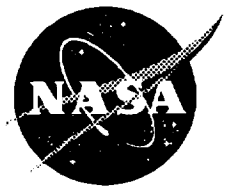
NASA Contractor Report 194913

ICASE Report No. 94-34



ICASE

EXTENDING HPF FOR ADVANCED DATA PARALLEL APPLICATIONS

Barbara Chapman
Piyush Mehrotra
Hans Zima

Contract NAS1-19480
May 1994

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681-0001

Operated by Universities Space Research Association

Extending HPF For Advanced Data Parallel Applications *

Barbara Chapman[a] *Piyush Mehrotra*[b] *Hans Zima*[a]

[a]Institute for Software Technology and Parallel Systems,
University of Vienna, Brünner Strasse 72, A-1210 Vienna, Austria
E-Mail: {barbara, zima}@par.univie.ac.at

[b]ICASE, MS 132C, NASA Langley Research Center, Hampton VA. 23681 USA
E-Mail: pm@icase.edu

Abstract

The stated goal of High Performance Fortran (HPF) was to "address the problems of writing data parallel programs where the distribution of data affects performance". After examining the current version of the language we are led to the conclusion that HPF has not fully achieved this goal. While the basic distribution functions offered by the language – regular block, cyclic, and block cyclic distributions – can support regular numerical algorithms, advanced applications such as particle-in-cell codes or unstructured mesh solvers cannot be expressed adequately. We believe that this is a major weakness of HPF, significantly reducing its chances of becoming accepted in the numerical community. The paper discusses the data distribution and alignment issues in detail, points out some flaws in the basic language, and outlines possible future paths of development. Furthermore, we briefly deal with the issue of task parallelism and its integration with the data parallel paradigm of HPF.

*The work described in this paper was partially supported by the Austrian Research Foundation (FWF Grant P8989-PHY) and by the Austrian Ministry for Science and Research (BMWF Grant GZ 308.9281-IV/3/93). This research was also supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

1 Introduction

The High Performance Fortran Forum (HPFF), which convened during 1992, set itself the task of defining language extensions for Fortran to facilitate data parallel programming on a wide range of parallel architectures without sacrificing performance [8]. Much of the work focussed on extending Fortran 90 with directives for specifying alignment and distribution of a program's data. These enable the programmer to influence the locality of computation by controlling the manner in which the data is mapped to processors. Other major extensions include data parallel constructs, such as the **FORALL** statement and construct, and the **INDEPENDENT** directive, along with a number of library routines.

A stated goal of the HPF forum was to "address the problems of writing data parallel programs where the distribution of data affects performance" [8]. Even though the defined extensions provide the first steps towards such a portable programming interface, it is our contention that they fall short of the overall goal.

In this paper we show that there is some important functionality which is missing from the current HPF language definition. We indicate how this might be added to the language, and state why it is needed. In several places, we give examples to illustrate the use of these additional features. The set of all features that we propose for inclusion into the current version of HPF will be informally subsumed under the name **HPF$^+$**. We do not claim, however, that HPF$^+$ is in any sense complete. In particular, we refrain from discussing any of the issues surrounding I/O and the handling of large data sets. Our prime consideration is functionality and semantics; we do not attempt to provide a full definition of the proposed features, and sometimes use an ad-hoc syntax. In a few places, we use syntax from Vienna Fortran [2, 17], which already provides solutions for some of the problems discussed in this paper.

We assume throughout that the reader is familiar with the basic mechanisms of HPF for mapping data. The full details are to be found in the HPF Language Specification [8].

The core of this paper is Section 2, in which we identify a range of application problems that cannot be adequately dealt with in the context of the distribution and alignment features of current HPF. We propose a number of new language features that address these issues. Furthermore, we also point out some difficulties with the present definition of the basic language. In Section 3, we outline some of the requirements posed by multidisciplinary problems, which need a capability to express task parallelism and integrate it with data parallelism. We propose a scheme that relies on the explicit creation of asynchronous tasks and an object-oriented mechanism for providing these tasks with access to shared data. Concluding remarks are to be found in Section 4.

2 Generalization of Data and Work Distributions

Much of HPF consists of constructs which may be used to specify the mapping of data in the program to an abstract set of processors. This is achieved via a two-level mapping: first, data arrays are aligned with other objects, and then groups of objects are distributed onto an array of abstract processors. These processors are then mapped to the physical processors of the target machine in an implementation-dependent manner.

The **ALIGN** directive is used to align elements of data arrays to other data arrays or *templates*. Templates are abstract index spaces which can be used as a target for alignment and may then be distributed in the same way as arrays. The **DISTRIBUTE** directive is provided to control distribution of the dimensions of arrays or templates onto an abstract set of processors. The distribution of a dimension may be described by selecting one of a set of predefined primitives which permit block, cyclic or block-cyclic mapping of the elements. The rules for alignment are more flexible: a linear function may be used to specify the relationship between the mappings of two different data arrays. Mechanisms are also provided to enable dynamic modification of both alignment and distribution; these are the **REALIGN** and **REDISTRIBUTE** directives, respectively.

These language constructs suffice for the expression of a range of numerical applications operating on regular data structures. However, more complex applications pose serious difficulties. For example, modern codes in such important application areas as aircraft modeling and combustion engine simulation often employ multiblock grids. Even if these grids are to be distributed by block to processors, the constructs of the current HPF language specification do not permit an efficient data mapping for these programs; as we will show in the next subsection, this requires distributions to sections of the processor array in general. HPF is even less equipped to handle advanced algorithms such as particle-in-cell codes, adaptive multigrid solvers, or sweeps over unstructured meshes. Many of these problems need more complex data distributions if they are to be executed efficiently on a parallel machine. Some of them may require the user to control the execution of major do loops by specifying which processor should perform a specific iteration. These are not provided by HPF.

Some programs will require that data be distributed onto processor arrays of different ranks: the current language definition does not permit the user to prescribe or assume any relationship between such processor arrays.

We discuss some of these topics in more detail in the remainder of this section, and propose a number of language extensions which provide the required functionality, including

- distribution to processor subsets (Section 2.1 & 2.2),

- processor views (Section 2.3),
- general block distributions (Section 2.4),
- indirect distributions (Section 2.5.1),
- user-defined distribution functions (Section 2.5.2), and
- on-clauses for the control of the work distribution in an **INDEPENDENT** loop (Section 2.5.3).

Some of these problems and their solutions were discussed during the HPFF meetings and can be found in the Journal of Development, Section 11 of the language draft [8].

2.1 Distribution to Processor Subsets – Multiblock Grid Codes

The HPF **DISTRIBUTE** directive specifies the distribution of data onto a processor array which has been declared by the user. It does not permit distribution onto a part of the processor array. Here, we give an example of a problem which may need this capability for efficient execution, and describe a simple extension to current HPF which would permit it.

Scientific and engineering codes from diverse application areas may use multiple grids to model the underlying problem domain. For example, in computational fluid dynamics a complex aircraft structure may be modeled using multiple structured grids [15], so that the spacing of the grids and their shapes can be individually chosen to match the underlying physical structure. A typical application run may use anywhere from 10 to 100 grids of widely varying sizes and shapes.

