NASA/CR-2000-210321
ICASE Report No. 2000-31

High Performance Fortran for Aerospace Applications

Piyush Mehrotra
ICASE, Hampton, Virginia

Hans Zima
University of Vienna, Vienna, Austria

ICASE
NASA Langley Research Center
Hampton, Virginia

Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center under
Contracts NAS1-19480 and NAS1-97046

December 2000

HIGH PERFORMANCE FORTRAN FOR AEROSPACE APPLICATIONS*

PIYUSH MEHROTRA† AND HANS ZIMA‡

Abstract. This paper focuses on the use of High Performance Fortran (HPF) for important classes of algorithms employed in aerospace applications. HPF is a set of Fortran extensions designed to provide users with a high-level interface for programming data parallel scientific applications, while delegating to the compiler/runtime system the task of generating explicitly parallel message-passing programs. We begin by providing a short overview of the HPF language. This is followed by a detailed discussion of the efficient use of HPF for applications involving multiple structured grids such as multiblock and adaptive mesh refinement (AMR) codes as well as unstructured grid codes. We focus on the data structures and computational structures used in these codes and on the high-level strategies that can be expressed in HPF to optimally exploit the parallelism in these algorithms.

Key words. distributed memory multprocessing, high-level language, distribution directives

Subject classification. Computer Science

1. Introduction. Exploiting the full potential of parallel architectures requires a cooperative effort between the user and the language system. There is a clear trade-off between the amount of information the user has to provide and the amount of effort the compiler has to expend to generate optimal parallel code. The spectrum ranges from low-level languages in which the user has to explicitly encode all the parallelism while the compiler effort is minimal, to sequential languages where the compiler has the full responsibility for extracting the parallelism. High Performance Fortran (HPF) takes the middle ground - sharing the responsibility between the user and the compiler/runtime system. It does this by providing Fortran directives which allow the user to express the parallelism and control the data locality at a very high level while utilizing a compiler which uses this information to generate the low-level details such as the required communication statements.

In this paper, we focus on applications from Computational Fluid Dynamics (CFD) and show how HPF can be used to express the parallelism for algorithms used in this area. As the requirements of the computational aerodynamicists have increased, applications with single grids have given way to those employing multiple grids and even unstructured grids. We start by providing a brief overview of HPF and use some simple single grid applications to show how HPF directives are used. In Section 3, we focus on applications which use multiple grids in order to generate flow solutions over complex bodies. Section 4 presents unstructured grid applications, describing how the HPF directives can be used to control the data and work distributions required for such codes. Concluding remarks can be found in Section 5.

Note that in this paper we are not concerned with the physics of the computations in these algorithms. Rather we focus on the data structures and the computational structures so that we can describe at a high level the approaches that can be used when employing HPF for exploiting the parallelism in these codes.

*The work described in this paper was partially supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480 and NAS1-97046, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

†ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199 (email: pm@icase.edu).

‡Institute for Software Science, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria. (email: zima@par.univie.ac.at). This research was also supported by the Priority Research Project F011 "AURORA" funded by the Austrian Science Fund.

Also, we do not discuss the compiler optimizations required to generate low-level code from HPF code. Other publications, including [13, 23], cover the required analysis and transformations in great detail.

2. Overview of HPF. High Performance Fortran is a set of Fortran extensions designed to allow specification of data parallel algorithms for a wide range of architectures. The user annotates the program with distribution and alignment directives to specify the desired layout of data. The underlying programming model provides a global name space and a single thread of control. Explicitly parallel constructs allow the expression of fairly controlled forms of parallelism, in particular data parallelism. Thus, the code is specified in 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 wide variety of architectures including SIMD, MIMD shared and distributed-memory machines.

The key concept of HPF – high level directives which allow the sharing of responsibility for exploiting parallelism between the user and the compiler/runtime system – is based on language research done by several groups over the years including [2, 8, 11, 15, 16, 18, 22].

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 defines the basic HPF features which each HPF compiler must support. The Approved Extensions include advanced features 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.

In the next two subsections we provide a brief description of the base language and the approved extensions, respectively. A more complete description of the language can be found in the HPF Language specification [12].

2.1. The Base Language. The HPF 2.0 Base Language supports the following features for specifying the mapping of data and the parallelism in the code.

Data mapping directives. HPF provides an extensive set of directives to specify the mapping of array elements to memory regions associated with "abstract processors." Arrays are first aligned relative to each other and then the aligned group of arrays are distributed onto a rectilinear arrangement of abstract processors. The alignment directives support the mapping of a dimension of an array relative to the dimension of another array. The following types of alignments are allowed: identical alignment, alignment with offset and stride, collapsing, embedding, replication and permutation.

The distribution directives allow each dimension of an array to be independently distributed using the **block** or **cyclic** distribution. The former partitions a dimension of the array into equal-sized contiguous blocks which are distributed across the target set of abstract processors while the latter distributes the elements cyclically across the abstract processors.

Data parallel directives. The current version of HPF (version 2.0) is based on the Fortran 95 standard. Thus, the array constructs of Fortran 90 can be used to specify the data parallelism in the code. Also, the forall statement and construct (which were introduced in HPF version 1.1 and later adopted in Fortran 95) provide a more general mechanism to specify such parallelism.

HPF itself provides the **independent** directive which asserts 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.

```
!HPF$ PROCESSORS  P(NUMBER_OF_PROCESSORS())
      REAL U(1:N, 1:N), F(1:N, 1:N)
!HPF$ ALIGN U :: F
!HPF$ DISTRIBUTE U (*, BLOCK)
      FORALL (I=2:N-1, J = 2:N-1)
            U(I, J) = 0.25 * (F(I, J) + U(I-1, J) + U(I+1, J) + U(I, J-1) + U(I, J+1))
      END FORALL
```

FIG. 2.1. *HPF version of a simple Jacobi procedure*

Intrinsic and library functions. HPF provides a set of new intrinsic functions including system functions to inquire about the underlying hardware, inquiry functions to inquire about the mapping 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 provides *extrinsic* procedures. These define an explicit interface and allow codes expressed using a different language, e.g., C, or a different paradigm, such as an explicit message passing, to be called from an HPF program.

