High Performance Fortran

History, Overview and Current Status

Harvey Richardson

Version 1.2

September 1995

Edinburgh Parallel Computing Centre

The University of Edinburgh

http://www.epcc.ed.ac.uk/epcc-tec/documents.html

Table of Contents

1 Introduction

High Performance Fortran (HPF) is a programming language designed to support the data parallel programming style. A primary design goal for HPF was that the language would enable top performance to be obtained on parallel computer architectures in a way that did not sacrifice portability.

The purpose of this document is to introduce the HPF language and place it in context with previous developments. The current status of the HPF effort is also described and details given of the implementations that are now available.

In the next section we briefly discuss the development of high-performance computer systems and programming environments. Section 3 details the formation and work of the High Performance Fortran Forum (HPFF), the body responsible for the definition and development of HPF. A detailed description of HPF follows in Section 4. The current implementation status of HPF is described in Section 5. The HPFF still continues and current issues are noted in Section 6. In the final section (Section 7) we provide pointers to further information sources relevant to HPF.

For reasons of space this document assumes familiarity with Fortran 90, although features of Fortran 90 relevant to data parallel programming are described briefly. However some terminology is used that may require reference to one of the Fortran 90 texts [1][2][3].

2 An Historical View

Since the dawn of the computer age there has been a demand for ever more powerful computing systems. In part this has been met by evolution in computer technologies - advances from computing mechanisms to electronics, to valves, to solid-state circuitry, to integrated circuits, to ECL for example. In the high-performance scientific market, notable advances have been made with vector processors and exotic circuit technology (GaAs). The more cost conscious high-volume sector has concentrated on integration and economy of scale. Recent RISC processor designs are of very high performance compared to traditional mainframe designs and even supercomputers. This is mainly a consequence of the short design cycle for new processors.

Another trend has been the attempt to use *n* processors in a computing system in the hope that an *n*-fold speed up can be achieved. Parallel computer designs have developed from the early 1970s (ILLIAC IV[4]) to the many `massively parallel' designs of today. The term *massively parallel* is often used loosely, meaning anything from around twenty to many thousands of processors.

We now introduce some terminology which will be required later. High performance parallel computer architectures can be categorised by memory access:

- **True shared memory** - Each processor has direct access to memory by use of a shared bus or network. Such designs are not scalable due to bus or network conflicts.
- **Virtual shared memory** - Each processor has local memory but may access remote memory by use of global addresses. This access is accomplished by low-latency messages sent over an interconnection network. Hence there is non-uniform memory access (NUMA) since it takes longer to access more distant memories.
- **Distributed memory** - Each processor has access to local memory via a local address space, the processors are connected by some network. Access to remote memories is only available via some message-passing system

These categories may overlap somewhat but serve as a useful classification.

Another distinction can be made between Multiple Instruction Multiple Data (MIMD) and Single Instruction Multiple Data (SIMD) hardware. In the former there are typically *n* complex processors executing up to *n* distinct instruction streams independently. An SIMD machine has each (simple) processor operating synchronously and obeying instructions from a single, central controller.

Most parallel machines are purchased in the hope that they can be used efficiently on a range of problems. For this to be possible a given task must be distributed onto the machine such that each processor can be kept busy on part of the problem. It is also desirable that as little data as possible are moved between processors since communication tends to be very time consuming compared to computation. Often there is a trade-off between parallelism and communication. Hence it is important that any programming environment gives adequate control over the distribution and communication of data.

Due to the complexity of highly parallel hardware and its inherent difference from a single-processor/single-memory architecture it is not possible to merely use traditional programming languages and expect to achieve good performance. Two programming paradigms have emerged and gained wide acceptance by the user community (particularly on larger machines):

- **message passing** - In this paradigm, individual programs written in a standard (serial) programming language are executed on each processor having access to memories associated with each processor. A number of message passing environments exist [5]. A common environment is the utilization of a communications library callable from programs running on each node. To transfer data between nodes the programmer can use send and receive routines provided by the library. Often more complex collective operations are also available.
- **data parallel** -In this paradigm a single program controls the distribution of, and operations on, data distributed across all processors. A data-parallel language will typically support array operations and allow whole arrays to be used in expressions. The compiler is responsible for producing code to distribute the array elements on the available processors. Each processor is `responsible' for the subset of the array elements which are stored in its local memory.

(Of course there are other equally valid approaches: for example see the survey by Henri Bal et. al.[6] and the more recent article by François Bodin et. al.[7]. Both articles contain numerous useful references.) Message Passing applications are only possible on MIMD hardware, data-parallel applications may be run on MIMD and SIMD hardware. One potentially confusing term is SPMD (Single Program Multiple Data) which has been used to mean different things in the past. We will use the term SPMD to mean a programming style where the same program is executed on each node of a MIMD system utilizing a message-passing library for communications.

The aim of this document is to discuss data-parallel programming and in particular High Performance Fortran so we now concentrate on these areas.

2.1 Data Parallel Developments

One definition of the data parallel programming style (used in the HPF standard[8]) is a style typified by a parallel environment with

- single-threaded control
- a global namespace
- loosely synchronous execution

By this we mean a single program, defining operations on arrays whose elements are distributed across processors by the compiler. An implementation may allow processors to operate asynchronously but many program constructs will force synchronisation - for example computation of a sum of all array elements. A data parallel compiler (or environment) would provide some basic functionality to enable a wide range of algorithms to be efficiently implemented. A data parallel language should provide most - if not all - of the following functionality:

- **whole-array elemental operations** - For example `a = b + 2*c` where `a`, `b` and `c` are arrays.
- **array sections** - The ability to refer to subsets of an array: a row or column for example.
- **conditional operations** - The ability to operate on a set of array elements based on conditions on array elements or the value of logical masks. This allows operations like taking the inverse of a selection of non-zero elements within an array.