Each sweep over the domain involves computation on the individual grids before data is exchanged between them. Thus, these types of applications exhibit at least two levels of parallelism. At the outer level, there is coarse grain parallelism, since the computation can be performed on each grid simultaneously. The internal computation on each grid, on the other hand, exhibits the typical loosely synchronous data parallelism of structured grid codes. An efficient execution of such a code would require that the work is spread evenly across the target machine; this means that the total number of grid points on each processor should be roughly the same, independent of the number of grids and their shapes and sizes.

One possible way of distributing data is be to distribute the array of grids to the processors so that each grid is mapped to exactly one processor. Along with exploiting the outer level of parallelism, this approach has two other drawbacks. First, the number of grids is not large in many applications, and may be significantly smaller than the number of processors of a massively parallel machine thus restricting the amount of parallelism that can be effectively

utilized. Secondly, the grids may vary greatly in size, resulting in an uneven workload on those processors which are involved in the computation.

Another possible strategy is to distribute each of the grids independently onto all of the processors of the machine, enabling the parallelism within a grid to be exploited. This will lead to a more even workload; however, the grids may not all be large enough for this to be a reasonable solution.

Both of the above distribution strategies are likely to be inefficient, particularly on machines with a large number of processors. A flexible alternative is to permit grids to be separately distributed to a suitably sized subset of the available processors. This approach allows both levels of parallelism to be exploited while providing the opportunity to balance the workload.

The current HPF definition does not, however, permit data arrays to be distributed directly to subsets of processors. That is, the target of a distribution directive must be a *processors-name* [8, page 26]. Hence we can only achieve the desired distribution either by aligning the data arrays representing individual grids to parts of a large template, which is then distributed to the processors, or by aligning each grid to a template of its own, which has been declared with exactly the size needed to ensure that, after distribution, the data for the grid is on the desired number of processors. In the latter case, the alignments must be carefully chosen so that the grids are mapped to different subsets of processors; otherwise, there would be a serious imbalance of the workload. In practice, both of these solutions are difficult to achieve, and will generally require a priori precise knowledge of the size of both the grid and the processor array. These template constructions, and alignment and distribution directives, are unnecessarily complicated and must be reimplemented for each modification of the problem. A simpler solution is to adopt the direct approach of Vienna Fortran, which permits a *processor-reference* [17, page 22] to be the target of a distribution. Thus a subsection of processors may be specified in a **DISTRIBUTE** directive, permitting straightforward descriptions of the desired distributions of the individual grids as shown in the following code fragment:

```
!HPF$  PROCESSORS R(128)
 ...
REAL G(32,2048)
 ...
!HPF$  DISTRIBUTE (*,BLOCK) ONTO R(N:M):: G
```

2.2 Distribution of Subobjects – Multigrid Codes

A further shortcoming of HPF is that it does not permit subobjects of a data object to be distributed. We use a simple multigrid example to show the kind of restrictions that this imposes on the choice of distribution for a problem.

Consider the following Fortran 90 program segment for a multigrid problem where the number of grids and their sizes is not known until runtime.

```
TYPE subgrid
  INTEGER xsize, ysize
  REAL, DIMENSION(:,:), ALLOCATABLE :: grid
    ...
END subgrid
TYPE(subgrid), DIMENSION(:), ALLOCATABLE :: grid_structure

    ...
READ(*,*) no_of_grids
ALLOCATE(grid_structure(no_grids))
DO i = 1, no_of_grids
   READ(*,*) ix, iy
   grid_structure(i)%xsize = ix
   grid_structure(i)%ysize = iy
   ALLOCATE(grid_structure(i)%grid(ix,iy))
END DO
```

In this example, each grid is declared as an allocatable array within a derived type. The set of all grids is also an allocatable array, where each element is a grid. Thus the number of grids and their individual sizes need not be specified until runtime. The multiblock application described in the last subsection also requires similar data structure declarations. HPF allows us to distribute the array of grids to the processors. However, we may not distribute the individual grids across processors, since these are subobjects and their distribution is explicitly prohibited [8, page 26].

But a multigrid problem does not exhibit the kind of parallelism between individual grids that is available in multiblock applications: the refinement and interpolation steps of the multigrid approach usually require a sequential processing of grids and it is the parallelism within a grid which must be exploited. Since this is not possible, we cannot describe a

satisfactory distribution for the above data structures. A more flexible approach to the mappings for objects and subobjects would solve this problem.

2.3 Processor Views

The shape of an abstract set of processors is defined in HPF by a **PROCESSORS** directive. Given a processor array, R, of rank k onto which a data array (or a template) is to be distributed, exactly k dimensions of the array must be distributed to the corresponding dimensions of R. That is, for an array with rank $r > k$, $r - k$ dimensions must remain undistributed, which is indicated by the symbol "*", while an array with a rank less than k cannot be distributed to R at all [8, page 26].

In other words, there is no way to view the same set of abstract processors as a processor array having a different shape, in particular a different rank.

Consider the following code fragment:

```
!HPF$ PROCESSORS P2(10,10), P1(100)
      REAL A(M,N), B(L)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO P2
!HPF$ DISTRIBUTE B(BLOCK) ONTO P1
```

Here, A is a two dimensional array with the two dimensions distributed onto to the two dimensional processor array $P2$. Meanwhile, B is a one dimensional array distributed onto the one dimensional processor array $P1$. Under the HPF definition, since $P1$ and $P2$ are of different shapes, the programmer cannot make any assumption about their relative mapping onto the physical processor array. What the programmer may want to do in this example is to "equivalence" $P2(i,j)$ with $P1(i+(j-1)*10)$, according to standard Fortran conventions. HPF cannot express this.

Again, Vienna Fortran provides a simple mechanism to provide "equivalenced" views of the same processor set [17, page 20]. Thus, the above example could be specified in Vienna Fortran as follows:

```
PROCESSORS P2(10,10) RESHAPE P1(100)
REAL A(M,N) DIST (BLOCK, BLOCK) TO P2
REAL B(L) DIST (BLOCK) TO P1
```

Using the **RESHAPE** clause provides a secondary (and in this case a one-dimensional) view of the same processor set. This allows different dimensional distributions to be specified for the same processor set.

A more general solution would allow the specification of *alignment* between processor arrays, using a linear function*:

!HPF$ **PROCESSORS** P2(10,10), P1(100)
!HPF$ **PROCESSOR_ALIGN** P2(I,J) **WITH** P1(F(I,J))

This means that $P2(I,J)$ and $P1(F(I,J))$, where F is a linear function in I and J, designate the same abstract processor. Equivalencing, as used in the above example, is a special case obtained by choosing $F(I,J) = I + (J-1) * 10$.

2.4 General Block Distributions – Particle-in-Cell Code

As mentioned in the introduction, dimensions of data arrays or templates are mapped in HPF by specifying one of a small number of predefined distributions, possibly with an argument. There are a number of problems for which these mappings are inadequate. In this section, we consider an application for which HPF's data distributions do not permit satisfactory load balancing. Here, a generalization of the block distribution gives the user a means to balance the workload on the target machine during execution.

