HPF tutorial

Tom Haupt
Northeast Parallel Architectures Center
at Syracuse University
Syracuse, New York, USA
haupt@npac.syr.edu

Contents

Chapter 1	Goals and Scope of High Performance Fortran	1
Chapter 2	The HPF Model	2
Chapter 3	Data Mapping	3
Section 1	Overview	3
Section 2	DISTRIBUTE	4
Section 3	ALIGN	5
Section 4	DYNAMIC	7
Section 5	Allocatable Arrays and Pointers	7
Section 6	PROCESSORS	9
Section 7	TEMPLATE	10
Section 8	INHERIT	11
Section 9	Alignment, Distribution, and Subprogram Interface	12
Section 10	Examples	14
Topic 1	Example 1	14
Topic 2	Example 2	16
Topic 3	Example 3	18
Chapter 4	Data Parallel Statements and Directives	20
Section 1	Overview	20
Section 2	Array Assignments (FORALL)	21
Topic 1	Fortran 90 array assignments	21
Topic 2	Array Sections	21
Topic 3	WHERE statement and construct	21
Topic 4	Elemental invocation of intrinsic functions	22
Topic 5	FORALL statement and construct	22
Topic 6	WHERE...ELSEWHERE construct	25
Section 3	Pure Procedures	25
Section 4	INDEPENDENT Directive	26

Chapter 5 Intrinsic and Library Procedures 28
Section 1 Elemental Functions 28
Topic 1 List of Fortran 90 elemental functions 29
Subtopic 1 New HPF Elemental Function ILEN 29
Subtopic 2 Numeric Computation Functions 29
Subtopic 3 Character Computation Functions 30
Subtopic 4 Bit Computation Functions 30
Subtopic 5 Conversion Functions 31
Section 2 Transormational Functions 31
Topic 1 List of Fortran 90 Transformational Intrisic Functions . . . 31
Subtopic 1 Array Reduction Functions 31
Subtopic 2 Array Construction Functions 32
Subtopic 3 Array Manipulation Functions 32
Subtopic 4 Array Reshape Functions 32
Subtopic 5 Array Computation Functions 32
Subtopic 6 Array Location Functions 32
Subtopic 7 Other Transformational Functions 33
Section 3 Inquiry Functions 33
Topic 1 List of Inquiry Intrinsic Functions 33
Subtopic 1 HPF System Inquiry Functions 33
Subtopic 2 Numeric Inquiry Functions 34
Subtopic 3 Kind Functions 34
Subtopic 4 Array Inquiry Functions 34
Subtopic 5 Pointer Association Inquiry Function 35
Subtopic 6 Argument Presence Inquiry Function 35
Subtopic 7 Character Inquiry Function 35
Subtopic 8 Bit Inquiry Function 35
Section 4 Fortran90 intrinsic subroutines 35
Topic 1 List of Intrinsic Subroutines 35

Section 5 HPF Library 35
Topic 1 Mapping Inquiry Subroutines 36
Topic 2 Bit Manipulation Functions 36
Topic 3 Array Reduction Functions 36
Topic 4 Array Combining Scatter Functions 36
Subtopic 1 Example 36
Topic 5 Array Prefix and Suffix Functions 37
Subtopic 1 Examples 38
Topic 6 Array Sorting Functions 39
Chapter 6 Extrinsic Procedures 39
Section 1 Restrictions 40
Chapter 7 Storage and Sequence Association 41
Section 1 Storage Associations 41
Section 2 HPF Storage Associations Rules 42
Section 3 Sequence Associations 43
Section 4 HPF Sequence Associations Rules 44
Chapter 8 Subset HPF 44
Section 1 Fortran 90 Features in Subset HPF 45
Section 2 HPF Features Not in Subset HPF 46
Chapter 9 More About HPF 46

.

1 Goals and Scope of High Performance Fortran

Nowadays, when increasing the speed of processors become more and more difficult, more and more computer experts admit that the future of high performance computing belongs to parallel computers. Many machines that allow for concurrent execution are commercially available for several years. Nevertheless, this is a very rapidly developing technology, and vendors come with newer, better concepts almost every year. Hardly ever parallel computers coming from different vendors have a similar architecture. To exploit specific features of the machine, vendors develop specific extensions to existing languages (Fortran, C, ..., etc.) and/or develop vendor specific runtime libraries for interprocessor communication. As a result, codes developed on these machines are not portable from platform to platform. Even worse, moving to the next version of the machine from the same vendor, usually requires recoding to obtain expected efficiency of the application. As the consequence, there is no surprise that they are not widely used, in particular, for commercial purposes. Users who traditionally require tremendous amount of CPU still prefer conventional CRAY supercomputers, recognizing parallel computing as a high risk technology, which does not protect software investment.

The problem of scalability and portability of the software for parallel computers, which is the key for protection of software investment, is addressed by High Performance Fortran (HPF). The idea behind HPF is to develop a minimal set of extensions to Fortran90 to support data parallel programming model, defined as single threaded, global name space, loosely synchronous parallel computation. The purpose of HPF is to provide software tools (i.e., HPF compilers) that produce top performance codes for MIMD and SIMD computers with non-uniform memory access cost. The portability of the HPF codes means that the efficiency of the code is preserved for different machines with comparable

number of processors. The HPF extensions to the Fortran 90 standard fall into three categories: compiler directives, new language features, and new library routines. The HPF compiler directives are structured comments that suggest implementation strategies or assert fact about a program to the compiler. They may affect the efficiency of the computation performed, but they do not change semantics of the program. In analogy to Fortran 90 statements, there are declarative directives, to be placed in the declaration part of a scoping unit, and executable directives, to be placed among the executable Fortran 90 statements. The HPF directives are design to be consistent with Fortran 90 syntax except for the directive prefix !HPF$, CHPF$ or *HPF$.

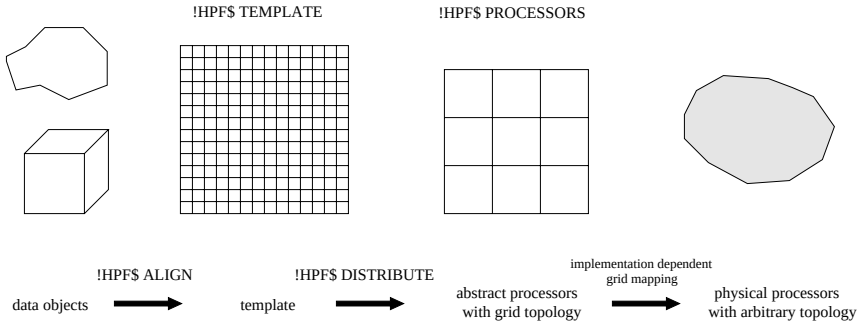
The new language features are FORALL statement and construct as well as minor modifications and additions to the library of intrinsic functions. In addition, HPF introduces new functions that may be used to express parallelism, like new array reduction functions, array combining scatter functions, arrays suffix and prefix functions, array sorting functions and others. Those functions are collected in a separate library, the HPF library. Finally, HPF imposes some restrictions to Fortran 90 definition of storage and sequence associations.

2 The HPF Model

The HPF approach is based on two key observations. First, the overall efficiency of the program can be increased, if many operations are performed concurrently by different processors, and secondly, the efficiency of a single processor is likely be the highest, if the processor performs computations on data elements stored in its local memory. Therefore, the HPF extensions provide means for explicit expression of parallelism and data mapping. It follows that an HPF programmer expresses parallelism explicitly, and the data distribution is tuned accordingly to control the load balance and minimize communication. On the other hand, given a data distribution, an HPF compiler may be able to identify operations that can be executed concurrently, and thus generate even more efficient code.

3 Data Mapping

3.1 Overview



HPF data alignment and distribution directives allow the programmer toadvise the compiler how to assign data object (typically arrayelements) to processors' memories. The model (c.f. figure) is that there is atwo-level mapping of data objects to memory regions, referred to as"abstract processors":

- arrays are first aligned relative to oneanother,
- and then this group of arrays is distributed onto a userdefined, rectilinear arrangement of abstract processors

The final mapping, abstract to physical processors is not specified by HPF and it is language-processor dependent.

