VPP Fortran and the Design of HPF/JA Extensions

Hidetoshi Iwashita, Naoki Sueyau, and Sachio Kamiya
Strategic Planning Division, Software Group
Fujitsu Ltd.
140 Miyamoto, Numazu-shi
Shizuoka 410-0396, Japan

Matthijs van Waveren
Fujitsu European Centre for Information Technology Ltd.
Hayes Park Central, Hayes End Road
Hayes, Middlesex UB4 8FE, UK

Abstract

VPP Fortran is a data parallel language, that has been designed for the VPP series of supercomputers. In addition to pure data parallelism, it contains certain low-level features that were designed to extract high performance from user programs. A comparison of VPP Fortran and HPF 2.0 shows that these low-level features are not available in HPF 2.0. The features include asynchronous inter-processor communication, explicit shadow, and the LOCAL directive. They were shown in VPP Fortran to be very useful in handling real-world applications, and they have been included in the HPF/JA extensions. They are described in the paper. The HPF/JA language specification version 1.0 is an extension of HPF 2.0 to achieve practical performance for real-world applications and is a result of collaboration in the Japan Association for HPF (JAHPF). Some practical programming and tuning procedures with the HPF/JA language specification are described, using the NAS Parallel Benchmark BT as an example.

Keywords: parallel computing, parallel languages, benchmark, asynchronous communication, data locality.

1 Introduction

In general, MPI can be used to achieve high performance in a distributed memory environment. However, for the last seven years, the data parallel language VPP Fortran has been used on the Fujitsu VPP series supercomputers as regularly as MPI. We think the
reason is that the VPP Fortran language specification possesses both ease of programming and the capability to deliver high performance.

It is our impression that the HPF 2.0 language specification [1, 2] has the power to encourage compiler optimization but that it does not yet provide sufficient features to handle real-world applications. The experience with VPP Fortran can be used in making HPF more useful for these types of applications.

The HPF/JA language specification version 1.0 [3] is an extension of HPF 2.0 to achieve practical performance for real-world applications and is a result of collaboration in the Japan Association for HPF (JAHF)
\(^1\). VPP Fortran has some important features which are not available in HPF, and we often used experience of VPP Fortran in the course of the discussions on the design of HPF/JA language extensions.

This paper discusses how the experience of VPP Fortran was used in the design and implementation of the HPF/JA language specification. We focus on the features of asynchronous communication, the LOCAL directive, and the explicit shadow. We also describe some practical programming and tuning procedures with the HPF/JA language specification, using the NAS Parallel Benchmark BT as an example.

This paper consists of the following sections: Section 2 introduces features of VPP Fortran, which are compared to HPF in Section 3; Section 4 describes three HPF/JA language features, which were designed from experience with VPP Fortran; Section 5 describes a programming example using the HPF/JA extensions; Section 6 discusses some open issues in HPF; and Section 7 is the conclusion.

2 Features of VPP Fortran

VPP Fortran [4, 5, 6, 7] was designed originally for the VPP series supercomputer in 1991.

On the one hand, VPP Fortran is much easier for programming than MPI due to the global name space, which is supported with many features. Data can be decomposed and located on multiple processors and the runtime system guarantees data consistency and the necessary synchronizations. Loop and task parallelism can also be specified in a consistent manner.

On the other hand, the low-level specifications of VPP Fortran were designed to supplement the pure data parallelism and to extract high performance. These specifications are explicit access of local data to each processor, broadcast and multicast communications, and barrier, critical section, and post/wait synchronizations. These low-level features are not always necessary but they are useful for performance tuning of critical portions in the user program.

\(^1\)http://www.tokyo.rist.or.jp/jahp/index-e.html
2.1 Data location

The programming model describing the target hardware is shown in Figure 1. Local memory is specific to each processor, and therefore always fast. Global memory, which is physically distributed across the processors, is shared by all processors. Every variable has either a global or local location attribute corresponding to the memory on which it is located.

![Diagram of VPP Fortran Programming Model]

Figure 1: VPP Fortran Programming Model

Global data can be distributed onto the processors and explicitly accessed as fast as local data under the following conditions:

- It is connected to a local variable name with the EQUIVALENCE statement of the Fortran standard and referred to with the local name, or
- It is referred inside the RESIDENT construct of VPP Fortran.

