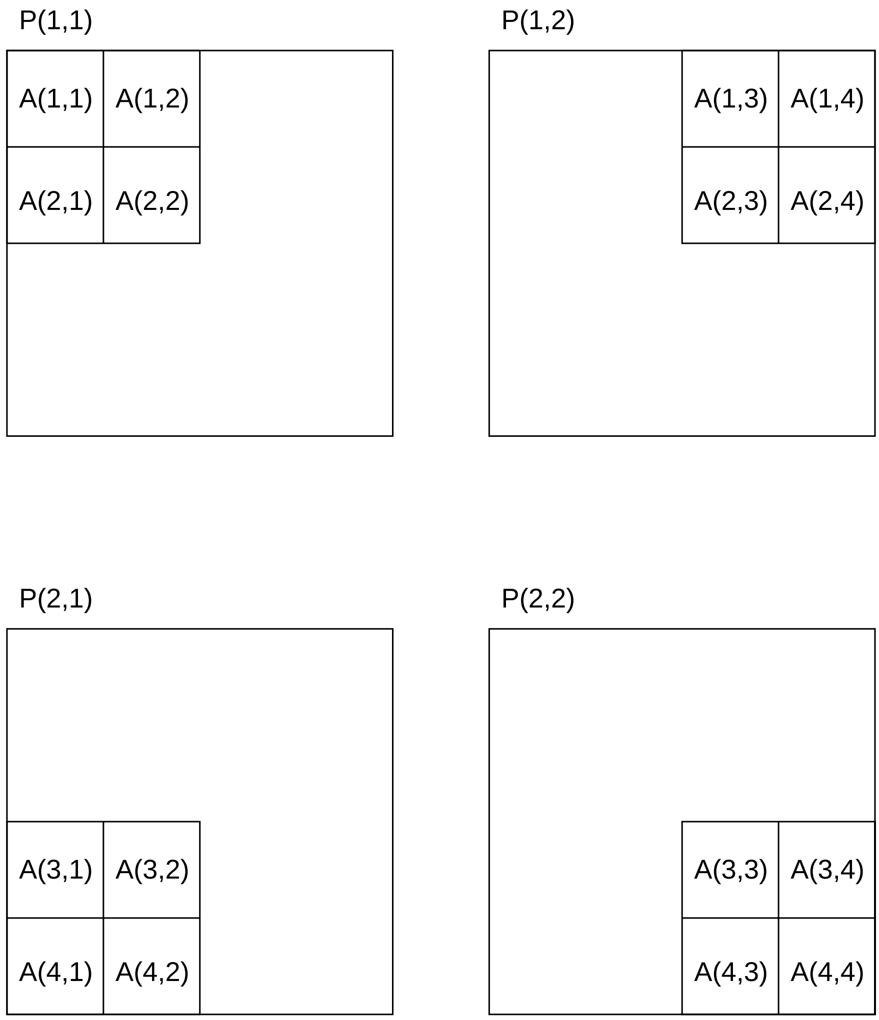


Figure 4.3: Allocation with full SHADOW attribute

Using this allocation method:

- Memory usage efficiency is poor.

Therefore, the size of usable data is limited. On the other hand, an object program can be executed at high speed by the following two features:

- The language processor need not perform subscript conversion from global subscript to local subscript, at the reference of an array when the full SHADOW attribute is statically recognized.

- The language processor need not allocate a new area at (explicit or implicit) dynamic remapping of an array nor copy a value from the original area.

Thus, this directive is useful for the following purposes:

- Executing a test at high speed when developing an application program
- Developing a program with execution speed and size balanced by pursuing the performance as long as memory permits.

Advice to users. A SHADOW directive cannot be specified for a dummy argument to which an INHERIT directive is specified. (For details, see Sections 4.4.2, 8.1, and 8.13 in the High Performance Fortran Language Specification Version 2.0.)

So when a dummy argument having the INHERIT attribute is associated with an actual argument having the full SHADOW attribute, the language processor cannot statically decide that the dummy argument is allocated in the same way as the object having the full SHADOW attribute. This could result in losing the above-described advantage of the full SHADOW attribute, the lack of need for the language processor to perform subscript conversion. Therefore, the object program may not be necessarily executed at high speed.

Taking into account this result, the user should not associate an object having the INHERIT attribute with one having the full SHADOW attribute. (*End of advice to users.*)

Advice to users. When an actual argument having the full SHADOW attribute is associated with a dummy argument not having it or an actual argument not having the full SHADOW attribute is associated with a dummy argument having it, execution-time overhead is required to convert the SHADOW attributes.

May lose the merit of not requiring large memory to allocate only a local part onto each processor.

The user should not therefore associate actual and dummy arguments by this method. (*End of advice to users.*)

Advice to implementors. This extension does not obstruct the implementors from using the allocation method shown in Figure 4.3 for an object not having the full SHADOW attribute. (*End of advice to implementors.*)

4.2.1 Syntax

Extend *shadow-spec*(H820) in the syntax rule of the SHADOW directive as follows:

J413	*shadow-spec-ja*	**is**	*width*
		or	*low-width* : *high-width*
		or	*full-width*
J414	*full-width*	**is**	*

Example.

```
      REAL A(100,100,100),B(100,100,100)
!HPF$ SHADOW A(5:5,0:1,3)                ! Conventional SHADOW directive
!HPFJ SHADOW B(*,*,*)                    ! Extension of SHADOW directive
```

4.2.2 Constraints

Constraint: The length of *shadow-spec-ja-list* must be equal to the rank of *shadow-target*.

Constraint: When *full-width* is specified as *shadow-spec-ja*, *full-width* must be specified in all dimensions.

4.2.3 Equivalence relation for extended SHADOW attributes

This section defines the equivalence relation for SHADOW attributes for an extended SHADOW directive. (For details, see Item 8.13 in the High Performance Fortran Language Specification Version 2.0.)

1. When the *shadow-spec-ja* is *full-width*, it is equivalent only to the *shadow-spec-ja* that is *full-width*.
2. When any of two *shadow-spec-ja* specifications are not *full-width* , the expressions w_1 and w_2 in them are equivelent iff they have the same value.
3. The *shadow-spec-ja* w is equivalent to the *shadow-spec-ja* w:w.
4. The *shadow-spec-ja* l_1:h_1 is equivalent to the *shadow-spec-ja* l_2:h_2 iff l_1 is equivalent to l_2 and h_1 is equivalent to h_2.
5. Other than this, no two lexically different *shadow-spec-ja* specifications are equivalent.

We then say that two SHADOW attributes are equivalent iff the *shadow-spec-ja-list* of one is elementwise equivalent to the *shadow-spec-ja-list* of the other.

Thus the equivalence relation is defined for the set of the mapping including the mapping of an object having the full SHADOW attribute.

4.3 Explicit Shadow

The approved extension feature, shadow, is used to promote the optimization of the compiler. Since the way of optimization depends on the compiler, it may be difficult to introduce general directives for assisting the optimization. However, there are many cases in which sufficient performance is not obtained only by the optimization of the compiler. So, we propose specification based on the concept of directly specifying the access of a shadow area, not assisting the optimization of the compiler. This function aims at enhancing performance by a user description not depending on the optimization by the compiler, while not preventing optimization by the compiler.

4.3.1 Terminology

In Fortran, the variable, constant, and constant subobject are called the data object. The data object can be defined and referenced in the method defined in Fortran. The data stored in the storage area declared by the SHADOW directive (approved extension specification) is called *shadow object*. The shadow object can be explicitly defined and referenced only by the below-described method.

Of the data allocated to a storage area other than a shadow area, data representing the same array element as that of a shadow object is called a *reflection source* of the shadow object. Conceptually, a shadow object and its reflection source are not mapped to one processor at the same time. A shadow object is not mapped to a processor to which its reflection source is mapped. In the same way, multiple shadow objects corresponding to the same array element are not mapped to one processor.

Example. Shadow object for block distribution

```
!HPF$ PROCESSORS P(4)
      REAL A(100)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
!HPF$ SHADOW A(1:2)
```

Data object A and its shadow object are mapped to processor P as follows:

Processor	Data object	Shadow object
P(1)	A(1), ···, A(25)	A(26), A(27)
P(2)	A(26), ···, A(50)	A(25), A(51), A(52)
P(3)	A(51), ···, A(75)	A(50), A(76), A(77)
P(4)	A(76), ···, A(100)	A(75)

Example. Shadow object with full SHADOW

```
!HPF$ PROCESSORS P(4)
      REAL A(100)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
!HPF$ SHADOW A(*)
```

When an array element is mapped to a processor as a data object, it is not mapped as a shadow object.