The alignment itself is logically accomplished in two steps. First, the index space spanned by an array that serves as an align target defines a natural template of the array. Then, an alignee is associated with this template. In addition, HPF allows users to declare a template explicitly; this is particularconvenient when aligning arrays of different size and/or different shape. It is the template (either a natural or explicit one) that is distributed onto abstract processors. This means, that all arrays'elements aligned with an element of the template are mapped to the same processor. This way locality of data is forced. Arrays and other data object that are not explicitly distributed using the compiler directives are mapped according to an implementation dependent default distribution.

One possible choice of the default distribution is replication: each processor is given its own copy of the data.

The data mapping can be declared using declarative directives:PROCESSORS, ALIGN, DISTRIBUTE, and, optionally, TEMPLATE. In addition, arrays may be remapped during the runtime. To this end, array must bedeclared using DYNAMIC directive, and the actual remapping is triggered by executable directives REALIGN and REDISTRIBUTE.

It is important to notice that the template is not a first-class Fortran 90 object, in the sense that it cannot be passed to a subprogram as an argument. As a consequence, a distributed array passed to a subprogram is aligned either to the natural template of the actual argument or it is aligned to the user defined template. Inboth cases it may lead to a runtime, implicit remapping of the array.To allow more efficient implementations, in particular when the mapping of the actual argument is known at the compile time, HPFprovides a directive INHERIT that specifies that a dummy argument should be aligned to a copy of the template of the corresponding actual argument in the same way the actual argument is aligned. In addition, user may use a special syntax of the ALIGN and DISTRIBUTE directives (with stars preceding the align and/or distribute attributes) that serve as assertion rather than declaration of the mapping of the dummy argument

3.2 DISTRIBUTE

The DISTRIBUTE directive specifies a mapping of data objects to abstract processors in a processor arrangement. Technically, the distribution step of the HPF model applies to the template of the object to which the array is ultimately aligned.

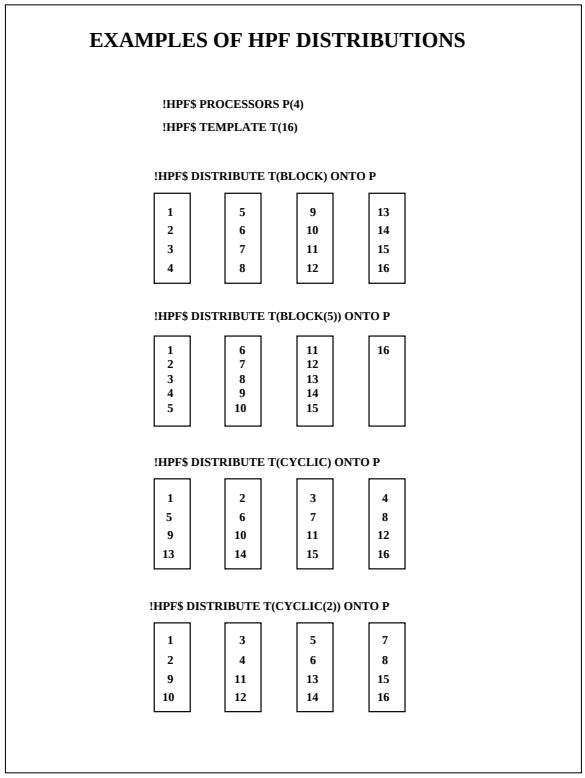
A template may be distributed (in each dimension) in the following ways:

- BLOCK
- CYCLIC
- BLOCK(N)
- CYCLIC(N)

In addition, any dimension of the template may be collapsed or replicated onto a processor grid (note, that it does not change the relative alignment of the arrays!).

The BLOCK distribution specifies that the template should be distributed across set of abstract processors by slicing it uniformly into blocks of contiguous elements. The BLOCK(n) distribution specifies that groups of exactly n elements should be mapped to successive abstract processors, and there must be at least (array size)/n abstract processors

if the directive is to be satisfied. The CYCLIC(n) distribution specifies that successive array elements' blocks of size n are to be dealt out to successive abstract processors in round-robin fashion. Finally, CYCLIC distribution is equivalent to the CYCLIC(1) distribution. The HPF distributions are illustrated in the following diagram.



Every object is created as if according to some complete set of specification directives; if the program does not include complete specifications for the mapping of some object, the compiler provides defaults. The default distribution is language-processor dependent, but must be expressible as explicit directives for that implementation.

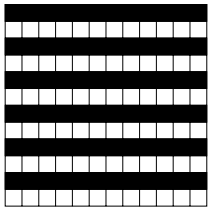
3.3 ALIGN

The ALIGN directive is used to specyfy that certain data objects are to be mapped in the same way as certain other data objects. Operations between aligned data objects are likely to be more efficient than operations between data objects that are not known to be aligned.

Data objects such as arrays may be aligned one with another in many ways. The repertoire includes shifts, strides, or any other linear combination of a subscript (i.e., n*i + m), transposition of indices, and collapse or replication of array's dimensions. Skewed or irregular alignments are, however, not allowed.

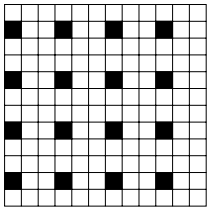
EXAMPLES OF HPF ALIGNMENTS

TRANSPOSITION AND STRIDE



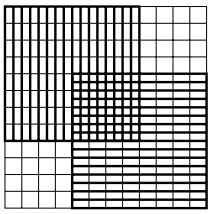
```
REAL, DIMENSION(12,6) :: A
!HPF$ TEMPLATE T(12,12)
!HPF$ ALIGN A(I,J) WITH T(2*J-1,I)
```

SHIFT AND STRIDES



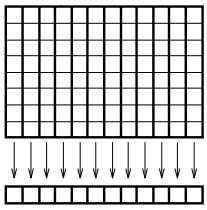
```
INTEGER, DIMENSION(4,4) :: B
!HPF$ TEMPLATE T(12,12)
!HPF$ ALIGN B(I,J) WITH T(2:12:3,1:12:3)
```

RELATIVE ALIGNMENT



```
REAL, DIMENSION(8,8) :: C,D
!HPF$ TEMPLATE T(12,12)
!HPF$ ALIGN C(:,:) WITH T(:,:)
!HPF$ ALIGN D(I,J) WITH T(I+5,J+5)
```

COLLAPSE



```
REAL, DIMENSION(8,12) :: E
!HPF$ TEMPLATE T(12)
!HPF$ ALIGN(*.:) WITH T(:)
```

If an object A is aligned with an object B, which in turn is aligned to an object C, this is regarded as an alignment of A with C directly. We say that A is*ultimately aligned* withC. If an object is not explicitly aligned with another object, we say that it is ultimately aligned with itself.

It is illegal to explicitly realign an object (REALIGN direcive) if anything else is aligned to it and it is illegal to explicitly redistribute an object (REDISTRIBUTE direcive) if it is aligned with another object.

3.4 DYNAMIC

ALIGN and DISTRIBUTE directives are declarative directives and must be placed in the declaration part of a scoping unit (e.g. subroutine). They define *static* data mapping: once declared it cannot be changed at execution time.

HPF allows also for dynamic mapping of data object (e.g. arrays). The *executable* directives REALIGN and REDISTRIBUTE are introduced for this purpose. These directives have the same syntax as static ALIGN and DISTRIBUTE directives, respectively, and they must be placed among executable Fortran 90 statements. In addition, data object to be remapped dynamically **must** be declared as dynamic using DYNAMIC directive, for example:

```
!HPF$ DYNAMIC A,B
!HPF$ DYNAMIC :: C,D
!HPF$ DISTRIBUTE(BLOCK,BLOCK), DYNAMIC :: X
```

Dynamic mapping is not included in Subset HPF.