2.2. HPF Approved Extensions. HPF 2.0 Approved Extensions include advanced features which allow more complex applications to be expressed using HPF.

Extensions to data mapping directives. These 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 blocks and the **indirect** distribution allows each element of the data object to be mapped individually using a mapping array.

Another important feature is the support for 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. In particular, redistribution of an array implies that all other arrays aligned with it have to be remapped.

Extensions to data parallel directives. 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 expressed.

2.3. Examples of HPF Codes. In this section we provide two code fragments using some of the HPF features described above. The first is the Jacobi iterative algorithm and the second is the Modified Gram-Schmidt algorithm.

The HPF version of the Jacobi iterative procedure which may be used to approximate the solution of a partial differential equation discretized on a grid, is shown in Figure 2.1. Such an algorithm, using a five-point stencil, is typical of many CFD applications.

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 array U is declared to distributed via the distribution clause (*, **BLOCK**), implying that the second dimension of the array is block distributed. That is, the columns of U (and thus those of F because of the alignment) are distributed by block, by default, across the processor array P. P has been declared to be an 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. Using this inquiry function, allows 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 such as explicit communication statements. 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.

As another example, consider the HPF version of the Modified Gram-Schmidt algorithm given in Figure 2.2[2]:

Again, the first directive declares that the columns of the array V are to be distributed by block across the memories of the underlying processor set. The outer loop, I, is sequential and is thus executed by all processors. Given the column distribution, in the Ith iteration of the outer loop, the first two K loops should be executed by the processor owning the Ith column.

The second directive declares the J loop to be *independent*, thus, the iterations of the J loop can be executed in parallel, i.e., each processor updates the columns that it owns in parallel. Since the Ith column is used for this update, it will have to be broadcast to all processors. Note that the variables K and *TMP* are declared to be *new* variables. That is, they act as private variables for each iteration and thus do not cause any inter-iteration dependences.

Since the columns are distributed by contiguous blocks across the processors, as the computation in the parallel J loop progresses, the processors will become idle. 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 distributed cyclically across the processors, and thus will force the work distribution of the inner J loop to be strip-mined in a cyclic rather than in a block fashion. Thus, all processors will remain busy until the tail end of the computation. Note, that all that is required is a change in the distribution directive. The code representing the computation itself is independent of the distribution and

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

[2] A Fortran 90 version of the code fragment, not shown here, would have used array constructs for the K loops. This would make the parallelism in the inner loops explicit.

```
      REAL V(N, N)
!HPF$ DISTRIBUTE V (*, BLOCK)
      DO I = 1, N
        TMP = 0.0
        DO K = 1, N
          TMP = TMP + V(K, I)*V(K, I)
        ENDDO
        XNORM = 1.0 / SQRT(TMP)
        DO K = 1, N
          V(K, I) = V(K, I) * XNORM
        ENDDO
!HPF$ INDEPENDENT, NEW(K, TMP)
        DO J = I+1, N
          TMP = 0.0
          DO K = 1, N
            TMP = TMP + V(K, I)*V(K, J)
          ENDDO
          DO K = 1, N
            V(K, J) = V(K, J) - TMP*V(K, I)
          ENDDO
        ENDDO
      ENDDO
```

FIG. 2.2. *HPF version of Modified Gram-Schmidt algorithm with a one-dimensional distribution*

hence does not need to be modified. Of course, the code needs to be recompiled so that the compiler can generate the required communications taking into account the new distribution.

The above distributions only exploit parallelism in one dimension, whereas the inner K loops can also run in parallel. This can be achieved by distributing both the dimensions of V as shown in Figure 2.3. Here, the processors are presumed to be arranged in a two-dimensional mesh and the array is distributed such that the elements of a column of the array are distributed by block across a column of processors whereas the columns as a whole are distributed cyclically. Thus, the first K loop becomes a parallel reduction, indicated by the *reduction clause* over the variable TMP, of the Ith column across the set of processors owning the Ith column. Similarly, the second K loop can also be declared to be *independent* and executed in parallel by the column of processors which owns the Ith column. The second set of K loops, inside the J loop, can be similarly parallelized.

In this section, we have provided a brief overview of the HPF language and illustrated the use of the basic directives through two simple examples. In the next two sections we discuss more complex examples and show how the HPF directives can be used to describe the data layout necessary for these codes.

3. HPF-Based Algorithms for Grid Collections. This section deals with HPF-based algorithms that operate on grid collections. More specifically, we define a *grid collection* as a set of structured grids all of which are defined over a given discretized domain in d-dimensional Cartesian space. A *structured (regular)* grid is a contiguous rectilinear arrangement of equal-sized cells in d-dimensional space. It can be

```
      REAL V(N, N)
!HPF$ DISTRIBUTE V (BLOCK, CYCLIC)
      DO I = 1, N
        TMP = 0.0
!HPF$   INDEPENDENT, REDUCTION(TMP)
        DO K = 1, N
          TMP = TMP + V(K, I)*V(K, I)
        ENDDO
        XNORM = 1.0 / SQRT(TMP)
!HPF$   INDEPENDENT
        DO K = 1, N
          V(K, I) = V(K, I) * XNORM
        ENDDO
!HPF$   INDEPENDENT, NEW(K, TMP)
        DO J = I+1, N
          TMP = 0.0
!HPF$     INDEPENDENT, REDUCTION(TMP)
          DO K = 1, N
            TMP = TMP + V(K, I)*V(K, J)
          ENDDO
!HPF$     INDEPENDENT
          DO K = 1, N
            V(K, J) = V(K, J) - TMP*V(K, I)
          ENDDO
        ENDDO
      ENDDO
```

FIG. 2.3. *Second HPF version of Modified Gram-Schmidt algorithm with a two-dimensional distribution*

characterized by its *origin*, and two vectors, the *meshsize* and the *extent*, which respectively specify the size of each cell and the number of cells in each dimension. Different grids in a collection may have different mesh sizes and different extents.

We will deal in some detail with grid collections of two different types, multiblock grid collections in Section 3.1, and AMR (adaptive mesh refinement) grids in Section 3.2. In Section 3.3, we discuss a range of HPF-based approaches for both types of grids.