Processor	Data object	Shadow object
P(1)	A(1), ···, A(25)	A(26), ···, A(100)
P(2)	A(26), ···, A(50)	A(1), ···, A(25), A(51), ···, A(100)
P(3)	A(51), ···, A(75)	A(1), ···, A(50), A(76), ···, A(100)
P(4)	A(76), ···, A(100)	A(1), ···, A(75)

Example. Shadow object for cyclic distribution

```
!HPF$ PROCESSORS P(3)
      REAL,DIMENSION(20) :: B12,B33
!HPF$ DISTRIBUTE (CYCLIC(3)) ONTO P :: B12,B33
!HPF$ SHADOW B12(1:2),B33(3:3)
```

Data object A and its shadow object are mapped to processor P as shown below. ■ indicates that the array element is mapped as a data object. □ indicates that the array element is mapped as a shadow object.

```
Array B12
        1 2 3 4 5 6 7 8 9 1011121314151617181920
P(1)    ■ ■ ■ □ □       □ ■ ■ ■ □ □       □ ■ ■
P(2)        □ ■ ■ ■ □ □       □ ■ ■ ■ □ □
P(3)              □ ■ ■ ■ □ □       □ ■ ■ ■ □ □

Array B33
        1 2 3 4 5 6 7 8 9 1011121314151617181920
P(1)    ■ ■ ■ □ □ □ □ □ □ ■ ■ ■ □ □ □ □ □ □ ■ ■
P(2)    □ □ □ ■ ■ ■ □ □ □ □ □ □ ■ ■ ■ □ □ □
P(3)          □ □ □ ■ ■ ■ □ □ □ □ □ □ ■ ■ ■ □ □
```

Rationale. As shown in this example, in block-cyclic distribution the same array element may be mapped to one processor as a data object and shadow object or multiple shadow objects may be mapped to one processor as the size of the shadow area enlarges. (See a in Figure 4.4) In this case, data to be accessed by LOCAL specification (see Section 4.6) is not determined uniquely. Although the LOCAL specification can be extended to determine which shadow is selected, such specification becomes complicated.

To cope with this problem, we can present the following concept. (See b in Figure 4.4)

- When a data object is specified for a specific array element, a shadow object is not allocated to the data object on the same processor in the conception.
- When a shadow object is specified for a specific array element, it is assumed to be one in the conception.

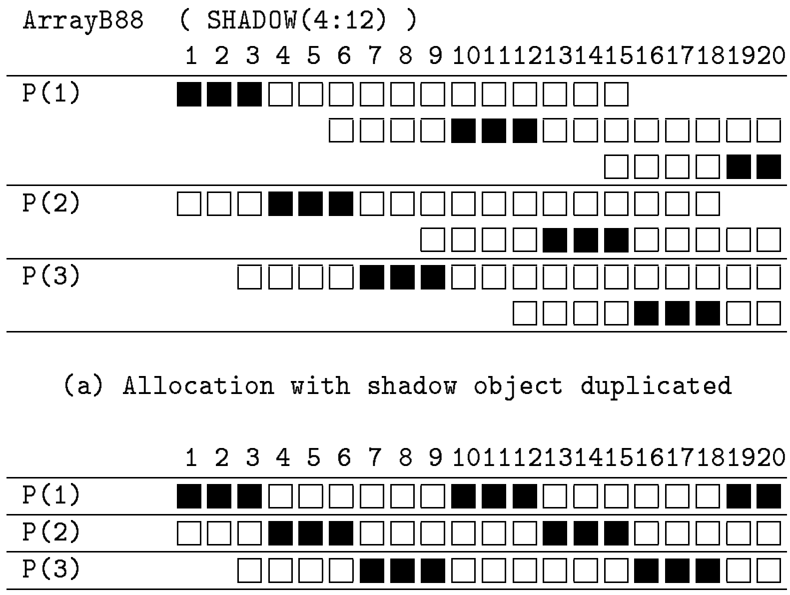
When this concept applies to the language specification, in the implementation of allocation method (a) the language processor must assure that the duplicated data object and shadow object are the same value. In this case, however, high-speed processing (the original purpose of LOCAL specification and shadow specification) cannot be expected.

In the HPF/JA specification, the shadow size for block-cyclic distribution is limited within a range in which the above-shown duplication does not occur (See Chapter 5) to prevent complication of the language specification and obstruction of high-speed processing.

(*End of rationale.*)

4.3.2 Definition and reference of shadow object

The value of a shadow object can be defined and referenced only in either one of the following methods:



(a) Allocation with shadow object duplicated

(b) Allocation with shadow object not duplicated

Figure 4.4: Duplication of shadow object

1. Initialization

   No shadow object can be initialized. The initial value of a shadow object is always undefined.

2. Value definition

   (a) Specification by `REFLECT` directive (See Section 4.4)
   The same value as that of the reflection source (provided by another processor) is stored to.

   (b) Specification by `LOCAL` directive
   A value definition of a variable within the valid range of the `LOCAL` directive is performed regardless whether the variable is a data or shadow object in each processor.

   However, to use the `LOCAL` directive, the constraints described in Section 4.6.3 must be satisfied.

3. Value reference

   (a) Specification by `LOCAL` directive
   A reference to a variable within the valid range of the `LOCAL` directive is performed regardless whether the variable is a data or shadow object in each processor.

Advice to users. The `EXT_HOME` clause (see Section 4.5) in the `ON` directive aims at only specifying active processors. Even if a specific variable is specified in the

EXT_HOME clause, if it is not specified in the LOCAL directive, it is not guaranteed to be defined or referenced within each processor. (*End of advice to users.*)

4.3.3 Defined and undefined states for shadow object

Like the data object, the shadow object has a defined or undefined state. An undefined shadow object does not necessarily have a valid value.

The processing of placing a data object into the defined or undefined state is already defined in Section 14.7 of the Fortran standard. That for placing a shadow object into the defined or undefined state is defined as follows:

1. Initial state of shadow object
   (a) The initial state of a shadow object is always undefined.
2. Conditions under which a shadow object is placed into the defined state

   A shadow object is placed into the defined state by any one of the following conditions:
   (a) A shadow object is placed into the defined state when it is defined in either method described in Section 4.3.2, Item 2.
   (b) At the entry of a procedure, a shadow object of a dummy argument is placed into the defined state if the following shadow inheritance conditions are satisfied and a shadow object of the corresponding actual argument is in the defined state.
   (c) Just after returning from a procedure, a shadow object of an actual argument is placed into the defined state when the following shadow inheritance conditions are satisfied and a shadow object of the corresponding dummy argument is placed into the defined state just before the END or RETURN statement is executed.
3. Conditions under which a shadow object is placed into the undefined state

   A shadow object is placed into the undefined state by any one of the following conditions:
   (a) A shadow object in the same active processor set as the reflection source is made undefined when the reflection source is made undefined.
   (b) A shadow object in the same active processor set as the reflection source is made undefined when a value is defined in the reflection source without LOCAL directive.
   (c) A shadow object in the same active processor set as the reflection source is made undefined unless its value matches the reflection source when exiting from the valid range of the LOCAL directive.
   (d) A shadow object in a different active processor set from the reflection source is made undefined if its value doesn't match the reflection source or the reflection source is undefined when the shadow object enters the same active processor set as the reflection source.
   (e) A shadow object is made undefined when its parent object is remapped.

Shadow inheritance conditions

When the following conditions are satisfied, the value of a shadow object is inherited between actual and dummy arguments:

1. The mapping of an actual argument is the specialization of the mapping of a dummy argument. (See Section 8.13 in the HPF specification.)
2. A dummy argument does not have the DYNAMIC attribute.

Advice to implementors. This definition allows language processors to process in the following way.

1. When a reflection source is updated or made undefined, no processing is required for the corresponding shadow area.
2. When the reflection source and shadow object are in the same active processor set, the language processor may copy the value of the reflection source to the corresponding shadow object at any time.
3. The value of a shadow area need not be assured at remapping.
4. When remapping is not required at a call or return of a procedure, no processing need be performed for the shadow area. If remapping is required, the value of the shadow area need not be assured.

Condition 2 is defined supposing that the constraints related to the LOCAL directive (see Section 4.6.3) are assured by the user. (*End of advice to implementors.*)

4.4 REFLECT Directive

The REFLECT directive assigns the value of a reflection source to a shadow object for variables having the shadow attribute.

4.4.1 Syntax

Add *reflect-directive* to *executable-directive-extended*(H207).