- **reductions** - For example a *sum* operation that adds together all elements in an array.
- **shifts** - The ability to shift elements along given axes of an array. This is very important for stencil operations which find applications in image processing and solution of differential equations for example.
- **general communication** -For example sending data from a vector into specified elements of a 3-dimensional array.
- **parallel prefix/suffix operations** - For example a running total operation.
- **control over data layout** - The ability to control how data is distributed among the processors and to control the alignment of objects to minimise communication.

Notice that these are high-level operations which can be implemented efficiently. It is much harder for a compiler to parallelise an equivalent set of loops defining operations on individual array elements.

As data parallel languages were developed, more and more features from the list above became available. The earliest data parallel languages probably relate to the ILLIAC IV[4] machine around 1973. IVTRAN[9] for example was based on FORTAN and had a parallel **`DO`** construct (**`DO FOR ALL`**) which could be used in conjunction with **`IF`** to express conditional operations in parallel. It was also possible to express a preferred axis for computation as a way to give the compiler a hint as to how data should be distributed for minimal communication. In the same year a data-parallel ALGOL 60 based language Glypnir [10] also had array-like expressions and a parallel IF. However the parallel objects were so called Super Words (swords) which were constrained to match the machine size (number of processors). Later, in 1979 a Pascal based language, Actus [11], added circular and end-off shifts (*rotate, shift*) and some reductions (*and, all*) to define a more complete set of operations.

More recently a number of data parallel languages have emerged which are quite neat extensions of serial languages. Some examples being: Parallel Pascal [12] (NASA MPP, 1983), Fortran-Plus enhanced [13] (AMT DAP, 1990) and Connection Machine C* [14] (Thinking Machines, 1990).

Although a number of languages were available at the start of the 1990s there were too many dialects and also features which were machine specific. This meant that users could be tied to one manufacturer and prospective users were less keen to learn a particular language. In an effort to introduce some standardisation the High Performance Fortran Forum was formed with a mission to define a data parallel Fortran standard.

3 The High Performance Fortran Forum

The discussions on a High Performance Fortran standard were started at Supercomputing '91 where Digital Equipment Corporation had organised a discussion meeting for interested parties. The first HPFF meeting proper was held in Houston, Texas in January 1992 and was attended by 130 people. Detailed work began in March when a working group of approximately 40 people started fleshing out a standard with the intention to finish by January 1993. A further eight meetings were held during 1992 leading to the publication of the HPF Specification v1.0 in May 1993 [8].

The working group contained members from industry, universities and (US) government laboratories. A list of affiliations of those who were present at more than two meetings is given in Table 1.

Table 1: Representation at more than two HPFF meetings

Alliant Computer Systems Corporation	Meiko, Inc.
Amoco	nCUBE, Inc.
Applied Parallel Research	Ohio State University
Archipel	Oregon Graduate Institute
Convex	The Portland Group, Inc.
Cornell Theory Center	RIACS
Cray Research, Inc.	Rice University
Digital Equipment Corporation	Schlumberger
Fujitsu	Shell
GMD	SUNY Buffalo
Hewlett Packard	SunPro, Sun Microsystems
IBM	Syracuse University
ICASE	Technical University, Delft
Intel Supercomputer	Thinking Machines
Lahey Computer	Unified Technologies
Lawrence Livermore National Laboratory	University of Stuttgart
Los Alamos National Laboratory	University of Southampton
Louisiana State University	University of Vienna
MasPar Computer Corporation	Yale University

A wider discussion was facilitated by the creation of a mailing list for those interested in the work in progress. The HPFF has no official connection with recognised standards bodies. Work on Fortran standardisation is undertaken by the ANSI X3J3 committee which is effectively under directions from an international group WG5 (more formally ISO/IEC JTC1/SC22/WG5). Some members of X3J3 were also active within the HPFF and hence there was informal contact between the two groups.

The HPFF set out to define a language to address the following goals:

1. to support the data parallel programming style
2. to achieve top performance on MIMD and SIMD machines with non-uniform memory access costs (and at the same time not be detrimental to performance on other machines.)
3. to support code tuning on various architectures.

In addition, some secondary goals were to: maintain compatibility with existing standards (particularly Fortran 90), achieve simplicity, and define open interfaces to other languages and programming styles. The task of the HPFF was perhaps made easier since there was a wealth of existing work on data parallel Fortran issues. Furthermore the recently finalised Fortran 90 standard [15] included a significant set of array features which form the core of a data parallel language. Other influences on the development of HPF were (in no particular order): various compiler projects by Compass Inc., Connection Machine Fortran [16] (Thinking Machines), Cray MPP programming model [17], Fortran 77D [18] (Rice University), Fortran 90D (Syracuse University) and Vienna Fortran [19] (Vienna University).

The HPF standard was based on Fortran 90 with the addition of features targeted at array distribution, new data parallel features and additional intrinsic and library functions to support efficient use of parallel architectures.

Some features - parallel I/O and elemental pure procedures for example - were not included in the standard because agreement could not be reached on them. These features are documented in the *High Performance Fortran Journal of Development v1.0* which is available from the HPFF World Wide Web server (see Section 7.)