3.5 Allocatable Arrays and Pointers

The fundamental difference between mapping of static and allocatable arrays is in time when the directives take effect. Mapping of allocatable arrays takes effect not on entry to subroutine but only at time when the array is allocated by an ALLOCATE statement.

As a consequence, the following code is not HPF-conforming:

```
      SUBROUTINE ILLEGAL(A)
      REAL, DIMENSION(:) :: A(:)
      REAL, ALLOCATABLE  :: B(:)
!HPF$ ALIGN A(I) WITH B(I)
```

The directive ALIGN in the above example cannot be processed since array B is not yet allocated. In the next example:

```
      SUBROUTINE ILLEGAL
      REAL, ALLOCATABLE  :: A(:),B(:)
!HPF$ ALIGN A(I) WITH B(I)
      ALLOCATE(A(1000))
```

ALLOCATE statement is nonconforming because A needs to be aligned but at that point in time B is still unallocated. On the other hand,

```
      SUBROUTINE GOOD
      REAL, ALLOCATABLE  :: A(:),B(:)
!HPF$ ALIGN A(I) WITH B(I)
      ALLOCATE(B(1000))
```

is correct because the alignment action does not take place until A is allocated.

It is forbidden for any data object (except for itself) to be aligned to an array at the time the array becomes undefined by reason of deallocation.

Another feature that a programmer must be careful about is that values of specification expressions in ALIGN and DISTRIBUTE directives are determined only once on entry to the subprogram. For example:

```
      SUBROUTINE BE_CAREFUL(N,M)
      REAL, ALLOCATABLE, DIMENSION(:) :: A,B
!HPF$ ALIGN B(I) WITH A(I+N)
!HPF$ DISTRIBUTE A(BLOCK(M*2)
      N=5
      M=50
      ALLOCATE(A(495))
      ALLOCATE(B(500))
```

The values of amount of shift N and block size M on entry to the subroutine are retained by ALIGN and DISTRIBUTE directives. Consequently, the data mapping is performed according to the original values of N and M and not 5 and 50, respectively.

One can gain more flexibility of mapping of allocatable arrays declaring them as dynamic (using DYNAMIC directive) and using REDISTRIBUTE and/or REALIGN directives that immediately follow ALLOCATE statement.

An array pointer may be used in REALIGN and REDISTRIBUTE as an alignee, align-target or distributee, *if and only if* it is currently associated with a whole array (thus not an array section). One may remap an object by using a pointer as an alignee or distributee only if the object was created by ALLOCATE but is not an ALLOCATABLE array.

3.6 PROCESSORS

The PROCESSORS directive declares (one or more) processor arrangement(s). Only rectilinear processor arrangements are allowed in HPF. Therefore they are completely defined by their names, their ranks (number of dimensions), and the extent in each dimension. For example:

```
!HPF$ PROCESSORS P(N)
!HPF$ PROCESSORS BIZARRO(1972:1997, -20:17)
```

The final mapping, abstract to physical processors, is not specified by HPF, and it is language-processor dependent. The intent is, however, that if two object are mapped to the same abstract processor at given instance during the program execution, the two object are mapped to the same physical processor at that instant.

Every dimension of the processor arrangement must have nonzero extent. An HPF compiler is required to accept any PROCESSORS declaration in which the product of the extents is equal to the number of physical processors that would be returned by the intrinsic function NUMBER_OF_PROCESSORS. Other cases may be handled as well, depending on implementation. As a consequence, program that declare two processors arrangements of different sizes, e.g.,

```
!HPF$ PROCESSORS P1(10) !HPF$ PROCESSORS P2(4,4)
```

may not be portable, while

```
!HPF$ PROCESSORS P1(16) !HPF$ PROCESSORS P2(4,4)
```

must be allowed by any HPF compiler.

If no shape is specified, then the declared processor arrangement is conceptually scalar

```
!HPF$ PROCESSOR SCALARPROC
```

Depending on the implementation architecture, such a processor arrangement may reside in a "host" processor, or may reside in an arbitrary chosen processor, or may be replicated over all processors.

The intrinsic functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE may be used to inquire about the total number of actual physical processors used to execute the program. This information may then be used to calculate appropriate sizes for the declared abstract processor, for example

```
!HPF$ PROCESSORS Q(NUMBER_OF_PROCESSORS())
!HPF$ PROCESSORS R(8,NUMBER_OF_PROCESSORS()/8)
```

3.7 TEMPLATE

The TEMPLATE directive declares one or more templates, specifying for each the name, the rank (number of dimensions), and the extent in each dimension, for example:

```
!HPF$ TEMPLATE T(100), TMPL2(N,2*N)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) :: A(N)
```

A template is simply an abstract space of indexed positions. Practically, the only use of a template is to be an abstract align-target that may then be distributed. Explicitly declared templates are useful in the particular situation when one must align several arrays relative to one another but there is no need to declare a single array that spans the entire index space of interest (cf. example program no. 3).

Unlikely arrays, templates does not occupy any memory space, cannot be passed as subprograms parameters or be in COMMON blocks. Therefore two templates declared in different scoping units (e.g. subroutines) will always be distinct, and the template to which a dummy argument is aligned is always distinct from the template to which the actual argument is aligned. The only way for two program units to refer to the same template is to declare the template in a module that is then used by the two program units

(Note: Modules are not in Subset HPF)

It may help to understand HPF mapping by assuming that each array is aligned with a template, even though an actual implementation may be different. By default, each array is aligned with its natural template, i.e. an index space that is identical to that declared for the array. Therefore, one can think that the following code:

```
      REAL, DIMENSION(100,100) :: A,B
!HPF$ PROCESSORS P(4,4)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO P
!HPF$ ALIGN B(:) WITH A(:)
```

is processed in the following way. First compiler create implicitly a natural template of A, as if the user declared `!HPF$ TEMPLATE natural_template_of_a(100,100)` and then the template is distributed onto processor arrangement P, and finally, array B is aligned with the template. The explicitly declared templates allow to align the array to a template that is larger than its natural one without unnecessary waste of memory.

When passing an array to a subprogram, by default the actual argument is aligned with its natural template. For example, for the following call

```
      REAL, DIMENSION(100) :: A
      ...
      CALL FOO(A(11:90))
      ...
```

the natural template of the actual argument has shape [80]. The user may then chose to let the compiler to implicitly declare the template of the dummy argument to be an exact copy of the natural template of the actual argument or the user may force use of different template by using appropriate mapping directives (c.f. INHERIT directive and section Alignment, Distribution and Subprogram Interface).

3.8 INHERIT

The INHERIT directive specifies that a dummy argument should be aligned to a copy of the template of the corresponding actual argument in the same way that the actual argument is aligned. In other words, the dummy argument is to be aligned to the *inherited* template rather than to the natural template of the actual argument. For example:

```
      REAL, DIMENSION(100) :: A
      ...
      CALL FOO(A(11:90))
      ...
      SUBROUTINE FOO(B)
      REAL B(80)
!HPF$ INHERIT B
```

Here, the directive INHERIT force dummy argument B to be aligned to a template of shape [100] as if the user declared

```
 !HPF$ TEMPLATE inherited(100)
 !HPF$ ALIGN B(I) WITH inherited(I+10)
```

Note that this allows to avoid costly data remapping on subroutine boundaries at the price of possible introducing of some load imbalance.

If the actual argument is not a regular array section, say, it is an array expression, then the inherited template is chosen arbitraily by the language processor. Consequently, the programmer cannot know anything a priori about its distribution.

The INHERIT attribute implies a default distribution of DISTRIBUTE * ONTO * *(that is, the distribution of the inherited template is assumed to be exactly the same as that declared in the calling procedure),* and this default distribution is not part of Subset HPF.

Therefore, if a program uses INHERIT, it must override the default distribution with an explicit mapping directives in order to conform Subset HPF.