The framework developed here can also be used for semi-coarsening multigrid algorithms as proposed by Overman and Van Rosendale [17]. Such algorithms operate on a hierarchical grid structure; multiple grids at any level of this hierarchy can be processed in parallel using the distribution strategies outlined in Section 3.3.

3.1. Multiblock Codes. Geometrically complex objects, such as aircraft, cannot be easily modeled using a single structured grid. A uniform mesh with a spatial resolution small enough to resolve the localized features in the solution, is often impractical due to the size of the required mesh and the wasted resources away from the region of interest. This section discusses a class of applications called **block-structured** or **multiblock** codes which operate on a set of interacting structured grids connected in an irregular man-

```
program  PROCESSING A MULTIBLOCK GRID COLLECTION

begin
read_number_of_grids
read_grid_parameters
allocate_and_setup_grids   ! allocate and initialize all grids in G

do while ( not done( G))
     boundary_update( G)
     for every  g ∈  G do
          solve_grid(g)
     end for
end do while
end program
```

FIG. 3.1. *Pseudocode for processing a multiblock grid collection*

ner [21]. Using multiple grids to discretize the domain, allows the individual grids in the collection to be tailored. Thus, fine grids can be used in areas of greater interest near the body while coarse grids, requiring less computation, can be used in the far field regions. Multiblock applications, used in grand-challenge applications such as computational fluid mechanics, aircraft simulation, galaxy formation, large-scale climate modeling, and computational combustion dynamics, can be characterized as follows:

- The data domain is partitioned into subdomains that are called *blocks*. Blocks are structured grids representing a self-contained region for computation that can, except for boundary conditions, be operated on independently of the other blocks.
- The number of blocks is relatively small (usually between 10 and 100) and may not be known until runtime. In general, the sizes of blocks are determined at runtime, and different blocks may have widely different sizes and shapes.
- The processing of individual blocks uses regular data access schemes. The functions applied to different blocks may be different. We assume that the amount of processing to be done for each block is proportional to the size of the block.
- Blocks need to interact. The interaction pattern is in general determined at runtime.

3.1.1. Processing of Multiblock Grid Collections. All grids in a multiblock grid collection can be processed independently. As a consequence, a decentralized approach can be taken to determine a solution for the multiblock problem: the equation is not solved over the whole domain, but for each grid separately and in parallel to the solution for the other grids. Boundary updates between grids that abut each other handle the information flow between grids.

We will define a generic algorithm which exploits the level of parallelism across the component grids of a multiblock grid collection, G, as well as the parallelism inherent in solvers for the individual grids. An abstract pseudocode version for such an algorithm is given in Figure 3.1.

We assume a dynamic scenario in which the number of grids in the collection as well as the parameters of the individual grids (i.e., their origins, mesh sizes, and extents) are determined at runtime. The algorithm

reads these parameters and allocates and initializes the individual grids in the collection as well as the data structures required to represent the topology of the collection and its boundaries. In this scenario, once the grid collection is set up it remains invariant.

The core of the algorithm consists of a do-while loop, which is executed until a termination condition is satisfied. Such a condition could either depend on the properties of the solution such as its precision, or could just count the number of iterations performed up to a pre-defined limit. This loop is inherently sequential; each iteration begins with a boundary update phase, in which the boundary information between abutting grids is exchanged, followed by a call to the solver for each component grid in the collection G.

A key assumption we make here is that the **for every** loop is parallel, so that *solve_grid(g)* can be executed independently for all grids in G. Since each g is a structured grid, any method for solving such a grid can be used here. As a consequence, the algorithm exploits two levels of parallelism: the inter-grid parallelism expressed by the **for every** loop, and the intra-grid parallelism of the *solve_grid* routine. During pre-processing, the boundary of each grid is updated. This involves an explicit assignment of solutions. For an internal boundary, solution vectors from neighboring grids are transferred. External boundaries are defined using local knowledge about the boundary of the global domain. The details of this approach depend on the specifics of the algorithm and the structure of the grid collection.

3.2. Adaptive Mesh Refinement (AMR). Adaptive mesh refinement techniques are useful for reducing the computational resources required for solving a system of hyperbolic PDEs. As in the case of multiblock codes, a collection of grids is used to discretize the flow-field. The adaptive mesh refinement algorithm, introduced by Berger and Oliger [6], starts with a structured coarse mesh and adaptively places a finer grid on regions which require a finer resolution. This is continued recursively giving rise to a hierarchy of levels with multiple grids at each level. The computation then consists of using standard finite-difference techniques to approximate the solution on each grid with interpolation and projection operators being used to transfer data between grids at different levels of the hierarchy. Since these codes focus on time-dependent phenomena, such as tracking a shock, the hierarchy of grids are modified and reconstructed dynamically to match the underlying changing phenomena.

Similar to the multiblock codes of the last subection, these algorithms exhibit a fair degree of parallelism since the grids are resolved independently and hence the solutions on all the grids at a level can be computed simultaneously. Also, if the grids are large enough, parallelism can be exploited to speed up the computation within each grid. Exploiting such parallelism adds to the overall complexity of the code. As indicated before, the issue is that even though the grids themselves are structured, the hierarchy of grids is irregular leading to irregular patterns of communication. Also, since the grid hierarchy is dynamic here, in order to effectively parallelize these codes, not only do the grids have to be dynamically distributed so as to maximize the parallelism, but also the irregular inter-grid communication patterns have to be generated each time the grid hierarchy is modified.