2.2 Execution process

Figure 2 illustrates an example execution of a VPP Fortran program.

The program portion specified in the PARALLEL REGION construct is executed in parallel. The PARALLEL REGION directive causes a fork across the processors and the END PARALLEL directive causes a join. This behavior is similar to the behavior of the parallel region construct of the OpenMP language specification [8]. In VPP Fortran, which is used on distributed memory systems, both control and data are distributed among the processors. In OpenMP, which is used on shared memory systems, only control is distributed among the processors. VPP Fortran has two SPREAD constructs – SPREAD REGION for describing task decomposition (C and D in Figure 2), and SPREAD DO for loop decomposition ($F_1$ through $F_N$).

A body of a PARALLEL REGION construct or a SPREAD construct with respect to a certain processor group is called a region. Regions may be nested. In Figure 2, program segments $B$ through $G$ constitute a region. $C$, $D$, and each of $F_1$ through $F_N$ are individual regions.
2.3 Index partition

The VPP Fortran language specification attaches importance to loop decomposition as well as to data decomposition. To describe the relationship of data and loop decomposition, VPP Fortran introduces the index abstraction which closely ties array index and loop iteration. An index partition, which defines the mapping between indices and processors, has the following four attributes:

INDEX specifies the range for the subscript of the array or the iteration.

PART specifies the partitioning type, BAND (same as BLOCK of HPF) or CYCLIC, optionally with a block width, or an irregular partitioning (same as GEN_BLOCK of HPF).

PROC specifies the processor group segments to be the target of the mapping. The segments of a \(n\)-dimensional processor group are the \((n - 1)\)-dimensional processor subgroups orthogonal to the dimension of the processor group.

OVERLAP specifies both widths of the overlap area (same as SHADOW of HPF).

Figure 3 illustrates an example describing the mappings of both an array \(A\) and nested loops \(I\) and \(J\) to a processor group \(P\) by the index partitions \(IP1\) and \(IP2\). \(P\) is specified as a \(2 \times 4\) matrix. Both the rows of the array \(A\) and the iterations of loop \(I\) are mapped to the rows of \(P\) according to the index partition \(IP1\). Similarly, the columns of \(A\) and the iterations \(J\) are mapped to the columns of \(P\) according to \(IP2\).
2.4 Explicit communication

Communication patterns often used in fluid applications are all-to-all aggregated communication for transpose data transfer and data swapping between neighbour processors for reading of the boundary data on the neighbour processors, as shown in Figure 4. VPP Fortran provides the SPREAD MOVE construct with the index partition for the former communication pattern and the OVERLAPFIX directive for the latter. Both of them can be performed asynchronously with other computations.

Figure 4: OVERLAPFIX communication

3 Comparison of HPF and VPP Fortran

In this section, we compare HPF and VPP Fortran in order to consider how to make HPF more practical.
3.1 General comparison

The approved extensions of HPF contain low-level directives which are powerful but ought to be understood precisely. This is similar to the low-level specifications in VPP Fortran. As a big difference, HPF has many communication library functions for a variety of communication and computational patterns while VPP Fortran has explicit communication directives instead.

3.2 Data mapping

Both of the languages are data parallel languages, but VPP Fortran has in addition the character of SPMD since it treats variables local to each processor. These local variables are not exactly similar to the NEW and RESIDENT variables of HPF, because a NEW variable is not local to each processor but local to each iteration of the loop and its physical location depends on the implementation. A RESIDENT variable is specified only to be located inside the active processor set, which is generally not a single processor. In addition, NEW or RESIDENT are not static attributes of a variable but belong to a block of execution statements.

Both languages have almost the same functionality, but HPF has the following features which VPP Fortran does not have:

- More varied expressions – e.g. alignment with stride, in direct distribution, replicated alignment, multiple alignment, pre-/de-/transcriptive and inherit mapping.
- Many alternatives to describe the same mapping – e.g., (1, j, k) and (::, ::) notations for alignment and any rank of a template to describe the same mapping.
- Omissible descriptions on any layer of mapping – e.g., distribution without explicit processors, alignment without explicit distribution.
- Implementation dependent definitions especially for processors – e.g., the processor set defined as scalar, and mutual alignment of processor sets.