Consider a simulation code designed to study the motion of particles in a given domain, such as plasmas for controlled nuclear fusion, or stars and galaxies. The computation at each time step can be divided into two phases. In the first phase, a global force field is computed using the current position of particles. In the second phase, given the new global force field, new positions of the particles are determined. The program can be structured by dividing the underlying domain into cells, with each cell owning a set of particles. The particles move from one cell to another as they change positions across the domain. Since the computation in each cell is dependent on the number of particles in the cell, the workload across the domain changes as the computation progresses.

If the set of cells is represented by an HPF array, then we need to distribute the cells across the processors such that the work per processor is approximately equal. This means that the distribution of the array across the processors must reflect the distribution of particles across cells, and thus is a function of values computed at execution time.

Two language features are required to deal with this situation: dynamic redistribution and irregular data distributions. While HPF satisfies the first requirement via the **REDISTRIBUTE** directive, it lacks functionality for meeting the second requirement.

The requirements of this problem can be met if we extend the set of distributions provided in HPF by **general block distributions**, as initially implemented in SUPERB [16] and Vienna Fortran [2, 17].

*This feature has been proposed by Robert Schreiber.

General block distributions are similar to the regular block distributions of HPF in that the index domain of an array dimension is partitioned into contiguous blocks which are mapped to the processors; however, the blocks are not required to be of the same size. Thus, general block distributions provide more generality than regular blocks while retaining the contiguity property, which plays an important role in achieving target code efficiency.

Consider a one-dimensional array A, declared as **REAL** $A[l:u]$, and assume that there are N processors $p_i, 1 \leq i \leq N$. If we distribute A using a general block distribution *GENERAL_BLOCK(B)*, where B is a one-dimensional integer array with N elements, and $B(i) = s_i$ (with $s_i > 0$ for all i) denotes the size of the i-th block, then processor p_1 owns the local segment $A[l : l + s_1 - 1]$, p_2 owns $A[l + s_1 : l + s_1 + s_2 - 1]$ and so on. B, together with the index domain of A, completely determines the distribution of A and provides all the information required to handle accesses to A, including the organization of the required communication.

A variant of this method determines general block distributions by specifying in B the sequence of upper bounds for the local segments – i.e., $l_1 + s_1 - 1, l_2 + s_1 + s_2 - 1, \ldots$ – rather than the segment lengths.

The above scheme can be readily generalized to multi-dimensional arrays, each dimension of which is distributed by regular or general block.

The example in Figure 1 illustrates a general block distribution applied to the rows of a two-dimensional matrix A, where the bulk of work is in the center of the region, reflected by smaller blocks for the associated processors.

Although the representation of general block distribution requires on the order of the number of processors to describe the entire distribution, optimization often permits a local description of the distribution to be limited to just a few processors, with which there will be communication. Also, the space overhead due to this representation is not large in general, since most problems do not require a large number of distinct general block distributions.

Arrays distributed in this way can be efficiently managed at compile time as well as runtime, allowing the use of the *overlap* concept [7, 16] to optimize communication related to regular accesses. Finally, codes can be easily parameterized with such distributions: for example, a procedure with a *transcriptive* formal argument[†] that is supplied with differently distributed actual arguments can be efficiently compiled if the representation of the argument's distribution is passed along as a set of additional implicit arguments created by the compiler [7, 16, 19].

We now show how a general block distribution may be used to implement a PIC code

[†]If such an argument is passed by reference, the distribution is left intact, and thus no movement of data will be necessary [8, page 48].

```
!HPF$ PROCESSORS R(8)
      INTEGER :: B(8) =(/400,400,200,100,100,100,500,800/)
      REAL A(2600,100)
!HPF$ DISTRIBUTE(GENERAL_BLOCK(B),*) :: A
```

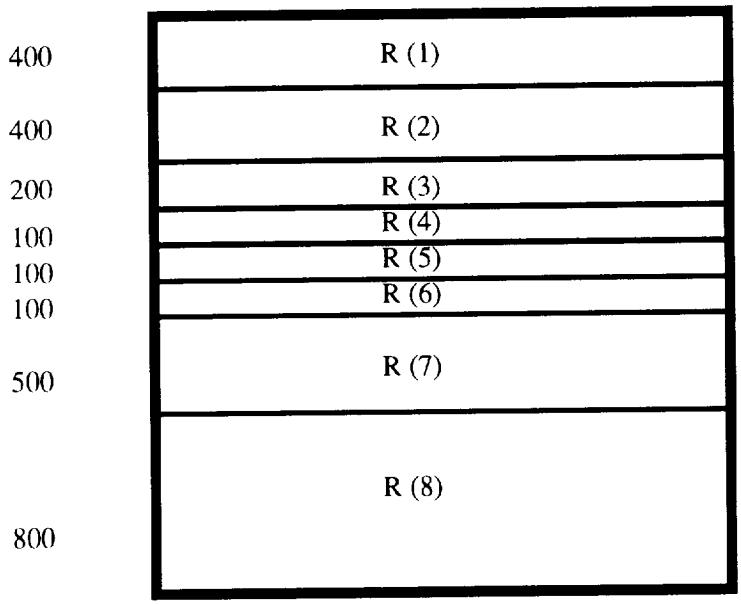


Figure 1: General Block Distribution in HPF+

as described above. The program in Figure 2 illustrates a simplified version of a PIC code expressed in HPF, extended by general block distributions.

Details irrelevant to our discussion are omitted. The cells are represented by the array *FIELD*. There are a maximum of *NCELL* cells and each cell is constrained to have a maximum of *NPART* particles. *FIELD* is declared to be **DYNAMIC** with the first dimension initially distributed into regular blocks. The procedure *initpos* determines the initial position of the particles and places them in the appropriate cells. Using the number of particles in each cell, the procedure *balance* computes the block sizes to be assigned to each processor and stores these in the array *BOUNDS*, such that *BOUNDS*(p) specifies the block size for processor p, where $1 \leq p \leq NUMBER\_OF\_PROCESSORS$. This array is then used to redistribute *FIELD*, using a general block distribution. The block sizes are selected so that each processor has roughly the same number of particles on its local part of the domain.

