High Performance Fortran: History, Status and Future*

Piyush Mehrotra[a] *John Van Rosendale*[a] *Hans Zima*[b]

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

[b]Institute for Software Technology and Parallel Systems,
University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria
E-Mail: zima@par.univie.ac.at

Abstract

High Performance Fortran (HPF) is a data-parallel language that was designed to provide the user with a high-level interface for programming scientific applications, while delegating to the compiler the task of generating an explicitly parallel message-passing program. The main objective of this paper is to study the expressivity of the language and related performance issues. After giving an outline of developments that led to HPF and shortly explaining its major features, we discuss in detail a variety of approaches for solving multiblock problems and applications dealing with unstructured meshes. We argue that the efficient solution of these problems does not only need the full range of the HPF Approved Extensions, but also requires additional features such as the explicit control of communication schedules and support for value-based alignment. The final part of the paper points out some classes of problems that are difficult to deal with efficiently within the HPF paradigm.

1 Introduction

The emergence of distributed-memory architectures more than a decade ago brought into sharp focus two issues: the necessity of controlling locality and the complexity of parallel programming. One approach to the control of locality is to use an explicitly parallel approach, e.g., C or Fortran coupled with message passing. However, the resulting programs tend to be messy and very low-level. It became clear that a higher level approach was also possible. One could use a high-level "data parallel" language with a single thread of control coupled with user-specified annotations for distribution of data and computation across processors. In effect, the user can leave some of the low-level details to the compiler and runtime system.

High Performance Fortran (HPF) is at the forefront of such languages, and is important both in and of itself, and as a milestone on the way to truly expressive parallel languages which will satisfy users' needs more fully. In this paper we look at the history and development of HPF and its current status and utility. In particular, we describe a range of computations it both can and cannot easily express well, and also give some thoughts on what we see as potential future directions.

The paper is organized as follows. In Section ??, we discuss the research in language and compiler technology which led to HPF. The next three sections form the technical core of the paper. Section ?? outlines the HPF language, illustrated by a few simple examples. In Section ??, a detailed discussion of parallelization strategies for block structured codes provides the background for the description of a range of advanced language features. Section ?? describes how unstructured grids can be represented and processed,

*The work described in this paper was partially supported by the ESPRIT IV Long Term Research Project 21033 "HPF+" of the European Commission, by the Austrian Ministry for Science and Transportation under contract GZ 613.580/2-IV/9/95, and 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.

using features of HPF and the new language HPF+ which provides support beyond the current HPF standard. Section ?? examines some limitations of HPF while Section ?? presents conclusions.

2 Language and Compiler Efforts Leading to HPF

The key concept of HPF – sharing responsibility for exploiting parallelism between the user and the compiler/runtime system – is based on research done by several groups over the years. In this section, we first describe some of the most significant language and compiler efforts which facilitated the development of HPF and then give a brief history of HPF itself.

One of the first languages to allow users to control the layout of data was IVTRAN [?], a language developed for the SIMD machine ILLIAC IV. Users could indicate the array dimensions to be spread across the processors and those which were to be local in a processor. Combinations resulting in physically skewed data were also allowed.

In the context of MIMD machines, Kali (and its predecessor BLAZE) [?, ?] were the first languages to introduce distribution declarations. Kali allows the dimensions of an array to be orthogonally mapped onto an explicitly declared processor array using simple regular distributions such as *block, cyclic* and *block-cyclic*, and more complex distributions such as *irregular* in which the mapping of each array element is explicitly specified. Simple forms of user-defined distribution were also permitted. Kali also introduced the idea of dynamic distributions which allow the user to change the distribution of an array at runtime. Parallel computation is specified by means of *forall* loops within a global name space. The language also introduced the concept of an *on clause* which allows the users to control the distribution of loop iterations across the processors. The Kali compiler [?] was the first compiler to integrate both a static and a runtime communication strategy. When enough information was available at compile-time, the compiler generated the required communication statically. In other cases, the compiler used the *inspector-executor* strategy [?, ?] to generate the communication at runtime.

In the same time period, Thinking Machines, a supercomputer manufacturer, together with COMPASS, a compiler software company, incorporated static layout directives, including alignment of arrays, for a subset of Fortran-8x on the Connection Machine [?]. The CM-Fortran compiler also included an element array assignment statement (a precursor for the HPF *forall*) which was not incorporated in Fortran 90. The COMPASS Fortran compiler technology for generating code for distributed-memory machines was later utilized in the Fortran compilers for MasPar and DEC as well.

SUPERB [?] was an interactive restructuring tool, developed at the University of Bonn, which translated Fortran 77 programs into message-passing Fortran for the Intel iPSC, the GENESIS machine, and the SUPRENUM machine. The user specified the distribution of the program's data via an interactive language. Program flow and dependence information, using both intraprocedural and interprocedural analysis techniques, was computed and made available to the user, who could select individual transformation strategies or request other services via menus. SUPERB performed coarse-grain parallelization for a distributed-memory machine and was also able to vectorize the resulting code for the individual nodes of the machine.

The Fortran D project [?, ?] followed a slightly different approach to specifying distributions. The distribution of data is specified by first aligning data arrays to virtual arrays known as *decompositions*. The decompositions are then distributed across an implicit set of processors using relative weights for the different dimensions similar to the approach used in CM-Fortran. The language allows an extensive set of alignments along with simple regular and irregular distributions. All mapping statements are considered executable statements, thus blurring the distinction between static and dynamic distributions.

Vienna Fortran [?, ?] was the first language to provide a complete specification of mapping constructs in the context of Fortran. Based largely on the Kali model, Vienna Fortran allows arrays to be aligned to other arrays, which are then distributed across an explicit processor array. In addition to the simple regular and irregular distributions, Vienna Fortran defines a generalized block distribution which allows unequal sized contiguous segments of the data to be mapped to the processors. Users can define their own distribution and alignment functions which provide a greater degree of control over the mapping. The language maintains a clear distinction between distributions that remain static during the execution of a procedure and those which can change dynamically, allowing compilers to optimize the code for the two situations. It defines multiple methods of passing distributed data across procedure boundaries including inheriting the distribution of the actual arguments. Distribution inquiry functions facilitate the writing of library functions which are optimal for multiple incoming distributions. The Vienna Fortran Compilation System (VFCS) [?, ?] extends the Superb compiler to HPF and HPF+ [?], targeting a wide range of distributed-memory machines.

Several other projects have also contributed to the understanding necessary for the development of HPF and the compiling technology required for such a language [?, ?, ?, ?, ?, ?, ?, ?, ?] including commercial efforts such as [?, ?, ?].

The Genesis of High Performance Fortran

A "birds of a feather" session on a high-level approach to programming distributed-memory machines was sponsored by D. Loveman of DEC and K. Kennedy of Rice University at Supercomputing '91 in Albuquerque. At this session, DEC put its version of High Performance Fortran [?] in the public domain as a strawman to get agreement on a unified approach to programming such machines. This was followed by the first HPF meeting organized by Rice University in January 1992, which was attended by more than 120 researchers and practitioners in the field of parallel computing. The consensus at the meeting was to form a Forum of interested participants who would together develop High Performance Fortran.

The HPF Forum, a group of about 40 researchers, met at intervals of six weeks starting March 1992 to discuss the approach to be taken and the features to be included in the language. After extensive discussions, including input from the wider high performance community, the Forum released Version 1.0 of HPF in May 1993.

During 1993 and 1994, the Forum continued to meet regularly in order to a) produce corrections and clarifications, and b) to identify a sequence of kernels and applications which would motivate the further development of the language. In November 1994, HPF Version 1.1, mainly incorporating corrections to the language, was released. Also released was a document which itemized a set of issues to be considered by the Forum for the next version of the language. The document also described a set of applications motivating the desired capabilities. The HPF Forum continued its third round of meetings in 1995 and 1996. Based on the requirements document and further input from the community, the Forum released HPF 2.0 in January 1997. All these documents are available at the HPF Web site at `http://www.crpc.rice.edu/HPFF/home.html`.

There are several commercial compilers available for HPF, most of them implementing the features of HPF 1.1 only. The most widely used ones are from Applied Parallel Research, DEC, IBM and Portland Group Inc. (a complete listing of all the commercial compiler efforts can be found at the above URL). Most of these efforts are targeted towards distributed-memory architectures such as Intel's Paragon, IBM's SP2, Cray's T3D and T3E and the Digital Workstation Cluster. Some of the compilers from the independent software vendors, such as APR and PGI, also target shared-memory machines and clusters of SMPs. The performance of these compilers for regular single grid codes is comparable to that of hand-written message passing code, however, they do not perform as well on more complex codes [?]. The First Annual HPF User's Group meeting held on February 24-26, 1997 in Albuquerque, NM, USA included several talks which described the performance of these compilers on codes from a wide variety of disciplines.