J415	*reflect-directive*	**is**	[*async-prefix*] REFLECT *reflect-object-list*
J416	*reflect-object*	**is**	*object-name*

async-prefix is defined in Section 4.1.4.

Constraint: All processors onto which a data or shadow object of *reflect-object* is distributed must be active.

Example.

```
REFLECT A
ASYNC(ID=ID1) REFLECT A,B,C
```

4.4.2 Semantics

The REFLECT directive copies the value of the reflection source of *reflect-object* to all shadow objects. When *async-prefix* is specified, the asynchronous transfer (see Section 4.1) is validated.

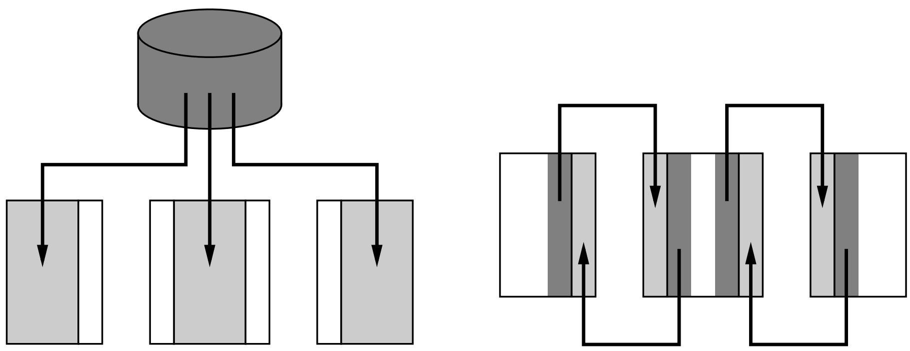
Introducing the REFLECT directive adds the following conditions for applying the INDEPENDENT directive to a DO loop, that is not being interfered with by a iteration of the loop (see Section 5.1 in the HPF2.0 specification).

- The REFLECT directive executed in a loop iteration interferes with reference of the same data, remapping of the same data, and another REFLECT directive for the same data in the other iteration.

4.4.3 Example

Read a value to a data object with a READ statement and copy the value to a shadow object with the succeeding REFLECT directive (see Figure 4.5)

```
!HPF$ PROCESSORS P(3)
      REAL A(M,N)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO P :: A
!HPF$ SHADOW(0,1) :: A
      ...
      READ(*) A
!HPFJ REFLECT A
      ...
```



(a) READ(*) A (b) REFLECT A

Figure 4.5: Example of REFLECT directive

4.5 Extension of HOME Clause in ON Directive

When a variable is specified in the HOME clause of the ON directive (approved extension), only processors to which the variable is mapped are assumed to be active. In this case, the computation mapping is performed, ignoring the shadow area of the variable. This section explains how to extend the HOME clause in the ON directive activating the processors to which shadow objects are mapped.

4.5.1 Syntax

Extend *home*(H907) in the ON directive as follows. Note that *on-stuff*(H903) is extended in Section 4.6

J417 *home-ja*	**is**	HOME (*variable*)
	or	HOME (*template-elmt*)
	or	EXT_HOME (*variable* [, *shadow-attr-stuff*])
	or	(*processors-elmt*)

Constraint: The length of *shadow-spec-list* specified by *shadow-attr-stuff* in the EXT_HOME clause must match the rank of the parent object of *variable*.

Constraint: The upper and lower shadow widths of each dimension specified by *shadow-attr-stuff* in the EXT_HOME clause must be equal to or less than the shadow width of the parent object of *variable* respectively.

Constraint: In *shadow-attr-stuff*, *shadow-spec-ja* must not be *full-width* (asterisk).

Example.

```
EXT_HOME(A(I))
EXT_HOME(B(I+1),(2))
EXT_HOME(Z(I,:),(1:0,0))
```

4.5.2 Semantics

In the ON directive, *home* specifies a set of active processors executing a valid range of ON directive. The semantics of the first syntax (HPF approved extension specification) for *home* is:

- All processors to which variable is mapped as a data object are active.

The semantics of the third syntax (HPF/JA extension) for*home-clause* is:

- When *shadow-attr-stuff* is not specified, all processors to which variable is mapped as a data or shadow object are active. If the parent object[3] of *variable* does not have a shadow area, this syntax has the same semantics as the first one.
- When *shadow-attr-stufff* is specified, all processors to which variable is mapped as a data object or shadow object in the range represented by *shadow-attr-stuff* are active.

  Rationale. The following alternative ideas were presented as the semantics of the EXT_HOME clause:

  1. To access all variables in the valid range of the ON directive having the EXT_HOME clause, the LOCAL directive should be specified explicitly. (This can be described easily by using a LOCAL directive without variable.) By this rule, processing is always performed at high speed in the valid range of the ON directive having the EXT_HOME clause.
  2. The valid range of the ON directive having the EXT_HOME clause is processed as if the LOCAL directive without variable were specified. In other words, the semantics of the LOCAL directive is included in a EXT_HOME clause.

[3]For example, if *variable* is A(I), the parent ofject is the whole array A.

However, these semantics were not used for the following reasons:

- Data mapped outside the specified processors may be referenced even in the valid range specified by the ON directive having the EXT_HOME clause. Use is limited when the LOCAL directive is always required.
- In the HPF2.0 specification, the purpose of the home clause is only to specify a set of active processors; no definition of data mapping is described. The semantics of data mapping should not be included only in the EXT_HOME clause.

(*End of rationale.*)

The EXT_HOME clause is considered to be a simpler notation of a HOME clause. These two descriptions are the same, excluding the following differences.

```
ON EXT_HOME(A(I),(M:N))
ON HOME(A(I-N:I+M))
```

- Different ranges are allowed as the value of I. Assuming that the lower limit of the array declaration of A is l and the upper limit is u, $l \leq \mathtt{I} \leq u$ is obtained in the former case and $l + \mathtt{N} \leq \mathtt{I} \leq u - \mathtt{M}$ in the latter case.
- In the former case, the M and N sizes cannot exceed the lower and upper shadow widths respectively.
- The EXT_HOME clause without *shadow-attr-stuff* can be used for a variable whose shadow size is unknown (variable with the inherit attribute, etc.), but the same processing cannot be represented in the HOME clause.

Example. In the part indicated by (*1) of the following program, the correspondence between the I values and active processors is as shown in the table below.

```
!HPF$ PROCESSORS P(3)
      REAL A(9)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
!HPF$ SHADOW(1) :: A
      ...
      DO I=1,9
!HPFJ  ON EXT_HOME(A(I))
         ...                  ! (*1)
!HPF$  END ON
      END DO
```

I	Set of active processors
1	P(1)
2	P(1)
3	P(1), P(2)
4	P(1), P(2)
5	P(2)
6	P(2), P(3)
7	P(2), P(3)
8	P(3)
9	P(3)

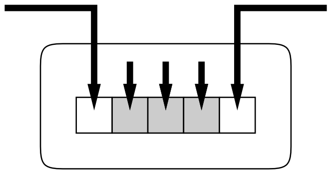
Advice to users. As shown in this example, the EXT_HOME clause complicates the changing of active processors; thus, efficiency may be reduced extremely. However, this clause is very effective in directly assigning the result of relatively easy calculation to a shadow area, for example, initialization of an array. To obtain high performance using the EXT_HOME clause, specify the LOCAL directive (see Section 4.6) together. (See Section 4.6.4.4) (*End of advice to users.*)

4.5.3 Example

```
       REAL A(100)
!HPF$ DISTRIBUTE(BLOCK),SHADOW(1) :: A
```

(a) ON HOME + REFLECT
Obtain the value of a data object by calculation and that of a shadow object by communication.

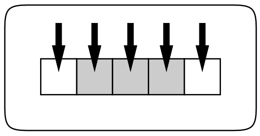
```
!HPF$ INDEPENDENT
      DO I=1,100
!HPF$   ON HOME(A(I))
          A(I)= ···
      END DO
!HPFJ REFLECT A
```



- The calculation area (range of I) does not overlap an adjacent processor.
- The value of a shadow object is obtained by communication from an adjacent processor.

(b) ON EXT_HOME
Calculate a range including a shadow area for each processor.

```
!HPF$ INDEPENDENT
      DO I=1,100
!HPFJ   ON EXT_HOME(A(I)),LOCAL(A(I))
          A(I)= ···
      END DO
```



- By the effect of the EXT_HOME clause, processors having A(I) as a shadow object also perform calculation.
- As a result of the effect of the LOCAL clause, the value of a shadow object is specified independently of an adjacent processor, not in cooperation with an adjacent processor.