3.3 Explicit communication

HPF requires the programmer to describe data decomposition and the compiler to generate computational decomposition. The result is that the efficiency of the generated code depends strongly on the ability of the compiler to generate computational decomposition. VPP Fortran does not expect high-level optimization of the compiler. It requires the programmer to describe decomposition of both data and computation.

The significant difference is in the handling of the shadow edge, or the overlap area in the terminology of VPP Fortran. While HPF provides only the sizes of the shadow edges, VPP Fortran allows users to declare and access the overlap area freely as local data in the usual manner of the Fortran language or with the OVERLAPFIX directive.
4 Contribution to HPF/JA Specifications

The HPF/JA language specification version 1.0 [3] is an extension of HPF2.0 to achieve practical performance for real-world applications and is a result of collaboration in the Japan Association for HPF (JAHPF). Since VPP Fortran has some important features which HPF does not have, we often used experience of VPP Fortran in the course of the discussions on the design of HPF/JA language extensions.

This section describes three features of HPF/JA language specification designed from experience of VPP Fortran.

4.1 Asynchronous communication

The asynchronous transfer function performs data transfer between processors in parallel with execution of another executable statement. It was rather smoothly ported from VPP Fortran to HPF. We designed the ASYNCHRONOUS directive and construct and ASYNC prefix to specify concurrent execution of statements and/or directives. To adapt it for the HPF language specification, we designed the asynchronous identifier as a new local entity and introduced the NOBUFFER clause in order to allow further performance tuning.

4.2 LOCAL directive

VPP Fortran has taught us that even if the data is located on the same processor as where the execution takes place, it is not accessed efficiently when the compiler does not know its locality at compile time. The reason is that finding out the locality at runtime involves a large number of instructions and runtime library calls which may suppress other compiler optimizations such as vectorization. Although data locality can in most cases be analyzed at compile time, we have added the LOCAL clause and directive in order to specify that data exists in active processors executing the execution statements. Using this information, the compiler knows that communication outside the active processor is not required for the data.

Some members of JAHPF hoped that local variables would be supported in HPF/JA as in VPP Fortran (see Figure 3). However, this might have broken the global name space, which is the basic concept of HPF. So we proposed the LOCAL directive as an extended executable directive for HPF/JA version 1.0, designed in analogy with the RESIDENT directive of the HPF approved extension. Therefore, the user must guarantee the data consistency at the exit point of the LOCAL block, just as at the exit point of the local procedures.

4.3 Explicit shadow

With the experience of VPP Fortran, we know that explicit access of the shadow area will contribute to the performance. But an adaptation to HPF is not so simple because it may
conflict with compiler optimization on the shadow area. The compiler might break the user data in the shadow area anytime or suppress optimization in order to avoid this.

To solve this issue, we designed the explicit shadow access in HPF/JA version 1.0 as follows:

- Only a shadow object defined explicitly can be referred explicitly. The user cannot expect a shadow object to be defined by the compiler automatically.

- A shadow object can hold only the same value as the corresponding data on the next processor (say reflection source). The state of a shadow object becomes undefined if the value becomes different from the value of its reflection source. This rule implies that the compiler can use a shadow area as a buffer of its reflection source whenever it needs to.

- The condition of becoming undefined is precisely defined. For example, a shadow object becomes undefined when its parent object is remapped. This rule implies that the compiler does not have to keep the consistency between a shadow area and its corresponding reflection source in the case of remapping.

The user can refer the data in the shadow area explicitly with the LOCAL directive. The user can also define the shadow area either with the REFLECT directive or the EXT_HOME clause of the extended ON directive. The REFLECT directive causes regular communication to be the same as for the OVERLAPFIX directive of VPP Fortran as shown in Figure 4. The EXT_HOME clause, named from the extended HOME clause, makes the active processors the owners of the specified variable taking also account of its shadow area. For example, if an array element A(I) is defined in the block of ON EXT_HOME(A(I)), the shadow objects of the array A will be filled on all processors.