The standard was announced at Supercomputing '92 where an initial draft was distributed. The *High Performance Language Specification, version 1.0* [8] was produced in May 1993 and made available by FTP from a number of sites (see Section 7)

4 The HPF Language

High Performance Fortran is based upon the Fortran 90 [15] programming language. Fortran 90 includes the FORTRAN 77 language and adds many new features which include:

- Array operations
- Improved facilities for numeric computation
- User-defined data types
- New control constructs
- Facilities for modular data and procedure definition
- New source form
- Optional procedure arguments and recursion
- Dynamic storage allocation
- Pointers

As noted previously the (then relatively new) Fortran 90 standard made an ideal basis for HPF. In particular, the Fortran 90 standard was extended by the definition of new directives, language syntax and library routines. Some restrictions were necessary in order to deal with the linear memory model inheritance of some Fortran 77 codes.

HPF extended Fortran 90 in the following ways. There is support for controlling the alignment and distribution of data on a parallel machine. New data parallel constructs are added. An extended set of intrinsic functions and a standard library provide much useful functionality at a high level of abstraction. EXTRINSIC procedures standardise the interface with other languages or styles and there are directives to address some sequence and storage association issues.

We now look in more detail at these elements of HPF.

4.1 Array Mapping

In any parallel programming environment it is crucial that the programmer have control over the layout and distribution of data. Compiler technology is not advanced enough to enable data mapping decisions to be fully automated and at the same time yield acceptable performance.

The HPF model for data mapping is illustrated in Figure 1.

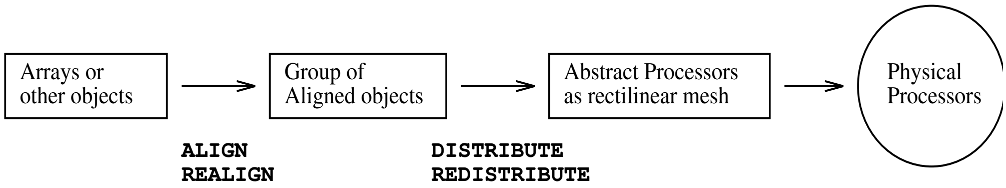

Figure 1: HPF data mapping model

The essential idea is that the relative *alignment* between two data objects may be defined - typically by relations involving array elements. A group of aligned objects (arrays) are then *distributed* onto an abstract rectilinear grid of processors. The mapping of abstract to physical processors is implementation dependent. In Figure 2 the arrays **`A`** and **`B`** are aligned with respect to each other so that **`A(I+1,J+1)`** and **`B(I,J)`** are aligned.

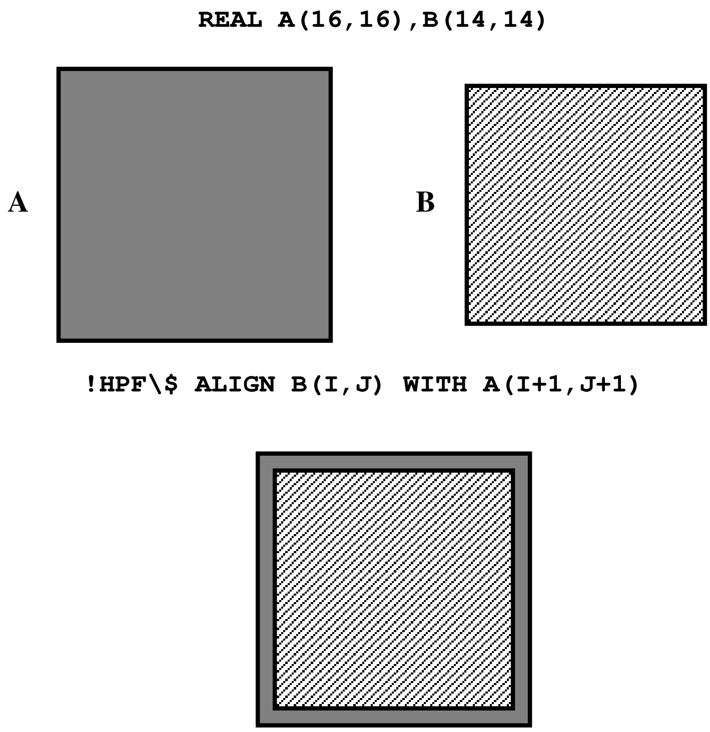

Figure 2: Array alignment

The **ALIGN** directive is used to specify the alignment between objects. In Figure 3 we show some potential distributions of the arrays **A** and **B** onto a set of 4 (abstract) processors.