The SAMR algorithm. The structured adaptive mesh algorithm can be described at an abstract level as follows. The algorithm starts with a structured coarse mesh representing a discretization of the physical domain under consideration and places finer grids over regions which need better resolution. This is continued recursively, as depicted by the recursive routine *amr* in Figure 3.2. Here, G^l represents the set of grids at level l while G represents the union of all the grids across all levels. Thus, at each level, first the solution on each of the grids at the level is computed. Then, the decision to regrid is made based on the error estimates. If there exists a finer level *l+1*, then the grids on the finer level are initialized by interpolating values from the coarser level l and the routine *amr* is recursively executed on the finer level.

```
amr( G, l)
  do i = 1, r^l
    for every g ∈ G^l do
      solve_grid (g)              ! solve for every grid g at level l
    end for
    if ( regridding required )
      adapt_grids( G, l)
    endif
    if exists level l+1
      interpolate( G, l, l+1)     ! initialize level l+1
      amr( G, l+1)                ! call amr recursively for level l+1
      project( G, l+1, l)         ! update values on level l
    endif
  end do
end amr
```

FIG. 3.2. *An abstract representation of the adaptive mesh refinement algorithm.*

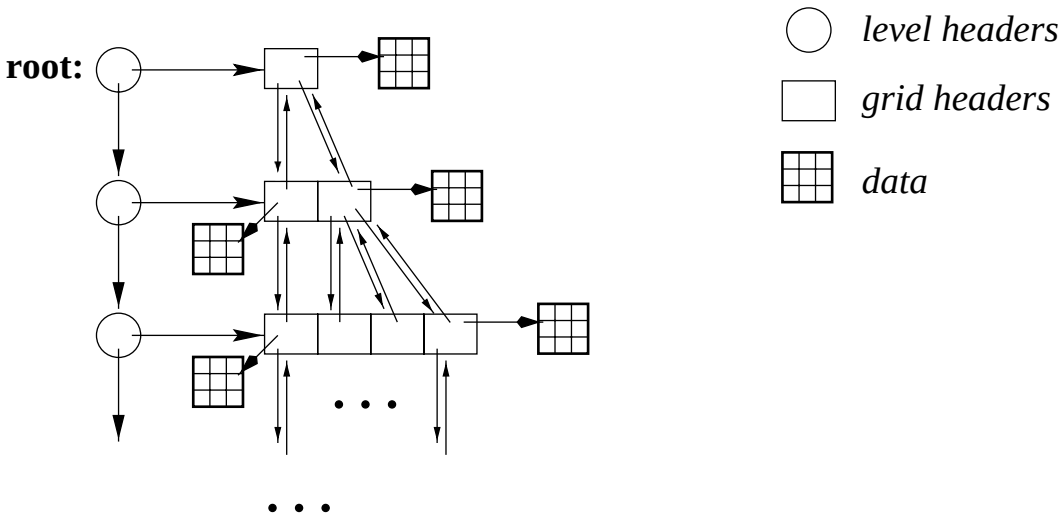

FIG. 3.3. *The grid hierarchy for an AMR algorithm*

Once the solution on the finer level has been computed, it is projected up to update the values at the current level.

As in the case of multiblock codes in the last subsection, the algorithm, as described, exhibits at least two levels of parallelism. First, on any given level, the computation on each of the grids at the level can be executed independently and in parallel. Second, the computation internal to each grid exhibits the typical loosely synchronous data parallelism of structured finite-difference 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, from each level in the hierarchy, should be roughly the same, independent of the number of grids and their shapes and sizes.

In Figure 3.3, we show a picture of the data structures required to maintain a grid hierarchy. This structure has been designed keeping in view the potential parallelism in the algorithm. In the next subsection, we explore how the HPF directives can be used to control the locality of such a collection of grids.

3.3. Processing of Grid Collections in HPF. In this section we study the application of HPF to grid collections. We focus on the algorithm introduced in the context of multiblock problems (Figure 3.1), which in fact provides a generic framework for dealing with arbitrary grid collections. Thus, the ideas described here are also applicable to the AMR codes except that in the latter case the grid hierarchy is dynamic and thus grid interactions have to be reconstructed everytime the grid structure changes.

A Fortran 90 version of the algorithm for two-dimensional grids is given in Figure 3.4. We introduce the data structures required for the representation of the grid collection and outline the grid construction as well as the algorithm processing the grid collection. We do not explicitly describe the algorithm used by the *solve_grid* routine or the *boundary_update* routine.

The fact that we operate on a parallel grid collection may suggest a representation of G as a linked list of grids. However, neither Fortran nor HPF provides a construct for expressing the parallel evaluation of all elements of a linked list. As a consequence, we choose to represent a grid collection as a one-dimensional array, each element of which represents an individual grid in the collection. The type of the array elements is specified as a derived type, *GRID_TYPE*, which we describe in a prototypical form. For each class of algorithms, fields may have to be added to this type. In the algorithm of Figure 3.4, *GRID_TYPE* contains the following fields explicitly:

- extent of the grid
- a pointer to an array of grid data

Depending on the particular application, other fields may be required, such as for storing boundary and topology information. However, we are not concerned about such fields here since they are specific to the particular type of algorithm and do not directly affect the parallelism. Also, for the AMR algorithm, this would represent the grids at one level. Additional data structures would be needed to keep track of the different levels and the relationships (parents, children and sibling) between the levels.

The array *GRID_COLL*, whose elements are of type *GRID_TYPE*, represents the grid collection. Since we assume that the number of grids in the collection is determined at runtime, this array must be declared as allocatable. After reading *n_grids* and allocating *GRID_COLL* accordingly, the algorithms reads the extent of each grid, which determines the dimensions of the associated two-dimensional array *data_array* which is allocated dynamically. Following this, the activation of the procedure *set_up* sets up the grid collection for further processing. This includes defining the boundary of each grid and initializing its data.