5 Programming Experiment

This section describes a practical example on programming and performance tuning using the HPF 2.0 specifications and the HPF/JA extensions. The sample program we chose is the block tridiagonal (BT) benchmark [9] of NAS Parallel Benchmark 2.3-serial. It is a CFD application program with about 3700 lines describing the ADI method. We used the Fujitsu HPF compiler version UXP/V HPF V20L20 L01041 on the VPP5000 vector-parallel supercomputer [10]. The benchmark code was gradually modified.

1. Trial execution with Fortran compiler

   We first compiled BT Class S and A using the sequential Fortran compiler without any special code modification.

2. Vector tuning
do k=1,grid_points(3)-2
  do j=grid_points(2)-2,0,-1
    do i=1,BLOCK_SIZE
      do n=1,BLOCK_SIZE
        rhs(m,i,j,k) = rhs(m,i,j,k)
      enddo
    enddo
  enddo
enddo

(a) Step 1

Figure 5: An example of vector tuning

We modified the source code as follows in order to encourage vectorization in the Fortran compiler:

- The specified shape (m,i,j,k) of variables u and rhs was transposed to (i,j,k,m), and the shape (m,n,c,i,j,k) of lhs was transposed to (i,j,k,m,n,c). The vector length is extended by this conversion.
- A manual loop interchange was performed. The loops amenable to vectorization were moved to the inner level, while the loops amenable to parallelization were moved to the outer level.

As the result of this step, the loop in subroutine y_backsubstitute was changed as shown in Figure 5, for example. Even though this vector tuning is not perfect, it is sufficient to show the effect of HPF programming.

3. Basic parallelization

The main variables have three dimensions I, J and K and many nested DO-loops can be parallelized along dimension K. So we distributed the variables along dimension K and specified INDEPENDENT directives on the DO K loops.

We could not use the DISTRIBUTE directive directly on the variables because the upper bounds in their K dimensions are different expressions, viz. KMAX-1, KMAX/2*2, and (KMAX+1)/2*2. We used the ALIGN and TEMPLATE directives instead. We aligned the main variables along the K dimension to the same template, which was a vector of shape (0:(KMAX+1)/2*2) and was distributed BLOCK onto the processors (see Figure 6).

According to the static analysis information generated by the HPF compiler, we found that the compiled code still includes many inter-processor communications.
!hpf$ processors pp(HPE)
!hpf$ template tt(0:(KMA1+1)/2*2)
!hpf$ distribute tt(block) onto pp

double precision
> us (0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2),
> vs (0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2),
> ws (0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2),
> qs (0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2),
> rho_i (0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2),
> square (0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2),
> forcing (0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2, 5),
> u   (0:(IMAX+1)/2*2, 0:(JMAX+1)/2*2, 0:(KMAX+1)/2*2, 5),
> rhs (0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2, 5),
> lhs (0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2, 5, 5, 3)
common /fields/ u, us, vs, ws, qs, rho_i, square,
> rhs, forcing, lhs

!hpf$ align (*, kk) with tt(kk) :: us, vs, ws, qs, rho_i, square
!hpf$ align (*, kk, *) with tt(kk) :: forcing
!hpf$ align (*, kk, *) with tt(kk) :: u, rhs
!hpf$ align (*, kk, *, *) with tt(kk) :: lhs

double precision fjac(5, 5, 0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX-1),
> njac(5, 5, 0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX-1),
> tmp1, tmp2, tmp3
common /work_lhs/ fjac, njac, tmp1, tmp2, tmp3

!hpf$ align (*, *, *, kk) with tt(kk) :: fjac, njac

Figure 6: Employed template and mapping
double precision
> forcing_t ( 0:JMAX/2+2, 0:JMAX/2+2, 0:KMAX/2+2, 5),
> u_t (0:(JMAX+1)/2+2, 0:(JMAX+1)/2+2, 0:(KMAX+1)/2+2, 5),
> rhs_t (0:JMAX/2+2, 0:JMAX/2+2, 0:KMAX/2+2, 5),
> lhs_t (0:JMAX/2+2, 0:JMAX/2+2, 0:KMAX/2+2, 5, 5, 3)
common /fields_t/ u_t, rhs_t, forcing_t, lhs_t

