High Performance Applications: State-of-the-Art and Future Requirements*

Barbara Chapman, Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria; *Piyush Mehrotra,* ICASE, MS 132C; NASA Langley Research Center, Hampton VA. 23681 USA; and *Hans Zima,* Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria

ABSTRACT: *In the past, most high performance applications dealt with numerical simulation, and parallel machines were essentially used only by a relatively small group of dedicated professionals which had to cope with the idiosyncrasies of the machine at a low level of abstraction. This situation is changing quickly, driven by enhanced hardware and software support offered by a new generation of parallel computers, and the rapid expansion of global networks that lead to the feasibility of applications distributed over geographically widely distant areas.*

In this paper, we discuss the language support required for the efficient handling of advanced applications. We will outline the major features of Vienna Fortran and compare the language to High Performance Fortran (HPF), a de-facto standard in this area. A significant weakness of HPF is its lack of support for many advanced applications, which require irregular data distributions and dynamic load balancing. We introduce HPF+, an extension of HPF based on Vienna Fortran, that provides the required functionality.

1 Introduction

The continued demand for increased computing power has led to the development of **highly parallel scalable multiprocessing systems (HMPs)**, which are now offered by all major vendors and have rapidly gained user acceptance. These machines are relatively inexpensive to build, and are potentially scalable to large numbers of processors. However, they are difficult to program: most of the architectures exhibit non-uniformity of memory access which implies that the locality of algorithms must be exploited in order to achieve high performance, and the management of data becomes of paramount importance.

' The work described in this paper was partially supported by the Austrian Research Foundation (FWF) and by the Austrian Ministry for Science and Research (BMWFK). This research was also supported by the National Aeronautics and Space Administration under NASA Contract No. NASI-18605, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

Traditionally, HMPs have been programmed using a standard sequential programming language (Fortran or C), augmented with message passing constructs. In this paradigm, the user is forced to deal with all aspects of the distribution of data and work to the processors, and to control the program's execution by explicitly inserting message passing operations. The resulting programming style can be compared to assembly language programming for a sequential machine; it has led to slow software development cycles and high costs for software production. Moreover, although MPI is evolving as a standard for message passing, the portability of MPI-based programs is limited since the characteristics of the target architectures may require extensive restructuring from the code.

As a consequence, much research and development activity has been concentrated in recent years on providing higher-level programming paradigms for HMPS. Vienna Fortran, building upon the KALL programming language [8] and experiences from the SUPERB parallelization system [12], was the first fully specified data-parallel language for HMPS. It provides language features for the high-level specification of data distribution and alignment, as well as explicitly parallel loops. High Performance Fortran (HPF) [7], a de-facto standard developed by a consortium including participants from industry, academia and research laboratories, is based on concepts of CM Fortran [11], Vienna Fortran, and Fortran D [6]. It provides support for regular applications, alleviating the task of the programmer for a certain segment of applications. However, it is generally agreed that the current version of the language, *HPF-1*, is not adequate to handle many advanced applications, such as multiblock codes, unstructured meshes, adaptive grid codes, or sparse matrix computations, without incurring significant overheads with respect to memory or execution time. This fact has been acknowledged by the HPF Forum in its decision to start the development of HPF-2 at the beginning of 1995.

In this paper, we outline the major features of Vienna Fortran and compare the language to *HPF-1*. We then identify some of the weaknesses of HPF-1 by considering language requirements posed by irregular algorithms and dynamic load balancing. This study leads to the description of an HPF extension, "**HPF+**", which, based upon Vienna Fortran, solves many of these problems by introducing proper extensions, and thus contributes to the present effort of the HPF Forum for defining a suitable successor to *HPF-1*.

2 Vienna Fortran: A Short Overview

Vienna Fortran is based on the *SPMD* (Single Program Multiple Data) or data parallel model of computation. With this method, the data arrays in the original program are each partitioned and mapped to the processors. This is known as *distributing* the arrays. The specification of the mapping of the elements of the arrays to the set of processors is called the *data distribution* of that program. A processor is then thought of as *owning* the data assigned to it; these data elements are stored in its local memory. Now the work is distributed, in general according to the data distribution: computations which define the data elements owned by a processor are performed by it - this is known as the *owner computes* paradigm. The processors then execute essentially the same code in parallel, each on the data stored locally.

The compiler analyzes the source code, translating global data references into local and non-local data references based on the distributions specified by the user. The non-local references are satisfied by inserting appropriate message-passing statements in the generated code. Finally, the communication is optimized where possible, in particular by combining messages and by sending data at the earliest possible point in time.