3 The HPF 2.0 Language

High Performance Fortran is a set of Fortran extensions designed to allow specification of data parallel algorithms for a wide range of architectures. The underlying programming model provides a global name space and a single thread of control. The user annotates the program with distribution and alignment directives to specify the desired layout of data. Explicitly parallel constructs allow the expression of fairly controlled forms of parallelism, in particular data parallelism. Thus, the code is specified in a high level, portable manner with no explicit tasking or communication statements. The goal is to allow architecture specific compilers to generate efficient code for a variety of architectures including SIMD, MIMD shared and distributed-memory machines.

The HPF 2.0 language consists of three parts: a) the Base Language, b) the Approved Extensions, and c) Recognized Extrinsic Interfaces. The base language provides the basic set of HPF features which must be supported by an HPF compiler. The Approved Extensions consist of advanced constructs that meet specific needs but are not likely to be supported by the initial compilers. The Recognized Extrinsic Interfaces are a set of interfaces approved by the HPF Forum but which have been designed by others to provide a service to the HPF community.

HPF 1.1 was based on Fortran 90 rather than Fortran 77 since several Fortran 90 constructs such as the array statements provide a natural mechanism for expressing data parallelism. HPF 1.1 extended these features by providing the *forall* constructs and also several new intrinsic and library routines for data parallel operations. The current version, HPF 2.0, is based on the current Fortran standard, Fortran 95, which has absorbed some of the HPF 1.1 features. In particular, the *forall* extensions are now part of Fortran 95 and hence have been dropped from HPF.

Fortran supports sequence and storage association of data based on a single linearly addressable memory model. Since HPF is targeting architectures which do not necessarily support this memory model, it does not support such association for data that has been explicitly distributed. The consequences of this restriction are twofold. First, data objects that have been equivalenced with other objects using EQUIVALENCE or COMMON statements cannot be explicitly distributed. Second, distributed arrays cannot be redimensioned across procedure boundaries, i.e., both the formal and actual array arguments must have the same rank. This restriction on Fortran has major implications when converting legacy Fortran code to HPF since either such data objects cannot be distributed or the code has to be rewritten to avoid sequence and storage association. Note that the new Fortran standard provides more powerful features, such as modules, interfaces, and dynamic data allocation, obviating the need for sequence and storage association.

The next subsection describes the HPF 2.0 Base Language presenting a few examples to illustrate the use of the supported constructs. The section concludes with a short description of the Approved Extensions. A complete description of all the features of HPF 2.0 can be found in the HPF Language specification [**?**].

3.1 The Base Language

Some of the features of the HPF 2.0 Base Language are as follows.

- *Data mapping directives:* HPF provides an extensive set of directives to specify the distribution and alignment of arrays onto explicitly specified sets of abstract processors.

- *Data parallel directives:* HPF provides data parallel mechanisms which extend the array statements and the *forall* constructs of Fortran 95. The **INDEPENDENT** directive can be used to assert that iterations of a loop do not have any loop-carried dependences [**?**] and thus can be executed in parallel. A **REDUCTION** clause can be used with this directive to identify variables which are updated by different iterations using associative and commutative operators.

- *New intrinsic and library functions:* HPF provides a set of new intrinsic functions including system functions to inquire about the underlying hardware, mapping inquiry functions to inquire about the distribution of the data structures and a few computational intrinsic functions. A set of new library routines have also been defined so as to provide a standard interface for highly useful parallel operations such as reduction functions, combining scatter functions, prefix and suffix functions, and sorting functions.

- *Extrinsic procedures:* HPF is well suited for data parallel programming. However, in order to accommodate other programming paradigms, HPF supports *extrinsic* procedures. These define an explicit interface and allow routines expressed using a different language, e.g., C, or a different paradigm, such as explicit message passing, to be called from an HPF program.

Examples

In this section we provide two code fragments which use some of the features of the HPF Base Language.

Given below is the HPF version of an iteration of the Jacobi iterative procedure which at each step updates the current approximation at a grid point, represented by the array u, by computing a weighted average of the values at the neighboring grid points and the value of the right hand side function represented by the array f.

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
```

```
        REAL u(n, n), f(n, n)
!HPF$ ALIGN f WITH u
!HPF$ DISTRIBUTE u (*, BLOCK ) ONTO P
        FORALL (i = 2:n-1, j = 2:n-1)
          u(i, j) = 0.25 * (u(i-1, j) + u(i+1, j) + u(i, j-1) + u(i, j+1)) + f(i, j)
        END FORALL
```

In this code fragment, the data objects are mapped as follows. The array f is aligned with the array u using the identity alignment. [1] The columns of u (and thus those of f) are then distributed by block across the processor array P. P is declared as array of abstract processors whose size is determined by the system inquiry function *NUMBER_OF_PROCESSORS*, which returns the number of processors being used to execute the program at runtime. Thus the above code can be run on varying numbers of processors without recompilation. The computation is expressed using a **FORALL** construct, where all the right hand sides are evaluated using the old values of u before assignment to the left hand side.

To reiterate, the computation is specified using a global index space and does not contain any explicit data motion constructs. Assume now that the *forall* loop is strip-mined by the compiler using the *owner computes rule*, where the owner of a data object executes the statements which compute the value of the object. Since the underlying arrays are distributed by columns, the edge columns will have to be communicated to neighboring processors. It is the compiler's responsibility to analyze the code and translate it into an explicitly parallel code with the appropriate communication statements inserted to satisfy the data requirements.

A second example is a simplified version of the forward elimination phase of the Gaussian Elimination algorithm without pivoting:

```
        REAL v(n, n), p
!HPF$ DISTRIBUTE v (*, BLOCK )
        DO k = 1, n-1
          DO i = k+1, n
            p = -v(i, k)/v(k, k)
            v(i, k) = 0.0
!HPF$         INDEPENDENT
            DO j = k+1, n
              v(i, j) = v(i, j) + p*v(k, j)
            END DO
          END DO
        END DO
```

The first directive declares the columns of the array v to be distributed by block. The scalar p is not explicitly mapped and will be replicated by most compilers. The k and i loops are sequential and are executed by all the processors. Thus, the values of $v(i,k)$ and $v(k,k)$ are broadcast by the owning processor to all processors each of which can then compute p independently. The inner loop is declared independent and thus will be executed in parallel. Since the array v is distributed by block, the element $v(k,j)$ required for updating $v(i,j)$ is local to the processor. Hence no communication is required for the update.

The distribution of the columns by contiguous blocks implies that processors will become idle as the computation progresses. A *cyclic* distribution of the columns would eliminate this problem. This can be achieved by replacing the distribution directive with the following:

```
!HPF$ DISTRIBUTE v (*, CYCLIC )
```

This declares the columns to be distributed cyclically across the processors, and thus forces the inner j loop to be strip-mined in a cyclic rather than in a block fashion. This keeps all processors busy until the tail end of the computation. Note that since the computation code is written using a global index space and hence is independent of the data mapping, it does not need to be changed when the mapping is changed. The new version just needs to recompiled for parallel execution.

[1] The language provides more complex mechanisms for aligning arrays to other objects including translation, dimension collapsing, dimension exchange and replication.

The above distributions only exploit parallelism in one dimension, whereas the i loop can also run in parallel. This can be achieved by distributing both the dimensions of v as follows:

```
        REAL v(n, n), p
!HPF$ DISTRIBUTE v (CYCLIC, CYCLIC)
        DO k = 1, n-1
!HPF$    INDEPENDENT, NEW(p, j)
         DO i = k+1, n
           p = -v(i, k)/v(k, k)
           v(i, k) = 0.0
!HPF$       INDEPENDENT
           DO j = k+1, n
              v(i, j) = v(i, j) + p*v(k, j)
           END DO
         END DO
        END DO
```

Here, the processors are presumed to be arranged in a two-dimensional mesh and the array is distributed cyclically in each dimension. To exploit the parallelism in the second dimension, the i loop is also declared parallel via the **INDEPENDENT** directive. The **NEW** directive indicates that the scalar variable p acts as a private variable for each of the parallel iterations and allocating one copy of p for each processor would suffice. Again, the values of $v(i, k)$ and $v(k, k)$ are broadcast to all processors in order to compute p. However, now the elements of the pivot row have to be broadcast down each column of processors so as to update the elements. Thus, we have increased the parallelism of the code at the expense of some additional communication overhead.

3.2 HPF Approved Extensions for Advanced Applications

The Approved Extensions of HPF 2.0 include advanced features which allow more complex applications to be expressed using HPF. In particular, extensions allow greater control of the mapping of data objects. For example, users can map pointers and components of derived types, and can map objects to subsets of processors directly. New distribution formats allow irregular distributions. The **GEN_BLOCK** distribution generalizes the block distribution by allowing non-equal sized blocks and the **INDIRECT** distribution allows each element of a data object to be mapped individually using a mapping array.

Another important feature is the support of dynamic remapping of data. If an object has been declared **DYNAMIC** then it can be remapped at runtime using the the **REALIGN** or **REDISTRIBUTE** directives. Redistribution of an array implies that all arrays aligned with it have to be remapped.

In addition to mapping data, the **ON** directive allows users to map computation onto processors. The **RESIDENT** directive allows the specification of information about accesses to data objects within the scope of an associated **ON** block.

The **TASK_REGION** directive extends HPF beyond the realm of data parallelism by allowing some forms of control parallelism to be expressed within the language. This directive can be used to indicate regions of code that can be executed in parallel on different subsets of processors. Even though this is a very restricted form of task parallelism, since no communication or synchronization is allowed within these regions, simple forms of control parallelism, such as pipelining, can be specified.

We motivate some of these extensions using two classes of applications in the next two sections. Section **??** will, in addition, exploit new features for value-based alignment and explicit schedule control which were introduced in the language HPF+ [**?**].

4 Multiblock Codes

In this section we describe multiblock codes and show how HPF Approved Extensions can be used to express the data mappings necessary to exploit the parallelism in these codes.

Scientific and engineering codes from diverse application areas sometimes use a set of grids, instead of a single grid, to model the underlying problem domain. For example, in computational fluid dynamics a complex aircraft structure may be modeled using multiple block structured grids 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.

Computations utilizing multiple structured grids typically exhibit at least two levels of parallelism. At the outer level, there is coarse-grain parallelism arising out of independent computations on the individual grids. In addition, the internal computation on each grid exhibits the typical loosely synchronous data parallelism of structured grid codes. The distribution of the grids determines the level and degree of parallelism that is exploited. An efficient execution of such a code requires 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. While there exist runtime libraries to parallelize multiblock codes [?], in the rest of the section we discuss how we can use a high level language such as HPF to specify different distributions of the underlying grids.

4.1 Distributing the Grids

One possible way of mapping the data is to distribute the array of grids to the processors so that each grid is mapped to exactly one processor. Note that in such a distribution a processor can own multiple grids. Along with only exploiting the outer level of parallelism, this approach has several other drawbacks. In many applications, the number of grids is not large 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. Also, the grids may vary greatly in size, resulting in an uneven workload on those processors which are involved in the computation. Thus, processors with the large grids become a bottleneck while others are idle.

A second strategy is to distribute the elements of each grid independently onto all of the processors of the machine, enabling the parallelism within a grid to be exploited. Thus each processor owns a part of each grid in the domain. This will lead to a more even workload; however, some of the grids may not be large enough to effectively exploit all the processors in the system.

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 individually 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.

Before describing how to map the grids using the HPF directives, we touch upon some issues of declaring the required data structures in Fortran 95. The code fragment below shows the declaration of the grids and their allocations in Fortran. Note that these have been simplified for the purposes of the presentation here. For example, we presume that the grids are one-dimensional only.

```
      TYPE GRID
         INTEGER SIZE
         REAL, POINTER BL(:)
      END TYPE GRID

      TYPE (GRID), ALLOCATABLE :: DOMAIN(:)

      READ (*, *) GRID_COUNT
      ALLOCATE (DOMAIN(GRID_COUNT))

      DO I = 1, GRID_COUNT
         READ(*, *) DOMAIN(I)%SIZE
      END DO

   ! Allocate each grid
      DO I = 1, GRID_COUNT
        ALLOCATE( DOMAIN(I)%BL( DOMAIN(I)%SIZE ) )
      END DO
```

The code has been written so as to allow the number of grids and the size of each grid to be chosen at runtime, i.e., based on values input to the program. Since Fortran does not allow an array of pointers, we have to declare *DOMAIN*, the set of all grids, as an allocatable array of derived type *GRID*. The derived type *GRID* includes the *SIZE* of the grid along with a pointer, *BL*, to an array which will store the data values for the grid.

The *GRID_COUNT* is read from input and used to allocate the *DOMAIN* array. Then the sizes of each grid are read (the first *I* loop) and the individual grids are allocated (the second *I* loop).

We now describe how to distribute the grids using the three strategies described above. The distributions require one or more of the following features from the approved extensions: mapping of pointers, mapping of components of derived types, mapping to subsets of processors, indirect distributions, and the dynamic redistribution of data.

To use the first strategy for distributing the grids, the array *DOMAIN* can be declared to be block distributed by the following directive:

```
      TYPE (GRID), ALLOCATABLE :: DOMAIN(:)
!HPF$ DISTRIBUTE ( BLOCK ) :: DOMAIN
```

Such a directive will distribute the elements of the array *DOMAIN* – i.e., the grids – by block across the processors. However, since the sizes of the grids are not known until runtime, using such a distribution would , in general, not lead to a balanced workload. In order to have finer control over the load balance, the **INDIRECT** distribution can be used to map each grid onto a specific processor as shown below:

```
      INTEGER, ALLOCATABLE :: MAP(:)
      TYPE (GRID), ALLOCATABLE :: DOMAIN(:)
!HPF$ DYNAMIC, DISTRIBUTE ( BLOCK ) :: DOMAIN

      READ (*, *) GRID_COUNT
      ALLOCATE (DOMAIN(GRID_COUNT), MAP(GRID_COUNT))

      DO I = 1, GRID_COUNT
          READ(*, *) DOMAIN(I)%SIZE
      END DO

      CALL COMPUTE_MAPPING ( DOMAIN, MAP )
!HPF$ REDISTRIBUTE DOMAIN( INDIRECT (MAP) )

 ! Allocate each grid
      ...
```

Here, the *DOMAIN* array is declared **DYNAMIC** and with an initial block distribution. The mapping array, *MAP*, is allocated to be the same size as the *DOMAIN* array. However, since one expects the number of grids to be small it is not distributed and thus would be replicated on all the processors. After reading the sizes of the grids, the *COMPUTE_MAPPING* routine is called to determine the optimal mapping and to initialize *MAP* appropriately. The **REDISTRIBUTE** directive is then used to remap the *DOMAIN* array using the computed mapping array. Once the outer array is remapped, the individual grids can now be allocated.

The second distribution strategy requires that the individual grids be mapped across all processors. This can be achieved as follows:

```
!HPF$ PROCESSORS P( NUMBER_OF_PROCESSORS() )
      TYPE GRID
        INTEGER SIZE
        REAL, POINTER BL(:)
!HPF$   DISTRIBUTE BL( BLOCK ) ONTO P
      END TYPE GRID
```

Here, the pointer *BL*, in the derived type *GRID*, is declared to be distributed by **BLOCK**. Thus, when the individual grids are allocated each of them will be distributed across the whole set of processors *P*.

The third strategy requires more complex declarations as shown below:

```
!HPF$ PROCESSORS P( NUMBER_OF_PROCESSORS() )
      TYPE GRID
        INTEGER SIZE
        INTEGER LO, HI            ! bounds of the target subset of processors
        REAL, POINTER BL(:)
!HPF$   DYNAMIC BL
      END TYPE GRID
      TYPE (GRID), ALLOCATABLE :: DOMAIN(:)

      READ (*, *) GRID_COUNT
      ALLOCATE (DOMAIN(GRID_COUNT))

      DO I = 1, GRID_COUNT
         READ (*, *) DOMAIN(I)%SIZE
      END DO

 ! Compute processor subsets for each grid, setting the LO and HI values
      CALL COMPUTE_PROCS_SUBSET ( DOMAIN )

 ! Allocate each grid and distribute to the computed processors subset
      DO I = 1, GRID_COUNT
        ALLOCATE( DOMAIN(I)%BL( DOMAIN(I)%SIZE ) )
!HPF$   REDISTRIBUTE DOMAIN(I)%BL( BLOCK ) ONTO P(DOMAIN(I)%LO:DOMAIN(I)%HI)
      END DO
```

In this version, the outer array *DOMAIN* is not distributed while the pointer *BL* in the derived type *GRID* is declared dynamic. After reading in the size of each of the grids, the routine *COMPUTE_PROCS_SUBSET* is called to compute the subset of processors on which each grid should be mapped, storing the bounds in the components *LO* and *HI*. These bounds are then used to distribute the array *BL* for each grid, at the time of allocation, to the appropriate subset of processors.

4.2 Exploiting the Parallelism

As noted above, the distribution strategy determines which level of parallelism will be exploited in such codes. The structure of the sequential computation, at a very high-level, is as follows:

```
DO ITERS = 1, NITERS
   CALL UPDATE_BDRY (DOMAIN)
   DO I = 1, GRID_COUNT
     CALL SOLVE_GRID( DOMAIN(I), ... )
   END DO
END DO

SUBROUTINE SOLVE_GRID( GRID, ... )
   ...
   DO I = 1, GRID%SIZE
     GRID%BL(I) = ...
   END DO
   ...
```

In each iteration, first the boundaries of each grid are updated. Among other computation, this requires grids that abut each other to exchange data on their boundaries. Once the boundaries have been updated, the solver routine *SOLVE_GRID* is called on each grid to update the data values within the grid. This is generally a data parallel computation similar to that found in typical structured grid codes.

The three distribution strategies require code extensions to exploit the parallelism appropriate for the mapping. Using the first distribution strategy maps each grid onto exactly one processor, thus exploiting the parallelism across the grids rather than within each grid. This can be expressed by declaring the loop which calls the *SOLVE_GRID* routine as parallel using the **INDEPENDENT** directive as follows:

```
        CALL UPDATE_BDRY (DOMAIN)
!HPF$   INDEPENDENT
        DO I = 1, GRID_COUNT
!HPF$      ON ( HOME (DOMAIN(I)), RESIDENT
           CALL SOLVE_GRID( DOMAIN(I), ... )
        END DO
```

However, just declaring the loop to be independent is not enough in this case. This is because the **INDEPENDENT** directive asserts that there are no loop-carried dependences but does not prohibit the routine to read the same distributed global data through common blocks or modules. In such a situation the processors owning the global data have to be executing the call to the routine since they have to send the data to the processors executing the code within the routine[2]. The code within the solver routine can be set up such that it does not access any global data, however the compiler cannot determine this without aggressive (and expensive) inter-procedural analysis. This can be avoided by using the declarations given above. The **ON** directive indicates that the call to the routine *SOLVE_GRID* is to be executed only on the processor owning the Ith grid. Along with this, the **RESIDENT** directive asserts that the routine accesses data resident only on this processor and does not access any data resident on other processors.

Given these declarations, the loop iterations, (and in turn the call to the solver routine) can be executed in parallel, while the code within the routine itself is executed sequentially. Thus, there is no communication in the solver routine – all the communication is confined to the *UPDATE_BDRY* routine since two grids exchanging data may be mapped to different processors.

The second mapping strategy distributes each grid onto all the processors thus exploiting only the parallelism within each grid. This can be expressed by declaring the loop inside the routine *SOLVE_GRID* to be parallel:

```
        SUBROUTINE SOLVE_GRID( GRID, ... )
          ...
!HPF$     INDEPENDENT
          DO I = 1, GRID%SIZE
             GRID%BL(I) = ...
          END DO
          ...
```

Note that the outer loop is executed sequentially by all processors and all processors simultaneously call the solver routine on each grid. Here, the communication required for the routine *SOLVE_GRID* is similar to that necessary for a typical structured grid code and can be generated by the compiler in a similar fashion. The communication required for the *UPDATE_BDRY* routine is a little more complicated since the actual patch to be transferred from one grid to another grid is not known until runtime. However, since the distribution of the grids is known this is not very difficult to generate.

The third distribution implies that the two loops – the one calling the routine *SOLVE_GRID* and the one inside the routine – can both be potentially executed in parallel. For example, consider the situation in which one grid is mapped to processors 1 and 2 while a second grid is spread across processors 3, 4 and

[2] This is under the assumption that the underlying system does not support one-sided communication since in that case the processor owning the data does not need to be involved in the communication.

5. Then, the first subset of processors can execute the call to the solver routine on the first grid while the second subset can execute the call on the second grid in parallel. Furthermore, since each of the calls is being executed on more than one processor, the code within the routine can also be executed in parallel. Thus, the loops outside and within the routine have to be declared parallel by combining the above two versions of the declarations:

```
        CALL UPDATE_BDRY (DOMAIN)

!HPF$  INDEPENDENT
        DO I = 1, GRID_COUNT
!HPF$      ON ( HOME (DOMAIN(I)), RESIDENT
            CALL SOLVE_GRID( DOMAIN(I), ... )
        END DO

         ...

        SUBROUTINE SOLVE_GRID( GRID, ... )
          ...
!HPF$     INDEPENDENT
          DO I = 1, GRID%SIZE
             GRID%BL(I) = ...
          END DO
           ...
```

Note that one still requires the **ON** and **RESIDENT** directives on the outer loop, otherwise the calls to the solver routines would be sequentialized.

4.3 Performance Results

We have implemented a model multiblock code in HPF and studied the performance of the code generated by Portland Group's HPF compiler for the IBM SP2. Since Version 2.2 of the compiler is restricted to HPF 1.1 and does not support distribution to subsets of processors, we could implement only the first two of the distribution strategies described above. Similarly, the compiler does not support mapped pointers or mapped components of derived types; thus we had to use a fixed number of grids whose sizes were known at compile time.

The performance of the first distribution strategy is presented in Table **??**, where parallel speedup is computed with respect to the time for the HPF code on a single processor and the sequential speedup is computed with respect to the execution time using IBM's XLF (sequential) Fortran 77 compiler. As expected, the code achieves perfect parallel speedup for the solver routine since it does not contain any communication. Also note that the speedups are essentially constant as the number of processors is increased from 4 to 7 since the processors with more than one grid control the total execution time. It is only when the eight grids are distributed over eight processors that we again see a jump in speedup. The table also presents the performance of a hand-coded MPI version of the code. As can be seen, the performance of the HPF code is comparable to that of the MPI code which shows similar trends.

Table **??** presents the performance of the HPF code using the second distribution strategy in which the elements of two equal-sized grids are distributed across all the processors. The HPF code shows the typical performance of structured grid codes with nearest neighbor communication in that the speedup trails off as the number of processors is increased, since the communication starts to dominate the per-processor computation.

5 Processing Unstructured Grids

5.1 Overview

Many simulations deal with problem domains characterized by complex geometries and require adaptivity to handle shocks, vortex sheets and other regions of strong gradients. There are two alternatives for discretizing

Performance of HPF code using PGI's HPF compiler Version 2.2

8 Equal Sized Blocks: 100x28x20; First Distribution Strategy							
Processors	Exec Time/Iteration (sec)			Parallel Speedup		Sequential Speedup	
	Solver	Bdry	Entire	Solver	Entire	Solver	Entire
1	2.34	0.13	2.48	1.00	1.00	0.86	0.89
2	1.19	0.55	1.75	2.00	1.42	1.68	1.26
3	0.90	1.31	2.22	2.67	1.11	2.24	0.99
4	0.60	1.34	1.95	4.00	1.28	3.35	1.14
5	0.60	1.44	2.06	4.00	1.21	3.35	1.07
6	0.60	1.50	2.06	4.00	1.20	3.35	1.07
7	0.60	1.56	2.17	4.00	1.14	3.35	1.02
8	0.30	1.42	1.73	8.00	1.44	6.70	1.28

Performance of MPI code

8 Equal Sized Blocks: 100x28x20; First Distribution Strategy							
Processors	Exec Time/Iteration (sec)			Parallel Speedup		Sequential Speedup	
	Solver	Bdry	Entire	Solver	Entire	Solver	Entire
1	2.08	0.13	2.22	1.00	1.00	0.97	1.00
2	1.04	0.41	1.48	2.00	1.50	1.93	1.49
3	0.78	0.90	1.69	2.66	1.31	2.58	1.31
4	0.52	0.93	1.46	4.00	1.52	3.86	1.51
5	0.52	1.00	1.53	4.00	1.45	3.86	1.44
6	0.52	1.03	1.56	4.00	1.42	3.86	1.42
7	0.52	1.06	1.62	4.00	1.37	3.86	1.36
8	0.26	1.01	1.27	8.00	1.75	7.73	1.74

Table 1: Performance of multiblock code on the IBM SP2 using the first distribution strategy

Performance of HPF code using PGI's HPF compiler Version 2.2

2 Equal Sized Blocks: 28x12x128; Second Distribution Strategy									
Procs	Exec Time/Iteration (sec)			Parallel Speedup			Sequential Speedup		
	Solver	Bdry	Entire	Solver	Bdry	Entire	Solver	Bdry	Entire
1	0.373	0.1573	0.535	1.00	1.00	1.00	0.75	0.23	0.60
2	0.201	0.0758	0.281	1.86	2.08	1.90	1.40	0.48	1.14
4	0.108	0.0397	0.151	3.45	3.96	3.54	2.60	0.91	2.11
8	0.057	0.0226	0.083	6.54	6.96	6.54	4.93	1.60	3.84
16	0.032	0.0140	0.051	11.66	11.23	10.49	8.78	2.59	6.25

Table 2: Performance of multiblock code on the IBM SP2 using the second distribution strategy

a set of equations over a domain [?]: (1) use a coordinate transformation to map the flow domain to a rectangular domain for which a Cartesian grid can be used, or (2) cover the domain with an *unstructured grid*. For complicated domains, the second approach is often preferred. For example, an unstructured grid used for modeling the flow of air over the body of an airplane may consist of a set of triangles in 2D space (or tetrahedra in 3D space), with the density of the gridpoints depending on the detail of modeling required.

The logical simplicity of regular grids – where the coordinates of one gridpoint can be used to immediately determine the coordinates of all its neighbors – is lost for unstructured grids: the numbering of the vertices in such a grid reflects properties of the grid generation algorithm, the object geometry, and the refinement strategy. In general, it cannot be assumed that the associated order is correlated with the physical location of gridpoints. As a consequence, the neighborhood relation must be explicitly represented.

Assume that the problem space is a bounded region in s-dimensional space, $\Re^s$, where $\Re$ denotes the set of real numbers. Then we can define a **grid** formally as a system $G = (X, E, C)$, where

- $X = \{x_1, \ldots, x_n\}$ is a discrete finite set of *vertices* or *gridpoints*. Every $i, 1 \leq i \leq n$, is called a *grid index*; the set $[1:n]$ is the *grid index space*.
- (X, E) is an undirected graph, where $E \subseteq X \times X$ is the set of edges connecting neighbors, and
- $C : X \rightarrow \Re^s$ defines the *coordinates* for each vertex in the physical problem space.

The data structures representing an unstructured grid, and thus the associated data access patterns, are in general not known at compile time. Many analysis tasks – such as determining the communication pattern for accessing nonlocal data in a parallel loop – must therefore be performed at runtime.

In this section, we will discuss a number of representations for unstructured grids. We begin with a simple data structure used in the context of a relaxation algorithm. Assume that grid $G = (X, E, C)$ is either generated during program execution or read in, and that it remains invariant thereafter. We represent, allocate and initialize G in a Fortran program as illustrated in Figure **??**. *GRID* is a one-dimensional, allocatable array, whose elements correspond to the x_i and collectively store all the information pertaining to the grid. The derived type *NODE* represents the element type of *GRID*; it contains the following pieces of information related to each vertex:

- its coordinates x, y (representing C, under the assumption $s = 2$),
- linkage information (representing E), specifying the number of neighbors (*NNB*) of the vertex, and their grid indices (*NB*),
- a weight, W, for each neighbor, used in the relaxation algorithm, and a flow variable, V.

During the sweep over the grid, the neighbors of each vertex are accessed indirectly, using *NB*.

Now assume that this algorithm is to be executed on a scalable parallel machine, based upon a reformulation in HPF. Note that the outermost loop is sequential, the I-loop does not contain any loop-carried dependences (if we ignore the dependences related to T, which is a private variable of each iteration), and the access pattern to the neighbors of the grid vertices – which has to be determined at runtime – remains invariant over the iterations of the IT-loop.

In the remainder of this section we discuss the related issues of data and work distribution (Section **??**) and methods for controlling the computation of communication schedules for loops with irregular accesses (Section **??**).

5.2 Data and Work Distribution

Consider the array *GRID* in the code shown in Figure **??** in the light of the data distributions offered by HPF 2.0. In general, the size of this array will be very large. As a consequence, replication of the array across all processors is impractical. Using a block or cyclic distribution would balance the load by allocating distribution segments of approximately the same size, if the amount of work per gridpoint is about equal. However, note that since no assumption can be made about the order of gridpoints such a distribution – which subdivides the index domain in a regular way – may be highly irregular in terms of the physical