!hpf$ align (*,jj,**) with tt_t(jj) :: forcing_t
!hpf$ align (*,jj,***) with tt_t(jj) :: u_t, rhs_t
!hpf$ align (*,**,***) with tt_t(jj) :: lhs_t

double precision jfac_t(5, 5, 0:JMAX/2+2, 0:JMAX/2+2, 0:KMAX-1),
> nfac_t(5, 5, 0:JMAX/2+2, 0:JMAX/2+2, 0:KMAX-1)
common /work_rhs_t/ jfac_t, nfac_t

!hpf$ align (*,*,jj,***) with tt_t(jj) :: jfac_t, nfac_t

Figure 7: Specification of alternative variables mapped with J-axis

The subsequent steps 4 and 5 are necessary in order to get a reasonable performance.

4. Remapping strategy

In contrast with the parts x_solve and y_solve, z_solve cannot be parallelized along the K dimension. Therefore, we chose to parallelize along the J dimension instead and found that variables rhs, lhs and u needed to be remapped before and after z_solve.

The REDISTRIBUTE or REALIGN directives would normally be used to specify this remapping, however we adopted the array assignment statement instead to describe data motion between two variables with different mappings. For this purpose, we specified alternative variables with a J-axis mapping as shown in Figure 7. These alternative variables correspond to the original variables with a K-axis mapping.

The values of the variables rhs, lhs and u are copied to the alternative variables rhs_t, lhs_t and u_t respectively, then z_solve is executed using the alternative variables instead, and finally the result values are copied back to the original variables. Variable forcing_t is used as the alternative of forcing in a portion of subroutine exact_rts. jfac_t and nfac_t are only used as work storage in subroutine lhsz.

This ping-pong method has the following benefits in comparison with the use of remapping:

- Dynamic variables, which may be remapped, often suppress compiler optimizations especially when the remapping is performed across procedures boundaries.
header.h

!hpf$ shadow (0,0,1) :: us, vs, ws, qs, rho_i, square
!hpf$ shadow (0,0,2,0) :: u

rhs.f

!hpf$ reflect u,us,vs,ws,qs,rho_i,square

!hpf$ independent, new(j,i,wi,j,wp1,wm1)
   do k = 1, grid_points(3)-2
!hpf$   on home(tt(k)), local(u,us,vs,ws,qs,rho_i,square)
   do j = 1, grid_points(2)-2
      do i = 1, grid_points(1)-2
         wij = ws(i,j,k)
         wp1 = ws(i,j,k+1)
         wm1 = ws(i,j,k-1)
         rhs(i,j,k,1) = rhs(i,j,k,1) + dz1t1 *
            (u(i,j,k+1,1) - 2.0d0*u(i,j,k,1) +
             u(i,j,k-1,1)) -
            tz2 * (u(i,j,k+1,4) - u(i,j,k-1,4))
   end
   end
   end

Figure 8: Code fragment around SHADOW and REFLECT

- Using two explicit variables in the ping-pong method provides an opportunity
to apply asynchronous communication, as described in step 6.

5. Utilizing the shadow region

In order to get a higher performance in subroutine compute_rhs, the variables u,
us, vs, ws, qs, rho_i and square need to be specified with a shadow region.
According to the static analysis information, we found that shadow elements of
these variables are referred to approx. 58N^3 times in total for each invocation of
subroutine compute_rhs, where N is the problem size 64 for Class A. The REFLECT
and LOCAL directives of HPF/IA guarantee the fetch and guarantee an efficient access
to the shadow regions.

Code fragments using these features are shown in Figure 8.