The remainder of the algorithm follows directly from the pseudocode as given in Figure 3.1.

```
  TYPE GRID_TYPE
    INTEGER t1, t2                          ! extent of grid
    REAL, POINTER :: grid_data(:, :)
      ...
  END TYPE GRID_TYPE

  TYPE(GRID_TYPE), ALLOCATABLE::GRID_COLL(:)  ! GRID_COLL represents G
  READ (*, *) n_grids                     ! read number of grids in the grid collection
  ALLOCATE(GRID_COLL(n_grids))

  DO I = 1, n_grids
     READ(*, *) GRID_COLL(I)%t1, GRID_COLL(I)%t2    !read grid extents
  END DO

  DO I = 1, n_grids
     ! allocate individual grids in the grid collection:
     ALLOCATE(GRID_COLL(I)%grid_data(GRID_COLL(I)%t1+1, GRID_COLL(I)%t2+1))
  END DO

  CALL set_up(GRID_COLL)   ! define boundaries and initialize grid data

  DO WHILE ( .NOT. termination(GRID_COLL))
     CALL boundary_update(GRID_COLL)
     DO I = 1, n_grids
        CALL solve_grid(GRID_COLL(I))
     END DO
  END DO WHILE

SUBROUTINE solve_grid(G)
 TYPE(GRID_TYPE), POINTER :: G
   ...
   DO I = 1, G%t1
      DO J = 1, G%t2
         G%grid_data(I, J) = ...
      END DO
   END DO
   ...
END SUBROUTINE solve_grid
```

FIG. 3.4. *Fortran 90 program for processing a grid collection*

3.3.1. Distributing the Grids Using HPF. Starting with the Fortran 90 version of the algorithm for processing a grid collection, we will now develop parallel versions using HPF. Three variants will be discussed which use different approaches for distributing the grids of the collection, with different consequences for the degree of parallelism in the resulting HPF program.

The three approaches can be characterized as follows:

D1 : distribute the grid collection, mapping each component grid to exactly one processor

D2 : map each component grid to all processors

D3 : map different component grids to disjoint subsets of processors.

The first two distribution strategies are likely to be inefficient, particularly on machines with a large number of processors. Both strategies can only exploit one level of parallelism. The third approach permits grids to be individually distributed to a suitably sized subset of the available processors. This allows both levels of parallelism inherent in the algorithm to be exploited while providing the opportunity to balance the workload. 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.

We will now discuss these methods in more detail separately.

Distributing Each Component Grid to Exactly One Processor. In our first approach, we distribute G such that each component grid is mapped to exactly one processor. Note that we do not exclude the possibility that different component grids are mapped to the same processor. That is, a processor owns several grid components. This strategy implies that only the outer level of parallelism in the code – the parallelism across G – can be exploited.

We consider two options for expressing such a distribution in HPF. The simplest way would be to distribute G by block (which is the initial distribution chosen in the algorithm). In this approach, some processors may remain idle; furthermore, the sizes of grids – which may radically differ – are not taken into account which may lead to an unbalanced computational load. In order to have finer control over the load balance, the algorithm in Figure 3.5 uses an **INDIRECT** distribution. Such a distribution is controlled by a *mapping array*, *MAP*, which is of the same size as *GRID_COLL* and can be used to explicitly control the distribution of *GRID_COLL*. This is done in such a way that for each element, i, in the index domain of *GRID_COLL*, the index of the associated processor is placed into *MAP(i)*. The mapping array will in general be defined dynamically, depending on data determined at runtime. Here, the *COMPUTE_MAPPING* routine is called to determine a suitable mapping and to initialize *MAP* appropriately. The **REDISTRIBUTE** directive is then used to remap *GRID_COLL* using the computed mapping array. Once the array is remapped, the individual grids can be allocated. Note that the *GRID_COLL* has to be declared **DYNAMIC** (with an initial block distribution) in order to allow its final distribution to determined at runtime.

We assume here that the number of grids in the collection is relatively small; therefore *MAP* is not distributed but would be replicated across all the processors.

As mentioned above, the distribution strategy discussed here can only exploit the parallelism across the set of component grids. This can be expressed in HPF by declaring the loop iterating over the grids of the collection as parallel using the **INDEPENDENT** directive. 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

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
    TYPE GRID_TYPE
      INTEGER t1, t2                          ! extent of grid
      REAL, POINTER :: grid_data(:, :)
        ...
    END TYPE GRID_TYPE

    TYPE(GRID_TYPE), ALLOCATABLE::GRID_COLL(:)  ! GRID_COLL represents G
!HPF$ DYNAMIC, DISTRIBUTE(BLOCK) :: GRID_COLL
    READ (*, *) n_grids                       ! read number of grids in the grid collection
    ALLOCATE(GRID_COLL(n_grids))
    CALL COMPUTE_MAPPING (GRID_COLL, MAP)       ! define MAP
    DO I = 1, n_grids
       READ(*, *) GRID_COLL(I)%t1, GRID_COLL(I)%t2    !read grid extents
    END DO

    DO I = 1, n_grids
       ALLOCATE(GRID_COLL(I)%grid_data(GRID_COLL(I)%t1+1, GRID_COLL(I)%t2+1))
    END DO

    CALL set_up(GRID_COLL)   ! define boundaries and initialize grid data

    DO WHILE ( .NOT. termination(GRID_COLL))
       CALL boundary_update(GRID_COLL)
!HPF$   INDEPENDENT
       DO I = 1, n_grids
!HPF$        ON (HOME(GRID_COLL(I))), RESIDENT
            CALL solve_grid(GRID_COLL(I))
       END DO
    END DO WHILE