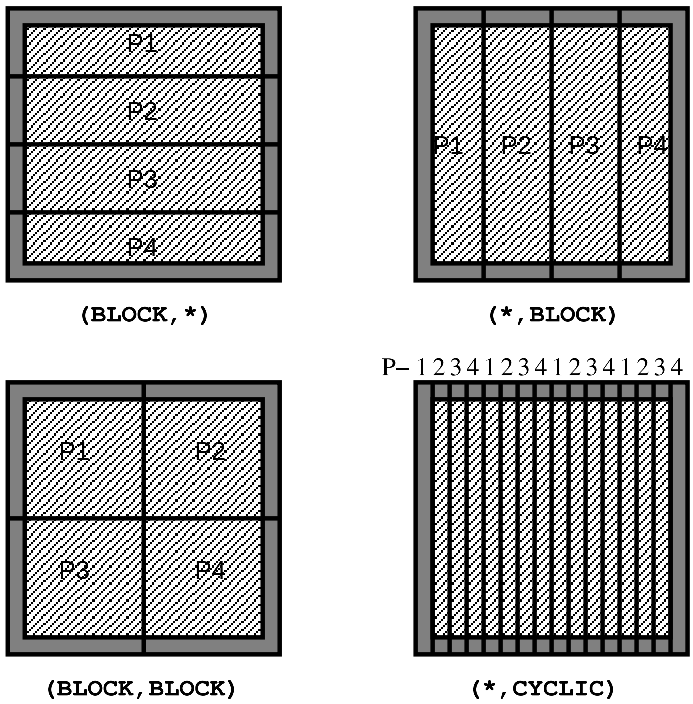

Figure 3: Array distribution

We are displaying axis 1 vertically and axis 2 horizontally. For the distribution **(BLOCK,*)** axis 1 is block distributed among the processors and all elements of axis 2 are local to each processor. The situation is reversed for the **(*,BLOCK)** distribution. With both axes block distributed **(BLOCK,BLOCK)** then each quadrant of the array **A** is distributed to a different processor. Cyclic distributions are also possible and the final example shows a cyclic distribution on axis 2. In this case each element along axis 2 is distributed to a different processor in sequence. Hence neighbouring elements along axis 2 are distributed to different processors. The distribution is defined by a directive, for example:

```
!HPF$ DISTRIBUTE A(*,BLOCK)
```

One example where a cyclic distribution would be useful is Gaussian Elimination. This method operates progressively on smaller blocks of an array. With a block distribution only a few processors (maybe only one) would be in use at some stages. It would not be sensible to cyclicly distribute an axis if many shift operations were required on it since that would involve maximal communication between processors.

There are a number of more complex forms to the **ALIGN** and **DISTRIBUTE** directives but we will not consider them further. It is also possible to align arrays onto an abstract index space, a *template*, rather than allocating an array merely to define an alignment. Again we will not consider this further. There are dynamic forms of the alignment and distribution directives (**REALIGN, REDISTRIBUTE**) so that data mappings can be changed during execution of a program.

It is also possible to specify a processor arrangement to be the target of a distribution. Consider the code below where the **PROCESSORS** directive is introduced and we assume that 6 physical processors are available.

```
      REAL A(16,16),B(14,14)
!HPF$ ALIGN B(I,J) WITH A(I+1,J+1)
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS()/3,3)
!HPF$ DISTRIBUTE A(CYCLIC,BLOCK) ONTO P
```

As before, the arrays **A** and **B** are aligned with respect to each other. Then we define a two dimensional abstract grid of processors **P(i,j)** for **i=1,2;j=1,2,3**. **NUMBER_OF_PROCESSORS** is a system enquiry function returning the actual number of processors available. In this example the elements of **A** are distributed so that:

A(1:15:2,:) → **processors P(1,1:3)**

A(2:16:2,:) → **processors P(2,1:3)**

Figure 4 shows which elements of **A** are mapped to different processors. Again we have axis 1 vertically and the highlighted areas within the arrays **A,B** correspond to an assignment to processor **P(1,2)**.

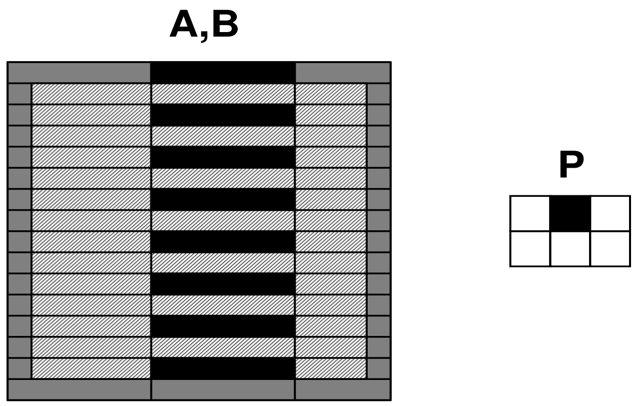

Figure 4: Array distribution **A(16,16)** *to array of processors* **P(2,3)**

As noted previously the mapping from abstract to real processors is implementation dependent. However it is guaranteed that array elements mapped to the same abstract processor will be mapped to the same physical processor. In HPF it is also possible to define a mapping to a *scalar* processor, this could be useful for data that need not be distributed. The implementation may assign this data to a control processor (if the target architecture is a SIMD machine), a particular processor in a MIMD machine or even replicate to all processors in a MIMD machine.

Another issue is the consideration of data mapping across procedure boundaries. Fortran compilers typically only compile one program unit at a time and hence cannot decide globally on distributions. The programmer must ensure consistency between the distributions of actual and dummy arguments if excessive remapping is not to occur. There are facilities in HPF to inherit distributions from actual arguments. It is also possible to use interface blocks to give the compiler enough information so that arguments can be efficiently remapped on invocation of a subprogram.

In summary, HPF has a flexible model to describe the mapping of data to processors and powerful directives to support this model.

4.2 Data Parallel Constructs

The array features of Fortran 90 form a basis for expressing collective operations on many array elements. For example consider the routine below which scales a matrix by its Frobenius norm:

```
          SUBROUTINE SCALE(A)
            REAL, DIMENSION(:,:) :: A
!HPF$       DISTRIBUTE A(BLOCK,BLOCK)
            REAL FNORM

            FNORM = SQRT( SUM( A*A ))
            IF (FNORM>0.) A=A/FNORM
          END SUBROUTINE
```

As well as array assignment, masked assignment and array sectioning, there are constructs to spread arrays along extra dimensions, permute array subscripts, shift array elements and more. However the array assignment statement in Fortran 90 has a (sensible) restriction that the shapes of the right and left hand sides of the assignment must be `conformable` (similarly for expressions). HPF introduces the **FORALL** statement which can express a richer set of assignments without this restriction. Interestingly, the **FORALL** statement was proposed for Fortran 90 (then 8x) but was dropped. The next revision of Fortran is expected to reintroduce it as a standard feature.