```
PARAMETER (MAXNNB = ...)          ! maximum number of neighbors for a gridpoint
INTEGER :: N, I , IT              ! number of vertices
REAL       :: T                   ! temporary variable
TYPE NODE
   REAL :: x, y                   ! coordinates
   REAL ::  V                     ! flow variable
   INTEGER :: NNB                 ! number of neighbors
   INTEGER :: NB(MAXNNB)          ! list of neighbors
   REAL   :: W(MAXNNB)            ! weights of neighbors
END TYPE NODE

TYPE (NODE), ALLOCATABLE :: GRID(:)
REAL, ALLOCATABLE :: TMP(:)
        ...
! Read gridsize and allocate GRID and TMP
READ (*, *) N
ALLOCATE (GRID(N), TMP(N))
CALL INIT_GRID(GRID)             ! initialize grid data structure
! Grid processing
DO IT = 1, maxtime               ! timestep loop
     TMP = GRID%V
! Sweep over the grid
     DO I = 1, N
          T = 0.0
          DO J = 1, GRID(I)%NNB
               T = T + GRID(I)%W(J) * TMP(GRID(I)%NB(J))
          ENDDO
          GRID(I)%V = GRID(I)%V + T
     ENDDO
ENDDO
```

Figure 1: Relaxation sweep over an unstructured grid in Fortran 90

locations of gridpoints.

We will discuss grid partitioning in Section **??**, and then deal with two data distribution strategies that reflect the properties of a given partition:

1. *Reordering.* With this method, the elements of the array *GRID* are permuted in such a way that each component of a grid partition is mapped to a contiguous subset of the index domain. After this permutation has been applied, *GRID* can be distributed by *block* or *general block*, depending on whether or not the number of gridpoints differs for different components. This will be discussed in Section **??**.

2. *Indirect Distribution.* An alternative strategy distributes *GRID* in such a way that the distribution segments are directly modeled after the components of the partition. Such sets are specified by an explicit mapping from indices to processors. This method will be discussed in Section **??**.

5.2.1 Grid Partitioning

Grid partitioning subdivides a grid into contiguous, mutually disjoint regions which are mapped to the processors of a parallel machine, thus specifying for each processor a portion of the grid which is local to that processor.

The overall criterion for partitioning is the minimization of the total execution time, which depends on many parameters, including the total processing load, the degree of parallelism in the algorithm, the amount of communication, the amount of processing in each node, and the load balance. A key role for the amount of communication is played by the number of *cut edges*. Minimizing this number is a good metric for the quality of a partitioning strategy.

In the following, assume that the vertex set, X, of grid $G = (X, E, C)$ is to be partitioned with respect to the processor set $\mathbf{P}$, where np denotes the number of processors. A **grid partition** of G is a total function $\delta^G : [1:n] \rightarrow \mathbf{P}$, where for all grid indices i, $\delta^G(i)$ determines the processor to which x_i is being mapped. Given δ^G, we define for each $p \in \mathbf{P}$ the **component** of p as $\lambda^G(p) := \{i \mid \delta^G(i) = p\}$.

In general, we cannot assume that different components are of the same size: depending on the amount of processing to be performed for individual gridpoints, the partitioner may balance the load by trying to equate the overall weights in a component rather than the number of gridpoints.

The following two subsections will illustrate the two different methods for imposing the structure of a grid partition on the distribution of a related array.

5.2.2 Reordering

Let A denote an array of shape $[1:n]$, and δ^G a grid partition. We wish to use a general block distribution for A, such that each block corresponds to a component associated with δ^G. This can be enforced by permuting the elements of A correspondingly. The following steps have to be performed:

1. Determine a permutation, π, of the grid index space, such that in the transformed space, the indices associated with a processor form a contiguous set.

   For example, if np=*4*, $n = 12$, and $\delta^G = (1,4,2,2,1,4,3,4,3,1,1,4)$, then $\lambda^G(1) = \{1,5,10,11\}$, $\lambda^G(2) = \{3,4\}$, $\lambda^G(3) = \{7,9\}$, and $\lambda^G(4) = \{2,6,8,12\}$. A suitable permutation of the index domain is therefore, for example, $\pi = (1,9,5,6,2,10,7,11,8,3,4,12)$.

2. Apply π to A, i.e., for all indices i, replace $A(\pi(i))$ by $A(i)$.

3. Define the distribution of A as $\delta^A(i) := \delta^G(\pi^{-1}(i))$. δ^A is a general block distribution.

   For the above example, $\delta^A = (1,1,1,1,2,2,3,3,4,4,4,4)$.

4. Replace each reference, j, to an element of A by the permuted value $\pi(j)$.

The example code in Figure **??** illustrates the application of this method to the code of Figure **??**. The algorithm is specified in HPF 2.0, using dynamic redistribution and general block distributions.

Processors	Block Distribution	Indirect Distribution
4	30.25	27.81
8	24.66	19.50
16	22.69	13.50

Sequential Time: 43.47 secs

Table 3: Timings for conjugate gradient solver (in seconds)

Dynamic redistribution is required since the grid is partitioned at runtime, and therefore the array element permutation and the related general block distribution can only be determined dynamically. The general block distribution is necessary since in most cases the number of elements in different components of the grid partition will not be the same. If this load balancing problem is not an important factor, the algorithm can be coded in the base language, relying on a static regular block distribution.

5.2.3 Indirect Distributions

Indirect distributions can be used to model arbitrarily structured grid partitions by setting up a *mapping array*. Let A, of shape $[1 : n]$, be given as above, and let δ^G denote the grid partition. The mapping array, say *MAP*, with the same shape as A, is defined by setting $MAP(i) := \delta^G(i)$. An indirect distribution for A via mapping array *MAP* is represented in the program by *INDIRECT(MAP)* and defines the distribution of A to be identical to the grid partition: $\delta^A = \delta^G$.

The application of this method to the example code of Figure **??** yields the following HPF program:

```
!HPF$  PROCESSORS P(NUMBER_OF_PROCESSORS())
       TYPE (NODE), ALLOCATABLE :: GRID(:)
       REAL , ALLOCATABLE :: TMP(:)
       INTEGER , ALLOCATABLE :: MAP(:)          ! mapping array
!HPF$  DYNAMIC , DISTRIBUTE( BLOCK ) :: GRID
!HPF$  DYNAMIC , ALIGN WITH GRID     :: TMP
!HPF$  DISTRIBUTE( BLOCK ) :: MAP
        ...
! Read grid size and allocate GRID, TMP, and MAP
       READ (*, *) N
       ALLOCATE (GRID(N), TMP(N), MAP(N))
! Initialize grid data structure
       CALL GRID_PARTITIONER_IND(GRID, MAP)  ! this routine partitions the grid and defines MAP
!HPF$  REDISTRIBUTE GRID(INDIRECT(MAP))     ! redistribution of GRID and TMP
! Grid processing: same as for the reordering method in Figure ??
```

Note that a (physical) reordering of array elements (as in Figure **??**) or a resetting of links is not required here. However, the implementation of the access to elements of *GRID* is more complex: while, for a block distribution of *GRID*, the processor on which an element *GRID(I)* resides can be easily determined by essentially computing *I/block-length*[3], the corresponding access for an indirect distribution requires a look-up in a *translation table*, which represents *MAP*. Since translation tables are, due to their size, usually also distributed (in the code above, a block distribution is used), such an access may be expensive and require two communication steps.

Table **??** provides timings for a conjugate gradient kernel, comparing performance figures for *block* and *indirect* distributions on an Intel iPSC/860 (NX) with 16 processors. The timings are based on an unstructured grid with 19061 cells and 338 solver iterations.

[3] This procedure is only slightly more complicated for general block distributions.

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
      PARAMETER (MAXNNB = ...)
      INTEGER :: N, I, IT
      REAL    :: T
      TYPE NODE
         ...
      END TYPE NODE

      TYPE (NODE), ALLOCATABLE :: GRID(:)
      REAL, ALLOCATABLE :: TMP(:)
      INTEGER, ALLOCATABLE :: PI(:)             ! permutation array
!HPF$ DYNAMIC, DISTRIBUTE( BLOCK ) :: GRID
!HPF$ DYNAMIC, ALIGN WITH GRID    :: TMP
!HPF$ DISTRIBUTE( BLOCK ) :: PI
      INTEGER :: H(NUMBER_OF_PROCESSORS())   ! auxiliary array for general block distribution
        ...
! Read grid size and allocate GRID, TMP, and PI
      READ (*, *) N
      ALLOCATE (GRID(N), TMP(N), PI(N))
      CALL INIT_GRID(GRID)                  ! initialize grid data structure
      CALL GRID_PARTITIONER_RO(GRID, PI)    ! this routine partitions the grid, constructs an
                                            ! associated permutation and assigns it to PI
      CALL CONSTRUCT(PI, H)                 ! uses PI to construct H
      GRID(PI) = GRID                       ! reordering
!HPF$ REDISTRIBUTE GRID( GEN_BLOCK (H))     ! redistribution of GRID and TMP
! Reset links
!HPF$ INDEPENDENT, NEW (J), ON HOME (GRID(I)),
      DO I = 1, N
         DO J = 1, GRID(I)%NNB
            GRID(I)%NB(J) = GRID(I)%NB(PI(J))
         ENDDO
      ENDDO

! Grid processing
      DO IT = 1, maxtime
         TMP = GRID%V
      ! Sweep over the grid
!HPF$     INDEPENDENT, NEW (J, T), ON HOME (GRID(I))
         DO I = 1, N
            T = 0.0
            DO J = 1, GRID(I)%NNB
               T = T + GRID(I)%W(J) * TMP(GRID(I)%NB(J))
            ENDDO
            GRID(I)%V = GRID(I)%V + T
         ENDDO
      ENDDO
```

Figure 2: The reordering method for the unstructured grid computation of Figure **??**

```
!HPF$   PROCESSORS P(NUMBER_OF_PROCESSORS())
        INTEGER :: N, J              ! number of vertices
        INTEGER :: M                 ! number of edges
        TYPE NODE
          ...
        REAL ::  V1, V2            ! flow variables
        END TYPE NODE

        TYPE(NODE), ALLOCATABLE :: GRID(:)
        REAL , ALLOCATABLE :: EDGE(:, 2)
        INTEGER, ALLOCATABLE :: MAP(:)    ! mapping array
!HPF$  DYNAMIC, DISTRIBUTE(BLOCK) :: GRID
!HPF$  DYNAMIC, DISTRIBUTE(BLOCK, *) :: EDGE
!HPF$  DISTRIBUTE(BLOCK) :: MAP
          ...
! Read n, allocate GRID and MAP, initialize GRID, and determine m
        CALL GRID_PARTITIONER_IND(GRID, MAP)
!HPF$ REDISTRIBUTE GRID(INDIRECT(MAP))
       ALLOCATE(EDGE(M, 2))
! Code for initialization of EDGE
!HPF+  REALIGN EDGE(J, :) WITH GRID(EDGE(J, 1)) ! value-based alignment of edges with gridpoints
! Sweep over the edges of the grid
!HPF$ INDEPENDENT, ON HOME(EDGE(J, 1)), NEW(N1, N2, DELTAX), REDUCTION(V2)
       DO J = 1, M
        ...
        N1 = EDGE(J, 1);    N2 = EDGE(J, 2)
        ...
        DELTAV = F(V1(N1), V1(N2))
        V2(N1) = V2(N1) - DELTAV
        V2(N2) = V2(N2) + DELTAV
       ENDDO
```

Figure 3: Sweep over edges of an unstructured grid

5.2.4 Sweep Over the Edges of an Unstructured Grid

The code in Figure **??** is based upon an unstructured grid, $G = (X, E, C)$, of triangles used in the context of a 2D Euler solver. Here, the computation is different from that presented earlier, and models iterations over the set of *edges* of the grid, accumulating the contributions on the incident nodes.

Since the algorithm performs a sweep over the edges, performance considerations imply that the grid representation as used in the previous codes is not adequate (although it contains all the required information). We represent the edges of the grid explicitly by an array *EDGE*, where *EDGE(J, K)*, $1 \leq K \leq 2$, is the grid index of vertex K for edge J. As before, we do not make any assumptions about the ordering of gridpoints or edges.

In order to reduce communication, the array *EDGE* has to be properly aligned with the array *GRID.* This requires a generalized form of "value-based" alignment, which is not offered by HPF. HPF+ [**?**] provides support for this feature via the following construct:

```
!HPF+  REALIGN EDGE(J, :) WITH GRID(EDGE(J, 1))
```

This has the following meaning. Given the distribution of *GRID* (as specified by *MAP*), the distribution of *EDGE* is determined as follows: $\delta^{EDGE}(J,:) = \delta^{GRID}(EDGE(J,1))$, i.e., elements *EDGE(J, 1)* and *EDGE(J, 2)* are mapped to the same processor as the *GRID* element which is the first vertex of that edge.

5.3 Control of Communication Schedules

In all examples for parallel loops shown in this section, arrays are accessed indirectly, using index arrays. This is due to the fact that the grid is generated dynamically, and the neighborhood of objects is not reflected by a similar neighborhood of their indices, as is the case in dense regular computations.

As a consequence, the analysis of data accesses in such loops cannot be done by the compiler, but must be performed at runtime. Since a parallel loop does not have loop-carried dependences, all the communication required for the loop can be done immediately before and/or immediately after its execution. The standard approach to translating such a loop has been the *inspector/executor* paradigm [?, ?]. In this paradigm, the inspector determines the work distribution of the loop and analyzes the access patterns to arrays, resulting in the computation of communication schedules for nonlocal accesses. Schedules may either be *gather schedules* – for nonlocal read accesses, or *scatter schedules* – for nonlocal writes. The executor performs the actual communication according to the schedules determined by the inspector and executes the iterations of the loop.