  SUBROUTINE solve_grid(G)
    TYPE(GRID_TYPE), POINTER :: G
      ...
      DO I = 1, G%t1
         DO J = 1, G%t2
            G%grid_data(I, J) = ...
         END DO
      END DO
      ...
  END SUBROUTINE solve_grid
```

FIG. 3.5. *Grid Collection Processing: First HPF version*

within the routine[3]. The code within the *solve_grid* routine can be set up such that it does not access any global data, however the compiler cannot determine this without aggressive (and expensive) interprocedural analysis. This can be avoided by using the declarations shown in Figure 3.5. The **ON** directive indicates that the call to *solve_grid* is to be executed only on the processor owning grid *GRID_COLL(I)*. 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 *solve_grid* routine) can be executed in parallel, without communication. Thus, all component grids of the grid family are processed in parallel, however, each individual execution of *solve_grid* is strictly sequential. All communication occurs only in the *boundary_update* routine when two grids which abut each other (and thus have to exchange boundary information) are mapped to different processors.

Along with only exploiting the outer level of parallelism, this approach has several other drawbacks. In many applications, the number of grids in a collection 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.

Distributing Each Component Grid to All Processors. Our second strategy does not distribute the array *GRID_COLL* as above, but maps each individual grid separately. That is, rather than constructing a single distribution which maps each grid as a whole to exactly one processor – we independently distribute the data arrays of each individual grid.

The HPF version of this code is given in Figure 3.6. The mapping is expressed by declaring the pointer, *grid_data*, in the derived type *GRID_TYPE* to be distributed by (* , **BLOCK**) **ONTO** P, where P is the set of all processors available to the program. The array *GRID_COLL* is not distributed, resulting in its replication across all processors.

This approach exploits the parallelism *within* each grid, but not the parallelism across the grids of a collection. Each processor may own a part of each grid, leading to a more even workload; however, some of the grids may not be large enough to effectively exploit all the processors in the system.

The parallelism in the code is made explicit by using the **INDEPENDENT** directive to declare both levels of the nested loop in the solver routine to be parallel.

Note that the loop which calls *solve_grid* is executed sequentially by all processors, and all processors simultaneously call the solver routine on each grid. Here, the communication required for *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 boundary update routine is more complicated here since the actual pattern of data to be transferred between neighboring grids is not known until runtime.

Distributing Each Component Grid to a Subset of Processors. Given the drawbacks of the previous two approaches, a more optimal approach is to map each grid of the collection separately to a suitably sized contiguous subset of processors; different grids are mapped to disjoint subsets. This allows both levels of parallelism in the algorithm to be exploited while providing the opportunity to balance the workload.

[3]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.

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
    TYPE GRID_TYPE
      INTEGER t1, t2                          ! extent of grid
      REAL, POINTER :: grid_data(:, :)
!HPF$ DISTRIBUTE grid_data(*, BLOCK) ONTO P
        ...
    END TYPE GRID_TYPE

    TYPE(GRID_TYPE), ALLOCATABLE::GRID_COLL(:)  ! GRID_COLL represents G
    READ (*, *) n_grids                         ! read number of grids in the grid collection
    ALLOCATE(GRID_COLL(n_grids))
    DO I = 1, n_grids
       READ(*, *) GRID_COLL(I)%t1, GRID_COLL(I)%t2    !read grid extents
    END DO

    DO I = 1, n_grids
       ! allocate individual grids according to statically specified (*, BLOCK) distribution
       ALLOCATE(GRID_COLL(I)%grid_data(GRID_COLL(I)%t1+1, GRID_COLL(I)%t2+1))
    END DO

    CALL set_up(GRID_COLL)   ! define boundaries and initialize grid data

    DO WHILE ( .NOT. termination(GRID_COLL))
       CALL boundary_update(GRID_COLL)
       DO I = 1, n_grids
          CALL solve_grid(GRID_COLL(I))
       END DO
    END DO WHILE

  SUBROUTINE solve_grid(G)
    TYPE(GRID_TYPE), POINTER :: G
      ...
!HPF$ INDEPENDENT, NEW J
      DO I = 1, G%t1
!HPF$     INDEPENDENT
         DO J = 1, G%t2
            G%grid_data(I, J) = ...
         END DO
      END DO
      ...
  END SUBROUTINE solve_grid
```

FIG. 3.6. *Grid Collection Processing: Second HPF version*

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
    TYPE GRID_TYPE
      INTEGER t1, t2                          ! extent of grid
      INTEGER lo, hi      ! lower and upper bounds for the target processor subset
      REAL, POINTER :: grid_data(:, :)
!HPF$  DYNAMIC grid_data
        ...
    END TYPE GRID_TYPE

    TYPE(GRID_TYPE), ALLOCATABLE::GRID_COLL(:)  ! GRID_COLL represents G
    READ (*, *) n_grids                         ! read number of grids in the grid collection
    ALLOCATE(GRID_COLL(n_grids))
    DO I = 1, n_grids
       READ(*, *) GRID_COLL(I)%t1, GRID_COLL(I)%t2    !read grid extents
    END DO
    CALL COMPUTE_PROCS_SUBSET (GRID_COLL)  ! compute processor subset (lo, hi) for each grid
    DO I = 1, n_grids
       ALLOCATE(GRID_COLL(I)%grid_data(GRID_COLL(I)%t1+1, GRID_COLL(I)%t2+1))
!HPF$   REDISTRIBUTE G(*, BLOCK) ONTO P(G%lo:G%hi)
    END DO
    CALL set_up(GRID_COLL)   ! define boundaries and initialize grid data
    DO WHILE ( .NOT. termination(GRID_COLL))
       CALL boundary_update(GRID_COLL)
!HPF$   INDEPENDENT
       DO I=1, l%ngrids
!HPF$      ON (HOME (GRID_COLL(I))), RESIDENT
          CALL solve_grid(GRID_COLL(I))
       END DO
    END DO WHILE