A major characteristic of this style of programming is that the performance of the resulting code depends to a very large extent on the data distribution selected by the programmer. The data distribution determines not only where computation will take place, it is also the main factor in deciding what communication is necessary. If, in a given scope, the association between an array and a distribution is invariant, we speak of a statically, otherwise of a dynamically distributed array.

'The Vienna Fortran language extensions provide the user with the following features:

- The **processors** which execute the program may be explicitly specified and referred to. It is possible to impose one or more structures, or *views*, upon them.
- The **distributions** of arrays can be specified using annotations. These annotations may use processor references related to processor structures introduced by the user.
  - Intrinsic functions are provided to specify the most common distributions.
  - Distributions may be defined indirectly via a map array.
  - Data may be replicated to all or a subset of processors.
  - The user may define new distribution functions.
- An array may be **aligned** with another array, providing an implicit distribution. Alignment functions may also be defined by the user.
- The distribution of arrays may be changed **dynamically.** However, a. clear distinction is made between arrays which are statically distributed and those whose distribution may be changed at runtime.
- In **procedures**, formal array parameters may
  - inherit the distribution of the actual argument, or
  - be explicitly distributed, possibly causing some data motion.
- A **forall** loop permits explicitly parallel loops to be written. Intrinsic reduction operations are provided, and others may be defined by the user. Loop iterations may be executed
  - on a specified processor,
  - where a particular data object is stored, or
  - as determined by the compiler.
- Arrays in **common blocks** may be distributed.
- **Allocatable** arrays may be used in much the same way as in Fortran 90. Array sections are permitted as actual arguments to procedures.

Vienna Fortran does not introduce a large number of new constructs, but those it does have are supplemented by a number of options and intrinsic functions, each of which serves a specific purpose. They enable the user to exert additional control over the manner in which data is mapped or moved, or the code is executed. A full specification of the language is given in [13]; its use for solving a range of typical application problems is described in [1].

3 A Comparison of Vienna Fortran and HPF

The main concepts in HPF have been derived from a number of predecessor languages, including mainly CM Fortran [11], Kali [8], Fortran D [6], and Vienna Fortran, with the last two languages having the largest impact.

The basic elements of HPF's language model are - similarly to Vienna Fortran - *abstract processors*, *distributions*, and *alignments*. In addition, HPF has introduced the concept of a template, which is essentially a named index domain that can be used as an alignment base. The implications of this construct which significantly complicates the underlying semantic model are discussed in [2].

HPF follows Vienna Fortran closely in a number of features. This includes in particular

- abstract processor arrays
- direct distribution and alignment of arrays
- distinction between static and dynamic distributions
- definition of the procedure interface, in particular inherited and enforced distributions
- FORALL loops (called **INDEPENDENT** loops in HPF)

On the other hand, a number of advanced concepts of Vienna Fortran have not been included in HPF. Among them are

- different *processor views*
- distribution of arrays to *processor sections*
- GENERAL-BLOCK distributions
- INDIRECT distributions
- user-defined distribution functions

These omissions, in particular the absence of language features for the formulation of more general distribution functions, significantly impairs the applicability of HPF to advanced algorithms using, for example, irregular or adaptive grids.

4 HPF+

The above discussion has indicated that the use of HPF-1 for advanced applications leads to problems related to expressivity and performance. In this section we discuss extensions to HPF-1 that are needed to address these problems. The discussion informally introduces an HPF extension, called "**HPF+**", using an ad-hoc HPF-like syntax. The extensions to HPF-1 incorporated into HPF+ are of two sorts: first, a generalization of the data distribution mechanisms, and, secondly, control facilities providing coarse-grain task parallelism integrated with the data-parallel HPF computation model. They will be discussed in individual subsections below.

4.1 Distribution to Processor Subsets and Subobject Distribution

The HPF-1 **DISTRIBUTE** directive specifies the distribution of data to a processor array which has been declared by the user. It does not permit distribution to a part of the processor array. Also, HPF-1 allows only the distribution of top-level objects - components of a derived type cannot be distributed. Here, we show that multiblock problems need both features, and describe simple extensions to HPF-1 which provide these functionalities. This will be illustrated by a program skeleton creating a corresponding grid structure.

