Final Report on HPF2 and a comparison with HPF+

L.M. Delves
N.A. Software Ltd

Deliverable: D3.2b
Date: March 1998
Status: draft
Confidentiality: public
Responsible: N.A. Software Ltd

Revised:

The **HPF+** Consortium:
AVL GesmbH - Austria
European Centre for Medium-Range Weather Forecasts (ECMWF) - England
Engineering System International SA (ESI) - France
NA Software Ltd (NAS) - England
University of Pavia - Italy
University of Vienna - Austria

©1998 by the **HPF+** Consortium

Contents

1 Introduction **2**

2 Overview of HPF2 **2**
2.1 Base HPF2 . . . 3
2.2 Approved Extensions . . . 3

3 HPF2 Language: Selected Examples **3**
3.1 Remapping at Procedure Boundaries . . . 4
3.1.1 Example . . . 4
3.2 EXTRINSIC program units . . . 4
3.3 ON Directive . . . 5
3.4 RESIDENT . . . 6
3.5 TASK_REGION directive . . . 6

4 Comparison of HPF2 with HPF+ **7**
4.1 Processor Array Subsets . . . 7
4.2 Data Mappings . . . 8
4.2.1 GEN_BLOCK, INDIRECT . . . 8
4.2.2 Multi–Block Distributions . . . 8
4.2.3 Templates . . . 8
4.3 Derived Type Mappings . . . 8
4.4 Independent Loops . . . 8
4.4.1 HPF2 facilities . . . 8
4.4.2 PUREST procedures . . . 9
4.4.3 Schedule Re–Use . . . 9
4.5 The RANGE directive . . . 9
4.6 SHADOW and HALO Directives . . . 10
4.7 TASK_REGION . . . 10

5 Conclusions **10**

1 Introduction

HPF2 is an amended and extended version of HPFV1.1; many of the extensions owe much to work by Vienna both within and outside the HPF+ project. In this document we provide an overview of HPF2 and an outline comparison with the final version of the HPF+ language (Deliverable D2.1b). The comparison is based on a study of the HPF2 draft standard; on discussions with HPFF members carried out during attendance at HPFF meetings in September 96(LMD) and December 96 (DW); and on discussions within the HPF+ (Siegfried Benckner VCPC; Dave Watson NASL) and Pharos (Thomas Brandes GMD) projects.

2 Overview of HPF2

HPF2 provides a reasonably graceful upgrade to HPF1.1:

- Nearly all of the language content of 1.1 has been retained.
- Some facilities have been changed slightly (so that HPF2 is *not* a superset of HPF1.1). But the changes are restricted to those features of 1.1 which have not in practice been implemented by vendors; few if any existing users will notice the changes.

  The removed features are:

  - Sequential arrays may no longer be explicitly mapped.
  - An explicit interface is always required for any call needing remapping of one or more arguments.
  - `INHERIT` and `DISTRIBUTE` may not both be specified for a dummy argument.

  These restrictions were already imposed by the N.A. Software HPFPlus compiler.
- HPF2 is described as based on Fortran95 rather than Fortran90. So, `FORALL, PURE`, and extensions to `MINLOC, MAXLOC` are no longer described within HPF2 since they have been adopted within Fortran95. To the user of course, this is irrelevant: the features are still available.
- Subset HPF is no longer recognised. Instead, the language has been split differently:

  1. Base HPF2 contains all of Fortran95, plus those HPF1.1 features which have been most commonly implemented.
  2. An Approved Extensions Annexe describes many more features which are *not* part of Base HPF2, but which provide a standard syntax and semantics for those vendors wishing to implement them.

  The HPFF hopes that all vendors will implement Base HPF2 as quickly as possible. They expect that different vendors will implement different selections from the Approved Extensions (there are so many that it is *most* unlikely that all will be implemented in the foreseeable future).

2.1 Base HPF2

Compared with HPF1.1, Base HPF2 has additions and subtractions:

Additions : A `REDUCTION` clause has been added to the `INDEPENDENT, NEW` feature to help parallelise existing loops without major rewriting.

Minor extensions to the intrinsic library have been made.

The syntax for extrinsic routines has been extended.

Subtractions : Dynamic mappings (`REDISTRIBUTE, REALIGN`) are not part of Base HPF2 but are in the Approved Extensions. We strongly approve of this move.

