HPF/JA: An HPF Extension for Real-World Scientific Simulations

Yoshiki Seo
C&C Media Research Laboratories
NEC Corporation
seo@ccm.cl.nec.co.jp

Abstract

JAHPF is a coalition of about 20 compiler vendor members and 20 HPC user members in Japan to promote the HPF language. In this paper, outline of HPF/JA 0.9, a preliminary draft of set of HPF extensions to enhance applicability of HPF in developing real-world parallel applications is presented.

1 Introduction

JAHPF(Japan Association for High Performance Fortran) is a coalition of about 20 compiler vendor members and 20 HPC user members in Japan to promote the HPF language. It started 1996 and its major activities under way are; a) proposing a set of HPF extensions (HPF/JA) to enhance applicability of HPF in developing real-world parallel applications, b) accumulating experiences of using HPF and making feedback to HPF specification and compiler implementations, c) developing HPF benchmark programs, d) translating HPF documents into Japanese and e) educating HPF users.

We developed the preliminary version of HPF extensions (HPF/JA 0.9) which includes the functionalities such as asynchronous communication, user controllable SHADOW, reduction kind, communication schedule reuse and parallelization support of loops with procedure calls.

We believe that HPF is the most promising language interface for distribute memory parallel computers in the near future. However, it was designed assuming very high-level capability of language processing systems, which seems to be the main reason why HPF has not been used widely in parallelizing real-world scientific applications. The major objective of HPF/JA is to complement the gap between the ideal and the current compiler technologies by defining a set of HPF extensions.

Asynchronous communication 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.

In the HPF/JA SHADOW, the coherence of the SHADOW data can be controled with a REFLECT directive to reduce the communication to it. Moreover, overlapped computation in it to reduce communication can be specified with EXT_HOME clause in an ON directive.

The REDUCTION directive is extended so that reduction variables may be referred to in any syntax, as long as a user specifies the reduction kind in the REDUCTION clause, while HPF2.0 allows reduction variables to appear only in restricted forms of reduction statements.

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 invocation of the enclosing DO loop, in order to reduce the cost of generating communication patterns.

HPF/JA allows a user to describe exact array access range information in INTERFACE BLOCK for parallelizing a loop including procedure calls with side effects.

In the talk, first, specification of the above extensions is presented. Then, their applicability to the application programs such as CFD code, particle in cell code, and ab-initio molecular dynamics code.

In the following sections, outline of the HPF/JA extensions are presented.

2 Communication Pattern Reuse

Generating efficient communications for unstructured programs is essential to expand the applicability of data parallel languages. In particular, irregular data

accesses are commonly used in many of the FEM (finite element method) applications or to handle sparse matrices. With the inspector-executor technique , compilers can pack the communication and generate the required communications outside of loops which include indirect array accesses. However, the cost of generating the communication patterns in the inspector phases are usually very expensive, and it reduces the efficiency of the parallel execution. In this case, when the data access pattern for the indirect array accesses remains the same for the multiple invocations of a certain `DO` loop, the inspector cost can be reduced by the reuse of the generated communication patterns. The technique is originally proposed by researchers in Swiss CSCS and Vienna University.

Specification

```
!HPF$  UNCHANGED_INDEX (<logical-expression>) &
!HPF$                  <variable-list>
```

The directive may be placed just before an `INDEPENDENT DO` loop. It gives compilers the information that array access indices of arrays specified in the variable-list remain the same as those in the previous invocation of the loop when the value of `<logical-expression>` is `TRUE`. Communication pattern is determined by computation mapping of the loop, data mapping of the target array and the value of array access index. When a compiler can assert the three factors remain the same in the consecutive invocation of a loop, it can reuse of the communication pattern generated in the previous inspector calculation.

Example

```
  LOGICAL LFLAG

  IF(pattern is changed) LFLAG=.FALSE.
!HPF$  UNCHANGED_INDEX (LFLAG) A
  DO I=1,N
    idx = ...
    B(I) = A(idx) + ...
  ENDDO
  LFLAG=.TRUE.
```

Communication pattern can be determined by computation mapping of the loop, data mapping of the target array and the value of array access index. In this example, when the computation mapping of the loop and the data mapping of A remain unchanged and `LFLAG` is `.TRUE.`, the communication schedule can be reused.