When the cost required to calculate the value of each element is lower than that for communication from the reflection source to a shadow object, method (b) (specifying an active processor with shadow) is more efficient.

4.6 LOCAL Clause and Directive

The RESIDENT clause and directive (approved extension) declare that data exists in active processors executing the execution statements. By this information, the compiler knows that communication outside the active processor is not required for the data. However, when there are multiple active processors, communication between these active processors may be required.

The LOCAL clause and LOCAL directive proposed in this section specify that communication is not required for specified data at all, improving the concept of the RESIDENT clause and RESIDENT directive.

4.6.1 Syntax

4.6.1.1 Extension of ON directive

Modify *on-stuff*(H903) as follows. *home-ja* is defined in Section 4.5.

J418	*on-stuff-ja*	**is**	*home-ja* [, *on-optional-clause-list*]
J419	*on-optional-clause*	**is**	*resident-clause*
		or	*local-clause*
		or	*new-clause*

Rationale. Specify the `RESIDENT` and `NEW` clauses in any order and add a new `LOCAL` clause. (*End of rationale.*)

Example.

```
ON HOME(B(I)), RESIDENT(A), LOCAL(A(I),A(I-1),A(I+1))
ON (PE(1:4)), NEW(I,J), LOCAL
```

4.6.1.2 LOCAL clause, directive, and construct

Define *local-clause*, *local-directive*, and *block-local-directive* as shown below. Then add the definition to *executable-directive-extended*(H207).

J420	*local-clause*	**is**	`LOCAL` *local-stuff*
J421	*local-stuff*	**is**	[(*local-object-list*)]
J422	*local-directive*	**is**	`LOCAL` *local-stuff*
J423	*local-construct*	**is**	*hpfja-directive-origin block-local-directive* *block* *hpfja-directive-origin end-local-directive*
J424	*block-local-directive*	**is**	`LOCAL` *local-stuff* `BEGIN`
J425	*end-local-directive*	**is**	`END LOCAL`
J426	*local-object*	**is**	*object*

The syntax rules are the same as those of `RESIDENT` syntax, except that the keyword is `LOCAL`. *local-object* is specified by the lexically matching way.

Example. `LOCAL` Clause

```
LOCAL(A(I),S)
LOCAL
```

Example. Specification by LOCAL directive

```
!HPFJ LOCAL(A(I),S)
      A(I) = S
```

Example. Specification by LOCAL construct

```
!HPFJ LOCAL BEGIN
      A(I) = S
      CALL SUB(A,I,S)
!HPFJ END LOCAL
```

4.6.2 Semantics

LOCAL clause in ON directive, and LOCAL directive specify how to reference the value and attributes of a specified variable as well as define the value of the variable within the valid range of these directives as follows.

Reference All active processors read data from a copy of a variable within each processor. They do not reference a copy of a variable of another processor.

Definition All active processors write to a copy of a variable within each processor.

The copy referenced or defined by each processor may be a data object or shadow object.

The LOCAL directive without variable has the same effect as that obtained when LOCAL is specified for the reference and definition of all variables (including all variables in procedures called directly and indirectly in the valid range) within the valid range.

Like the RESIDENT specification, the LOCAL specification is related to the ON directive enclosing it and the mapping of an object variable. Therefore, the assertion of LOCAL may not be reliable unless the language processor accords with the specified ON directive and mapping.

> *Rationale.* The purpose of the LOCAL specification is that the user can ensure a specified access can be performed without communication among processors. (*End of rationale.*)

4.6.3 Constraints

1. The copy of a variable specified by LOCAL must be mapped onto all active processors as a data object or shadow object.

2. When any variable specified by LOCAL is defined, the copy of the variable must not be mapped as a data object onto processors not included in the active processor set. (However, it may be mapped as a shadow object onto processors not included in the active processor set.)

3. For a variable specified by LOCAL, the following processing must not be specified in the valid range of LOCAL:

   - Remapping
   - Allocation or deallocation for the variable

- Specifying as *reflect-object* in REFLECT directive
- Specifying in REDUCTION clause
- Specifying in RESIDENT clause or RESIDENT directive

4. A global procedure (global model) must not be called in the valid range of LOCAL without variable.

5. When exiting the valid range of LOCAL or existing the valid range of ON defined in the valid range of LOCAL, the value of the copy of the data object for a specified variable must match within active processors. (The value of a shadow object need not match that value, in which case the value of the shadow object becomes undefined. (See Section 4.3.3)

   Rationale. Purposes of constraints 1 and 2

   By constraint 1, to reference a variable, a processor referencing the variable always has the data. The language processor can therefore omit a test requiring communication and reference data on the processor. In the same way, to define a variable, a processor that defines a variable always owns its data area. The language processor can therefore omit a test overwriting an invalid area and set an appropriate value.

   By constraint 2, the variable data is assigned to all the copies of data objects by defining the variable in active processors. The language processor need not therefore perform communication to assure the matching between variable values. (*End of rationale.*)

4.6.4 Example

4.6.4.1 Referencing a variable specified by LOCAL

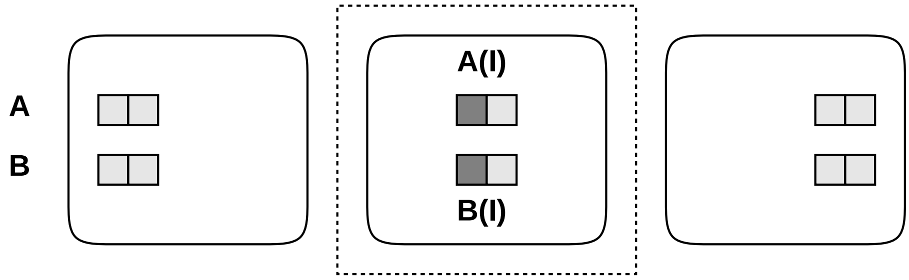
```
      SUBROUTINE SUB1(A,B)
!HPF$ INHERIT A,B              ! Mapping of A and B unknown statically
      ...
      DO I=1,100
!HPFJ   ON HOME(A(I)), LOCAL(B(I)) ! Communication unnecessary to read from B(I)
          A(I) = B(I)
      END DO
```

When the active processors specified by the ON directive and the mapping of variable B(I) are as shown in example (a) , (b), or (c) in Figure 4.6, this program is correct. In this description, variable B(I) is referenced without communication. When the active processors and the mapping of variable B(I) are as shown in example (d), the correct result cannot be expected for this program.

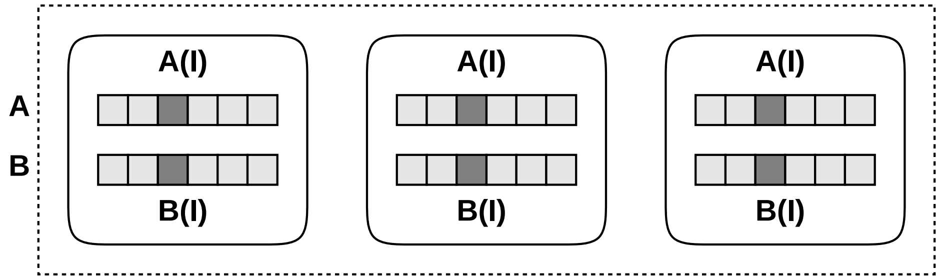
In example (d), a correct program is obtained by replacing LOCAL(B(I)) with RESIDENT(B(I)).