2.2 Approved Extensions

These include:

Extensions for Data Mapping :

- Mappings to processor subsets to assist MIMD description and load balancing
- Mapped Derived Type components (arrays of derived types may already be mapped within Base HPF2; mapped arrays may not have mapped components).
- Irregular distributions: `GEN_BLOCK, INDIRECT`.
- `REALIGN, REDISTRIBUTE`
- A `RANGE` directive to reduce combinatorial explosion with `INHERIT`.
- A `SHADOW` directive to specify overlap regions

Intrinsic and Library procedures : have been extended

Asynchronous I/O : but no parallel I/O

Extrinsic Interfaces : Extensions to these.

ON directive : to specify where to compute

RESIDENT directive : An assertion that no communication outside a specified set of processors, will be required.

Tasking Directive : to specify a simple form of MIMD parallelism with separate tasks being carried out on disjoint processor subsets (but with semantics such that no multitasking is ever needed on a given virtual processor).

3 HPF2 Language: Selected Examples

In this section we illustrate some of the parts of HPF2 which we expect to be the most important in practice; but start with an important simplification over HPF1.1.

3.1 Remapping at Procedure Boundaries

In HPF1.1, `descriptive` and `prescriptive` mappings are defined. A descriptive mapping provided a promise that the actual arguments would match the specification in the procedure so that no remapping would be needed. An implicit interface was then allowable under some circumstances. Now, HPF2 removes the distinction between these two classes of procedure argument (though both are kept syntactically for backwards compatibility).

The HPFPlus compiler already implements the HPF2 rules (as does that of DEC) by insisting on an explicit interface whenever any argument is mapped.

3.1.1 Example

```
Program EX1
interface
  subroutine cyclic_map(a,b)
  real,dimension(:)::a,b
  !hpf$ distribute(cyclic)::a,b
  end subroutine
end interface
real, dimension(1000):: x,y
!hpf$ distribute(block)::x,y
....
call cyclic_map(x,y)
...
end program
```

In this example, the explicit interface is mandatory because the actual arguments must be remapped at the call.

3.2 EXTRINSIC program units

The HPF2 extended syntax is:

```
EXTRINSIC(LANGUAGE=.., &
          MODEL=.., &
          EXTERNAL_NAME=..) [TYPE] FUNCTION/SUBROUTINE(...)
```

with three models being supported: `MODEL = 'GLOBAL', 'LOCAL', 'SERIAL'` and languages `LANGUAGE = 'HPF', 'F77', 'FORTRAN', 'C'`.

There are abbreviations for backwards compatibility:

Abbreviation	LANGUAGE	MODEL
HPF	HPF	GLOBAL
HPF_LOCAL	HPF	LOCAL
HPF_SERIAL	HPF	SERIAL
F77_LOCAL	F77	LOCAL
FORTRAN_LOCAL	FORTRAN	LOCAL
C_LOCAL	C	LOCAL

HPFPlus currently (at Release 2.1)implements `HPF_SERIAL, HPF_LOCAL` (and `HPF`). Note that the syntax used is *not* that of a directive but of a first class language feature. However, it seems possible that Fortran2000 will adopt the syntax to formalise the calling of C and other alien language routines.

3.3 ON Directive

This is an Approved Extension, with syntax exemplified by:

```
!hpf$ ON HOME(a(2:4))
 statement
```

or in block form:

```
!hpf$ ON HOME(a(2:4)) BEGIN
   block
!hpf$ END [ON]
```

where a block will typically be a loop. The semantics are as follows:

- Restricts the set of active processors, which is specified by the HOME clause. This can mention a program variable, a template, or a processor arrangement.
- The ON clause does *not* assert anything about data residency; so communication with other processors may be needed.
- ON clauses may be nested provided the nesting makes sense.
- ON can be applied to subprogram invocations:

```
!hpf$ processors p(10)
!hpf$ distribute x(block) onto p
!hpf$ on (p(4:7))
   call collect(x)
```

will cause `X` to be mapped to P(4:7) before the call; any local variables used in `collect` will also be stored on P(4:7).

3.4 RESIDENT

This directive asserts that the data accessed by the computations in the scope of the directive, is local to the set of processors specified. The compiler can use this information to avoid generating communication tests, or to simplify address calculations.

The syntax is exemplified by:

```
!hpf$ on home (z(i)), resident (x,y, record(i))
x(i) = y(i+1) + record(i)%field1 + record(i+1)%field2
```

This directive applies to one statement. It asserts that `x(i), y(i+1)`, and `record(i)` all lie on the same processor as `z(i)`. But `record(i+1)` may be on a different processor.