3 Asynchronous Communication

Generally speaking, communications are very expensive compared to computations on distributed memory parallel systems. Efficiency of parallel executions can be improved if the communication cost can be hidden behind the computational cost by overlapping the communications and the computations. The asynchronous communication functionality in HPF/JA gives users a means to specify it in the similar way as the asynchronous I/O in the HPF2.0 approved extensions.

Specification There are two forms of directive to specify the asynchronous communication, namely `ASYNCHRONOUS` construct and `ASYNC` prefix.

```
!HPF$ ASYNCHRONOUS (ID=<async-id>) BEGIN
  statements #1
!HPF$ END ASYNCHRONOUS
  statements #2
!HPF$ WAIT (ID=id)
```

`<async-id>` is a variable name of newly introduced type to describe the correspondence of the asynchronous communication and `WAIT` for it. The variable should be declared in an `ASYNCID` declaration.

```
!HPF$ ASYNCID :: <async-id-list>
```

A set of communications to be carried out asynchronously are specified in statements #1, and they can include:

- an intrinsic assignment [1]
- a `FORALL` statement which includes just an intrinsic assignment statement as its body.
- REDISTRIBUTE directive
- REALIGN directive
- REFLECT directive (HPF/JA).

The `ASYNCHRONOUS` construct specifies that the communications cased by the above statements and the succeeding computations (statements #2) can be carried out simultaneously. A set of statements #2 can be a block of arbitrary Fortran statements, but they may not interfere with the execution of statements #1.

The simplified prefix form `ASYNC` is spported as well. The meaning is almost the same as the above, but the asynchronous communication can be expressed in a single directive by putting the prefix `ASYNC` on top of a (`REDISTRIBUTE, REALIGN or REFLECT`) directive as follows.

```
!HPF$ ASYNC (ID=id) REDISTRIBUTE (BLOCK)
  statements #2
!HPF$ WAIT (ID=id)
```

[1] Assignment statement which does not include user defined operator or pointer assignment.

4 User Controllable Shadow

Shadow area is an extra memory space for remote data so as to make the address space of them and that of local data contiguous. By allocating the area, the cost of address translations of global to local for distributed array accesses can be reduced. Because computing the sufficient shadow area for each distributed array throughout the program execution is very difficult, and it requires compiler's sophisticated interprocedural analysis, HPF2.0 supports a `SHADOW` directive as an approved extension to allow users explicitly declare the width of the shadow area.

However, in the current HPF2.0 specification, the control of the area owes completely to a compilation system, and there are no user's control on it, which frequently results in redundant communications. In the HPF/JA shadow area, users can control the communication on it. Moreover, it is possible to reduce communication cost by overlapped computation. Computation mapping considering the shadow area can be explicitly specified so that boundary calculations with the shadow area can be computed by its owner of the shadow area as well as its actual owner.

Specification

```
!HPF$ REFLECT <object-list>
!HPF$ ON HOME <data-object> &
!HPF$         [WITHSHADOW[<shadow-attr-stuff>]]
!HPF$ ON HOME..., [LOCAL(object-list)]
```

The REFLECT directive specifies that each value of the shadow data specified in the `<object-list>` should be made the same as that of its original data object stored in its owner. In other words, the coherence of the data is guaranteed at the point where the directive is placed. Users can eliminate the redundant communication for the shadow area by combining the `REFLECT` and the `LOCAL` directive.

When `WITHSHADOW` is specified with `ON HOME`, the owner area is interpreted including the `SHADOW` area. For example, when the original area of A(I) is owned by an abstract processor, and its corresponding `SHADOW` area by its adjacent processor, the computation is carried out by both of these processors. In this way, overlapped computation can be specified with the directive.

In the HPF2.0 approved extension, the `RESIDENT` clause is introduced to explicitly tell the compiler the no 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 variable.

Example

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

  DO I=1,N
   A(I) = ...
  ENDDO

!HPF$ REFLECT <object-list>

  DO I=2,N-1
!HPF$ ON HOME(B(I)),LOCAL(A)
    B(I)=A(I-1)+A(I)+A(I+1)
  ENDDO
```

In this case, the value of `A` is set in the first loop. Then, the value of the `SHADOW` area for `A` is updated by the `REFLECT` directive. The LOCAL clause in the second loop asserts that communication for `A` is unnecessary.