4.2.1 FORALL

The **FORALL** statement and corresponding block construct express collective assignments (optionally masked) defined in terms of array subscripts. For example the following statement

```
FORALL( i=1:m,j=1:n ) A(i,j) = i + j
```

defines the assignment to A at all index values in the range specified. Similar to array assignment, the **FORALL** expression is completely evaluated at all active index positions before assignment to the left hand side. In this case the equivalent Fortran 90 code

```
A = SPREAD( (/(i=1,m)/), DIM=2, NCOPIES=n) + &
    SPREAD( (/(i=1,n)/), DIM=1, NCOPIES=m)
```

is more cumbersome. This is typical when the right hand side contains the value of the index variables rather than them appearing only as subscripts. Further examples of a more complex nature are:

```
FORALL(K=1:N, A(K,K).NE.0D0) D(K) = 1D0/A(K,K)

FORALL( I=1:M,J=1:N )
  AB(I,J) = A(I)*B(J)
  V(I,J) = SUM( V,MASK=(C==C(I,J)) )
END FORALL

FORALL (K=1:N) S(K) = SUM( A(:K) )
```

In the first example we assign the reciprocal of the diagonal elements of **A** to a vector **D**. Note the use of a scalar mask expression to control which elements are assigned. Secondly, the outer product of **A** and **B** is assigned to **AB**. The next statement shows just how complex things can become. Here the elements of **V** are assigned to the sum of all elements of **V** corresponding to identical values of an array **C**. The equivalent Fortran 90 expression is more complicated. Finally the **FORALL** statement can implement a parallel prefix operation and here we accumulate partial sums in **S**.

There are restrictions on the **FORALL** statement so that a compiler can produce code that executes in parallel (or alternatively that operates on the index positions in any order). In particular, multiple values may not be assigned to the same *atomic* data object. For example the statement

```
FORALL( i=1:n ) A( p(i) ) = B(i)
```

is not standard conforming if **p** contains repeated values.

It is possible to use user-defined functions in the body or mask expression of the **FORALL** but the functions must meet certain requirements to ensure that they are free of side effects. HPF introduces the **PURE** attribute to be used in defining such functions.

4.2.2 The PURE Attribute

A pure function is guaranteed to be free of any side effects in that it only returns a value and does not modify global data, data mappings, pointer associations or perform I/O. Pure subroutines may modify **INTENT(OUT)** or **INTENT(INOUT)** arguments. A procedure *must* be pure if used in the mask expression or body of a **FORALL** statement or construct, and if used within the body of a pure procedure or as an actual argument in a pure procedure reference. There are some syntactic constraints on the definition of a pure procedure which follow from the restrictions above. As an illustration consider

```
PURE REAL FUNCTION vol(h,w,d)
  REAL h,w,d
  INTENT(IN) :: h,w,d
  vol = h*w*d
END FUNCTION vol
```

and note that this routine could then be used in the statement

```
FORALL( i=1:n) v(i,:) = vol( h(i),w(i),d(i) )
```

under the condition that the interface for **vol** was explicit (for example there was an interface block for **vol**).

4.2.3 The INDEPENDENT Directive

For a compiler to fully optimise code it is essential that it can recognise dependencies between objects in a program. This is particularly important for loops (**DO** and **FORALL**) since execution in a particular order or indeed in parallel can yield significant performance benefits. The HPF **INDEPENDENT** directive is a way for the programmer to assert that there are no interactions between computations for different active index values or alternatively that the loop may be executed in any order or in parallel.

This is particularly useful where compiler analysis alone is not enough. For this code

```
!HPF$ INDEPENDENT
      DO i=1,n
        B( index(i) ) = A(i)
      END DO
```

the assertion is that there are no repeated values in **index**. If there were then the result would depend on the order of evaluation and might even vary from run to run in a parallel environment. Note that the **INDEPENDENT** directive does not change the meaning of the code. There is also a more general form of the directive which allows the declaration of temporaries in the enclosed loop(s).

4.3 Intrinsic and Library Procedures

Fortran 90 has a rich set of intrinsic procedures. To these HPF adds three new intrinsic functions, extends two existing functions and defines a library module.

4.3.1 Intrinsic Functions

The Fortran 90 intrinsic functions **MAXLOC** and **MINLOC** are extended by the addition of an optional **DIM** argument:

```
MAXLOC(ARRAY[,DIM][,MASK])
MINLOC(ARRAY[,DIM][,MASK])
```

(The []s denote optional arguments.) A consequence of this extension is that the second argument may appear ambiguous when keyword arguments are not used. For example with

```
L1 = MINLOC(C,2)
L2 = MINLOC(C,C>2)
```

the second arguments define a dimension and mask respectively. The ambiguity is resolved by noting the types of the arguments.

The system enquiry intrinsic functions

```
NUMBER_OF_PROCESSORS([DIM])
PROCESSORS_SHAPE()
```

return the *physical* number of processors and shape of the processor array in the machine. These functions may be used wherever Fortran 90 requires a *specification-expr* and hence may be used in array declarations or HPF directives.

HPF also defines an additional intrinsic function **ILEN** which returns the number of bits required to store an integer value.

4.3.2 The HPF_LIBRARY Module

HPF defines a library of functions and subroutines which contains a large number of routines useful for expressing data parallel algorithms at a high level. There are also routines for querying the actual alignment and distribution of arrays at run-time. The routines are collected in a library module (this must be provided by an HPF implementation.) To access these routines a **USE** statement is required to place the definitions in scope. Note that these routines are not intrinsic (and hence automatically known to the compiler) because the flexibility of varying argument lists and type specific behaviour was deemed unnecessary.