In each time step (represented by one iteration of the outer loop), the procedure *update_field* computes the new force field based on the current particle positions. Then, the procedure *update_part* is called to update the positions of the particles. Based on the new

```
      PARAMETER (NCELL = ..., NPART = ...)

      INTEGER BOUNDS(NUMBER_OF_PROCESSORS())
      REAL FIELD(NCELL,NPART,...)
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK,*,*) :: FIELD

C     Compute initial position of particles
      CALL initpos(FIELD, NCELL, NPART, ...)

C     Compute initial partition of cells
      CALL balance(BOUNDS, FIELD, NCELL, NPART, ...)
!HPF$ REDISTRIBUTE FIELD(GENERAL_BLOCK(BOUNDS))

      DO k = 1, MAX_TIME
C       Compute new field
        CALL update_field(FIELD, NCELL, NPART, ...)
C       Compute new particle positions and reassign them
        CALL update_part(FIELD, NCELL, NPART, ...)

C       Rebalance every 10th iteration if necessary
        IF ( MOD(k,10) .EQ. 0 .AND. rebalance() ) THEN
          CALL balance(BOUNDS, FIELD, NCELL, NPART, ...)
!HPF$     REDISTRIBUTE FIELD(GENERAL_BLOCK(BOUNDS))
        ENDIF

      ENDDO
```

Figure 2: High level PIC code in HPF+ with general block distributions

positions, the new owner cell for each particle is determined. If a particle has moved from one cell to another, it is explicitly reassigned. This obviously requires communication if the new cell is on a different processor. Since this communication is based on the locations of the current and the new cell, it is highly irregular in nature. Thus, the compiler will have to generate runtime code using the inspector/executor paradigm [9, 12] to support this particle motion.

If the number of particles on each processor remains roughly equal for the duration of the simulation, then load balance will be maintained. Some problems of this kind display sufficient uniformity such that a simple block distribution will suffice to provide a reasonable load balance. For other problems, the motion of particles during the simulation may lead to a severe load imbalance. The code, as shown here, checks whether rebalancing is required (by

calling function *rebalance*), on every 10th iteration. If so, a new *BOUNDS* array is computed and the cells redistributed to balance the workload.

2.5 Irregular Distributions

General block distributions provide enough flexibility to meet the demands of some irregular computations: if, for instance, the nodes of an unstructured mesh are partitioned prior to execution and then appropriately renumbered, then the resulting distribution can be described in this manner. However, this approach is not appropriate for each irregular problem. For example, a data distribution as shown in Figure 3 – which may be the outcome of a dynamic partitioner – cannot be represented in this way. A system based on this kind of distribution has been developed by Baden [1].

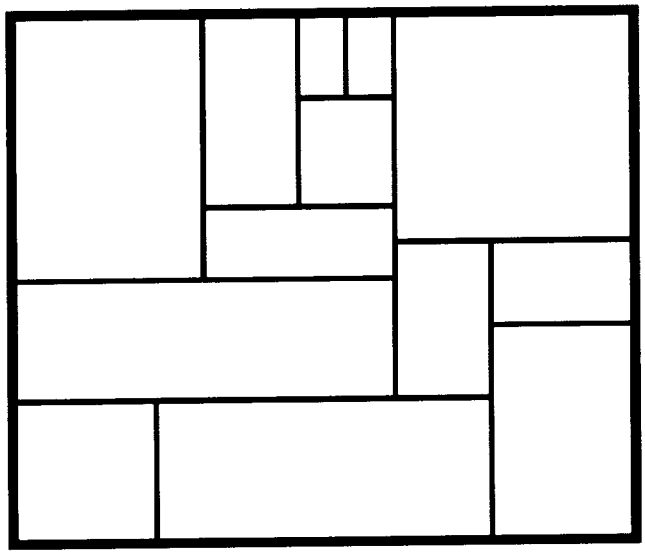


Figure 3: An example for an Irregular Block Distribution

Rather than proposing a special syntax for this kind of distribution, we will in the following subsections deal with a range of mechanisms, at different levels of abstraction, to handle arbitrarily complex data distributions.

We begin with *indirect distribution functions*, which allow the specification of a distribution via a mapping array (Section 2.5.1), and continue with *user-defined distribution functions* (Section 2.5.2). After discussing an extension of HPF's **INDEPENDENT** loop concept (Section 2.5.3), we give an example of a sweep over an unstructured mesh, based on all three extensions (Section 2.5.4). We conclude with the discussion of language features that directly support the binding of partitioners to **INDEPENDENT** loops (Section 2.5.5).

2.5.1 Indirect Distributions

Indirect distribution functions can express *any* distribution of an array dimension that does not involve replication. Consider the following program fragment in HPF+:

```
!HPF$ PROCESSORS R(M)
      REAL A(N)
      INTEGER MAP(N)
      ...
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK) :: A
!HPF$ DISTRIBUTE (BLOCK) :: MAP
      ...
!  Compute a new distribution for A and save it in the mapping array MAP:
!  The j-th element of A is mapped to the processor whose number is stored in MAP(j)

      CALL PARTITIONER(MAP, A,...)
      ...
!  Redistribute A  as specified by MAP:
!HPF$ REDISTRIBUTE A(INDIRECT(MAP))
      ...
```

Array A is dynamic and initially distributed by block. MAP is a statically distributed integer array that is of the same size as A and used as a **mapping array** for A; we specify a reference to an indirect distribution function in the form *INDIRECT(MAP)*. When the reference is evaluated, all elements of MAP must be defined and represent valid indices for the one-dimensional processor array R, i.e., they must be numbers in the range between 1 and M. A is then distributed such that for each $j, 1 \leq j \leq N$, $A(j)$ is mapped to $R(MAP(j))$. In this example, MAP is defined by a **partitioner**, which will compute a new distribution for A and assign values to the elements of MAP accordingly. (This distribution will often be used for a number of arrays in the program).

Indirect arrays were introduced in [10, 12]. They must be supported by a runtime system, which manages the internal representation of the mapping array and handles accesses to the indirectly distributed array. The mapping array is used to construct a *translation table*, recording the owner of each datum and its local index. Note that this representation has **O**(N) elements, on the same order as the size of the array. Most codes require only a very small number of indirect mappings (this is usually between 1 and 3 distinct mappings). The PARTI routines [12] represent a runtime library which directly supports indirect distribution functions, and has been integrated into a number of compilers.

2.5.2 User-Defined Distribution Functions

Indirect distribution functions incur a considerable overhead both at compile time and at runtime. A difficulty with this approach is that when a distribution is described by means of a mapping array, any regularity or structure that may have existed in the distribution is lost. Thus the compiler cannot optimize the code based on this complex but possibly regular distribution.

User-defined distribution functions (UDDFs) provide a facility for extending the set of intrinsic mappings defined in the language in a structured way. The specification of a distribution function introduces a class of *distribution types* by establishing mappings from (data) arrays to processor arrays. Such UDDFs were first defined in Kali [10] and Vienna Fortran [17]; the following discussion is based on the UDDFs as described in Vienna Fortran.

Syntactically, UDDFs are similar to Fortran functions; however, their activation results in the computation of a distribution rather than in the computation of a value. Apart from this, no side effects may occur as a result of executing these functions. UDDFs have two implicit formal arguments, representing the data array to be distributed and the processor array to which the distribution is targeted. Specification statements for these arguments can be given using the keywords **TARGET_ARRAY** and **PROCESSOR_ARRAY**, respectively. Other local data structures may be declared as well. UDDFs may contain Fortran executable statements along with at least one *distribution mapping statement* which maps the elements of the target array to the processors.

UDDFs constitute the most general mechanism for specifying distributions: any arbitrary mapping between array indices and processors can be expressed, including partial or total replication. We illustrate their use by two examples, representing indirect and skewed distributions.