More commonly, the assertion will be used to label a loop:

```
real, dimension(1000):: x,y
integer,dimension(1000)::ix1,ix2
!hpf$ processors p(np)
!hpf$ distribute(block) onto p:: x,y,ix,iy
....
!hpf$ independent
do i = 1,n
  !hpf$ on home(x(i)), resident(y(ix(i)))
  x(i) = y(ix(i)) - y(iy(i))
end do
```

3.5 TASK_REGION directive

This provides a simple form of heterogeneous task parallelism which we expect to be of substantial importance in porting applications. Here is a code fragment which illustrates it:

```
!hpf$ processors procs(8)
....
done1 = .false.
do while(.true.)
! First task on first four processors:
!hpf$ ON (procs(1:4)) BEGIN, RESIDENT
  read(unit=iu, end=100)a1
   call rowffts(a1)
   goto 101
   100 done1 = .true.
   101 continue
!hpf$ END ON
   IF(done1) EXIT
   a2 = a1
```

```
! Second task on processors 5-8
!hpf$ ON(procs(5:8)) BEGIN, RESIDENT
   call colffts(a2)
   write(unit=ou)a2
!hpf$ END ON
end do
!hpf$ END TASK_REGION
```

4 Comparison of HPF2 with HPF+

Apart from the inclusion of full Fortran95 (eg: derived types, pointers) Base HPF2 is quite close to Subset HPF1.1. The HPF+ language includes extensions aimed at improving the efficiency of description and (hopefully) compilation of Fortran codes using highly irregular data distributions. Comparison should therefore be made with Base HPF2 *together* with the Approved Extensions Annexe. This comparison ignores the question of when commercial compilers might implement specific features; we include here such comments as we can for the NAS HPFPlus compiler. The comparison is with the final version of HPF+ as described in [1].

4.1 Processor Array Subsets

HPF+ and HPF2 each address this, providing the facility to distribute data to regular subsections of processor arrays; but HPF+ provides greater functionality via the inclusion of: `RESHAPE, SHAPE=,ORDER=` for a processor array. These allow a programmer to ensure an appropriate correspondence between processors defined in arrays of differing rank or shape.

This facility fills a significant gap in HPF2. The NASL HPFPlus compiler implementation provides a subset of the facility implicitly by making the association between different processor arrays using array element ordering; but an explicit mechanism is much preferable.

Examples (taken from [1]):

```
!HPF$ PROCESSORS R(8)
!HPF+ PROCESSORS Q(4,2) = RESHAPE(R)
!HPF+ PROCESSORS R_ODD(4) = R(::2), R_EVEN(4) = R(2::2)

!HPF$ PROCESSORS R(16), R2(4,4), UPPER_LEFT_R2(2,2) = R2(1:2,1:2)
```

The use of processor subsets provides a convenient alternative to the use of `ALIGN`, and one which is often easier for the programmer to visualise.

Note that the use of `RESHAPE` with the optional `order` parameter gives a mechanism for describing an irregular distribution of data to a processor array.

4.2 Data Mappings

4.2.1 GEN_BLOCK, INDIRECT

Compared with HPF1.1, both languages add `GEN_BLOCK, INDIRECT` mappings. However, the HPF+ `INDIRECT` mappings are more powerful than those in HPF2: HPF2 provides indirection only on a per dimension basis, while multidimensional indirect distributions are allowed in HPF+. Additionally, HPF+ will accept *array expressions* as indirection arguments.

`GEN_BLOCK` distributions are easy to implement efficiently and provide a natural and useful extension of `BLOCK(m)` distributions. `INDIRECT` mappings require sophisticated compilation techniques to be available before they can hope to be efficient (as [1] notes). NASL will implement `GEN_BLOCK` in its next compiler release, but we are likely to be more cautious over `INDIRECT`.

There is a minor (but to programmers potentially annoying) difference in semantics between HPF2 and HPF+ indirect distributions: in HPF2 the mapping array values must lie between the declared lower and upper bounds of the processor array dimension, while in HPF+ they must lie between 1 and the appropriate dimension size. NASL will be forced to stay with the standard HPF2 semantics, and this would currently implie that we could not implement multi–dimensional indirect distributions.

4.2.2 Multi–Block Distributions

HPF+ also provides a `MULTI_BLOCK(array)` distribution; this allows the description of an iregular distribution of a set of contiguous blocks, and provides for those common cases where there is some locality in the problem as well as some irregularity. NASL hopes to implement `MULTI_BLOCK` at the same time as `INDIRECT`.