The PARTI and CHAOS libraries [?] provide a set of primitives that support the inspector/executor paradigm. They have been integrated into a number of compilers, including the Vienna Fortran Compilation System(VFCS) [?] and the Fortran D system [?].

The inspector phase may be very expensive and represents a significant runtime overhead, which in some cases may dominate the overall execution time [?, ?]. It is therefore important to be able to deal with situations in which the schedule computed by the inspector remains invariant over multiple executions of the parallel loop (for example, if the parallel loop is enclosed in a time-step loop). If this is the case, the inspector needs to be executed only when the loop is entered for the first time, suppressing subsequent computations of the communication schedule. Three approaches to this problem have been studied:

- *Inspector hoisting* [?] determines at compile time the set of program statements and data on which the inspector depends. If the compiler can assert invariance, it can modify the code generated for the inspector/executor paradigm to suppress redundant computations of the communication schedule.

  This is difficult for many "real" programs, in which calls to parallel loops tend to occur in a deep nest of procedure calls which may be impossible to analyze statically.

- *Timestamping* denotes a method in which assignments to indirection arrays are monitored at runtime. If, at the time the inspector is activated, the communication schedule has already been computed in a previous iteration and none of the indirection arrays on which the access pattern depends has changed in the meantime, then a recomputation of the communication schedule can be avoided. Experiments with this method have been reported in [?].

- *Explicit schedule control* gives the programmer linguistic mechanisms to explicitly control the computation and application of communication schedules. These language extensions, described in detail in [?], have been included in HPF+ and implemented in VFCS. This is illustrated by the example discussed below.

Explicit Schedule Control

The code fragment below illustrates the use of *schedule variables* to control the computation of a communication schedule.

```
!HPF+ SCHEDULE :: S
        ...
        DO t = 1, max_time
        ...
!HPF+        INDEPENDENT, ON HOME(C(I)), GATHER (B::S)
           L: DO I = 1, N
                 ...
                 A(I) = B(IX(I)) + C(I)
                 ...
```

Times without schedule reuse

Processors	Total (Unoptimized)	Inspector	Gather	Scatter	Speedup
2	598.98	347.63 (58%)	2.31	2.38	0.9
4	304.75	172.48 (57%)	4.33	6.22	1.8
8	160.19	89.60 (56%)	5.33	7.81	3.4
16	100.04	50.66 (51%)	8.68	10.78	5.5
32	80.99	36.34 (45%)	10.78	11.65	6.7

Times with schedule reuse

Processors	Total (Schedule Reuse)	Inspector	Speedup	Unoptimized/Schedule Reuse
2	285.22	1.47	1.9	2.1
4	153.00	0.81	3.6	2.0
8	79.10	0.47	6.9	2.0
16	47.47	0.31	11.5	2.1
32	35.31	0.20	15.4	2.3

Sequential Time: 545.45 secs

Table 4: Times for Finite Element kernel.

```
              END DO
                ...
              IF RECONFIGURE(...) THEN CALL RECOMPUTE(IX)
!HPF+              RESET S
              END IF
         END DO
```

Here, S is an explicitly declared schedule variable which is initially undefined. Therefore, the first time the parallel loop is encountered, the inspector is fully executed and determines a communication schedule as discussed above. This schedule is assigned to S. Upon subsequent executions of the parallel loop, this schedule is reused, as long as the call to *RECONFIGURE* yields **false**. Once this call yields **true**, S is *reset*, resulting in the disassociation of S from the previously computed schedule, and setting its value to *undefined*. With the next execution of the parallel loop, the inspector has to be run again, and the above cycle is restarted.

We show the effect of redundant inspector elimination for a kernel based on PAM-CRASH [?], a commercial Finite Element code, with a grid containing 35571 nodes and 35000 elements. Table **??** shows the timings for the kernel as parallelized by VFCS and executed on the Meiko CS-2, without and with explicit schedule reuse.

6 Beyond HPF and Its Paradigm

HPF was primarily designed to support data parallel programming on architectures for which memory locality is important. Consequently, the language is not necessarily well suited to other classes of applications or to other kinds of parallel architecture. In this section, we will discuss a few aspects of this issue.

First, we will take another look at the capabilities of HPF for dealing with complex data parallel problems (Section **??**). Second, we study the relevance of HPF with respect to applications that require task parallelism, possibly in combination with data parallelism (Section **??**). Finally, we will briefly discuss the suitability of HPF for new classes of parallel architectures that have become commercially available over the past few years (Section **??**).

6.1 Difficult Data Parallel Problems

HPF does best on simple, regular problems, such as dense linear algebra and partial differential equations on regular meshes. It is also effective for many irregular problems, if the full scope of the language is used. This has been illustrated in this paper for multiblock problems (Section **??**) and sweeps over unstructured meshes (Section **??**); other work in this area includes adaptive N-body simulations [**?**] and sparse matrix computations [**?**].

However, there are data parallel applications which cannot be handled well in HPF. One example of such a method is Discrete Simulation Monte Carlo (DSMC). This is a numerical algorithm for computing flows of rarefied gases in three dimensions by tracking the motion of millions of discrete particles. DSMC is a grid-based method, which typically uses a regular three-dimensional mesh. Associated with each mesh cell is a list of the particles currently residing in that cell. The simulation advances by updating the positions and velocities of all particles. There are two steps involved:

1. Pairs of particles are randomly selected in each cell as candidates for collision. The "collision" then changes the velocity of each of these particles.

2. Once the collisions are done, all particles advance in time, moving some distance along their current trajectories.

In Step 2, as particles move from one cell to another, the membership of each particle list changes. There are typically a million or more cells and several million particles. So updating the lists of particles associated with each cell is, in effect, a large sorting problem. The sequential program generally uses an intermediate index array which allows updating the cell ownership lists without actually moving the data associated with the particles.

There are several ways of expressing the code in HPF [**?**]. Simple, straightforward approaches lead to implementations which either incur unnecessary communication or have large memory overheads thus limiting the size of the problems that can be run. What one requires is some way of expressing the code such that particle motion from one cell to a destination cell on the same processor occurs through index manipulation while if the destination cell is on another processor through data copying involving communication. This is a problem in HPF since conceptually the processor boundaries should be transparent in the code.

A better HPF solution for DSMC can be found if the processors are explicitly modeled in the HPF code. However, this requires significant rewriting of the original program. In contrast, a message-passing solution for DSMC can be developed in a relatively simple way and yields a highly efficient target program.

6.2 Task Parallelism

The data parallel paradigm represented by HPF supports the coordinated execution of a set of identical processes working on different segments of the same problem. For some scientific applications, the discipline enforced by this model is too rigid: they need a capability to express parallelism in a less structured way. Examples of such task parallel applications include real-time signal processing, branch and bound problems, and multidisciplinary applications. They can be generally characterized by the fact that tasks may be created dynamically in an unstructured way, different tasks may have different resource requirements and priorities, and that the structure and volume of the communication between a pair of tasks may vary dramatically, from an exchange of boolean signals to data transfers involving millions of numbers.

HPF is not designed to deal with such problems adequately. However, a number of methods have been proposed to address this issue in the context of the language.

There are two ways of incorporating task parallelism in HPF. One way is to spawn threads within an HPF program, to carry out fine-grain parallel tasks. The other approach is to activate an entire HPF program as a task. This provides coarse-grain parallelism, and within each task one can still express and exploit fine-grain loop or array parallelism. We look in turn at each of these approaches.

There has been little work done in introducing fine-grain task parallelism in HPF. The independent loop construct can be considered as a mechanism for the implicit spawning of fine-grain tasks. Similarly, a pure procedure can be spawned as an independently executing task. Finally, the **TASK_REGION** directive provides a simple mechanism for indicating segments of code which can be executed concurrently similar to parallel sections found in other languages.

While these features allow the spawning of tasks, there are no mechanisms for synchronization or communication, although reductions are allowed in independent loops. Thus these constructs suffice for algorithms like divide and conquer, which require no inter-task communication; however, they will not work for branch and bound, and other algorithms where inter-task communication is essential.

Use of coarse-grain tasks, each comprising an entire HPF program, is both easy and natural. One is, in effect, wrapping HPF in a coordination language. There have been a number of efforts along this line. One is Fortran M[?], in which coarse-grain tasks communicate via "channels". This is almost like message-passing, except that the channels are set up at the beginning of execution, in effect defining the communication topology. In contrast, in message-passing languages, the topology is constructed "on the fly". The advantage of using channels instead of message-passing is subtle, and has more to do with software engineering issues than performance issues.