Example: A UDDF Specifying an Indirect Distributions

The distribution function *INDIRECT*, as introduced in the previous section, can be easily expressed by a UDDF as shown below. For simplicity we assume that A and MAP have the same shape.

```
!HPF$ DFUNCTION INDIRECT(MAP)
!HPF$ TARGET_ARRAY A(*)
!HPF$ PROCESSOR_ARRAY R(:)
!HPF$ INTEGER MAP(*)

!HPF$ DO I=1,SIZE(A)
```

```
!HPF$    A(I) DISTRIBUTE TO R(MAP(I))
!HPF$  ENDDO
!HPF$  END DFUNCTION INDIRECT
```

Example: A UDDF for a Skewed Distribution

The UDDF *SKEW*, as defined below, specifies a two-dimensional skewed block distribution, as may sometimes be needed to satisfy the requirements for locality. We assume here that $M1$ divides $N1$, and $M2$ divides $N2$.

```
!HPF$  DFUNCTION SKEW
!HPF$  TARGET_ARRAY  A(0:,0:)
!HPF$  PROCESSOR_ARRAY R(0:,0:)
!HPF$  INTEGER N1,N2,M1,M2

!HPF$  N1=UBOUND(A,1)+1; N2=UBOUND(A,2)+1
!HPF$  M1=UBOUND(R,1)+1; M2=UBOUND(R,2)+1
!HPF$  DO I=0,N1
!HPF$    DO J=0,N2
!HPF$      A(I,J) DISTRIBUTE TO R(I/(N1/M1), MOD(J/(N2/M2)+I,M2+1))
!HPF$    ENDDO
!HPF$  ENDDO
!HPF$  END DFUNCTION SKEW
```

The distribution generated for X, as declared in the program fragment below, is shown in Figure 4.

```
!HPF$ PROCESSORS R2(8,8)
      ...
      REAL     X(0:7,0:7)
      ...
!HPF$ DISTRIBUTE (SKEW), ON TO R2(1::2,1::2) :: X
```

A facility similar to UDDFs can be provided for **user-defined alignment**; the (implicitly transferred) processor array of the UDDF must be replaced in this case by the source array to which the target array is to be aligned [17].

R2(1,1)	R2(1,3)	R2(1,5)	R2(1,7)
x(0,0) x(0,1)	x(0,2) x(0,3)	x(0,4) x(0,5)	x(0,6) x(0,7)
x(1,6) x(1,7)	x(1,0) x(1,1)	x(1,2) x(1,3)	x(1,4) x(1,5)
R2(3,1)	R2(3,3)	R2(3,5)	R2(3,7)
x(2,4) x(2,5)	x(2,6) x(2,7)	x(2,0) x(2,1)	x(2,2) x(2,3)
x(3,2) x(3,3)	x(3,4) x(3,5)	x(3,6) x(3,7)	x(3,0) x(3,1)
R2(5,1)	R2(5,5)	R2(5,5)	R2(5,7)
x(4,0) x(4,1)	x(4,2) x(4,3)	x(4,4) x(4,5)	x(4,6) x(4,7)
x(5,6) x(5,7)	x(5,0) x(5,1)	x(5,2) x(5,3)	x(5,4) x(5,5)
R2(7,1)	R2(7,3)	R2(7,5)	R2(7,7)
x(6,4) x(6,5)	x(6,6) x(6,7)	x(6,0) x(6,1)	x(6,2) x(6,3)
x(7,2) x(7,3)	x(7,4) x(7,5)	x(7,6) x(7,7)	x(7,0) x(7,1)

Figure 4: A Skewed Distribution

2.5.3 Extensions of the INDEPENDENT Loop Concept

Whenever a do loop contains an assignment to an array for which there is at least one indirect access within the loop, the compiler will not be able to determine whether the iterations of the loop may be executed in parallel. Since such loops are common in irregular problems, and may contain the bulk of the computation, the user must assert the independence of the do loop's iterations.

For this, HPF provides the **INDEPENDENT** directive, which asserts that a subsequent do loop does not contain any loop-carried dependences [18], allowing the loop iterations to be executed in parallel. The **INDEPENDENT** directive may optionally contain a **NEW** clause which introduces *private* variables that are conceptually local in each iteration, and therefore cannot cause loop-carried dependences.

There are two problems with this feature:

- There is no language support to specify the **work distribution** for the loop, i.e., the mapping of iterations to processors. This decision is left to the compiler/runtime system.
- **Reductions**, which perform global operations across a set of iterations, and assign the result to a scalar variable, violate the restriction on dependences and cannot be used in the loop (note that HPF and Fortran 90 provide intrinsics for some important reductions).

The first problem can be solved by extending the **INDEPENDENT** directive with an **ON** clause that specifies the mapping, either by naming a processor explicitly or referring to the *owner* of an element. The concept of *on clause* was first introduced in Kali [10] and later adopted in Fortran D [6] and Vienna Fortran [2]; a similar proposal was discussed in the HPF Forum but not included in the language (see [8], Journal of Development, Section 11). For example, in

```
!HPF$ PROCESSORS R(M)
      ...
!HPF$ INDEPENDENT, ON OWNER (EDGE(I,1)), ...
      DO I = 1, NEDGE
       ...
```

iteration *I* of the loop is executed on the processor that owns the array element *EDGE(I,1)*. If we assume that *R(F(I))* is this processor, the above example could also be written in the form

```
!HPF$ PROCESSORS R(M)
      ...
!HPF$ INDEPENDENT, ON R(F(I)), ...
      DO I = 1, NEDGE
       ...
```

The second problem can be solved by extending the language with a reduction directive – which is to be permitted within independent loops – and imposing suitable constraints on the statement which immediately follows it. It could be augmented by a directive specifying the order in which values are to be accumulated. Note that simple reductions could be detected by most compilers.

For example, in the code fragment below, the contributions *INCR(I)* of different iterations are accumulated in the variables *Y(EDGE(I,1))* (where *EDGE(I,1)* may yield the same value for different values of *I* – see Section 2.5.4 for an example with a geometric interpretation):

```
!HPF$ PROCESSORS R(M)
!HPF$ INDEPENDENT, ON OWNER (EDGE(I,1)), ...
      DO I = 1, NEDGE
       ...
!HPF$ REDUCTION
       Y(EDGE(I,1)) = Y(EDGE(I,1)) + INCR(I)
       ...
```

Vienna Fortran provides a language extension for reduction operations which is more general, but which is not a directive.

2.5.4 Sweep Over An Unstructured Mesh

We now illustrate some of the language features introduced above by reproducing a section of code from a two-dimensional unstructured mesh Euler solver.

The mesh for this code consists of triangles; values for the flow variables are stored at their vertices. The computation is implemented as a loop over the edges: the contribution of each edge is subtracted from the value at one node and added to the value at the other node.

Figure 5 illustrates one solution to this problem. The mesh is represented by the array $EDGE$, where $EDGE(I,1)$ and $EDGE(I,2)$ are the node numbers at the two ends of the Ith edge. The arrays X and Y represent the flow variables, which associate a value with each of the $NNODE$ nodes.