The rationale behind the provision of many of these routines is that the operations are essential components in a wide range of data parallel algorithms. It is to be expected that vendor supplied implementations will be much more efficient than a user implementation which will necessarily be optimised only for a particular architecture.

The routines available in the HPF_LIBRARY module are briefly described below. For more detail, refer to the HPF documentation mentioned in Section 7.

Mapping inquiry subroutines

The mapping of arrays is done by an implementation taking into account any advisory directives in the program. The routines **HPF_ALIGNMENT**, **HPF_TEMPLATE** and **HPF_DISTRIBUTION** are available to enquire about the mapping at run-time.

Bit manipulation functions

These bit manipulation functions are provided

```
LEADZ(I)
POPCNT(I)
POPPAR(I)
```

and return the number of leading zero bits, the number of one bits and the bit parity of the integer **I** respectively.

Array reduction functions

Additional reduction functions are defined which extend the set already available in Fortran 90 (**ANY, ALL, COUNT, PRODUCT, SUM, MAXVAL and MINVAL**). The new routines are **IALL, IANY, IPARITY** and **PARITY**. These correspond to the binary operations **IAND, IOR, IEOR** and **.NEQV.** respectively.

Array combining scatter functions

These routines generalize the assignment

```
Y(V) = X
```

to allow repeated values in the vector **V**. Values sent to the same array element are combined by a combiner **XXX** where **XXX** ∈ (**ALL, ANY, COPY, COUNT, IALL, IANY, IPARITY, MAXVAL, MINVAL, PARITY, PRODUCT, SUM).** The functions have the form

```
XXX_SCATTER(ARRAY,BASE,INDX1,...,INDXn,MASK)
```

where **XXX** specifies the combiner. Essentially these routines return a result that is equal to **BASE** combined with some combination of elements from **ARRAY**. These elements are sent to destinations defined by the coordinates in the **INDX** arrays.

Array prefix and suffix functions

A prefix function - or scan - applies an operation along a vector such that any element is the result of operations on preceding elements. For example:

```
SUM_PREFIX( (/1,2,0,1/) ) = (/1,3,3,4/)
```

The same combiners are available as for the scatter functions and a number of options control the prefix or suffix operation:

```
XXX_PREFIX(ARRAY,DIM,MASK,SEGMENT,EXCLUSIVE)
XXX_SUFFIX(ARRAY,DIM,MASK,SEGMENT,EXCLUSIVE)
```

The suffix function operates in a different direction to the prefix operation: the result at a given position depends on succeeding elements. The scan is applied along a particular dimension of the array **ARRAY** with active elements defined by the mask. The scan may also be segmented. If the (optional) **DIM** argument is not present then **ARRAY** is treated as a vector with elements taken in array-element order.

Array sorting functions

Two functions are available to produce grading permutations of arrays. These can operate on whole multidimensional arrays or along particular dimensions of arrays

```
GRADE_DOWN(ARRAY,DIM)
GRADE_UP(ARRAY,DIM)
```

4.4 Extrinsic Procedures

Extrinsic routines are provided by HPF to meet the goal of providing a standard interface to other programming languages or styles. The mechanism for doing this is to define the type of extrinsic procedure in an interface block by using the **EXTRINSIC** prefix to the function or subroutine definition.

For example imagine that an implementation provided a mechanism to call routines written in a hypothetical data parallel C which we will call *Performance C*. We could define the interface for such a routine as follows

```
      INTERFACE
        EXTRINSIC(PC) FUNCTION COMPRESS(V,N)
          INTEGER, DIMENSION(:) :: V
          INTEGER COMPRESS
!HPF$     DISTRIBUTE(BLOCK) :: V
          INTENT(INOUT) :: V
          INTENT(IN) :: N
        END FUNCTION COMPRESS
      END INTERFACE
```

HPF defines two choices for the *extrinsic-kind-keyword* (which is the `argument' to **EXTRINSIC**), namely HPF and HPF_LOCAL. HPF is merely the standard interface to HPF routines. However the HPF_LOCAL extrinsic interface is an interface to an SPMD model of execution where multiple copies of the routine execute (one to each processor) and these routines only have access to data mapped to them via the dummy arguments. The local routines are free to utilize message passing libraries if desired. On return from the subroutine all processors are synchronised. There are however restrictions on what the local routines can do - for example not remapping data that is returned. A set of routines to support this model are available in an HPF_LOCAL_LIBRARY module. Note that the HPF_LOCAL interface is not a proper part of HPF since it cannot be provided on some machines. However, if the HPF_LOCAL interface is implemented it should be done so as described in the HPF standard.

4.5 Sequence and Storage Association

There are two concepts in the Fortran language which grew out of an assumption of a linear memory model, namely sequence and storage association. Loosely defined these are

- **Sequence association** -The mapping of array elements to a linear order: first axis varies fastest. This defines how array elements are associated across procedure boundaries when the shapes of the actual and dummy arguments differ.
- **Storage association** -The association of two or more objects that occurs when two or more storage sequences share or are aligned with one or more storage units.

Much old code uses these features of the language. However they present severe difficulties for an implementation where data are mapped to a set of processors. HPF provides a **`SEQUENCE`** directive that can define variables and **`COMMON`** blocks to have the *sequential* property. Data declared sequential may be safely sequence and storage associated.

This is the one area where an existing Fortran code may not work with an HPF compiler as **`SEQUENCE`** directives may need to be applied. This is because all data is considered mappable by default.