4.2.3 Templates

Note that HPF+ does *not* support `TEMPLATE` declarations: it provides a single level distribution model. There has been ongoing arguments over whether templates are either necessary or a Good Idea; Vienna has consistently argued that they are Not A Good Idea. NASL agrees on the whole with Vienna - they add little to the expressive power of the language, but are a substantial pain to the compiler - but we are forced to implement them.

4.3 Derived Type Mappings

Both languages provide mapping of derived type components and mapping of pointers.

4.4 Independent Loops

4.4.1 HPF2 facilities

Both languages provide `INDEPENDENT, NEW, REDUCTION, RESIDENT` clauses and `ON, ON HOME`.

In HPF+, the `INDEPENDENT` directive may be affixed to an array statement, and asserts to the compiler that the right hand side may be computed in parallel with the left hand side.

This is a surprisingly useful tool for avoiding unnecessary temporary arrays, and was provided already some time ago for our Fortran90 compiler, with the same semantics.

4.4.2 PUREST procedures

HPF+ provides also `PUREST` procedures as a way of guaranteeing to the compiler that an invocation requires *no* communication (a desirable attribute which `PURE` signally fails to provide).

4.4.3 Schedule Re–Use

For irregular parallel loops it is a major task for the compiler to generate efficient code. HPF+ provides a number of facilities to help the compiler, and these facilities, with `HALO`, provide a major step forward in handling irregular codes.

The compiler's problems stem from its inability, when indirect addressing is involved, to analyse the pattern of communications needed within a statement (the *schedule*. To run a loop in parallel, this analysis has to be carried out at runtime; this involves an "inspector" phase which gathers the information, and an "executor" stage which does the required work. Since the inspector is likely to be expensive, efficiency requires that schedules, once computed, be re–used as long as possible.

The user needs to help here. The constructs provided in HPF+ include:

REUSE : gives permission to re–use a computed schedule over a whole loop, possibly conditionally.

SCHEDULE variables : provide more explicit control over schedule re–use.

RESET : to indicate that a given schedule variable now needs recomputing

GATHER, SCATTER : to bind the schedules associated with specific arrays to schedule variables.

Implementing these facilities within a compiler is very costly; but the benchmarks carried out within the HPF+ project show the benefits which result. They also show that the facilities are reasonably straightforward to use by an applications programmer: the result of developing the syntax and semantics in close collaboration with end users.

4.5 The RANGE directive

HPF2 introduced `RANGE` to enable the potential complexity of `INHERIT`ed arguments and `DYNAMIC` variables to be delimited. HPF+ implements `RANGE` but with the additional feature that the processor array onto which the object is distributed, can be specified in addition to the type of distribution.

4.6 SHADOW and HALO Directives

A shadow region is a rewctilinear, contiguous area around a local section of an array which is used to store copies of data otherwise resident on another (neighbouring) processor. Both HPF2 and HPF+ include a `SHADOW directive`; the construct is useful for describing the details of regular stencil operations to the compiler.

HPF2 provides also `HALO` directives; these generalise shadows to provide *irregular* local buffering of remotely held data. Indications from experiments within HPF+ suggest that the facility can be used very powerfully to improve efficiency.

4.7 TASK_REGION

This is not included in HPF+. NASL believes that:

1. Multitasking is a natural complement to the ability to address processor subsets; and is potentially a very valuable language addition;

2. The HPF2 `TASK_REGION` construct is unnecessarily restricted in providing only wholly independent tasks with no mutual synchronisation points; a richer construct would be preferable.

5 Conclusions

Overall, HPF+ and HPF2 are quite close on both content and syntax. This is not surprising: the developments in HPF+ were influential in steering the HPFF development of HPF2, and in particular `GEN_BLOCK, INDIRECT` are directly attributable to University of Vienna input to HPF2.

However, the HPF2 Approved Extensions contain features (eg: `TASK_REGION`) not addressed in HPF+; while HPF+ includes major features (HALO, schedule re–use, multidimensional indirect distributions) not provided by HPF2.

We believe that this experience gained with industrial codes in HPF+ has been very beneficial. The major HPF+ innovations appear to be both powerful and useable. We very much hope that they will be added to a future upgrade of the HPF de facto standard; but in any event we plan to add them, as extensions, to the NASL HPF compiler.

References

[1] *Definition of HPF+*, HPF+ Deliverable D2.1c, March 1998 (VCPC responsible)

[2] *High Performance Fortran Language Specification*, HPFF Forum, August 1996