Scientific and engineering codes from diverse application areas may use multiple grids to model the underlying problem domain. These grids may be structured, unstructured or a mixture of both types, individually chosen to match the underlying physical structure and allocate the computational resources efficiently with high node densities in selected areas. A typical application may use anywhere from 10 to 100 grids of widely varying sizes and shapes. Each sweep over the domain involves computation on the individual grids before data is exchanged between them. Thus, these types of applications exhibit at least two levels of parallelism. At the outer level, there is coarse grain parallelism, since the computation can be performed on each grid simultaneously. The internal computation on each grid, on the other hand, exhibits the typical loosely synchronous data parallelism of structured grid codes.

Distributing the array of grids to the processors so that each grid is mapped to exactly one processor offers limited parallelism since it only exploits the outer level, the number of grids may be too small to allow the use of all processors, and the grids may vary significantly in size resulting in an uneven workload. Another strategy is to distribute each grid independently to all processors, enabling the parallelism within a grid to be exploited. This will lead to a more even workload; however, the grids may not all be large enough for this to be a reasonable solution.

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

HPF-1 does not, however, permit data arrays to be distributed directly to subsets of processors. In HPF-1 this can be expressed indirectly by using templates and alignment, but these solutions are difficult to achieve and will generally require a priori precise knowledge of the size of both the grid and the processor array, and must be reimplemented for each modification of the problem. A simpler solution is to adopt a direct approach, which permits a processor subsection to be the target of a distribution. An example of this is shown in Figure 1.

Consider the case of a multiblock problem where the number of grids and their sizes are not known until runtime. Each grid can be declared as a pointer within a derived type and the set of all grids can be an allocatable array, where each element is a grid. HPF-1 allows us to distribute the array of grids to the processors. However, we may not distribute the individual grids across processors, since these are subobjects and their distribution is explicitly prohibited. This is necessary for exploiting the parallelism present within the individual grids. The algorithm in Figure 1 illustrates a solution for this problem, by providing a notation for the distribution of subobjects. We assume that at most one level in a nested structure can be distributed in this way.

```
!HPF$ PROCESSORS R(NPR)
TYPE subgrid
   INTEGER xsize, ysize
   REAL , DIMENSION (:,:), POINTER ::grid
!HPF+$  DYNAMIC ::grid
        ...
END subgrid

TYPE (subgrid), DIMENSION (:),  ALLOCATABLE :: grid_structure
            ...
READ (*,*) no_of_grids
ALLOCATE (grid_structure(no_of_grids))
DO i = 1, no_of_grids
   READ (*,*) ix, iy
   grid_structure(i)%xsize = ix
   grid_structure(i)%ysize = iy
ENDDO
! Compute processor array section for each grid: low(i), high(i)
DO i = 1, no_of_grids
ALLOCATE (grid_structure(i)%grid( grid_structure(i)%xsize,grid_structure(i)%ysize))
!HPF+$ REDISTRIBUTE grid_structure(i)%grid (BLOCK,*) ONTO R(low(i):high(i))
ENDDO
```

Figure 1: Creation of the grid structure for a multiblock code

4.2 *General Block Distributions*

Dimensions of data arrays or templates can be mapped in HPF-1 by specifying either block or cyclic distributions. There are a number of problems for which these regular mappings do not result in an adequate balance of the workload across the processors of the target machine, but which can be handled by general block distributions, a relatively simple extension of HPF-l's regular block distributions.

General block distributions were initially implemented in SUPERB and Vienna Fortran. They are similar to the regular block distributions of HPF-1 in that the index domain of an array dimension is partitioned into contiguous blocks which are mapped to the processors; however, the blocks are not required to be of the same size. Thus, general block distributions provide more generality than regular blocks while retaining the contiguity property, which plays an important role in achieving target code efficiency.

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

The following code fragment illustrates an array *A* whose rows are distributed in blocks of sizes 400,400,200,100,100,100,500, and 800.

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

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

Arrays distributed in this way can also be efficiently managed at runtime, allowing the use of the *overlap* [15] concept to optimize communication related to regular accesses. Finally, codes can be easily parameterized with such distributions: for example, a procedure with a transcriptive formal argument[1] that is supplied with differently distributed actual arguments can be efficiently compiled if the representation of the argument's distribution is passed along as a set of additional implicit arguments created by the compiler.