3.9 Alignment, Distribution, and Subprogram Interface

When calling subroutines, one typically faces the following situations:

- the mapping of the dummy arguments is known at compile time and it is to be forced regardless the mapping of the actual argument
- the mapping of the dummy argument is known at compile time, moreover, it is known at compile time that it is exactly the same as that of the actual argument
- the mapping of the dummy argument is not known at compile time and it should be exactly the same as that of the actual argument (in particular, it may be different at each invocation of the subprogram)

A dummy argument always has a fresh template to which is aligned. This template is contructed in one of three ways:

- The template is declared explicitely (a natural template of other array may serve as the explicit template here), and the dummy argument is aligned to it explicitly by ALIGN directive.
- The dummy argument is not explicitly aligned and it have the INHERIT attribute. In this case the argument is aligned to the inherited template, that is, the natural template of the actual argument as declared in the callee (or it is implementation dependent if the actual argument is not a regular array section).
- The dummy argument is not explicitly aligned and it does not have INHERIT attribute. In this case, the dummy argument is aligned to the natural template of the actual argument (which has the same shape and boundaries as the dummy argument).

To force a specific mapping of the dummy argument, the mapping must be defined explicitly, and it must also appear in interface blocks.

To assert the language processor that the actual argument is mapped in the same way as the dummy argument one uses *descriptive* form of mapping directives with asterisks proceeding the mapping specifications. For example:

```
!HPF$ DISTRIBUTE A *(BLOCK) ONTO *P
```

asserts the compiler that A is already distributed BLOCK onto processor arrangement P so, if possible, no data movement should occur.

One can mix *prescriptive* and *descriptive* modes:

```
!HPF$ DISTRIBUTE A(BLOCK) ONTO *P
```

Here, the compiler should do whatever it takes to cause A to have BLOCK distribution on the processor arrangement P; A is already distributed onto P, though it might be with some other distribution format.

To force the language processor to copy all or some aspects of the distribution from that of the actual argument, one uses *transcriptive* format of mapping directives (with distribution parameters replaced by asterisks). For example:

```
!HPF$ DISTRIBUTE A * ONTO *
```

specifies that mapping of A should not be changed from that of the actual argument.

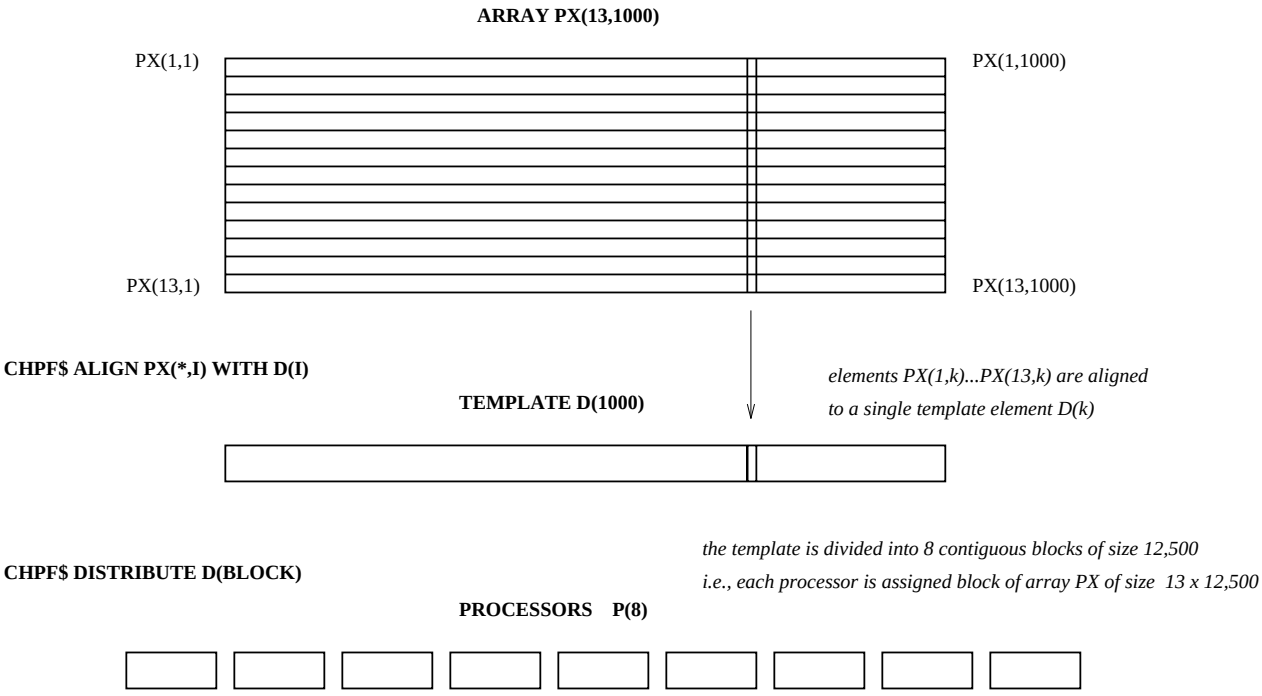
The transcriptive format in not included in Subset HPF.

3.10 Examples

Example 1

(adopted from PARKBENCH, Low Level HPF Compiler Benchmarks, Kerenel AA)

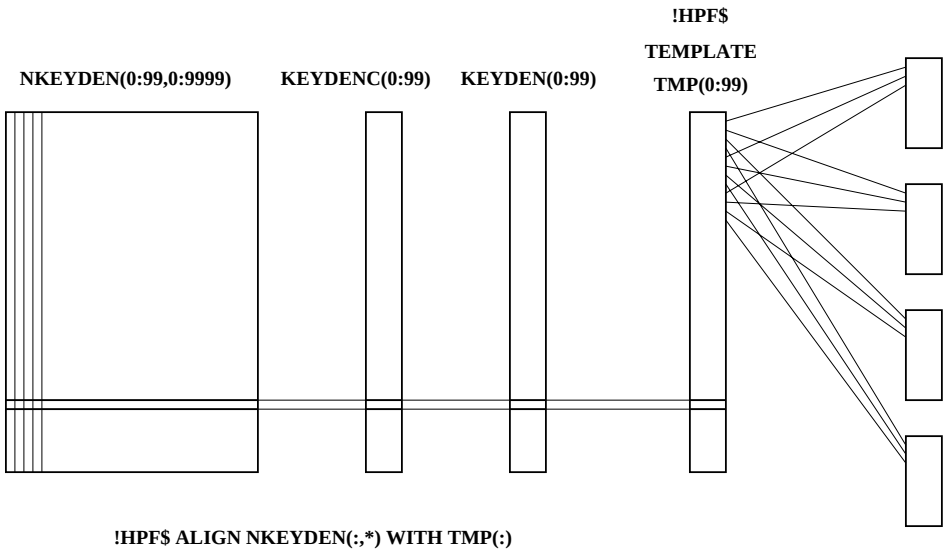
```
program AA
parameter ( N = 100000 )
         real PX(13,N), Q
         integer i

!HPF$ PROCESSORS P(8)
!HPF$ TEMPLATE D(N)
!HPF$ ALIGN PX(*,I) WITH D(I)
!HPF$ DISTRIBUTE D(BLOCK) ONTO P

        FORALL ( i = 1:N )
     *  PX(1,i) =      10.0 * PX(13,i) + 0.2  * PX(12,i) +
     *                  2.6 * PX(11,i) + 0.25 * PX(10,i) +
     *                 0.11 * PX(9,i)  + 2.5  * PX(8,i)  +
     *                 1.01 * PX(7,i)  + 0.5  * (PX(5,i) +
     *                 PX(6,i))        + PX(3,i)
      STOP
      END
```


ARRAY PX(13,1000)
PX(1,1)
PX(1,1000)
PX(13,1)
PX(13,1000)
CHPF$ ALIGN PX(*,I) WITH D(I)
elements PX(1,k)...PX(13,k) are aligned
to a single template element D(k)
TEMPLATE D(1000)
CHPF$ DISTRIBUTE D(BLOCK)
the template is divided into 8 contiguous blocks of size 12,500
i.e., each processor is assigned block of array PX of size 13 x 12,500
PROCESSORS P(8)