4.6 Subset HPF

One of the goals of the HPFF was to promote early adoption of HPF. As an aid to this a subset standard was defined in the hope that subset compilers could soon be produced. This is understandable, since the effort involved in producing a full HPF implementation is considerable, not least because HPF contains all of Fortran 90.

Many Fortran 90 features are included in the subset, for example nearly all of the array features and intrinsic functions. Rather than give a comprehensive list here it will suffice to mention the important features *omitted* from the subset which are:

- The **`REALIGN,REDISTRIBUTE`** and **`DYNAMIC`** directives
- The **`PURE`** function attribute
- The **`FORALL`** *construct*
- The HPF_LIBRARY module
- The **`EXTRINSIC`** function interface
- The following Fortran 90 features: free source form, pointers, the **`TARGET`** attribute, derived types and operators, modules and new I/O features.

It is interesting to note that the idea of a subset standard is by no means a new one. The ISO 1539-1980(E) standard (Fortran 77) defined a subset FORTRAN.

The usefulness of the subset HPF definition is diminishing now that implementations of HPF are available --- albeit with varying feature lists. It is likely that a future revision of the HPF language specification will remove the HPF subset definition.

5 Current Implementations

The implementation status of HPF (at the time of writing) is summarised in Table 2.

Table 2: HPF implementation status

Announced Product:	
Applied Parallel Research	Digital
Hitachi	Intel
Meiko	Motorola
NASoftware	NEC
Pacific Sierra Research	The Portland Group
SofTech	
Announced Effort:	
ACE	Convex
Fujitsu	IBM
Lahey	MasPar
Nag	nCUBE
Thinking Machines Corporation	
'Interested':	
Cray Research	Edinburgh Portable Compilers
Hewlett Packard	Silicon Graphics
Sun	

A number of companies have announced products which are either available or in beta test. Others have announced that they are actively working on an HPF implementation. Finally there are the companies who have expressed an interest in HPF (Cray Research, HP and Silicon Graphics attended HPFF meetings) but have not announced an effort for various reasons. Other bodies are also known to be working on HPF or have related projects. For example the JISC funded work at the University of Southampton (UK) and the Adaptor tool from GMD, Germany.

The first HPF products to become available were from Applied Parallel Research and The Portland Group, Inc.(PGI). Both companies offer portable products that target a wide range of architectures --- from workstations and clusters to massively parallel supercomputers. Initial releases were targeted at the HPF subset but recent releases have increased in functionality. As well as basic compilation, there is provision for performance analysis of the parallel application. Digital were the first to announce a full Fortran 90/HPF compiler (called *DEC Fortran 90 v1.0*). This was initially targeted at DEC OSF1/AXP systems.

Since HPF compilers are under constant development this report will no longer attempt to give detailed information on specific products. The compiler vendors should be approached for definitive information.

6 Current Developments

This section covers the work undertaken by the HPFF from early 1994 to date.

In January 1994 a meeting was held in Houston, Texas to formally initiate a High Performance Fortran Forum II effort. This was followed by further meetings in August and October. The stated charter for HPFF '94 was:

- CCI - Corrections, Clarifications, Interpretations including clarifying the ways that HPF 1.0 already supports course-grained parallelism - e.g. independent loops.
- Encourage `Industrial Strength' Implementations:
    - More notes to implementors
    - Collect and publish practical programming kernels
    - Support establishment of a validation suite
- Identify, flesh out requirements for HPF 2.0 but don't `develop' these requirements yet. Candidates are: parallel I/O, irregular grids, additional tasking support.
- Produce an HPF 1.X document.

Significant work was undertaken within 1994 to address these goals. A status meeting on HPF issues was held at *Supercomputing 94* in Washington, D.C. A document ``*HPF-2 Scope of Activities and Motivating Applications*'' was distributed there and describes the then status of HPF activities and the set of motivating applications. The document also contains numerous references to related work. It is available by ftp from the host **`hpsl.cs.umd.edu`** in the file **`pub/hpf_bench/hpf2.ps`**.

The HPFF is still active and has the following objectives for 1995/1996:

1. To produce, by November 1996, a High Performance Fortran version 2.0 including additional features that broaden the applicability of HPF, and whose implementation can exploit successful research experience.
2. To provide correction, clarification, and interpretation of the current language standard, HPF 1.1.
3. To encourage high quality implementations.

Details on current HPFF efforts are available from the HPFF WWW pages (see Section 7).

The following sections give information on work completed or in progress. Note that the sections do not correspond directly to working groups within the HPFF.

6.1 Corrections, Clarifications and Interpretations

The HPF 1.0 standard was produced in a relatively short time and contained a number of errors and ambiguities. The groups within HPFF have been working to define corrections to the HPF standard, to clarify points that are not explicitly covered and to provide interpretations of the standard. A CCI document is available from the HPFF WWW pages, this contains details of many questions submitted to the HPFF for clarification along with related formal replies from the HPFF. A new revision of HPF, *High Performance Fortran Language Specification version 1.1* [20], was made available in November 1994.

6.2 Tasking Issues

HPF does not address tasking issues. One group is considering how tasking directives (parallel section definitions for example) could be combined with HPF data distribution. Existing work in this area are Convex SPP directives and the ANSI X3H5 group. The current proposal defines TASK REGIONs and assumes mechanisms which allow distribution of variables to processor subgroups and direct execution on groups of processors for specific blocks of code.

6.3 Implementations

Ones group keeps track of the implementation status of HPF and intends to compile a compendium of papers and essays. The hope is that implementors will share experience so that quality implementations are quickly available. For more details on the implementation status of HPF see Section 5.