Consider the distribution of the data across the one-dimensional array of processors, $R(M)$. Each of the arra[illegible]as to be dynamically distributed, since the mesh is to be distributed at runtime, in or[illegible]dance the computational load across the processors.

The array X is declare[illegible]dynamically distributed with an initial block distribution. Later in the code, this array is distributed indirectly, using the mapping array MAP. The user-specified routine $PARTITIONER$, whose code has been omitted from the example, will generate a mesh partition and store it in MAP.

Y is also declared with the keyword **DYNAMIC** and is aligned to X. Whenever X is redistributed, Y is automatically redistributed with exactly the same distribution function.

Consider now how the array $EDGE$ is used in the algorithm. Since the elements of $EDGE$ are pointers to flow variables – in iteration I, $X(EDGE(I,1))$, $X(EDGE(I,2))$ and the corresponding components of X and Y are accessed – we relate the distribution of $EDGE$ to the distribution of X and Y in such a way that $EDGE(I,:)$ is mapped to the same processor as $X(EDGE(I,1))$.

This kind of relationship between data structures occurs in many codes, since a mesh is frequently described in terms of elements, whereas values are likely to be accumulated at the vertices. It can be simply expressed if we extend the **REDISTRIBUTE** directive as shown in the example.

The computation is specified using an extended **INDEPENDENT** loop. The work distribution is specified by the **ON** clause: the Ith iteration is to be performed on the processor that owns $EDGE(I,1)$.

```
      PARAMETER (NNODE = ...)
      PARAMETER (NEDGE = ...)
!HPF$ PROCESSORS R(M)
      ...
      REAL     X(NNODE), Y(NNODE)
      INTEGER  MAP(NNODE)
      REAL     EDGE(NEDGE,2)
,2)
      INTEGER N1, N2
      REAL  DELTAX
      ...
!HPF$ DYNAMIC :: X,Y,EDGE
!HPF$ DISTRIBUTE (BLOCK) :: X, MAP
!HPF$ ALIGN WITH X :: Y
!HPF$ DISTRIBUTE (BLOCK,*) :: EDGE
      ...
      CALL PARTITIONER(MAP,EDGE)
      ...
!HPF$ REDISTRIBUTE X(INDIRECT(MAP))
!HPF$ REDISTRIBUTE EDGE(I,:) ONTO R(MAP(EDGE(I,1)))
      ...
!HPF$ INDEPENDENT, ON OWNER (EDGE(I,1)), NEW (N1, N2, DELTAX)
      DO I = 1, NEDGE
       ...
       N1 = EDGE(I,1)
       N2 = EDGE(I,2)

       ...
       DELTAX = F(X(N1), X(N2))

!HPF$ REDUCTION
       Y(N1) = Y(N1) - DELTAX
!HPF$ REDUCTION
       Y(N2) = Y(N2) + DELTAX
      END DO
      ...
     END
```

Figure 5: Code for Unstructured Mesh in HPF+

The variables $N1$, $N2$ and $DELTAX$ are private, so conceptually each iteration is allocated a private copy of each of them. Hence assignments to these variables do not cause flow dependencies between iterations of the loop.

For each edge, the X values at the two incident nodes are read and used to compute the contribution $DELTAX$ for the edge. This contribution is then accumulated into the values of Y for the two nodes. But since multiple iterations will accumulate Y values at each node, different iterations may write to the same array elements. As a consequence, we have indicated that these are reductions.

The dominating characteristic of this code, from the point of view of compilation, is that the values of X and Y are accessed via the edges, hence a level of indirection is involved. In such situations, either the mesh partition must be available to and exploitable by the compiler, or runtime techniques such as those developed in the framework of the *inspector-executor paradigm* [9, 12] are needed to generate and exploit the communication pattern.

2.5.5 Sweep Over Unstructured Mesh: Revisited

The code for the unstructured Euler solver discussed in the previous section represents a low-level approach to parallelization, in which the programmer assumes full control of data and work distributions, using the **ON** clause, and indirect distribution functions; a user-defined partitioning routine explicitly constructs a mapping array which can then be referred to.

This process may be further automated. Recent developments in runtime support tools and compiler technology, such as the CHAOS system developed at the University of Maryland [11] and integrated in the Vienna Fortran Compilation System, show how a higher level language interface may be provided in which control over the data and work distributions in an **INDEPENDENT** loop can be delegated to a combination of compiler and runtime system. We illustrate this approach by the example in Figure 6 which uses an ad hoc notation.

The *use-clause* of the **INDEPENDENT** loop enables the programmer to select a partitioner from those provided in the environment (in the example, this is SPECTRAL_PART) and the arrays (in the example, X) to which it is to be applied. SPECTRAL_PART is called with implicit arguments specifying the iteration space of the **INDEPENDENT** loop and the data flow pattern associated with the use of X in the loop. The call has two effects: First, a new distribution is computed for X, and X along with its associated secondary array, Y, is redistributed accordingly. Secondly, a new work distribution is determined for the **INDEPENDENT** loop, based on the combined objectives of minimizing load imbalances and maximizing locality.

```
      PARAMETER (NNODE = ...)
      PARAMETER (NEDGE = ...)
!HPF$ PROCESSORS R(M)
      ...
      REAL      X(NNODE), Y(NNODE)
      REAL      EDGE(NEDGE,2)
      INTEGER N1, N2
      REAL    DELTAX
      ...
!HPF$ DYNAMIC :: X,Y,EDGE
!HPF$ DISTRIBUTE (BLOCK) :: X
!HPF$ ALIGN WITH X :: Y
!HPF$ DISTRIBUTE (BLOCK,*) :: EDGE
      ...
!HPF$ INDEPENDENT, NEW (N1, N2, DELTAX), USE(SPECTRAL_PART(X))
!HPF$ DO I = 1, NEDGE
      ...
       N1 = EDGE(I,1)
       N2 = EDGE(I,2)

       DELTAX = F(X(N1), X(N2))

!HPF$ REDUCTION
        Y(N1) = Y(N1) - DELTAX
!HPF$ REDUCTION
        Y(N2) = Y(N2) + DELTAX
      END DO
      ...
      END
```

Figure 6: Code for Unstructured Mesh: Version 2

The actions described above represent an extension of the *inspector* in the inspector-executor paradigm [9, 12] and are performed *before* the actual execution of the loop. Note that the execution of the **INDEPENDENT** loop implicitly redistributes X and Y and that the arrays retain their new distributions after the loop has completed execution.

Other constructs may be useful in conjunction with the specification of a partitioner: for example, the user may wish to specify the array whose usage within the loop should form the basis of the loop's work distribution. It may sometimes be desirable to restore the distribution of one or more of the newly partitioned arrays after the loop has executed. If the partitioner is to be invoked at intervals throughout the program's execution, a condition for its invocation may be needed; this may depend on the value of the loop variable or some other program variable. Finally, it may be be necessary to combine this approach with low-level control, in which case some means of accessing the map array implicitly constructed for irregularly partitioned arrays such as X should be provided.