Example 2

(adopted from NAS benchmark, fragment of the IS kernel)

```
       program bucksort
       integer                              :: N,MAX,KEY,i,j
       parameter(MAXKEY=100)        parameter(N=10000)
       integer, dimension(0:N-1)            :: key
       integer, dimension(0:MAXKEY-1)       :: keyden,keydenc
       integer, dimension (0:MAXKEY-1,0:N-1) :: nkeyden

!HPF$ PROCESSORS PROC(4)
!HPF$ TEMPLATE TMP(0:MAXKEY-1)
!HPF$ DISTRIBUTE TMP(CYCLIC) ONTO PROC
!HPF$ ALIGN(:) WITH TMP(:) KEYDEN,KEYDENC
!HPF$ ALIGN NKEYDEN(:,*) WITH TMP(:)

      forall(i=0:N-1) key(i)=mod(i,100)
      nkeyden=0

      do j=0,MAXKEY-1
       where(key.eq.j) nkeyden(j,:)=nkeyden(j,:)+1
      enddo

      keydenc=0
      keyden=sum(nkeyden,dim=2)
      FORALL(i=1:MAXKEY-1) keydenc(i)=sum(keyden(0:i-1))

      end
```



!HPF$ ALIGN NKEYDEN(:,*) WITH TMP(:)
!HPF$ ALIGN KEYDEN(:) WITH TEM(:)
!HPF$ ALIGN KEYDENC(:) WITH TEM(:)

A relative alignment of the arrays is defined: elements NKEYDEN(i,0:9999), KEYDENC(i), and KEYDEN(i) are to be stored in the same abstract processor

!HPF$ DISTRIBUTE TMP(CYCLIC)

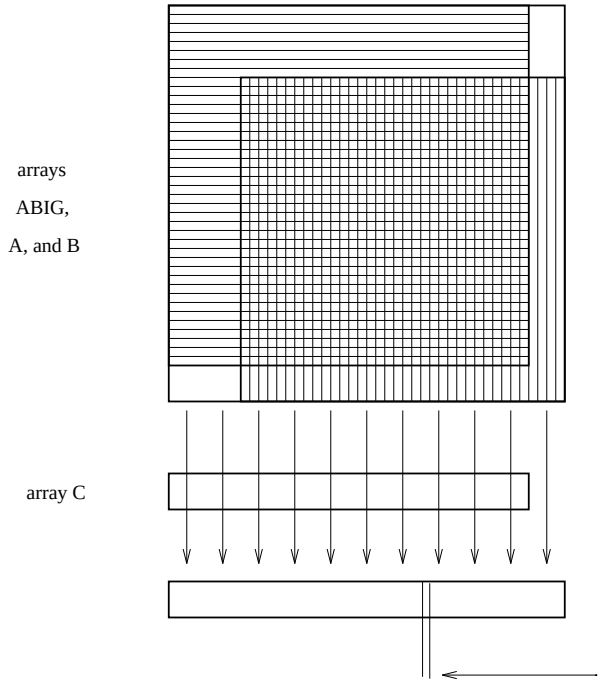
Elements of the template (and consequently, the array elements aligned to them) are distributed onto the processor grid in a round robin fashion. This way an approximate load balance is achived for concurrent execution of the final FORALL statement in this example code.

Note that array KEY is not distributed (it is assumed to be replicated) in order to minimize cost of communication. This may cause that the first FORALL statement will be serialized by the HPF compiler. This inelegant feature of this example code comes from the fact that we wanted to make the code simple and short, and therefore we skept the original initilaization phase of the IS kernel.

Example 3

(adopted from PARKBENCH, Low Level HPF Compiler Benchmarks, Kerenel TM)

```
      program TM
      INTEGER NDIM,MDIM,ND1,MD1
      PARAMETER (NDIM=1023,MDIM=2047,ND1=NDIM+1,MD1=MDIM+1)
      REAL, DIMENSION(MDIM)          :: R
      REAL, DIMENSION(NDIM)          :: C
      REAL, DIMENSION(NDIM,MDIM)     :: A
      REAL, DIMENSION(ND1,MD1)       :: ABIG
      REAL, DIMENSION(1,1)           :: ACORN
      REAL, DIMENSION(NDIM-1,MDIM-1) :: B


CHPF$ PROCESSORS P(8)
CHPF$ TEMPLATE TMPL(ND1)
CHPF$ DISTRIBUTE TMPL(BLOCK) ONTO P
CHPF$ ALIGN ABIG(I,*) WITH TMPL(I)
CHPF$ ALIGN A(I,*) WITH TMPL(I)
CHPF$ ALIGN C(:) WITH TMPL(:)
CHPF$ ALIGN B(I,*) WITH TMPL(I+2)

      forall(i=1:mdim) r(i)=1.0+i
      forall(i=1:ndim) c(i)=1.0-i
      forall(i=1:ndim,j=1:mdim) a(i,j)=i+j
      acorn = 0.5

      ABIG(1:NDIM,1:MDIM)=A
      ABIG(1:NDIM,MD1)=C
      ABIG(ND1,1:MDIM)=R
      ABIG(ND1,MD1)=ACORN(1,1)

      FORALL(I=1:ND1-2,J=1:MD1-2) B(I,J)=ABIG(I+2,J+2)
      STOP
      END
```



LEGEND:

array ABIG(ND1,MD1)

array A(ND1-1,MD1-1)

array B(ND1-2,MD1-2)

CHPF$ ALIGN ABIG(I,*) WITH TMPL(I)
CHPF$ ALIGN A(I,*) WITH TMPL(I)
CHPF$ ALIGN B(I,*) WITH TMPL(I+2)
CHPF$ ALIGN C(:) WITH TMPL(:)

CHPF$ TEMPLATE TMPL(ND1,MD1)

each element of the template TMPL(i) is associated with array elements ABIG(i,1:MD1), A(i,1:MD1-1), B(i+2,1:MD1-2), and C(i,MD1-1) with exceptions for TMPL(1), TMPL(2) not associated with any elements of B and TMPL(MD1) which is not associated with any alements of A or C.

CHPF$ DISTRIBUTE TMPL ONTO P

Contiguous blocks of the template (i.e. array elements associated with them) are stored in local memories of the abstract processors

CHPF$ PROCESSORS P(8)

As a result of this mapping directives assignments ABIG(i,k) = A(i,k), ABIG(j,k)=C(j) and B(i,j)=ABIG(i+2,k) will not cause interprocessor communication

4 Data Parallel Statements and Directives

The parallelism can be explicitly expressed in HPF using the following language features: Fortran 90 array assignments, masked array assignments WHERE, WHERE...ELSEWHERE construct, FORALL statements, FORALL constructs, INDEPENDENT assertions, intrinsic functions and the HPF library, and extrinsic functions

4.1 Overview

Several Fortran 90 features, notably array syntax, WHERE construct and many elemental intrinsic functions proved to be well suited to express parallelism explicitly (as it was intended to be). However, experience with early attempts to define data parallel versions of Fortran, such as CM-Fortran, demonstrated severe limitations of Fortran 90 at this respect. Inspired by research languages, including Fortran D, Vienna Fortran and others, Fortran 90 was augmented with new, HPF features to express data parallelism:

- FORALL statement and construct
- INDEPENDENT directive
- PURE directive
- parallel equivalents of transformational intrinsic functions such as SUM, MAXLOC, RESHAPE, etc.
- a rich set of new standard library functions
- extrinsic procedures

FORALL statement and construct are generalization of array assignments and WHERE construct to relax some restrictions imposed by Fortran 90. The INDEPENDENT directive provides a mechanism to assert the compiler about data dependencies in FORALL and DO loops (it may be obvious for a programmer that a loop can be safely executed in parallel, while it is very difficult to prove it at compile time). In addition within INDEPENDENT DO one can define a NEW instance of selected variables for each iteration without broadcasting its value to all processors involved in computation.

