

JAHPF (Japan Association for High Performance Fortran) is a coalition of about 20 compiler specialists of vendor and about 20 users of high performance computing in Japan to promote the HPF language. It started 1996 and its major activities under way are as follows;

- \*Proposing a set of HPF extensions (HPF/JA) to improve application performance,
- \*Accumulating experiences of using HPF, and making feedback to HPF specification and compiler,
- \*Developing HPF benchmark programs,
- \*Translating HPF documents into Japanese,
- \*Educating HPF users.

The HPF 2.0 basic specifications and approved extensions were published, but functionalities for an optimization of data transfers seem to be insufficient from the user's point of view. The problems in the current HPF were discussed in JAHPF, and the JAHPF introduced the original language extensions HPF/JA. HPF/JA 1.0 includes following functionalities;

## Reduction kind

In HPF/JA, reduction variables may be referred to in any syntax, even in procedure calls, as long as the user specifies the reduction kind in the REDUCTION clause, while HPF2.0 allows reduction variables to appear only in restricted forms of reduction statements.

### <Specification>

```
!HPFJ INDEPENDENT, REDUCTION &
!HPFJ& ( [reduction-kind:] reduction-spec-list )
```

Here *reduction-kind* is one of the following:

*operator*: +, \*, .AND., .OR., .EQV., .NEQV.

*intrinsic*: MAX, MIN, IAND, IOR, IEOR

*maxmin-kind*: FIRSTMAX, LASTMAX, FIRSTMIN, LASTMIN.

If *reduction-kind* is *operator* or *intrinsic*, *reduction-spec-list* is simply a list of *reduction-variables*. If *reduction-kind* is *maxmin-kind*, each member of *reduction-spec-list* is of the form *reduction-variable/loc-*

*variable-list*/. This format enables the user to specify the MAXLOC/MINLOC computation associated with the MAX/MIN computation for the *reduction-variable*.

## Asynchronous communication

It enables users to explicitly specify the possibility of overlapping communication and computation in the similar way to asynchronous I/O in the HPF2.0 approved extensions.

### <Specification>

```
!HPFJ ASYNCHRONOUS ( ID=async-id ) BEGIN
    statements #1
!HPFJ END ASYNCHRONOUS
    statements #2
!HPFJ ASYNC WAIT ( ID=async-id )
```

Statements #1 can be a set of the following;

- Intrinsic assignment statement,
- FORALL statement,
- REDISTRIBUTE directive,
- REALIGN directive,
- REFLECT directive (HPF/JA).

The ASYNCHRONOUS construct specifies that the communications caused by the statements #1 can be carried out simultaneously with the succeeding computations of the statements #2. The statements #2 can be arbitrary Fortran statements or HPF directives, but they may not interfere with the execution of the statements #1.

### <Example>

```
!HPFJ ASYNCHRONOUS ( ID ) BEGIN
!HPF$ REDISTRIBUTE A ( *, CYCLIC )
!HPFJ END ASYNCHRONOUS
    ...
!HPFJ ASYNC WAIT ( ID )
```

## User controllable SHADOW

In the HPF/JA SHADOW, the coherence of the SHADOW data can be controlled with the REFLECT and LOCAL directive to reduce the communication. Moreover, overlapped computations in the SHADOW area can be specified with the

EXT\_HOME and LOCAL clauses.

<Specification>

```
!HPFJ REFLECT object-list

!HPFJ LOCAL(object-list) BEGIN
    block
!HPFJ END LOCAL

!HPFJ ON EXT_HOME(variable [,shadow-attr-stuff]) , &
!HPFJ& LOCAL(object-list)
```

The REFLECT directive specifies that each value of the shadow data specified in the *object-list* should be updated so that its value is the same as that of its original data object stored in its owner. In other words, the consistency of the data is guaranteed at the point where the REFLECT directive is placed (Fig.1).

Users can eliminate the redundant communication for the shadow area by combining the REFLECT and the LOCAL directive.

When the EXT\_HOME and LOCAL clause are specified with ON directive, the owner area is interpreted including the SHADOW area. For example, the original area of A(I) is owned by an abstract processor, and its corresponding SHADOW area is owned by its neighboring processor, then the computation is carried out individually by both of these processors. In this way, overlapped computation can be achieved (Fig.2).

In the HPF2.0 approved extension, the RESIDENT clause is introduced to explicitly tell the compiler the need of communication. However, it only asserts communication requirements across the active processor set, and there is no guarantee whether the communication inside the processor set is required or not. On the other hand, the LOCAL clause asserts that there are no communications

required for the specified variables.

### Communication schedule reuse for irregular array accesses

Generating efficient communications for unstructured programs is essential to improve the applicability of data parallel languages.

UNCHANGED\_INDEX directive gives users a handle to assert that data access patterns for a specific indirect array access remain the same through multiple invocations of the enclosing DO loop, in order to reduce the cost of generating communication schedules.

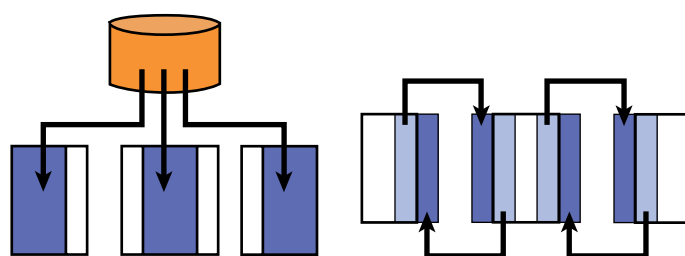
<Specification>

```
!HPFJ UNCHANGED_INDEX [(scalar-logical-expr)] &
!HPFJ& variable-list
```

The directive may be placed just before an independent DO or FORALL loop. It gives compilers the information that indices of the arrays specified in the *variable-list* remain the same as those in the previous invocation of the loop when the value of *scalar-logical-expr* is true. Communication schedules are determined mainly by computation mapping of the loop, data mapping of the target array and the values of access indices. When a compiler can assert those factors remain the same in the consecutive invocation of a loop, it may reuse the communication schedule generated in the previous calculation.

For more information please visit URL

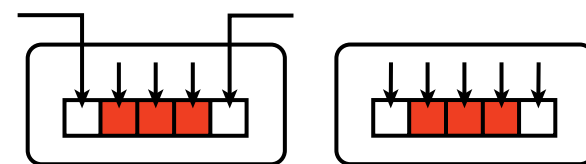
<http://www.tokyo.rist.or.jp/jahpf>



(a) READ (\*) A

(b) REFLECT A

Fig.1 REFLECT



(a) ON HOME + REFLECT

(b) ON EXT \_\_ HOME

Fig.2 CONTROLLING SHADOW DATA