4.3 Irregular Distributions

General block distributions provide enough flexibility to meet the demands of some irregular computations: if, for instance, the nodes of a simple unstructured mesh are partitioned prior to execution and then appropriately renumbered, the resulting distribution can be described in this manner. This renumbering process is similar to domain decomposition and can be a complex and computationally demanding task. However, this approach is not appropriate for all irregular problems. A general block distribution, even with two or three dimensions, may not be able to provide an equal workload per processor. Also, block distributions are always constrained by the adjacency of data. The single workspaces typically used in Fortran programs cannot necessarily be renumbered in such a way as to produce a sequential group of regions. Irregular distributions offer the ability to express totally unstructured or irregular data structures but at some cost in terms of the code the compiler must generate.

We will here introduce two different mechanisms to handle general data distributions. We begin with *indirect distribution* functions, which allow the specification of a distribution via a mapping array and continue with *user-defined distribution functions.*

4.3.1 Indirect Distributions

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

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

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

The example in Figure 2 illustrates the use of indirect distributions in the context of a sweep over an unstructured mesh.

Indirectly distributed arrays must be supported by a runtime system which manages the internal representation of the mapping array and handles accesses to the indirectly distributed array. The mapping array is used to construct a *translation table,* recording the owner of each datum and its local index. Note that this representation has **O**(*N*) elements, on the same order as the size of the array; however, most codes require only a very small number of distinct indirect mappings. The PARTI routines developed by J. Saltz and collaborators [91 represent a runtime library which directly supports indirect distribution functions, in connection with irregular array accesses.

4.3.2 User-Defined Distribution Functions

Indirect distribution functions incur a considerable overhead both at compile time and at runtime. A difficulty with this approach is that when a distribution is described by means of a mapping array, any regularity or structure that may have existed in the distribution is lost. Thus the compiler cannot optimize the code based on this complex but possibly regular distribution. **User-defined distribution functions (UDDFS)** provide a facility for extending the set of intrinsic mappings defined in the language in a structured way. The specification of a UDDF establishes a mapping from (data) arrays to processor arrays, using a syntax similar to Fortran functions. UDDFs have two implicit formal arguments, representing the data array to be distributed and the processor array to which the distribution is targeted. Specification statements for these arguments can be given using the keywords **TARGET_ARRAY** and **PROCESSOR_ARRAY,** respectively. UDDFs may contain local data structures and executable statements along with at least one distribution mapping statement which maps the elements of the target array to the processors.

UDDFs constitute the most general mechanism for specifying distributions: any arbitrary mapping between array indices and processors can be expressed, including partial or total replication. We illustrate their use by an example, representing indirect distributions. For simplicity we assume here that *A* and *MAP* have the same shape.

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

1 If such an argument is passed by reference, the distribution is left intact, and thus no movement of data will be necessary.

```
!HPF+$  Do I=1,SIZE(A)
!HPF+$     A(l) DISTRIBUTE TO R(MAP(I))
!HPF+$ ENDDO
!HPF+$ END DFUNCTION INDIRECT
```

4.4 *Extensions of the INDEPENDENT Loop Concept*

Whenever a do loop contains an assignment to an array involving an indirect access, the compiler will not be able to determine whether the iterations of the loop may be executed in parallel. Since such loops are common in irregular problems, and may contain the bulk of the computation, the user must assert the independence of its iterations.

For this purpose, HPF-1 provides the **INDEPENDENT** directive, which asserts that a subsequent do loop does not contain any loop-carried dependencies, allowing the loop iterations to be executed in parallel. A **NEW** clause introduces private variables that are conceptually local in each iteration, and therefore cannot cause loop-carried dependencies.

There are two problems with this feature:

- There is no language support to specify the **work distribution** for the loop, i.e., the mapping of iterations to processors. This decision is left to the compiler/runtime system.
- **Reductions,** which perform global operations across a set of iterations, and assign the result to a scalar variable, violate the restriction on dependencies and cannot be used in the loop.[2]

The first problem can be solved by extending the **INDEPENDENT** directive with an **ON** clause that specifies the mapping, either by naming a processor explicitly or referring to the *owner* of an element. For example, in Figure 2 **INDEPENDENT** is used for the *I* loop over edges so that the loop is executed on the processor that owns the array element *EDGE(I,*1*).*

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

In Figure 2, many proposed features of HPF+ are illustrated for a simple unstructured mesh code. The mesh for this code consists of triangles; values for the flow variables are stored at their vertices. The computation is implemented as a loop over the edges: the contribution of each edge is subtracted from the value at one node and added to the value at the other node. The mesh is represented by the array *EDGE*, where *EDGE(I,* 1*)* and *EDGE(I,* 2) are the node numbers at the two ends of the *I*-th edge. The arrays *X* and *Y* represent the flow variables, which associate a value with each of the *NNODE* nodes.