6. Asynchronous communication for REFLECT directive and array assignment statement

We found two opportunities to utilize asynchronous communication in the program.
First, in subroutine compute_rhs, variable us is defined at the first nested DO-loop
and its shadow region is referred to at the 15th nested DO-loop 254 lines later
(shown in Figure 9). So the REFLECT translation can be hidden behind the 254
lines of execution. Similarly, the REFLECT translation for all variables specified with
\texttt{rhs.f}
\begin{verbatim}
!hpf$ async id1, id2, id3, id4, id5, id6, id7
!hpf$ async(id1) reflect u
    ... (two nested DO loops defining us, etc.) ...
!hpf$ async(id2) reflect us
    ... (DO loops and other asynchronous reflect directives) ...
!hpf$ asyncwait(id1)
!hpf$ asyncwait(id2)
    ...
!hpf$ asyncwait(id7)
!hpf$ independent, new(j,i,wi,k,wp1,wm1)
    do k = 1, grid_points(3)-2
!hpf$    on home(tt(k)), local(u, us, vs, ws, qs, rho_i, square)
    ...
\end{verbatim}

\textbf{Figure 9: Use of asynchronous REFLECT communication}

A SHADOW region in Step 5 can be performed asynchronously. This asynchronous communication has been implemented with the ASYNCHRONOUS directive or the ASYNC prefix used in Figure 9.

Second, the variable \texttt{1hs} is defined in subroutine \texttt{y_solve_cell} but it is not updated in the following subroutine \texttt{y_backsubstitute}. The alternative variable \texttt{1hs_r} is referred to in the following subroutines in \texttt{z_solve}. In this case, the assignment statement describing data transfer of \texttt{1hs} to \texttt{1hs_r} can be performed in parallel with the execution of subroutine \texttt{y_backsubstitute}.

The performance after each step was measured with data sizes Class S and Class A as shown in Figure 10. We reached a performance of 6828 Mflop/s on 8 processors and 9086 Mflop/s on 12 processors.

We see from the figure that the performance of Step 4 (transpose remap) is not good enough. The reason is that the current compiler cannot treat a shadow region efficiently without manual directives. We can see that the REFLECT and \texttt{LOCAL} directives of HPF/JA contribute strongly to the performance. We think that the asynchronous communication feature will extract more performance if the program is substantially modified to increase the computation that can hide the asynchronous communication.
6 Further Discussion

6.1 Improvement of NEW clause

Specifying the local variables for each iteration with the NEW clause of the INDEPENDENT directive does not seem convenient for real-world applications. We have encountered an INDEPENDENT DO loop with over 247 NEW variables, which required 22 continuation lines of the INDEPENDENT directive. In addition, copying the DO indices inside the INDEPENDENT DO loop into the NEW clause is rather error-prone.

In fact, some implementations have implemented alternative language specifications which simplify the description of such syntax. For example, under some assumptions, the LOCAL clause without the variable name can be used instead of the NEW clause. While these alternative language descriptions can be convenient, it is not a desirable situation that an implementation-dependent solution becomes popular in some compilers without being part of the standard language specification. If this is the case, we should improve the language specification.

Figure 10: Result of NAS Parallel BT
6.2 Performance portability

With respect to the tuning of performance, we would like to recognize precisely where the data is distributed and located. However, variables distributed onto a scalar processor, variables specified in NEW and REDUCTION clauses, and sequential variables can be located on any processor depending upon the implementation.

The situation that these issues are implementation-dependent is not desirable, with respect to the portability of performance. The reason is that we must regard the location of variables in program segments around the LOCAL construct, around the TASK_REGION construct with the RESIDENT clause, and around the actual arguments of HPF LOCAL and FORTRAN LOCAL procedures. Compilers should have a common default interpretation of data locality.

7 Conclusion

We have shown that the HPF/JA extensions are suitable to extract high performance from actual applications. We have discussed the extensions of asynchronous inter-processor communication, explicit shadow, and the LOCAL directive. These extensions originate from VPP Fortran. An example of their usage has been shown. The HPF/JA extensions have been carefully designed not to restrict compiler optimizations that would be performed if they were absent. They are not orders to the compiler but suggestions to the compiler, in order to keep the semantics of the other directives in HPF.

Acknowledgment

We thank all members of JAHPF. The HPF/JA language specification was mainly designed by compiler developers of Hitachi, NEC and Fujitsu during worthwhile discussions in JAHPF. We thank the Commissariat à l’Énergie Atomique for allowing us to use their VPP5000 for the programming experiment, and Fujitsu Systems Europe for assisting us during the experiment.

References