A related approach is the authors' language Opus [?]. Opus encapsulates HPF programs as object-oriented modules, passing data between them by accessing *shared abstractions (SDAs)* which are monitor-like constructs. Opus was explicitly designed to support multidisciplinary analysis and design, and related applications with significant coarse-grain parallelism.

6.3 Challenges Posed by New Parallel Architectures

In recent years, a new generation of high performance architectures has become commercially available. Many of these machines are either symmetric shared-memory architectures (SMPs) or clusters of SMPs, where an interconnection network connects a number of nodes, each of which is an SMP. Thus, these machines display a hybrid structure integrating shared-memory with distributed-memory parallelism. One of their dominating characteristics is their use of a deep memory hierarchy, often involving multiple levels of cache.

As a consequence, these architectures have not only to deal with the locality problem typical for distributed-memory machines – which is addressed by HPF –, but also with cache locality. A cache miss in a program executing on a cluster of SMPs may be more expensive than a non-local memory access.

HPF and its compilers currently are not designed to deal with such issues. The future will show whether the (possibly extended) HPF paradigm will be able to efficiently cope with such architectures, or whether other programming methods will prove more adequate.

7 Conclusion

HPF is a well-designed language which can handle most data parallel scientific applications with reasonable facility. However, as architectures evolve and scientific programming becomes more sophisticated, the limitations of the language are becoming increasingly apparent. There are at least three points of view one could take:

1. HPF is too high-level a language — MPI-style languages are more appropriate.
2. HPF is too low-level a language — aggressive compiler technologies and improving architectures obviate the need for HPF-style compiler directives.
3. The level of HPF is about right, but extensions are required to handle some applications for some upcoming architectures.

All three of these alternatives are being actively pursued by language researchers. For example, HPC++ [?] is an effort to design an HPF-style language using C++ as a base. On the other hand, F - - [?] is an attempt to provide a lower-level data-parallel language than HPF. Like HPF, F - - provides a single thread of flow control. But unlike HPF, F - - requires all communication to be explicit using "get" and "put" primitives.

While it is difficult to predict where languages will head, the coming generation of SMP-cluster architectures may induce new families of languages which will take advantage of the hardware support for shared-memory semantics with an SMP, while covering the limited global communication capability of the architectures. In this effort the experience gained in the development and implementation of HPF will surely serve us well.

Acknowledgment

The authors would like to thank Kevin Roe (ICASE), Siegfried Benkner and Viera Sipkova (University of Vienna), Guy Lonsdale (NEC) and George Mozdzynski (ECMWF) for many fruitful discussions on this subject. The performance figures presented in Section ?? were provided by Kevin Roe and those in Section ?? were provided by S. Benkner and V. Sipkova. Execution time on the NASA IBM SP2s was provided by the NASA Ames/Langley Metacenter under the High Performance Computing and Communications Program.

References

[1] G. Agrawal, A. Sussman and J. Saltz. An Integrated Runtime and Compile-Time Approach for Parallelizing Structured and Block Structured Applications. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):747-754, July, 1995.

[2] E. Albert, K. Knobe, J. D. Lukas, and Jr. G. L. Steele. Compiling Fortran 8x array features for the Connection Machine Computer System. In *Proceedings of the Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, pages 42–56, New Haven, CT, July 1988.

[3] F. André, J.-L. Pazat, and H. Thomas. PANDORE: A system to manage data distribution. In *International Conference on Supercomputing*, pages 380–388, Amsterdam, The Netherlands, June 1990.

[4] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. M. Chapman, M. Egg, T. Fahringer, J. Hulman, Y. Hou, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender and H. P. Zima. *Vienna Fortran Compilation System - Version 1.2 - User's Guide.* Institute for Software Technology and Parallel Systems, University of Vienna, October 1995.

[5] S. Benkner, P. Mehrotra, J. Van Rosendale and H. P. Zima. High-Level Management of Communication Schedules in HPF-like Languages. Technical Report, TR 97-5, Institute for Software Technology and Parallel Systems, University of Vienna, April 1997.

[6] B. Chapman, M. Haines, P. Mehrotra, J. Van Rosendale, and H. Zima. Opus: A coordination language for multidisciplinary applications. *Scientific Programming (to appear)*, 1997.

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

[8] B. Chapman, P. Mehrotra, and H. Zima. Extending HPF for Advanced Data Parallel Applications. *IEEE Parallel and Distributed Technology*, Fall 1994, pp. 59-70.

[9] R. Das, J. Saltz, R. von Hanxleden. Slicing Analysis and Indirect Access to Distributed Arrays. In: *Proc. 6th Workshop on Languages and Compilers for Parallel Computing*, 152-168, Springer Verlag (August 1993).

[10] B. Numrich. A Parallel Extension to Fortran 90. In: *Proc. Spring'96 Cray User Group Conference*, Barcelona, March 1996.

[11] 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.

[12] 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.

[13] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and J. Anderson. A production quality C* compiler for hypercube machines. In *3rd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 73–82, April 1991.

[14] E. Haug, J. Dubois, J. Clinckemaillie, S. Vlachoutsis, G. Lonsdale. Transport Vehicle Crash, Safety and Manufacturing Simulation in the Perspective of High Performance Computing and Networking. *Future Generation Computer Systems*, Vol.10, pp. 173-181, 1994.

[15] High Performance C++. WWW page *http://www.extreme.indiana.edu/hpc++/index.html*.

[16] High Performance FORTRAN Forum. *High Performance FORTRAN Language Specification, Version 2.0*, January 1997.

[17] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[18] Y.C.Hu, S.Lennart Johnsson, and S.-H.Teng. A Data-Parallel Adaptive N-body Method. Proc.Eight SIAM Conference on Parallel Processing for Scientific Computing, March 14 – 17, 1997.

[19] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of The Fifth Distributed Memory Computing Conference*, pages 1105–1114, Charleston, SC, April 1990.

[20] 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.

[21] C. Koelbel, P. Mehrotra, J. Saltz, and H. Berryman. Parallel loops on distributed machines. In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 1097–1104, Charleston, SC, April 1990.

[22] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings of Supercomputing '90*, pages 865–876, New York, NY, November 1990.

[23] D. Loveman. High Performance Fortran: Proposal. In High Performance Fortran Forum, January 1992.

[24] P. Mehrotra. Programming parallel architectures: The BLAZE family of languages. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 289–299, Los Angeles, CA, December 1988.

[25] P. Mehrotra and J. Van Rosendale. Compiling high level constructs to distributed memory architectures. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[26] 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*, pages 364–384. Pitman/MIT-Press, 1991.

[27] J. H. Merlin. Adapting fortran 90 array programs for distributed memory architectures. In H. P. Zima, editor, *Proc. First International ACPC Conference, Salzburg, Austria*, pages 184–200. Lecture Notes in Computer Science 591, Springer Verlag, 1991.

[28] D. Middleton, P. Mehrotra, and J. Van Rosendale. Expressing Direct Simulation Monte Carlo Code in High Performance Fortran. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 698–703, February 1995.

[29] R. E. Millstein. Control structures in ILLIAC IV Fortran. *Communications of the ACM*, 16(10):621–627, October 1973.

[30] *MIMDizer User's Guide, Version 7.02.* Placerville, CA., 1991.

[31] D. Pase. MPP Fortran programming model. In High Performance Fortran Forum, January 1992.

[32] 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.

[33] A. P. Reeves and C. M. Chase. The Paragon programming paradigm and distributed memory multicomputers. In *Compilers and Runtime Software for Scalable Multiprocessors, J. Saltz and P. Mehrotra Editors*, Amsterdam, The Netherlands, Elsevier, 1991.

[34] K. Roe and P. Mehrotra. Evaluation of High Performance Fortran for CAS Applications. In *Proceedings for the Workshop on CAS Applications*, NASA Ames Research Center, pp. 133-138, August 1996.

[35] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 69–80, Portland, OR, June 1989.

[36] M. Rosing, R. W. Schnabel, and R. P. Weaver. Expressing complex parallel algorithms in DINO. In *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*, pages 553–560, 1989.

[37] R. Rühl and M. Annaratone. Parallelization of Fortran code on distributed-memory parallel processors. In *Proceedings of the International Conference on Supercomputing*. ACM Press, June 1990.

[38] 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.

[39] Schetz, J. A. and Fuhs, A. E. (Eds.). Handbook of Fluid Dynamics and Fluid Machinery, Volume Two, Chapter 19. John Wiley &Sons (1996).

[40] P. S. Tseng. A systolic array programming language. In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 1125–1130, April 1990.

[41] M.Ujaldon, E.L.Zapata, B.Chapman, and H.Zima. Vienna Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(11), November 1997.

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

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

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

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

[46] 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.