4.6.4.2 Defining a variable specified by LOCAL

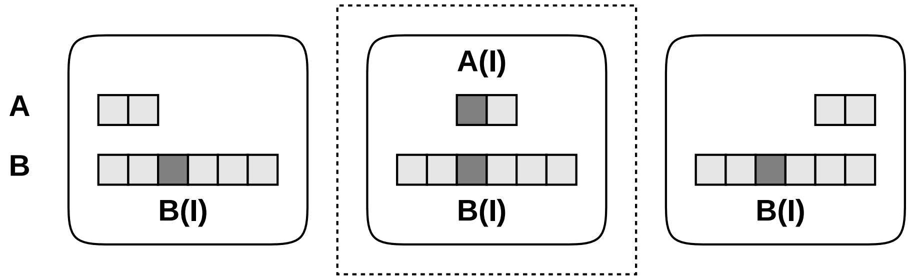
```
      SUBROUTINE SUB1(A,B)
!HPF$ INHERIT A,B                ! Mapping of A and B unknown statically
```



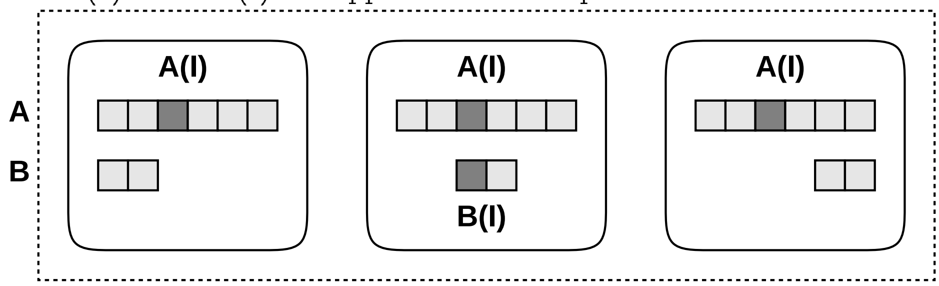
(a) When the mapping of B(I) matches an active processor



(b) When the mapping of B(I) matches multiple active processors



(c) When B(I) is mapped to all active processors and others



(d) When B(I) is mapped to some of active processors

Figure 4.6: Relationships between active processor(s) and the mapping of variable
The frame indicated by a dotted line indicates active processor(s) when ON HOME(A(I)) is specified.

```
      ...
      DO I=1,100
!HPFJ   ON HOME(A(I)), LOCAL(B(I)) ! Communication unnecessary to write to B(I)
          B(I) = A(I)
      END DO
```

When the active processors specified by the ON directive and the mapping of variable B(I) are as shown in example (a) or (b) in Figure 4.6, this program is correct. In this description, variable B(I) is defined without communication. When the active processors and the mapping of variable B(I) are as shown in example (c) or (d), the correct result cannot be expected for this program.

In example (d), a correct program is obtained by replacing LOCAL(B(I)) with RESIDENT(B(I)). In example (c), however, replacing LOCAL(B(I)) with RESIDENT(B(I)) does not correct the program.

4.6.4.3 LOCAL specification for variable with shadow

```
!HPF$ PROCESSORS P(4)
      REAL A1(400),A2(400),B(400)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A1,A2,B
!HPF$ SHADOW(1) :: A1,A2,B
      ...
!HPFJ REFLECT B          ! Can reference the shadow value of B.
!HPF$ INDEPENDENT
      DO I=2,399
!HPFJ   ON HOME(A1(I)), LOCAL(B,A2(I)) BEGIN
          A1(I)=B(I-1)+B(I)+B(I+1)
          A2(I)=B(I-1)+2*B(I)+B(I+1)
!HPFJ   END ON
      END DO
```

Variable B is specified as LOCAL; therefore, B(I-1), B(I), and B(I+1) are referenced in each processor. When the value of I is the lower boundary in a block distribution segment, the lower shadow is referenced for B(I-1). In the same way, when the value of I is the upper boundary in a block distribution segment, the upper shadow is referenced for B(I+1).

Variable A2(I) is also specified as LOCAL; therefore, its value is defined in each processor. In this case, a processor having A2(I) as a shadow object for any I is not regarded as an active processor. A value therefore is not assigned to the shadow area of A2.

4.6.4.4 LOCAL specification for ON directive extended with shadow

```
!HPF$ PROCESSORS P(4)
      REAL A1(400),A2(400),B(400)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A1,A2,B
!HPF$ SHADOW(1) :: A1,A2,B
      ...
!HPF$ INDEPENDENT
```

```
          DO I=2,399
!HPFJ   ON EXT_HOME(A1(I)), LOCAL BEGIN
            A1(I)=B(I)
            A2(I)=2*B(I)
!HPFJ   END ON
          END DO
```

The range of I executed by each processor as an active processor is extended to a range in which A1(I) is mapped as a shadow area by specifying the ON directive having the EXT_HOME clause. For example, processor P(2) is active in the range of $100 \leq I \leq 201$. For $I = 100$ and $I = 201$, the reference of B(I) means the reference of a shadow object, and the definition of A1(I) and A2(I) means the assignment of their values to a shadow object.

4.6.5 Comparison of RESIDENT with LOCAL [Reference]

4.6.5.1 Variable without shadow

RESIDENT represents that communication required for variable reference and definition is closed in the active processor set; LOCAL represents that the variable reference and definition are closed in each processor (no communication is required). In both cases, all the copies of a defined variable must be in the active processor set to prevent the value of the variable from differing inside and outside the active processor set.

These features are summarized in Table 4.1. From this table, the following result is obtained:

1. When the active processor set is composed of one processor, the semantics of LOCAL is the same as that of RESIDENT.

2. LOCAL implies RESIDENT. In other words, when LOCAL(v) is correct, even if it is replaced with RESIDENT(v), the program is always correct.

4.6.5.2 Variable with shadow

For RESIDENT, the mapping of a shadow area is not considered. So, when a variable is mapped to an active processor as a shadow object, RESIDENT cannot be specified. For LOCAL, the mapping of a shadow area is considered. Thus, a variable can be mapped to all processors in an active processor set as a data object or a shadow object.

For RESIDENT and LOCAL, when defining a value, a data object must not be mapped to outside the active processor set to prevent its value from being mismatched among processors. Since the correctness of the value of a shadow object is assured by the user, it may be mapped to outside the active processor set. Thus, the value of a shadow object may not match among processors (a data object is updated, but the result is not reflected in the corresponding shadow object).

These features are summarized as shown in Table 4.2.

4.7 Reusing Communication Schedule

The purpose of this directive is to enable efficient communication when array data is accessed irregularly in a parallel loop. The finite element method and sparse matrix processing are

Table 4.1: RESIDENT and LOCAL specifications for variable without shadow

	RESIDENT(v)	LOCAL(v)
v reference conditions	v exists on at least one processor in the active processor set. $P \cap H(v) \neq \phi$	v exists on all processors in the active processor set. $P \subseteq H(v)$
v definition conditions	v exists on at least one processor in the active processor set, and v does not exist outside the active processor set. $P \supseteq H(v)$	v exists on all processors in the active processor set, and v does not exist outside the active processor set. $P = H(v)$

v indicates a variable (named data object, array element, array section, structure component, or character substring).
P indicates the active processor set.
$H(v)$ indicates the set of processors to which v is mapped as a data object.

Table 4.2: RESIDENT and LOCAL specifications for variable with shadow

	RESIDENT(v)	LOCAL(v)
v reference conditions	v exists on at least one processor in the active processor set as a data object. $P \cap H(v) \neq \phi$	v exists on all processors in the active processor set as a data or shadow object. $P \subseteq H^{+}(v)$
v definition conditions	v exists on at least one processor in the active processor set as a data object, and v does not exist outside the active processor set as a data object. $P \supseteq H(v)$	v exists on all processors in the active processor set as a data or shadow object, and v does not exist outside the active processor set as a data object. $H(v) \subseteq P \subseteq H^{+}(v)$

v indicates a variable (named data object, array element, array section, structure component, or character substring).
P indicates the active processor set.
$H(v)$ indicates a set of processors to which v is mapped as a data object.
$H+(v)$ indicates a set of processors to which v is mapped as a data or shadow object. (($H^{+}(v) \supseteq H(v)$))

frequently used with the actual numerical simulation codes. In HPF2.0, it is often difficult for the compiler to generate codes for performing efficient communication even by using the approved extension. This directive aims at enabling efficient communication by the compiler using a means for describing the reusability of communication patterns in a program. The basic concept is planned by R. Ruehl and others in Swiss CSCS, and also used in Vienna Fortran of Vienna University.

For the indirect array reference in the irregular access pattern, the communication pattern is defined only at runtime. Most conventional compilers perform communication so that all array elements are mapped to each processors or perform communication for each element respectively. This results in a deterioration of the parallel execution performance of irregular processing.

There is an Inspector-Executor method advocated by J. Saltz and others of the University of Maryland to pack an irregular communication before and after a loop for efficient communication. This method divides loop processing into two parts: a part for obtaining an index for irregular array access (Inspector) and performing communication using the obtained index, and another part for performing loop processing (Executor). However, the processing overhead of the Inspector is too high, and parallel processing cannot be usually performed at high speed.

For simulation using the finite element method, the same data access pattern is often used repeatedly to indirectly access an array. This is because the data structure remains unchanged over a specific period to repeatedly execute a convergent loop or time evolution loop outside. The purpose of this directive is to enable efficient communication for these irregular array accesses using a means of informing the compiler that the same communication pattern appears repeatedly.

4.7.1 Syntax

Add *index-reuse-directive* to *executable-directive-extended*(H207).