Note that rather than attaching such attributes to a number of **INDEPENDENT** loops individually, language features can be defined that associate a partitioner with the whole program or a set of loops. We do not discuss their syntax here.

2.6 Libraries

Control of Dynamic Data Distributions One of the features that HPF has in common with some of its predecessors (in particular, Kali [10], Fortran D [6], and Vienna Fortran [2]) are *dynamic data distributions*. However, compared with Vienna Fortran, HPF has only rudimentary facilities for controlling this powerful feature – with significant consequences for the compiler as well as for the target code efficiency. It is possible to provide constructs for obtaining more precise information on distributions at compile time. Vienna Fortran provides the **RANGE** attribute and the **DCASE** statement, a generalization of the Fortran 90 case statement that allows the association of blocks of code with a combination of distribution types of a selected set of arrays. We show by means of an example below how these constructs can provide the compiler and/or runtime system with valuable information regarding the distributions of dynamic or inherited arrays.

If a formal array argument inherits its distribution from the actual argument associated with it at runtime (via a *transcriptive dist-format-clause*), then the compiler may not have any information on the distributions which it will assume. This has consequences for the efficiency of the code it generates.

Simple language extensions may alleviate this problem. If the user knows that only a few distributions will occur for a specific set of formal arguments, an additional directive would enable this information to be provided to the compiler. Vienna Fortran has this feature in

the form of the **RANGE** clause. The example below follows the Vienna Fortran syntax, in which inheritance of a distribution is specified by an asterisk. Only the specified distributions may actually occur for the array A at runtime.

```
SUBROUTINE RANGE_EG ( A, . . . )
REAL  A(N,M) DIST(*) RANGE( (BLOCK,BLOCK), (BLOCK,CYCLIC(100)))
```

Further, the efficiency of the computation within the subroutine may depend very heavily on the actual distributions of the arguments, thus yielding good performance in some cases and very poor performance in others.

There is a specific difficulty in resolving two legitimate demands of a general purpose subroutine: it should handle a variety of different arguments, which may be differently distributed, and it should handle them efficiently. Redistribution at procedure boundaries may be costly, and hence should be avoided if possible. Vienna Fortran provides a construct which may be used in this situation: the **DCASE** construct, which is modeled along the lines of the CASE construct in Fortran 90. It enables the selection of a block of statements according to the actual distribution of one or more arrays. In particular, this also gives the compiler knowledge of the distribution which will reach the encapsulated segment of code.

```
SUBROUTINE MMUL (A,B,C,N,M,L)
REAL A(N,M), B(M,L), C(N,L) DIST(*)
INTEGER LEN, LSUB

SELECT DCASE (C,A):
CASE((BLOCK,:),(BLOCK,:)) :
      IF (M*L .LE. MAXSIZE) THEN
          CALL MATMUL(A,B,C,N,M,L)
      ELSE  LEN = L / $NP
            DO J = 1, $NP
              CALL MATMUL1 (A,B,C,N,M,L,LEN,J)
            END DO
      ENDIF
CASE((BLOCK,BLOCK),(BLOCK,*)) :
...
CASE DEFAULT: ....
...
END SELECT
```

In the above code, the matrix operation is handled in a specific way depending on how the actual argument arrays are distributed[‡]. In this way, we can insert appropriate code or call further subroutines as required. The compiler has precise information on the distribution functions of the selected arrays for the block of statements within the cases. Only one of the case alternatives is executed; if none of the other specifications match, then the default (if present) is selected. The cases are examined in the order in which they occur textually. The first distribution expression is compared with the actual distribution of C, and the second with that of A. If C is distributed by block in the first dimension and not at all in the second, and A likewise, then the first case is selected and its code executed. Otherwise, the distribution of C is then compared with the next case: if it is distributed by block in both dimensions, then if A is distributed by block in the first dimension, this case is selected. An "*" matches any distribution whatsoever. A test may be associated with a side effect. If an array distribution is compared with (CYCLIC(K)), for example, then if the actual distribution is indeed cyclic, the variable K is set to the value of its width.

2.7 Data Distribution and Alignment – Other Issues

There are a number of other issues with the specification of data distribution and alignment in current HPF which we have not discussed in this paper. In the following, we point out some flaws and gaps in the basic language definition. This relates specifically to processor declarations, templates, and the procedure interface.

As already mentioned earlier in Section 2.3, different **processor arrangements** introduced in a **PROCESSORS** directive are not related to each other, except for the case where identical shapes are used [8, page 40]. As a consequence, the semantics of examples such as the one on top of page 49 of the HPF language draft:

!HPF$ **DISTRIBUTE** POLYHYMNIA * **ONTO** ELVIS

may not be well defined, if the size of *ELVIS* is different from the size of the processors arrangement to which the corresponding actual argument is distributed.

The use of the **TEMPLATE** directive for declaring **templates** to which data may be aligned introduces several additional problems. The size of templates is determined by a specification expression, and hence templates cannot be used for describing the alignment of allocatable arrays. Furthermore, since templates are not first class objects in the language, they cannot be passed across procedure boundaries, and thus do not provide sufficient generality to describe the distributions and alignments of procedure arguments. HPF tries to

[‡]In Vienna Fortran *$NP* is an intrinsic function which returns the number of processors executing the program

alleviate this problem by introducing the **INHERIT** directive, which is supposed to align a dummy argument "to a copy of the template of the corresponding actual argument in the same way that the actual argument is aligned" [8, page 44]. This makes things even worse, since there are a number of important cases – including actual arguments that are array expressions – where the language does not specify the meaning of this directive; furthermore, there is no conceivable Fortran 90-oriented syntax that allows the specification of a descriptive *dist-format-clause* in a situation where actual argument templates of different dimensions are mapped at different call sites to a given dummy argument.

A more detailed discussion of the problems associated with the **TEMPLATE** directive, along with a proposal for an alternative mapping scheme without templates is given in [3].

Procedure boundaries pose other problems as well, such as the restriction of not allowing array reshaping for distributed arguments. This causes severe problems in porting sequential Fortran codes which often use reshaping. Also, data arrays when passed as actual arguments are always remapped back to their original distribution on returning from a procedure call. Thus, any redistribution of data in a procedure cannot affect the distribution of the actual argument. This provides a clean interface between procedure calls; however, there are situations (such as unstructured grids), where it is more convenient to write a separate routine which computes the new distribution of an array argument and redistributes it.

3 Integration of Task With Data Parallelism

With the rapidly growing computing power of parallel architectures, the complexity of simulations developed by scientists and engineers is increasing fast. Many advanced applications are of a multidisciplinary and heterogeneous nature and thus do not fit into the data parallel paradigm represented by HPF, Vienna Fortran and similar languages.

Multidisciplinary programs are formed by pasting together modules from a variety of related scientific disciplines. For example, the design of a modern aircraft involves a variety of interacting disciplines such as aerodynamics, structural analysis and design, propulsion, and control. These disciplines, each of which is initially represented by a separate program, must be interconnected to form a single multidisciplinary model subsuming the original models and their interactions. The parallelism both within and between the discipline models needs to be exposed and effectively exploited.