In addition to Fortran 90 elemental intrinsics, HPF introduces a mechanism that is semantically equivalent to user defined elemental functions: PURE functions called in FORALL loops (not in Subset HPF). HPF superseeds Fortran 90 intrinsic transformational functions by their parallel equivalents and adds 48 new routines collected in the HPF Library *(the Library is not a part of Subset HPF)*. Finally, HPF standardize mechanism by which HPF programs call non-HPF subprograms (extrinsic procedures). This feature

allows for a mixture of sequential and parallel programs (example: visualization routines), as well as for taking full advantage of the machine's specific architecture on which the program is run, when the algorithm is difficult to express in a data parallel fashion.

To summarize, the parallelism can be explicitly expressed in HPF using the following language features: Fortran 90 array assignments, masked array assignments WHERE, WHERE...ELSEWHERE constructs, FORALL statements, FORALL constructs, INDEPENDENT assertions, intrinsic functions and the HPF library, and extrinsic procedures.

4.2 Array Assignments (FORALL)

Fortran 90 array assignments

Fortran77 forces the programmer to process array elements one at a time, in DO loops. In fact, the more natural way of the process is that it performs some operation on the whole array. In Fortran90 it is possible to treat a whole array as a single object. For example, supposing that A, B and C are 100x100 arrays of real values, the sum of arrays A, B resulting in array C can be simply expressed in Fortran 90 as

```
C = A + B
```

instead of tideous

```
DO i=1,100
  DO j=1,100
     C(i,j)=A(i,j)+B(i,j)
  END DO
END DO
```

Note, that the assignment of the array elements can be done in arbitrary order, in particular in parallel. This observation is a base for HPF semantics of the array assignments.

Array Sections

User can also refeference part of an array ("array sections"), such as B(:,1:99:2) which corresponds to the odd-numbered columns of B.

WHERE statement and construct

Fortran90 allows for masked array assignments, WHERE statement and WHERE...ELSEWHERE construct, for example:

```
WHERE(mask) A=B
```

Here, the assignment a(i,j)=b(i,j) is executed only for these pairs of indices (i,j) where elements of the logical array mask(i,j) evaluate to .TRUE.

Elemental invocation of intrinsic functions

Arrays and array sections can be arguments to a broad class of elemental initrinsic functions, such as ABS, ATAN, COS, COSH, EXP, to name a few. For example, if A, B are arrays as defined above

```
B = ABS(A)
```

is equivalent to Fortran77

```
DO i=1,100
  DO j=1,100
     b(i,j) = ABS(A(i,j))
  END DO
END DO
```

FORALL statement and construct

FORALL statement and construct are new language features to express data parallelism, that is, to provide a convenient syntax for simultaneous assignments to large groups of array elements. The functionality they provide is very similar to that provided by the array assignments and the WHERE constructs in Fortran 90. In fact, all Fortran 90 array assignments, including WHERE, can be expressed using FORALL statements. For example,

```
B = 1.0
A = B
A(1:98,3:100)=B(3:100,1:98)
WHERE(B.GT.0) A=2.*B
```

can be expressed using FORALL syntax as

```
FORALL(I=1:100,J=1:100) B(I,J)=1.0
FORALL(I=1:100,J=1:100) A(I,J)=B(I,J)
FORALL(I=1: 98,J=3:100) A(I,J)=B(I+2,J-2)
FORALL(I=1:100,J=1:100,B(I,J).GT.0) A(I,J)=2.*B(I,J)
```

However, Fortran 90 places several restrictions on array assignments. In particular, it requires that operands of the right side expressions be conformable with the left hand side ar-

ray. These restrictions are relaxed by FORALL statements. For example, the following assignments cannont be expressed easily in Fortran90:

```
FORALL(I=1:100,J=1:100) A(I,J)=(I+J)*B(I,J)
FORALL(I=1:100) A(I,I)=C(I)
FORALL(I=1:100) A(I,IX(I))=X(I)
```

In addition, a FORALL may call user-defined functions on the elements of an array, simulating Fortran 90 elemental function invocation (albeit with a different syntax). Functions that are allowed to be called in a FORALL loop must be declared as PURE and they must not produce any side effects. Example:

```
FORALL(K=1:9) X(K)=SUM(X(1:10:K))
```

(Note: Fortran90 intrinsic functions are pure by definition).

The FORALL statement essentially preserves the semantics of Fortran 90 array assignments: the array elements may be assigned in an arbitrary order, in particular, concurrently. To preserve determinism of the result, it is required that each array element is assigned only once.

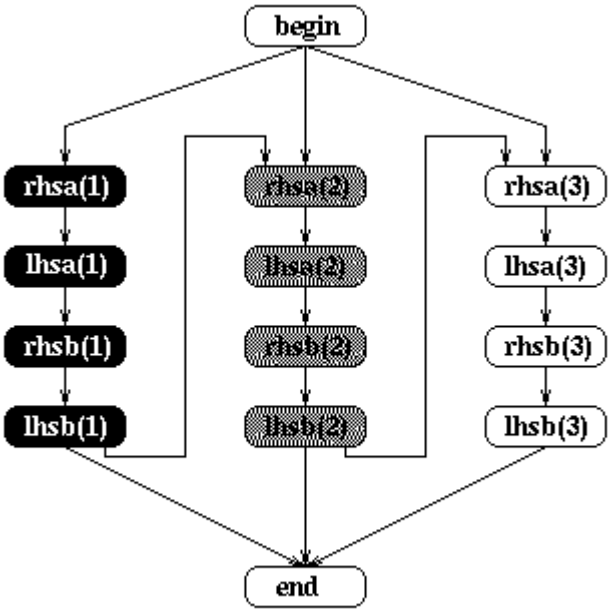


Fig. 4.1 Data dependencies in a DO loop

The FORALL construct is semantically equivalent to a sequence of the FORALL statements. This implies that no loop carried dependencies may by present in the body of the FORALL construct. These are shown in the fig. 4.1 which represents a DO loop

```
DO i=1,3
 lhsa(i)=rhsa(i)
 lhsb(i)=rhsb(i)
ENDDO
```

The lines on the diagram represent dependencies. Since assignment a in the second iteration of the loop depends on assignment b in the first iteration, this DO loop cannot be executed in parallel, and consequently it cannot be expressed as a FORALL construct.

The next diagram, in fig. 4.2 shows a parallelizable loop:

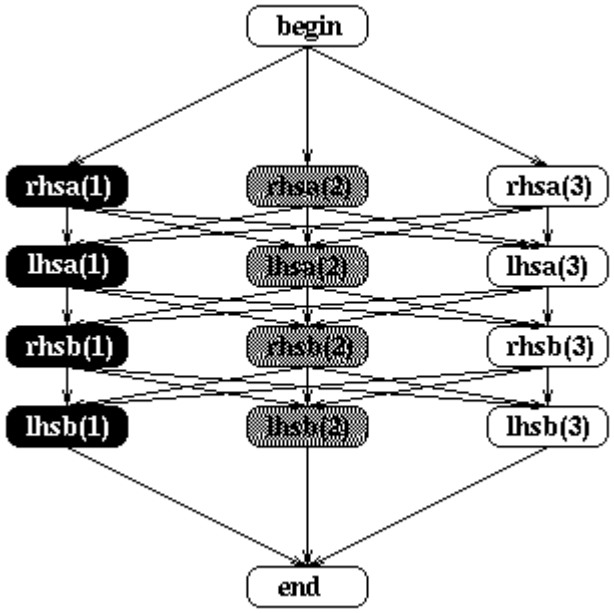


Fig. 4.2 Data dependencies in a FORALL loop

Here, the rhs expression of the assignment a can be safely evaluated concurrently for each loop iteration, since elements rhsa(i) do not depend on computations performed in other iterations. After this is completed (processors must be synchronized at this point), assignments lhsa(i)=rhsa(i) are taking place, again in parallel. Then, the processors must be synchronized once more, before the next array assignment is processed. With such

dependency pattern we may instruct the compiler that we want to execute this loop in parallel using FORALL construct:

```
FORALL (i=1:3)
 lhsa(i)=rhsa(i)
 lhsb(i)=rhsb(i)
ENDFORALL
```

In case of the masked array assignments (WHERE or FORALL with an optional mask) the mask (or condition) must be evaluated before the loop is executed. Therefore statements of the loop body may modify arrays that serve as a mask without influencing the mask itself:

```
FORALL(i=1:n, a(i) > 1.0) a(i)=1.0/a(i)
```

This may lead to an additional synchronization.