  SUBROUTINE solve_grid(G)
    TYPE(GRID_TYPE), POINTER :: G
      ...
!HPF$ INDEPENDENT, NEW J
      DO I = 1, G%t1
!HPF$    INDEPENDENT
         DO J = 1, G%t2
            G%grid_data(I, J) = ...
         END DO
      END DO
```

FIG. 3.7. *Grid Collection Processing: Third HPF version*

The HPF program embodying this approach, is shown in Figure 3.7. The array *GRID_COLL* is not distributed, while the pointer *grid_data* in the derived type *GRID_TYPE* is declared dynamic. After determining the extent of each grid, the routine *COMPUTE_PROCS_SUBSET* is called to compute the subset of processors to which each grid should be mapped, storing the indices of the lower and upper bound of the processor subset in the components *lo* and *hi*. These bounds are then used to distribute the data array associated with each grid at the time of allocation.

Both loops – the one iterating over the component grids and the nested loop in *solve_grid* – have to be declared parallel. Note that one still requires the **ON** and **RESIDENT** directives on the first loop, otherwise the calls to the solver routines would be sequentialized.

The mapping of the grids in the collection here is controlled by the user through the routine *COMPUTE_PROCS_SUBSET*. When mapped properly, multiple grids can be processed in parallel by subsets of processors. This allows the parallelism to be exploited at both levels while having a much better control on the overall load balance of the computation. The communication required for this strategy is similar to that for the second strategy.

Before closing the discussion on multiblock codes, we briefly describe the compiler analysis required to generate the communication required for the different distribution strategies. Since each of the grids in the collection is block-structured, the compiler can easily analyze the *solve_grid* routine and insert the required communication statements. Note that not knowing the target subset of processors for a grid in the third strategy is similar to not knowing the complete set of processors at compile time and just requires the compiler to generate message passing statements parameterized by the lower and upper bounds of the target processor subset.

The situation is a little more complicated for the communication required for the boundary update routine. The issue here is that not only that the distribution is known only at runtime but also the actual boundary portions that abut each other is dependent on the grid structure, i.e., is data dependent, and hence is also not known at compile time. Thus, the compiler needs to generate code which will at runtime determine the required communications based on the portions of the distributed arrays to be exchanged. The computation is generally quite simple and experience with such applications has shown that the generated codes achieve reasonable performance [1, 19].

4. Unstructured Grid Applications. Unstructured grid codes provide several advantages in modeling the flow over complex geometries. In particular, they provide added flexibility in generating and adapting meshes for complex configurations. However, such codes generally require more computational resources and are more difficult to parallelize.

Unstructured grid flow solvers generally use a finite element approach to spatially discretize the domain using piecewise linear flux functions over each individual triangle in 2D or tetrahedra in 3D. One approach is to use a compact vertex based scheme, with an edge based data structure. The flow variables are stored at each vertex in the mesh while the residuals are constructed by looping over edges that define the connectivity of the vertices.

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 and access to values inherently requires using indirection via index arrays. This complicates the parallelization of such

codes since the pattern of data accesses are now dependent on the data values, i.e., the grid connectivity, and hence are known only at runtime and cannot be analyzed at compile time.

Another consequence of the structure of the underlying data structures in such codes is that simple data distributions strategies, such as block and cyclic do not work. In fact, the partitioning of such grids for parallel execution is a complex problem. There exist several grid partitioning packages which attempt to subdivide the grid into contiguous, mutually disjoint regions to be mapped onto the processors of the underlying parallel machine. The overall criterion for partitioning is the minimization of the total execution time, which depends on many parameters, including the degree of parallelism in the algorithm, the amount of communication which would be generated by the partition, the amount of processing at each node, and the overall load balance. The issue in this paper is not how we partition the mesh but how the generated partitioning is represented in a generic manner using HPF directives.

Consider an abstraction of a two-dimensional unstructured mesh Euler solver in which the mesh is represented by triangles and the flow variables are stored at the vertices of the mesh. Figure 4.1 shows one way in which this computation may be specified. The mesh is represented by the array $GRID$ of $NODE$s each of which represents a vertex. Along with other fields such as the x-y coordinates (not shown here), the derived type for each node also contains the flow variables represented by $V1$ and $V2$. The connectivity of the overall mesh is represented by the array $EDGE$ such that $EDGE(I,1)$ and $EDGE(I,2)$ are the node numbers at the two ends of the Ith edge.

We reproduce only the main computational kernel of the code, an edge-based residual construction loop which updates the values at the end vertices of each edge based on calculation of flux across the edge. This is represented by the J loop in Figure 4.1 which uses array indirection to extract and store values of the flow variables at the two vertices of each edge.

Since the partitioning of the mesh is to be determined at runtime, the arrays constituting the mesh, $GRID$ and $EDGE$, are declared to be **DYNAMIC**. As indicated above, the irregularity of the vertex numbering implies that the **INDIRECT** distribution is needed to map the vertices to the processors. Thus, the routine *GRID_PARTITION* not only partitions the grid but also returns the mapping array $NODEMAP$ such that the value of its ith element represents the index of the processor on which the ith element of the $GRID$ array is to be mapped.

Once the partitioning of the vertices has been determined, we can also determine the mapping of array representing the edges. Given the structure of the computation, it would be useful to distribute $EDGE$ in such a way that the values at one or both of its nodes are on the same processor. We have chosen to distribute the elements of $EDGE$ to the processor which owns the values for the first of its nodes. We again use the **INDIRECT** distribution for this, assuming that the *GRID_PARTITION* routine will also setup the $EDGEMAP$ array based on the values in the $EDGE$ array.

Note that the mapping arrays are as large as the unstructured mesh itself and hence have to be distributed themselves. This is in contrast to the mapping array used with multiblock codes in the last section which was used to map the grids in a collection and hence was samll and could be replicated across the processors.

The computation is specified using a **INDEPENDENT** loop, with an **ON** clause to specify where each iteration is to be performed. Thus the iterations of the loop, over the edges in this case, can be executed in parallel. In Figure 4.1, the **ON** clause specifies that the Ith iteration should be performed on the processor that owns the $(I,1)$th element of $EDGE$.

The variables $N1$, $N2$ and $DELTAV$ are private variables for each iteration and hence are declared in the NEW clause. Thus assignments to these variables do not cause flow dependences between iterations

```
!HPF$   PROCESSORS P(NUMBER_OF_PROCESSORS())
        INTEGER :: N              ! 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(:, :)
        INTEGER, ALLOCATABLE :: NODEMAP(:), EDGEMAP(:)  ! mapping arrays
!HPF$  DYNAMIC, DISTRIBUTE(BLOCK) :: GRID
!HPF$  DYNAMIC, DISTRIBUTE(BLOCK, *) :: EDGE
!HPF$  DISTRIBUTE(BLOCK) :: NODEMAP, EDGEMAP
          ...