```
  REAL A(N),B(N)
!HPF$ DISTRIBUTE (BLOCK)::A,B
!HPF$ SHADOW(1)::A
  DO I=1,N
!HPF$ ON HOME(A(I)) WITHSHADOW
   A(I) = ...
  ENDDO

  DO I=2,N-1
!HPF$ ON HOME(B(I)),LOCAL(A)
    B(I)=A(I-1)+A(I)+A(I+1)
  ENDDO
```

This time, there are no communications required, because the `SHADOW` data are computed by the overlapped computation, which is specified with the WITHSHADOW clause. This kind of optimization is effective when the computational cost to set the SHADOW of `A` is less that the communication cost of the `SHADOW` area.

5 Reduction Kind

The current HPF2.0 specification on the reduction computation is dependent on the compiler's capability of automatically recognizing the kind of reduction operation, such as summation, and finding minimam or maximum values. However, it is often the case that the recognition is difficult, especially when the reduction takes place across the procedure boundaries. Therefore, we extend the reduction directive in HPF 2.0, and users can explicitly specify the kind of reduction operation.

```
!HPF$ INDEPENDENT,REDUCTION[:<rediction-kind>] &
!HPF$                     <reduction-variable-list>
```

Here, `<reduction-kind>` is `ALL`, `ANY`, `IALL`, `IANY`, `IPARITY`, `MAXVAL`, `MINVAL`, `PARITY`, `PRODUCT` or `SUM`. When `<reduction-kind>` is `MAXVAL` or `MINVAL`, `<reduction-variable>` may be decorated with the `FIRSTLOC` or `LASTLOC` sub-clause, which specifies a variable the first or the last location of `MAX/MIN` occurrence is preserved.

Example

```
  AMAX = -HUGE
!HPF$ INDEPENDENT, &
!HPF$   REDUCTION:MAXVAL(QMAX:FIRSTLOC(ILOC))
  DO I=1,N
    IF(QMAX.LT.Q(I)) THEN
      QMAX=Q(I)
      ILOC=I
    ENDIF
  ENDDO
```

In the example, the value of the smallest `I` when the maximum value is set to QMAX is preserved in `ILOC`. An example which demonstrates the effectiveness of specifying the REDUCTION kind without `FIRSTLOC` or `LASTLOC` is shown in the next section.

6 Parallelization of loops including procedure calls

Parallelization of a `DO` loop which includes procedure calls in its body is indispensable to a wide range of real-world applications. However, a user can not explicitly control the parallelization in the current HPF2.0 specification. Users can put an `INDEPENDENT` directive for it, or a very sophisticated compiler may be able to parallelize certain pattern of these loops by interprocedural analysis, but whether the loop is parallelize of not is compiler's implementation dependent.

In order to parallelize a loop including procedure calls, a compiler have to know the exact information about the accesses of data objects. Especially, array access region information is essential to handle communications at the caller sites.

HPF/JA allows a user to describe the exact array access range information by expanding the specification of the `INTENT` attribute.

Specification

```
INTENT(<intent-spec>) &
       array-name(<section-subscript-list>)
```

`<intent-spec>` is `IN`, `OUT`, `INOUT` or `NONE`. The attribute `NONE` asserts that there are no access on the specified array or array section. The meaning of the other attributes remains the same as those in Fortran90. Array access information of `COMMON` data can be specified in the `INTENT` statement, which allow users to access them in the callee routines. `<vector-subscript>` may not be specified in `<section-subscript-list>`.

Example

```
  REAL*8 A(1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
INTERFACE
  SUBROUTINE COMPUTE_COLUMN(A,I)
    REAL*8 A(1000)
    INTENT(INOUT):: A(1:N,I)
    END SUBROUTINE
END INTERFACE

!HPF$ INDEPENDENT, NEW(I)
  DO I=1,1000
    CALL COMPUTE_COLUMN(A,I)
  ENDDO
```

The array access range of `A` in the subroutine is explicitly described in the INTENT attribute. With the information, compiler can insert the required communication (transposition in this case) outside of the loop in the caller site.

7 Acknowledgement

The HPF/JA extensions presented here is the fruit of cooperative work by all the JAHPF members. The author would like to express sincere appreciation for their devotion to the work.