J427	*index-reuse-directive*	**is**	INDEX_REUSE [(*scalar-logical-expr*)] *index-reuse-variable-list*
J428	*index-reuse-variable*	**is**	*array-variable-name*

Example.

```
        LOGICAL REUSE
        ...
        REUSE = .FALSE.
!HPFJ INDEX_REUSE (REUSE) A, B
```

4.7.2 Semantics

A DO loop or FORALL loop just after the INDEX_REUSE directive is called a target loop of the INDEX_REUSE directive. When the value of *scalar-logical-expr* is true, the programmer assures the language processor that all the following conditions are satisfied for the target loop:

- The number of iterations for each loop in the target loop range matches that for the previous execution (in a serial execution sequence).

- For all appearance of the arrays specified in *index-reuse-variable-list* in an target loop, the part-name, value of subscript for each dimension, and value of substring range in all iterations of the target loop matches that in the corresponding iterations performed at the previous execution (in a serial execution sequence).
- When the target loop includes a control construct such as a conditional branch and one of the arrays specified in *index-reuse-variable-list* appears under the control, the control flow in all iterations of the target loop matches that defined at the previous execution (in a serial execution sequence).

In this case, the language processor considers the other conditions, and if possible, it reuses a communication schedule concerning the array configured at the preceding execution of the target loop.

When the value of *scalar-logical-expr* is false, the programmer informs the language processor that one of the above conditions may not be satisfied. In this case, the language processor reconfigures a communication schedule concerning the array and saves it.

When *scalar-logical-expr* is omitted, .TRUE. is assumed to be specified.

When executing a loop the first time, the value of *scalar-logical-expr* is ignored, and the language processor always configures a communication schedule concerning the array and saves the result.

The saved communication schedule is processed in the same way as a variable having the SAVE attribute. In other words, after the execution of a procedure including the INDEX_REUSE directive returns once, the communication schedule can be reused if the procedure is called again.

Rationale.

To reuse a communication schedule for an array irregularly accessed in a loop, generally the following conditions must be satisfied for the loop:

1. The mapping of the array is the same as that of the previous execution.
2. The loop calculation mapping (allocation of each loop iteration to a processor) is the same as that of the previous execution.
3. The upper and lower bounds of the array are the same as those at the previous execution.
4. The upper and lower bounds, and increment value are the same as those of the previous execution.
5. In each loop iteration, the control flow for accessing the array is the same as that of the previous execution.
6. In each loop iteration, the subscript value of the array is the same as that of the previous execution.

In addition, the following conditions may be required depending on how the language processor is implemented:

7. When the array is a variable in a common block or a module variable, it must not be referenced or defined in a procedure called from the target loop.

8. The array must not be an actual argument of a procedure called from the target loop.
9. A procedure including the target loop must not be called recursively from the target loop.
10. The target loop must not be in a task region.
11. The dependence relation resulting from a variable in *index-reuse-variable-list* must not exist in the target loop.
12. Others

Which condition is actually required varies depending on how the language processor is implemented. Therefore, it is difficult to unify these conditions as a language specification.

To solve this problem, the programmer should specify only a condition which cannot be determined automatically by the language processor, and the other conditions should be decided by the language processor.

Specifically, the programmer instructs to the language processor whether conditions 5 and 6 above are satisfied. Condition 7 is regarded as a restriction. (*End of rationale.*)

Advice to implementors. Conditions other than 5 to 7 above should be decided by the language processor. We recommend that a communication schedule be reused as much as possible. (*End of advice to implementors.*)

4.7.3 Constraints

- A DO statement, FORALL statement or construct must appear just after the INDEX_REUSE directive.

- When an array in *index-reuse-variable-list* is a module variable or a variable in a common block, or when the array is a variable that can be accessed outside a procedure by the host association, the variable must not be referenced and defined in a procedure called from a DO or FORALL loop following the INDEX_REUSE directive.

4.7.4 Example

Example.

```
!HPF$ DISTRIBUTE A(BLOCK)
      LFLAG=.FALSE.
      DO ITIME=1,IT ! Time evolution loop, etc.
        ...
!HPFJ   INDEX_REUSE (LFLAG) A
!HPF$   INDEPENDENT, NEW(IDX,IDX2)
        DO I=1,N
          IDX = IBANK(I)
          IDX2 = IBANK2(I)
          B(I) = A(IDX) + C(I)
          IF (D(I).LT.C(I)) THEN ! The value of expr is the same
```

```
                             !     as the previous
      D(I) = A(IDX2)
    END IF
  END DO
  ...
  IF (data structure must be changed)THEN ! For the next iteration
    (data structure change code)
    LFLAG=.FALSE.          ! The communication schedule isn't reused
  ELSE IF
    LFLAG=.TRUE.           ! Reused.
  END IF
  ...
END DO
```

When LFLAG is true, the INDEX_REUSE directive assures in the language processor that the values of the logical expression of the IF statement in the DO I loop are the same as that in the previous execution for each iteration and that the values of subscripts IDX and IDX2 of A are also the same as that in the previous execution for each iteration.

In this case, the ON directive is not specified explicitly. The communication schedule of A can therefore be reused when the language processor does not change a calculation mapping for each loop execution and the mapping of A remains unchanged.

Chapter 5

Restriction and Modification for HPF2.0

In HPF/JA, some functions are restricted and modified as compared with HPF2.0. The purposes of the restriction and modification are:

- Promoting the implementation of the language processor at an early stage by limiting functions at lower priority levels
- Preventing the program execution result from being changed by the language processor by limiting and clearly defining obscure items in the HPF2.0 specification

These restrictions, especially the ones for the first purpose above, will be relaxed in the future HPF/JA version.

5.1 Restriction for HPF2.0

In HPF/JA, the following restrictions are provided in respect to HPF2.0:

- Mapping of pointer and target

  In HPF/JA, a pointer and target cannot be mapped. In the HPF2.0 specifications, the entire description in Section 8.8 and all descriptions related to the mapping of the pointer and target in other parts are invalidated.

- Mapping of structure components

  In HPF/JA, structure components cannot be mapped. In the HPF2.0 specifications, the entire description in Section 8.9 and all descriptions related to the mapping of structure components in other parts are invalidated.

  Rationale. To enable the mapping of a pointer, target, or structure component, excessive functions are required as compared with the mapping of other variables. For example, various specifications described in Sections 8.8 and 8.9 of the HPF2.0 specifications must be implemented.

  From the viewpoint of users, the functions introduced from Fortran 90 such as pointer and structure are not used widespread in Japan. Therefore, even if the mapping of the pointer and structure component is restricted, many users do not feel inconvenienced. They rather hope for implementation of frequently used functions at an early stage.

For these reasons, in this HPF/JA version, the mapping of a pointer, a target, and structure component is restricted.

(*End of rationale.*)

- INDIRECT distribution format

  In HPF/JA, INDIRECT cannot be specified in the distribution format. In the HPF2.0 specification, the INDIRECT line is deleted from the right part of syntax H810 in Section 8.10. In addition, all descriptions related to the INDIRECT distribution format in other parts are deleted.

  Rationale. To enable the INDIRECT distribution, a large index conversion table is generally required at execution. Furthermore, to access this table at high speed, the entire table must be copied for each processor. As a result, scalability in memory use is reduced.

  To resolve this problem and enhance performance with the INDIRECT distribution, the inspector/executor method is necessary. This method is not however considered to be sufficiently matured. When using the INDIRECT distribution, performance will be unpredicted.

  From these reasons, in this HPF/JA version, the INDIRECT distribution is restricted. (*End of rationale.*)

- RANGE specification

  In HPF/JA, the RANGE directive cannot be specified. In the HPF2.0 specifications, the entire description in Section 8.11 and all descriptions related to the RANGE directive in other parts are invalidated.

  Rationale. The RANGE directive is most useful when informing the language processor that the distribution format is not INDIRECT. If the distribution can be INDIRECT, the language processor would generate code whose efficiency is very low.

  In this HPF/JA version, the INDIRECT distribution format is restricted; thus, the RANGE directive has become less necessary, and it also is restricted.

  When INDIRECT distribution is allowed in the future, the RANGE directive may be required. These items should be discussed together. (*End of rationale.*)

- Extrinsic procedure other than HPF_GLOBAL and Fortran_LOCAL

  In HPF/JA, an extrinsic procedure other than HPF_GLOBAL and Fortran_LOCAL cannot be used. In the HPF2.0 specification, only the parts related to HPF_GLOBAL and Fortran_LOCAL in the description of Chapter 11 are assumed to be valid.