FORALL construct and PURE directive are not part of Subset HPF

WHERE...ELSEWHERE construct

The FORALL construct does not replace the WHERE...ELSEWHERE construct, that is, there no generalization of ‘ELSE’ clause for FORALL. The semantics of the WHERE...ELSWHERE construct, as defined by Fortran 90 standard, is that both "WHERE" block of statements and "ELSEWHERE" blocks are always executed (as opposed to IF...ELSEIF construct). In particular, the execution of the "WHERE" block may affect data that are accessed by the "ELSEWHERE" block. Therefore assignments defined in the "WHERE" block must be completed before "ELSEWHERE" block of statements is executed.

4.3 Pure Procedures

A pure function is one that obeys certain syntactic constrains that ensure it produces no side effects. This means that the only effect of a pure function reference on the state of a program is to return a result - it does not modify the values, pointer associations, or data mapping of any of its arguments or global data, and perform no external I/O. A pure subroutine is one that produces no side effects except for modifying the values and/or pointer associations of INTENT(OUT) and INTENT(INOUT) arguments. These properties are declaraed by a new attribute (the PURE attribute) of the procedure.

A pure procedure may be used in any way that a normal procedure can. However, a procedure is *required to be pure* if it is used in any of the following contexts:

- the mask or body of a FORALL statement or construct
- within the body of a pure procedure
- as an actual argument in a pure procedure reference

Example:

```
INTERFACE  PURE FUNCTION f(x)
REAL, DIMENSION(3) :: f
REAL, DIMENSION(3), INTENT(IN) :: x
END FUNCTION f
END INTERFACE

REAL v(3,10,10)
...
FORALL (i=1:10, j=1:10) v(:,i,j)=f(v(:,i,j))
...
```

PURE procedures are not part of Subset HPF

4.4 INDEPENDENT Directive

The execution of the FORALL assignment may require an intra- and interstatement synchronizations: the evaluation of the left hand side expression of the FORALL assignment must be completed for all array elements before the actual assignment is made. Then, the processors must be synchronized again, before the next array assignment is processed.

In some cases these synchronizations may be not necessary. The following diagram illustates a situation where each loop iteration can be processed independently of any computations performed in other iterations *(lines in this diagram symbolize data dependencies)*:

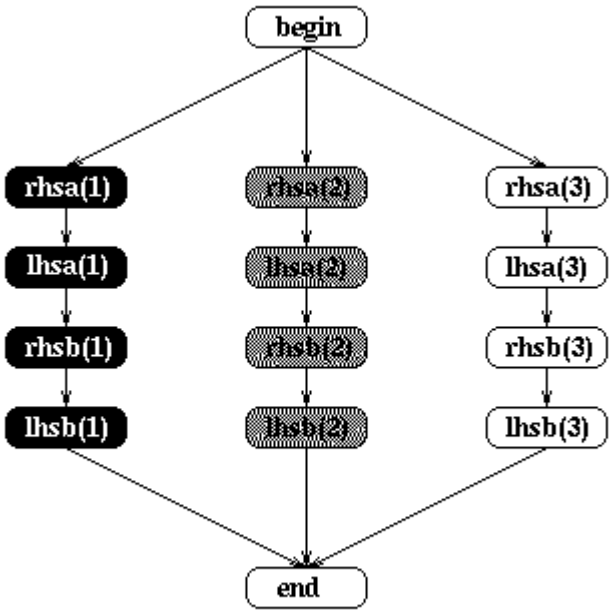


Fig. 4.3 Data dependencies in an INDEPENDENT DO loop

```
!HPF$ INDEPENDENT              !HPF$ INDEPENDENT
      DO i=1,3           or          FORALL(i=1:3)
       lhsa(i)=rhsa(i)                lhsa(i)=rhsa(i)
       lhsb(i)=rhsb(i)                lhsb(i)=rhsb(i)
      END DO                        END FORALL
```

A mature HPF compiler should be able to perform appropriate data dependency analysis to determine possible optimizations. Nevertheless, sometimes the dependency analysis may be extremely difficult, e.g., because of indirections. In that cases, the user, by putting INDEPENDENT directive, may assert the compiler that the operation in the following FORALL statement or construct may be executed independently - that is, in any order, or interleaved, or concurrently - without changing the semantics of the program.

The INDEPENDENT directive may also precede a DO loop, to assert the compiler that iterations in the following DO loop may be executed independently. In addition, in context of DO loop, the INDEPENDENT directive allows for declaration of new

instances of a variable for each iteration of the loop. This is an exception from the global space name paradigm (since each processor is allowed to assign a different value to a variable that has the same name on different processors). For the sake of determinism, the NEW variable becomes undefined beyond the scope of the DO loop.

5 Intrinsic and Library Procedures

HPF includes all Fortran90's intrinsic procedures. It also adds new intrinsic procedures: NUMBER_OF_PROCESSORS, PROCESSORS_SHAPE, ILE , and extends definition of MAXLOC and MINLOC by addition of an optional DIM argument. Not all Fortran90 intrinsic procedures are in Subset HPF.

In addition to the new intrinsic functions, HPF defines a library module, HPF_LIBRARY, that must be provided by vendors of any full HPF implementation (thus not Subset HPF).

5.1 Elemental Functions

Many intrinsic procedures have scalar dummy arguments, and many of these may be called with array actual arguments. These are called *elemental* intrinsic procedures. In Fortran90 there are 64 elemental intrinsic function and one elemental subroutine.

Elemental functions are defined to have scalar results as well as scalar dummy arguments. If the actual argument is an array or array section, the function is applied element-by-element, resulting in an array of the same shape as the aragument and whose element vaalues are the same as if the function had been individually applied to the corresponding elements of the argument.

For example, if A, B are real arrays 100 by 100

```
B = ABS(A)
```

is equivalent to Fortran77

```
DO i=1,100
   DO j=1,100
      b(i,j) = ABS(A(i,j))
   END DO
END DO
```

Since the result does not depend on order in which assignments of the array elements are done, the assignments can be performed concurrently.

HPF introduces an new elemental intrinsic function ILEN which computes the number of bits needed to store an integer value.

List of Fortran 90 elemental functions

New HPF Elemental Function ILEN

- ILEN(I) – returns one less than the lenght, in bits, of the two's-complement representation of an integer: if I is nonnegative, ILEN(I) has the value log2(I+1); if I is negative, ILEN(I) has the value log2(-I). Argument I must be of type integer, result type and type parameter are the same as I.