In this section, we outline the salient features of the language **OPUS**, an extension of Fortran 90, which addresses this issue [4]. OPUS provides a software layer on top of data parallel languages, designed to address the "programming in the large" issues as well as the

parallel performance issues arising in complex multidisciplinary applications. A program executes as a system of *tasks* which interact by sharing access to a set of *Shared Data Abstractions (SDAs)*. SDAs generalize Fortran 90 modules by including features from object-oriented data bases and monitors in shared-memory languages. They can be used to create persistent shared "objects" for communication and synchronization between coarse-grained parallel tasks, at a much higher level than simple communication channels transferring bytes between tasks.

A task is *spawned* by activating a subroutine with a list of arguments all of which must be of intent IN. Tasks are asynchronously executing autonomous activities to which *resources* of the system may be allocated. For example, the physical machine on which a task is to be executed, along with additional requirements pertaining to this machine, may be specified at the time a task is created.

Tasks may embody nested parallelism, for example by executing a data parallel HPF program, or by coordinating a set of threads performing different functions on a shared data set.

An SDA consists of a set of data structures along with the methods (procedures) which manipulate this data. A set of tasks may share data by creating an SDA instance of appropriate type, and making it accessible to all tasks in the set. Tasks may then asynchronously call the methods of the SDA, with each call providing exclusive access. Condition clauses associated with methods and synchronization facilities embodied in the methods allow the formulation of a range of coordination strategies for tasks.

The state of an SDA can be saved on external storage for later reuse. This facility can be seen as providing an I/O capability for SDAs, where in contrast to conventional byte-oriented I/O the structure of the object is preserved.

Other Fortran-based approaches to the problem of combining task with data parallelism include the programming languages *Fortran M* [5], which provides a message-passing facility in the context of a discipline enforcing determinism, and *Fx* [14], which allows the creation of parallel tasks that can communicate at the time of task creation and task termination by sharing arguments. DPC [13], based on C, uses a mechanism modeled on C file structures to support message passing between data parallel tasks.

4 Conclusion

In this paper, we have evaluated the capabilities of High Performance Fortran for expressing data parallel programs in an efficient manner. Based on the analysis of a number of advanced applications - including multiblock codes, particle-in-cell codes, and sweeps over

unstructured meshes – it was shown that HPF in the current form does not provide sufficient functionality to implement these problems in an efficient manner. The paper proposes a range of language features, with a particular emphasis on the problem of distributing data and work to the processors of a machine. These features span a broad spectrum, ranging from well-understood and efficient constructs such as data distribution to processor subsets and general block distributions to powerful facilities for binding dynamic partitioners to **INDEPENDENT** loops, which automatically distribute data and work. Furthermore, features for task parallelism and its integration with data parallelism were also discussed.

If HPF's functionality is to be successfully extended in the directions needed to support these advanced applications, it will be necessary to revise the basic language and clarify the semantics of a number of crucial features, in particular processor arrays, templates, and the procedure interface.

References

[1] S. Baden. Programming Abstractions for Dynamically Partitioning and Coordinating Localized Scientific Calculations Running on Multiprocessors. SIAM J. Sci. and Stat. Computation,12(1), January 1991.

[2] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming* 1(1):31-50, Fall 1992.

[3] B. Chapman, P. Mehrotra, and H. Zima. High Performance Fortran Without Templates: A New Model for Data Distribution and Alignment. Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego (May 19-22, 1993), ACM SIGPLAN Notices Vol. 28, No. 7, pp.92-101, July 1993.

[4] B. Chapman, P. Mehrotra, J. Van Rosendale, and H. Zima. A Software Architecture for Multidisciplinary Applications: Integrating Task and Data Parallelism. Proc. CONPAR'94, Linz, Austria, September 1994. Also: Technical Report TR 94-1, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, March 1994 and Technical Report 94-18, ICASE, NASA Langley Research Center, Hampton VA 23681.

[5] I. T. Foster and K. M. Chandy. Fortran M: A Language for Modular Parallel Programming. Technical Report MCS-P327-0992 Revision 1. Mathematics and Computer Science Division, Argonne National Laboratory, June 1993.

[6] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Department of Computer Science Rice COMP TR90079, Rice University, March 1991.

[7] H. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, December 1989.

[8] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Technical Report, Rice University, Houston, TX, May 3, 1993. Also available as Scientific Programming 2(1-2):1-170, Spring and Summer 1993.

[9] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[10] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pp. 364–384. Pitman/MIT-Press, 1991.

[11] R. Ponnusamy, J. Saltz, A. Choudhary. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. Technical Report, UMIACS-TR-93-32, University of Maryland, April 1993.

[12] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[13] B. Seevers, M. J. Quinn, and P. J. Hatcher. A parallel programming environment supporting multiple data-parallel modules. In *Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, October 1992.

[14] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting Task and Data Parallelism on a Multicomputer. Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego (May 19-22, 1993), ACM SIGPLAN Notices Vol. 28, No. 7, July 1993.

[15] V. N. Vatsa, M. D. Sanetrik and E. B. Parlette. Development of a flexible and efficient multigrid-based multiblock solver; AIAA-93-0677. *Proceedings of the 31st Aerospace Sciences Meeting and Exhibit*, January, 1993.

[16] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[17] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. ICASE Internal Report 21, ICASE, Hampton, VA, 1992.

[18] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series, Addison-Wesley, 1990.

[19] H. Zima and B. Chapman. Compiling for Distributed Memory Systems. *Proceedings of the IEEE*, Special Section on Languages and Compilers for Parallel Machines, pp. 264-287, February 1993. Also: Technical Report ACPC/TR 92-16, Austrian Center for Parallel Computation, November 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY(Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1994	Contractor Report

4. TITLE AND SUBTITLE
EXTENDING HPF FOR ADVANCED
DATA PARALLEL APPLICATIONS

5. FUNDING NUMBERS
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
Barbara Chapman
Piyush Mehrotra
Hans Zima

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 94-34

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-194913
ICASE Report No. 94-34

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
Submitted to IEEE Parallel and Distributed Technology

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The stated goal of High Performance Fortran (HPF) was to "address the problems of writing data parallel programs where the distribution of data affects performance". After examining the current version of the language we are led to the conclusion that HPF has not fully achieved this goal. While the basic distribution functions offered by the language - regular block, cyclic, and block cyclic distributions - can support regular numerical algorithms, advanced applications such as particle-in-cell codes or unstructured mesh solvers cannot be expressed adequately. We believe that this is a major weakness of HPF, significantly reducing its chances of becoming accepted in the numerical community. The paper discusses the data distribution and alignment issues in detail, points out some flaws in the basic language, and outlines possible future paths of development. Furthermore, we briefly deal with the issue of task parallelism and its integration with the data parallel paradigm of HPF.

14. SUBJECT TERMS
Data parallel programming, High Performance Fortran, data distributions, dynamic redistributions.

15. NUMBER OF PAGES
30

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		