  Rationale. To implement a language processor at an early stage, this system uses only Fortran_LOCAL (required as a local model at the minimum) and HPF_GLOBAL (that is HPF itself). (*End of rationale.*)

- Restriction of shadow width for CYCLIC distribution

  In HPF/JA, the shadow width for a dimension whose arrays are cyclically distributed must satisfy the following conditions:

$$(w_l + w_h) \times |s| \leq m \times (p - 1)$$

  Here,

w_l	Lower shadow width
w_h	Upper shadow width
s	Stride of the alignment to the ultimate aligned target
m	Block size at distribution of the ultimate aligned target
p	Extent of the processor arrangement

  Here, s, m, and p are the ones corresponding to the array dimension.

  The full SHADOW is excluded from this restriction.

  Rationale. This restriction is a condition provided to prevent the same array elements from being stored in multiple places of a shadow area on one processor. See Section 4.3. (*End of rationale.*)

- Asynchronous I/O

  In HPF/JA, asynchronous I/O defined for the HPF approved extension cannot be used. In the HPF2.0 specifications, the entire description of Chapter 10 and all descriptions related to asynchronous I/O in other parts are invalidated.

  Rationale. Asynchronous I/O is not a subject special to a distributed-memory parallel computer to be targeted by HPF. So, this is restricted in order to implement a language processor at an early stage. (*End of rationale.*)

5.2 Modification for HPF2.0

In HPF/JA, a part of the HPF2.0 specification is modified as follows:

- Mapping of DYNAMIC variable at return of procedure

  In HPF/JA, the remapping of a dummy argument executed in a called procedure must not be reflected in the mapping of an actual argument even if the dummy and actual arguments have the DYNAMIC attribute.

  In the HPF2.0 specifications, the entire second paragraph of the first item in Section 8.6,

  "The effect of any redistribution of the dummy after the procedure returns to the caller is dependent on the attribute of the actual argument....then the new mapping must match one of the formats specified in the range directive. "

  is modified as follows.

  "The effect of any redistribution of the dummy after the procedure returns to the caller does not remain regardless of the attribute of the actual argument. The mapping of an actual argument is the same as that defined before call. This is because *the remapping of an argument is not visible to the caller* as described in Section 4.2 of the HPF2.0 specification as a principle."

Rationale. In the HPF2.0 specification, when both actual and dummy arguments have the DYNAMIC attribute, the remapping of the dummy argument is reflected in the actual argument. But if an array is remapped in a callee, the mapping of other arrays ultimately aligned to the array in the caller is obscure. For example, this problem occurs in the following program:

```
      PROGRAM EX1
      REAL A(10),B(10)
!HPF$ DYNAMIC A,B
!HPF$ ALIGN A(I) WITH B(I)
!HPF$ DISTRIBUTE B(BLOCK)
          :
      CALL SUB(B)
          :
      END

      SUBROUTINE SUB(B)
      REAL B(10)
!HPF$ DYNAMIC B
!HPF$ DISTRIBUTE B(BLOCK)
        :
!HPF$ REDISTRIBUTE B(CYCLIC)
        :
      RETURN
      END
```

In this example, array B is remapped from BLOCK to CYCLIC in procedure SUB. If this remapping is reflected in the caller when returning, the mapping of array A aligned to array B is obscure: that is, we do not know whether array A should be remapped to CYCLIC together with array B.

In addition, there is another problem when array A is remapped as shown below.

```
      PROGRAM EX2
      REAL A(10),B(10)
!HPF$ DYNAMIC A,B
!HPF$ ALIGN A(I) WITH B(I)
!HPF$ DISTRIBUTE B(BLOCK)
          :
      CALL SUB(A)
          :
      END

      SUBROUTINE SUB(A)
      REAL A(10)
!HPF$ DYNAMIC A
```

```
!HPF$ DISTRIBUTE A(BLOCK)
          :
!HPF$ REDISTRIBUTE A(CYCLIC)
          :
      RETURN
      END
```

When an attempt is made to inherit the CYCLIC distribution in the callee to the caller , the alignment relation between arrays A and B is destroyed.

Moreover, when the array is realigned to a local variable or template in the callee, it loses its *align-target* after returning to the caller.

A clear description in respect to these problems is omitted in the HPF2.0 specification, and the language processor could interpret them at its convenience. To prevent this in HPF/JA, as a temporary specification, the mapping is returned to that in the caller when returning from the callee regardless whether the DYNAMIC attribute is specified. (*End of rationale.*)

- NEW and REDUCTION specifications for dummy argument

  HPF/JA allows a dummy argument to be specified in NEW and REDUCTION clause. In the HPF2.0 specification, the first item, "dummy argument," is deleted from restriction 4 in Section 5.1.

  Rationale. The purpose of restriction 4 above is to assure that the NEW and REDUCTION variables are not related to other variables with an alias. For the dummy argument, some restrictions are provided in the Fortran specification so that an alias problem does not occur. For example, if a dummy argument is related to another variable with an alias, the value cannot be assigned without using the dummy argument. Therefore, even if a dummy argument is specified as a NEW or REDUCTION variable, an alias problem does not occur.

  In an actual program, a situation in which a dummy argument is to be handled as a NEW variable often occurs: for example, when an array used as a temporary work area is passed to the callee procedure as an argument by allocating it in the caller. This temporary array may be processed as a NEW variable in many cases.

  For these reasons, in HPF/JA, the NEW and REDUCTION specifications are allowed for a dummy argument. (*End of rationale.*)

Annex A

Syntax Rules

This Appendix collects the formal syntax definitions of this HPF/JA Language Specification.

A.2 Notation and Syntax

A.2.2 Syntax of Directives

J201	*hpfja-directive-line*	**is**	*hpfja-directive-origin hpf-directive*
J202	*hpfja-directive-origin*	**is**	`!HPFJ`
		or	`CHPFJ`
		or	`*HPFJ`
J203	*specification-directive-ja*	**is**	*processors-directive*
		or	*subset-directive*
		or	*align-directive*
		or	*distribute-directive*
		or	*inherit-directive*
		or	*template-directive*
		or	*combined-directive*
		or	*sequence-directive*
		or	*dynamic-directive*
		or	*shadow-directive*
		or	*asyncid-directive*
J204	*executable-directive-ja*	**is**	*independent-directive-ja*
		or	*realign-directive-ja*
		or	*redistribute-directive-ja*
		or	*on-directive*
		or	*resident-directive*
		or	*asynchronous-directive*
		or	*asyncwait-directive*
		or	*reflect-directive*
		or	*local-directive*
		or	*index-reuse-directive*

J205 *executable-construct-ja* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *if-construct*
or *where-construct*
or *on-construct*
or *resident-construct*
or *task-region-construct*
or *asynchronous-construct*
or *local-construct*

A.3 HPF/JA Extension Related to Parallel Processing Specification

A.3.1 Specification of REDUCTION Kind

J301 *independent-directive-ja* **is** `INDEPENDENT` [, *new-clause*] [, *reduction-clause-ja-list*]

J302 *reduction-clause-ja* **is** `REDUCTION` ([*reduction-kind* :] *reduction-spec-list*)

J303 *reduction-kind* **is** *reduction-operator*
or *reduction-function*
or *maxmin-kind*

J304 *reduction-operator* **is** `+`
or `*`
or `.AND.`
or `.OR.`
or `.EQV.`
or `.NEQV.`

J305 *maxmin-kind* **is** `FIRSTMAX`
or `FIRSTMIN`
or `LASTMAX`
or `LASTMIN`

J306 *reduction-spec* **is** *reduction-variable* [/ *location-variable-list* /]

J307 *location-variable* **is** *scalar-variable-name*

Constraint: When *reduction-kind* is *maxmin-kind*, *reduction-spec* must have *location-variable-list*. When *reduction-kind* is not *maxmin-kind* or *reduction-kind* is omitted, *reduction-spec* must not have *location-variable-list*.

Constraint: When *reduction-kind* is *maxmin-kind*, *reduction-variable* in *reduction-spec* must be *scalar-variable-name*.

Constraint: The type of variable specified in *reduction-variable* must be defined for each *reduction-kind* value as follows:

Logical type for `.AND.`, `.OR.`, `.EQV.`, and `.NEQV.`

Integer type for `IAND`, `IOR`, and `IEOR`

Numeric type for `+` and `*`

Integer or real type for `MAX`, `MIN`, `FIRSTMAX`, `FIRSTMIN`, `LASTMAX`, and `LASTMIN`

Constraint: *reduction-variable* specified in *reduction-clause* without *reduction-kind* must be referenced in the reduction statement format in the loop defined in Section 5.1.3 of the HPF2.0 specification. (*reduction-variable* specified in *reduction-clause* with *reduction-kind* may be referenced in any format in a loop.)