Consider the distribution of the data across the one-dimensional array of processors, *R*(*M*). The array *X* is declared to be dynamically distributed with an initial block distribution. At runtime this array is distributed indirectly, as defined in the mapping array *MAP* obtained from the user-specified routine *PARTITIONER. Y* is also declared with the keyword **DYNAMIC** and is aligned to *X*. Whenever *X* is redistributed, *Y* is automatically redistributed with exactly the same distribution function.

Since the elements of *EDGE* are pointers to flow variables—in iteration *I*, *X(EDGE(I,* 1*))*, *X(EDGE(I,*2*))* and the corresponding components of *X* and *Y* are accessed—we relate the distribution of *EDGE* to the distribution of *X* and *Y* in such a way that *EDGE(I,:)* is mapped to the same processor as *X(EDGE(I,1)).*

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

The computation is specified using an extended **INDEPENDENT** loop. The work distribution is specified by the **ON** clause: the *I*-th iteration is to be performed on the processor that owns *EDGE(I,* 1*).* The variables *N*1, *N*2 and *DELTAX* are private, so, conceptually, each iteration is allocated a private copy of each of them. Hence assignments to these variables do not cause loop-carried dependencies [14].

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

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

4.5 *Data Distribution and Alignment —Other Issues*

There are a number of other issues with the specification of data distribution and alignment in HPF-1 which we have not discussed here. These include processor views, control of dynamic data distributions, library interfaces, and the procedure boundary; they are discussed in [4].

4.6 *Integration of Task With Data Parallelism*

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

2 Note however that HPF-1 and Fortran 90 provide intrinsics for some important reductions.

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

!HPF+$ INDEPENDENT, ON OWNER (EDGE(I,1)), NEW (N1, N2, DELTAX)

DO I = 1, NEDGE
      ...
    N1 = EDGE(I,1)
    N2 = EDGE(I,2)
      ...
   DELTAX = F(X(N1), X(N2))

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

END DO
  ...
END
```

Figure 2: Code for Unstructured Mesh in HF+

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

In this section, we propose language features that address this issue. These extensions provide a software layer on top of data-parallel languages, designed to address the "programming in the large" issues as well as the parallel performance issues arising in complex multidisciplinary applications. A program executes as a system of *tasks* which interact by sharing access to a set of **Shared Data Abstractions (SDAs)**. SDAs generalize Fortran 90 modules by including features from object-oriented data bases and monitors in shared-memory languages. They can be used to create persistent shared "objects" for communication and synchronization between coarse-grained parallel tasks, at a much higher level than simple communication channels transferring bytes between tasks.

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

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

An SDA consists of a set of data structures along with the methods (procedures) which manipulate this data. A set of tasks may share data by creating an SDA instance of appropriate type, and making it accessible to all tasks in the set. Tasks may then asynchronously call the methods of the SDA, with each call providing exclusive access. Condition clauses associated with methods and synchronization facilities embodied in the methods allow the formulation of a range of coordination strategies for tasks. The state of an SDA can be saved on external storage for later reuse. This facility can be seen as providing an 1/0 capability for SDAs, where in contrast to conventional byte-oriented 1/0 the structure of the object is preserved.

Other Fortran-based approaches to the problem of combining task with data parallelism include the programming languages Fortran M [5], which provides a message-passing facility in the context of a discipline enforcing determinism, and Fx [10], which allows the creation of parallel tasks that can communicate at the time of task creation and task termination by sharing arguments. These approaches address a small grain of parallelism.

5 Conclusion

In this paper, we have outlined the features of Vienna Fortran, and compared the language to HPF1. After an analysis of weaknesses of HPF-1 in the context of advanced algorithms, we proposed an extension, HPF+, which is based on Vienna Fortran and addresses these problems. HPF+ can be seen as a contribution to the work of the HPF Forum towards the standardization of parallel languages for scalable High Performance architectures.

6 References

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

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

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

[4] B. Chapman, P. Mehrotra, and H. Zima. Extending HPF for Advanced Data-Parallel Applications. *IEEE Parallel & Distributed Technology* 2(3):59-70, Fall 1994.

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

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

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

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

[9] 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.

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

[11] Thinking Machines Corporation. CM Fortran Reference Manual, Version 5.2. Thinking Machines Corporation, Cambridge, MA, September 1989.

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

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

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

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