! Read N, M, allocate arrays GRID, EDGE, NODEMAP and EDGEMAP
        ALLOCATE(GRID(N))
        ALLOCATE(NODEMAP(N))
        ALLOCATE(EDGEMAP(N))
        ALLOCATE(EDGE(M, 2))
          ...
! Code for initialization of GRID and EDGE
          ...
! Partition the grid, setting up the mapping arrays
        CALL GRID_PARTITIONER(GRID, NODEMAP, EDGE, EDGEMAP)
! Redistribute the GRID and EDGE arrays based on the values returned by the partitioner
!HPF$ REDISTRIBUTE GRID(INDIRECT(NODEMAP))
!HPF$ REDISTRIBUTE EDGE(INDIRECT(EDGEMAP))
          ...
! Sweep over the edges of the grid
!HPF$ INDEPENDENT, ON HOME(EDGE(J, 1)), NEW(N1, N2, DELTAV), REDUCTION(GRID)
        DO J = 1, M
            N1 = EDGE(J, 1);     N2 = EDGE(J, 2)
            ...
            DELTAV = F(GRID(N1)%V1, GRID(N2)%V1)
            ...
            GRID(N1)%V2 = GRID(N1)%V2 - DELTAV
            GRID(N2)%V2 = GRID(N2)%V2 + DELTAV
        ENDDO
```

FIG. 4.1. *Sweep over edges of an unstructured grid*

of the loop. For each edge, the value of the flow variable $V1$ at the two incident nodes are read and used to compute the flux contribution $DELTAV$ for the edge. This contribution is then accumulated into the residual $V2$ for the two nodes.

Since each vertex has multiple edges incident upon it, multiple iterations will accumulate $V2$ values at each node. Thus, the $GRID$ array is declared as a reduction allowing the compiler to generate correct computation for the accumulations.

In most cases, the two vertices of an edge are in the same partition and hence are mapped to the same processor. However, for cross-partition edges, the two vertices will be mapped to different processors. In this case the values have to be gathered before the loop body and the updates have to be scattered after the loop body. The compiler has to analyze the code and generate the communication required to gather and scatter these values. The problem is that the values of the flow variables for each node are accessed via the edges. Thus a level of indirection is involved in each access. Given that the data distribution of each of the arrays is determined at run time, the compiler cannot detect which references are local and which are not.

One of the techniques used to generate the communication in such situations is called the *inspector/executor* strategy [14, 20]. For each parallel loop, the compiler generates two loops: the first called the *inspector* utilizes the distribution and the edge connectivity to generate the communication schedule; the second, called the *executor*, uses this schedule to gather the node values before the loop execution and scatter the updates after the execution. Note that this confines the communication among the processors to the scatter/gather phase allowing the body of the loop to be executed completely in parallel. The overhead associated with the inspector loop is generally fairly large. However, many of the unstructured codes make several sweeps over the same mesh allowing the cost of the inspector to be amortized across the sweeps [1, 3].

5. Conclusion. HPF is a well-designed language that allows a reasonably efficient and concise formulation of most algorithms used in aerospace applications. HPF programs are much higher level than equivalent algorithms that use explicit message passing primitives such as those offered by MPI or PVM, and are thus easier to develop and less error-prone. On the other hand, HPF cannot in all cases provide the same degree of control over the parallelism of an application as an MPI program can, resulting in a potential performance penalty. Over the past few years, much research in language design, compilers, and runtime systems was devoted to deleting or minimizing this effect, in particular for programs with irregular data and work distributions. Although some problems remain, it has been shown that for many relevant benchmarks HPF can achieve almost the same performance as MPI programs [9, 4, 5].

The data parallel paradigm represented by HPF supports the "loosely synchronous" execution of a set of identical processes working on different segments of the same problem. Some applications, such as multidisciplinary optimization, need a more flexible way to express parallelism. 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.

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. One important approach uses coarse-grain tasks, each comprising an entire HPF program. In effect, HPF is wrapped in a coordination language. Proposals along this line have been made in the Fortran M [10] and Opus languages [7]. Opus encapsulates HPF programs as object-oriented modules, passing data between them by accessing *shared abstractions (SDAs)* which are monitor-like constructs.

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.

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, No. 7 (1995), pp. 747–754.

[2] E. Albert, K. Knobe, J.D. Lukas, and G.L. Steele, Jr., *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), pp. 42–56, New Haven, CT, July 1988.

[3] Benkner, S., *Vienna Fortran 90 and Its Compilation*, Ph.D. Thesis. Technical Report TR 94-8, University of Vienna, Institute of Software Technology and Parallel Systems, November 1994.

[4] S. Benkner, P. Mehrotra, J. Van Rosendale, and H.P. Zima, *High-Level Management of Communication Schedules in HPF-Like Languages*, Proc. International Conference on Supercomputing 1998 (ICS'98), Melbourne, Australia, July 1998.

[5] S. Benkner and H.P. Zima, *Compiling High Performance Fortran for Distributed-Memory Architectures*, to appear in Parallel Computing, Special Anniversary Issue, D. Trystram, ed., 2000.

[6] M.J. Berger and J. Oliger, *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations*, J. Comput. Phys., 53 (1984), pp. 482–512.

[7] B. Chapman, M. Haines, P. Mehrotra, J. Van Rosendale, and H. Zima, *Opus: A coordination language for multidisciplinary applications*, Scientific Programming, 6/4 (1997), pp. 345–362.

[8] B. Chapman, P. Mehrotra, and H. Zima, *Programming in Vienna Fortran*, Scientific Programming, 1, No. 1 (1992), pp. 31–50.

[9] B. Chapman, P. Mehrotra, and H. Zima, *Extending HPF for Advanced Data Parallel Applications*, IEEE Parallel and Distributed Technology, (1994), pp. 59–70.

[10] 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.

[11] 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.

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

[13] S. Hiranandani, K. Kennedy, and C. Tseng, *Compiling Fortran D for MIMD Distributed Memory Machines*, Communications of the ACM, 35, No. 8 (1992), pp. 66–80.

[14] C. Koelbel and P. Mehrotra, *Compiling global name-space parallel loops for distributed execution*, IEEE Transactions on Parallel and Distributed Systems, 2, No. 4 (1991), pp. 440–451.

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

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

[17] A. Overman and John Van Rosendale, *Mapping Robust Parallel Multigrid Algorithms to Scalable Memory Architectures*, ICASE Report No. 93-24, ICASE, NASA Langley Research Center, June 1993.

[18] D. Pase, *MPP Fortran programming model*, in High Performance Fortran Forum, January 1992.

[19] 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.

[20] 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, No. 2 (1990), pp. 303–312.

[21] 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.

[22] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, *Vienna Fortran — a language specification*, ICASE Interim Report No. 21, ICASE, NASA Langley Research Center, Hampton, VA, March 1992.

[23] 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.