Constraint: A variable specified as *reduction-variable* or *location-variable* must not be specified two or more times in the same *independent-directive*. It must not also be specified in *new-clause* and *reduction-clause* within the range of the succeeding *do-stmt*, *forall-stmt* and *forall-construct* (that is, loop body in the source program) to which the *independent-directive* applies.

A.4 HPF/JA Extension for Communication Optimization

A.4.1 Asynchronous Transfer Function

J401	*asyncid-directive*	is	`ASYNCID` *async-id-list*
J402	*async-id*	is	*async-id-name*

Constraint: When `SAVE` is defined in *combined-directive*, `ASYNCID` must also be defined.

J403	*asynchronous-directive*	is	`ASYNCHRONOUS` *asynchronous-stuff*
J404	*asynchronous-stuff*	is	`(` [`ID =`] *async-id* `)` [`,` *nobuffer-clause*]
J405	*asynchronous-construct*	is	*hpfja-directive-origin block-asynchronous-directive* *block* *hpfja-directive-origin end-asynchronous-directive*
J406	*block-asynchronous-directive*	is	`ASYNCHRONOUS` *asynchronous-stuff* `BEGIN`
J407	*end-asynchronous-directive*	is	`END ASYNCHRONOUS`
J408	*asyncwait-directive*	is	`ASYNCWAIT (` [`ID =`] *async-id* `)`
J409	*nobuffer-clause*	is	`NOBUFFER`
J410	*redistribute-directive-ja*	is	[*async-prefix*] *redistribute-directive*
J411	*realign-directive-ja*	is	[*async-prefix*] *realign-directive*
J412	*async-prefix*	is	`ASYNC (` [`ID =`] *async-id* `)`

A.4.2 Extension of SHADOW Directive

J413	*shadow-spec-ja*	is	*width*
		or	*low-width* `:` *high-width*
		or	*full-width*

J414 *full-width* **is** *

Constraint: The length of *shadow-spec-ja-list* must be equal to the rank of *shadow-target*.

Constraint: When *full-width* is specified as *shadow-spec-ja*, *full-width* must be specified in all dimensions.

A.4.4 REFLECT Directive

J415 *reflect-directive* **is** [*async-prefix*] `REFLECT` *reflect-object-list*

J416 *reflect-object* **is** *object-name*

Constraint: All processors onto which a data or shadow object of *reflect-object* is distributed must be active.

A.4.5 Extension of HOME Clause in ON Directive

J417 *home-ja* **is** `HOME (` *variable* `)`
or `HOME (` *template-elmt* `)`
or `EXT_HOME (` *variable* [, *shadow-attr-stuff*] `)`
or `(` *processors-elmt* `)`

Constraint: The length of *shadow-spec-list* specified by *shadow-attr-stuff* in the `EXT_HOME` clause must match the rank of the parent object of *variable*.

Constraint: The upper and lower shadow widths of each dimension specified by *shadow-attr-stuff* in the `EXT_HOME` clause must be equal to or less than the shadow width of the parent object of *variable* respectively.

Constraint: In *shadow-attr-stuff*, *shadow-spec-ja* must not be *full-width* (asterisk).

A.4.6 LOCAL Clause and Directive

J418 *on-stuff-ja* **is** *home-ja* [, *on-optional-clause-list*]

J419 *on-optional-clause* **is** *resident-clause*
or *local-clause*
or *new-clause*

J420 *local-clause* **is** `LOCAL` *local-stuff*

J421 *local-stuff* **is** [(*local-object-list*)]

J422 *local-directive* **is** `LOCAL` *local-stuff*

J423 *local-construct* **is**
hpfja-directive-origin block-local-directive
block
hpfja-directive-origin end-local-directive

J424 *block-local-directive* **is** `LOCAL` *local-stuff* `BEGIN`

J425 *end-local-directive* **is** `END LOCAL`

J426 *local-object* **is** *object*

A.4.7 Reusing Communication Schedule

J427 *index-reuse-directive* **is** `INDEX_REUSE` [(*scalar-logical-expr*)] *index-reuse-variable-list*

J428 *index-reuse-variable* **is** *array-variable-name*

Annex B

Syntax Cross-reference

This Appendix cross-references symbols used in the formal syntax rules. Rule identifiers beginning with "J" refer to syntax rules of this HPF/JA Language Specification; the full rule may be found in Appendix A. Rule identifiers beginning with "H" refer to syntax rules of this High Performance Fortran Language Specification. Rule identifiers beginning with "R" refer to syntax rules of the Fortran Language Standard ("Fortran 95").

B.1 Nonterminal Symbols That Are Defined

Symbol	**Defined**	**Referenced**			
action-stmt	R216	J205			
align-directive	H313	J203			
async-id	J402	J401	J404	J408	J412
async-prefix	J412	J410	J411	J415	
asynchronous-construct	J405	J205			
asynchronous-directive	J403	J204			
asynchronous-stuff	J404	J403	J406		
asyncid-directive	J401	J203			
asyncwait-directive	J408	J204			
block	R801	J405	J423		
block-asynchronous-directive	J406	J405			
block-local-directive	J424	J423			
case-construct	R808	J205			
combined-directive	H301	J203			
distribute-directive	H305	J203			
do-construct	R816	J205			
dynamic-directive	H804	J203			
end-asynchronous-directive	J407	J405			
end-local-directive	J425	J423			
executable-construct-ja	J205				
executable-directive-ja	J204				
full-width	J414	J413			
high-width	H823	J413			
home	H907				
home-ja	J417	J418			

hpf-directive	H203	J201		
hpfja-directive-line	J201			
hpfja-directive-origin	J202	J201	J405	J423
if-construct	R802	J205		
independent-directive-ja	J301	J204		
index-reuse-directive	J427	J204		
index-reuse-variable	J428	J427		
inherit-directive	H401	J203		
local-clause	J420	J419		
local-construct	J423	J205		
local-directive	J422	J204		
local-object	J426	J421		
local-stuff	J421	J420	J422	J424
location-variable	J307	J306		
logical-expr	R725	J427		
low-width	H822	J413		
maxmin-kind	J305	J303		
new-clause	H502	J301	J419	
nobuffer-clause	J409	J404		
on-construct	H904	J205		
on-directive	H902	J204		
on-optional-clause	J419	J418		
on-stuff	H903			
on-stuff-ja	J418			
processors-directive	H329	J203		
processors-elmt	H909	J417		
realign-directive	H803	J411		
realign-directive-ja	J411	J204		
redistribute-directive	H802	J410		
redistribute-directive-ja	J410	J204		
reduction-clause-ja	J302	J301		
reduction-function	H506	J303		
reduction-kind	J303	J302		
reduction-operator	J304	J303		
reduction-spec	J306	J302		
reduction-variable	H504	J306		
reflect-directive	J415	J204		
reflect-object	J416	J415		
resident-clause	H910	J419		
resident-construct	H913	J205		
resident-directive	H912	J204		
sequence-directive	H333	J203		
shadow-attr-stuff	H819	J417		
shadow-directive	H817	J203		
shadow-spec-ja	J413			
specification-directive-ja	J203			
subset-directive	H901	J203		
task-region-construct	H917	J205		

template-directive	H331	J203
template-elmt	H908	J417
variable	R601	J417
where-construct	R739	J205
width	H821	J413

B.2 Nonterminal Symbols That Are Not Defined

Symbol	**Referenced**
array-variable-name	J428
async-id-name	J402
object	J426
object-name	J416
variable-name	J307

B.3 Terminal Symbols

Symbol	**Referenced**
`!HPFJ`	J202
`(`	J302 J404 J408 J412 J417 J421 J427
`)`	J302 J404 J408 J412 J417 J421 J427
`*`	J304 J414
`*HPFJ`	J202
`+`	J304
`,`	J301 J404 J417 J418
`.AND.`	J304
`.EQV.`	J304
`.NEQV.`	J304
`.OR.`	J304
`/`	J306
`:`	J302 J413
`=`	J404 J408 J412
`ASYNC`	J412
`ASYNCHRONOUS`	J403 J406 J407
`ASYNCID`	J401
`ASYNCWAIT`	J408
`BEGIN`	J406 J424
`CHPFJ`	J202
`END`	J407 J425
`EXT_HOME`	J417
`FIRSTMAX`	J305
`FIRSTMIN`	J305
`HOME`	J417
`ID`	J404 J408 J412

INDEPENDENT	J301			
INDEX_REUSE	J427			
LASTMAX	J305			
LASTMIN	J305			
LOCAL	J420	J422	J424	J425
NOBUFFER	J409			
REDUCTION	J302			
REFLECT	J415			