6.4 Distribution of Data and Computation

The main criticism of data parallel programming and of HPF has been that there is little support for irregular data distribution. To address this, more general data alignments and distributions can be defined and these are under consideration in the light of the applications base. Current proposals define generalised block distributions that would allow block distributions of varying block size or more general mappings of array elements (within an axis) to processors.

Another issue is that it is not possible to specify *where* a computation is performed. It would to possible to extend **`DO INDEPENDENT`** and **`FORALL`** to do this and to specify which code blocks should be handled by a set of processors.

6.5 I/O Issues

Significant progress on I/O issues was made during the development of HPF 1.0 but no new I/O features were included in the standard - see the Journal of Development for more information. Recent work has concentrated on the impact of parallel I/O on Fortran standards and a number of recommendations to X3J3 made (for example to remove a restriction on file position being a default integer type which limits file size on some machines). Other topics under consideration include: asynchronous I/O, checkpoint/restart functions and out-of-core arrays.

6.6 Motivating Applications

The proposed extensions address a lack of functionality in HPF which precludes an efficient implementation of a number of applications. A set of *motivating applications* is being collected so that deficiencies in HPF can be highlighted and addressed by proposed extensions for HPF-2. The applications and documentation are available via anonymous **`ftp`** from the site **`hpsl.cs.umd.edu`** in the directory **`pub/hpf_bench`**. Among the applications are: Barnes-Hut N-body simulation, EULER 3-D hydrodynamics, Multigrid, Direct Simulation Monte Carlo, the FFT, Out of Core Matrix Transposition and Sparse LU Factorization.

6.7 Other Issues

Two other proposals relate to Kernel HPF and interoperability with C. Kernel HPF defines a minimal set of HPF features that can be implemented with maximum efficiency. The interoperability proposal defines a mechanism by which HPF can call external routines written in C. An interface definition for the external function gives enough information for the implementation to convert the HPF variables to ``equivalent'' C variables, performing any type conversion, dereferencing, reordering or remapping as appropriate.

7 Information Sources

There is a wealth of information available about High Performance Fortran, mostly in electronic form via the Internet. The HPF standard (v1.0) was published in a number of forms, the most accessible printed copy probably being the publication in the journal *Scientific Programming* [8]. A useful book is the "*High Performance Fortran Handbook*" [21], published by the MIT press.

On the World Wide Web there is a server containing information on the HPFF and this is available at the URL **`http://www.erc.msstate.edu/hpff/home.html`** These Web pages should be the first point of call for those interested in HPF, there are pointers to the standard itself, pointers to HPF documentation, details of HPFF meetings and information on mailing lists.

An alternative version of this document is available from sthe *Related Publications* link on the HPFF Web page.

8 Bibliography

[1] M. Metcalf and J. Reid. *Fortran 90 explained*. Clarendon, 1990.

[2] J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener. *Fortran 90 Handbook*. McGraw-Hill, 1992.

[3] T. M. R. Ellis, I. R. Philips, and T. M. Lahey. *Fortran 90 Programming*. Addison Wesley, Wokingham, 1994.

[4] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV Computer. *IEEE Transactions on Computers* **C-17**(8), 746-757 (1968).

[5] O. A. McBryan. An overview of message passing environments. *Parallel Computing* **20**, 417-444 (1994).

[6] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys* **21**(3), 261-322 (1989).

[7] F. Bodin, T. Priol, P. Mehrotra, and D. Gannon. *Directions in Parallel Programming: HPF, Shared Virtual Memory and Object Parallelism in pC++*. ICASE Report No. 94-54, NASA CR No. 194943. Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Va. USA, 1994.

[8] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming* **2**(1-2), 1-170 (1993).

[9] R. E. Millstein. Control Structures in Illiac IV Fortran. *Communications of the ACM* **16**(10), 621-627 (1973).

[10] D. H. Lawrie, T. Layman, D. Baer, and J. M. Randal. Glypnir - A Programming Language for Illiac IV. *Communications of the ACM* **18**(3), 157-164 (1975).

[11] R. H. Perrott. A Language for Array and Vector Processors. *ACM Transactions on Programming Languages and Systems* **1**(2), 177-195 (1979).

[12] A. P. Reeves. Parallel Pascal: An Extended Pascal for Parallel Computers. *Journal of Parallel and Distributed Computing* **1**, 64-80 (1984).

[13] Active Memory Technology Ltd, 65 Suttons Park Avenue, Reading, Berks, RG6 1AZ, UK. *Introduction to FORTRAN-PLUS enhanced*, 1990. AMT has been taken over by Cambridge Parallel Processing, check availability with them.

[14] Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1264, USA. *C* Programming Guide*, 1993.

[15] International Organisation for Standardization and International Electrotechnical Commission. *Fortran 90 [ISO/IEC 1539:1991 (E)]*, 1991.

[16] Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1264, USA. *CM Fortran Language Reference Manual*, 1994.

[17] D. M. Pase, T. MacDonald, and A. Meltzer. *MPP Fortran Programming Model*. Cray Research, Inc., 655F Lone Oak Drive, Eagan, Minnesota 55121, 1994.

[18] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. *Fortran D Language Specification*, Department of Computer Science Technical Report. Rice University, Houston, Tx. USA, 1991.

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

[20] High Performance Fortran Forum. High Performance Fortran Language Specification v1.1. available at **`http://www.erc.msstate.edu/hpff/hpf-report-ps/hpf-v11.ps`**, 1994.

[21] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, Ma., USA, 1994.