Numeric Computation Functions

- ABS(A) Absolute value.
- ACOS(X) Arc cosine function (radians).
- ASIN(X) Arc sine function (radians)
- ATAN(X) Arc tangent function (radians)
- ATAN2(Y,X) Argument of complex number (X,Y)
- CEILING(A) Least integer greater than or equal to its argument.
- COS(X) Cosine function (radians)
- COSH(X) Hyperbolic cosine function
- DIM(X,Y) MAX(X-Y,0)
- DPROD(X,Y) Double precision real product of two default real scalars.
- EXP(X) Exponential function
- FLOOR(A) Greatest integer less than or equal to its argument
- LOG(X) Natural (base e) logarithm
- LOG10(X) Common (base 10) logarithm
- MAX(A1,A2{,A3,...]) Maximum value
- MIN(A1,A2[,A3,...]) Minimum value
- MOD(A,P) Remainder modulo P, that is A-INT(A/P)*P
- MODULO(A,P) A modulo P
- SIGN(A,B) Absolute value of A times sign of B
- SIN(X) Sine function (radians)
- SINH(X) Hyperbolic sine function
- SQRT(X) Square root function
- TAN(X) Tangent function (radians)
- TANH(X) Hyperbolic tangent function

(DOT_PRODUCT and MATMUL are transformational intrinsic functions)

Character Computation Functions

- ADJUSTL(STRING) Adjust left, removing leading blanks and inserting trailing blanks.
- ADJUSTR(STRING) Adjust right, removing trailing blanks and inserting leading blanks.
- INDEX(STRING,SUBSTRING[,BACK]) Starting position of SUBSTRING within STRING
- LEN_TRIM(STRING) Length of string without trailing blanks
- LGE(STRING_A,STRING_B) True if String_A equals or follows String_B in ASCII collating sequence
- LGT(STRING_A,STRING_B) True if String_A follows String_B in ASCII collating sequence
- LLE(STRING_A,STRING_B) True if String_A equals or precedes String_B in ASCII collating sequence
- LLT(STRING_A,STRING_B) True if String_A precedes String_B in ASCII collating sequence
- SCAN(STRING,SET[,BACK]) Index of left-most (right-most if BACK is TRUE) character of STRING that belongs to SET; zero if none belong
- VERIFY(STRING,SET[,BACK]) Zero if all characters of STRING belong to SET or index of left-most (right most if BACK is TRUE) that does not.

(REPEAT and TRIM are transformational intrinsic functions)

Bit Computation Functions

- BTEST(I,POS) True if bit POS of integer I has value 1.
- IAND(I,J) Logical AND on the bits
- IBCLR(I,POS) Clear bit POS of I to zero.
- IBSET(I,POS) Set a Bit POS of I to one.
- IEOR(I,J) Exclusive OR on the bits
- IOR(I,J) Inclusive OR on the bits
- ISHFT(I,ISHFT) Logical shift on the bits (end off shift)
- ISHFTC(I,ISHFT[,SIZE]) Logical circular shift on a set of bits on the right.
- NOT(I) Logical complement of the bits

There is also an elemental subroutine:

- CALL MVBITS(FROM,FROMPOS,LEN,TO,POS) Copy bits

Conversion Functions

- ACHAR(I) Character in position I of ASCII collating sequence.
- AIMAG(Z) Imaginary part of complex number.
- AINT(A[,KIND]) Truncate to a whole number.
- ANINT(A,[KIND]) Nearest whole number
- CHAR(I[,KIND]) Character in position I of the processor collating sequence
- CMPLX(X[,Y][,KIND]) Convert to COMPLEX type
- CONJG(Z) Conjugate of a complex number
- DBLE(A) Convert to DOUBLE PRECISION real.
- IACHAR(C) Position of character C in ASCII collating sequence
- IBITS(I,POS,LEN) Extract a sequence of bits
- ICHAR(C) Position of character C in the processor collating sequence
- INT(A[,KIND]) Convert to integer type
- LOGICAL(L[,KIND]) Convert between KINDs of logicals
- NINT(A[,KIND]) Nearest integer
- REAL(A[,KIND]) Convert to REAL type

5.2 Transormational Functions

A transformational intrinsic procedure is one that is not elemental, i.e., array arguments are interpreted as a single objects rather than in elemental fashion. For example, function MATMUL returns an algebraic product of two matrices and not an array of products of the corresponding array elements.

HPF superseeds Fortran90 transformational functions by their parallel implementations optimized for specific architecure of the machine on which the program is run. In addition, HPF defines a rich set of new transformational functions collected in the HPF library *(not in Subset HPF)*. Together they constitute a very powerful set of language features to express paralelism.

List of Fortran 90 Transformational Intrisic Functions

Array Reduction Functions

- ALL(MASK,[DIM]) True if all elements are true.
- ANY(MASK[,DIM]) True if any element is true
- COUNT(MASK[,DIM]) Number of True elements

- MAXVAL(ARRAY[,DIM][,MASK]) Value of maximum array element
- MINVAL(ARRAY[,DIM][,MASK]) Value of minimum array element
- PRODUCT(ARRAY[,DIM][,MASK]) Product of array elements
- SUM(ARRAY[,DIM][,MASK]) Sum of selected array elements

Array Construction Functions

- MERGE(TSOURCE,FSOURCE,MASK) Combines two (comformable) arrays under control of a mask, i.e. TSOURCE when MASK is true and FSOURCE otherwise.
- PACK(ARRAY,MASK[,VECTOR]) Packs a masked array into a vector
- SPREAD(SOURCE,DIM,NCOPIES) Replicates an array by adding a dimension
- UNPACK(VECTOR,MASK,FIELD) Unpacks a masked array from a vector

Array Manipulation Functions

- CSHIFT(ARRAY,SHIFT[,DIM]) Performs circular shift(element i+1 -> i for shift of +1)
- EOSHIFT(ARRAY,SHIFT[,DIM]) Performs end off shift(element i+1 -> i for shift of +1)
- TRANSPOSE(MATRIX) Matrix transpose

Array Reshape Functions

- RESHAPE(SOURCE,SHAPE[,PAD][,ORDER]) Reshape SOURCE to shape SHAPE

Array Computation Functions

- DOT_PRODUCT(VECTOR_A,VECTOR_B) Dot product of two vectors
- MATMUL(MATRIX_A,MATRIX_B) Matrix multiplication

Array Location Functions Note: HPF extends Fortran90 definition of these function allowing for an optional parameter DIM

- MAXLOC(ARRAY[,DIM][,MASK]) Location of maximum array element
- MINLOC(ARRAY[,DIM][,MASK]) Location of minimum array element

Other Transformational Functions

- REPEAT(STRING,NCOPIES) Concatenates NCOPIES of STRING
- SELECTED_INT_KIND(R) Kind of type parameter for specified exponent range
- SELECTED_REAL_KIND([P][,R]) Kind of type parameter for specified precsion and exponent range
- TRANSFER(SOURCE,MOLD[,SIZE]) Same physical representation as SOURCE, but type of MOLD
- TRIM(STRING) Remove trailing blanks from a single string

5.3 Inquiry Functions

Fortran 90 has a number of intrinsic functions known as *inquiry functions* and *numeric manipulation functions* (sometimes called the "environmental intrinsics"). These functions, rather than performing some computation with their arguments, return information concerning the status or nature of the argument.

There are several classes of the inquiry intrinsics: Numeric (KIND, PRECISION, RADIX, EXPONENT, etc) to inquire or set the numerical environment , Array Inquiry (ALLOCATED, SHAPE, SIZE, LBOUND, UBOUND), Pointer Association Status (ASSOCIATED), Argument Presence (PRESENT) and others.

HPF defines a new class of inquiry functions: *System Inquiry Intrinsic Functions*:

- NUMBER_OF_PROCESSORS returns the total number of processors available to the program or the number of processors available to the program along a specified dimension of the processors array.
- PROCESSOR_SHAPE returns the shape of the processor array.

In addition, HPF introduces *Mapping Inquire Subroutines* that allow the program to determine the actual mapping of an array at runtime (particularly important when an EXTRINSIC subprogram is invoked). These subroutines are included in the HPF Library (not in Subset HPF).

List of Inquiry Intrinsic Functions

HPF System Inquiry Functions

- NUMBER_OF_PROCESSORS([DIM]) Total number of processors available to the program or the number of procesors available along a specified dimension of the procesor array

- PROCESSOR_SHAPE() Shape of the processor array

Numeric Inquiry Functions

- DIGITS(X) Number of significant digits in the model for X.
- EPSILON(X) Number that is almost negligble compared with one in the model for numbers like X.
- HUGE(X) Largest number in the model for numbers like X.
- MAXEXPONENT(X) Maximum exponent in the model for numbers like X.
- MINEXPONENT(X) Minimum exponent in the model for numbers like X.
- PRECISION(X) Decimnal precision in the model for X
- RADIX(X) Base of the model for numbers like X.
- RANGE(X) Decimal exponent range in the model for X.
- TINY(X) Smallest positive number in the model for numbers like X
- EXPONENT(X) Exponent part of the model for X
- FRACTION(X) Fractional part of the model for X
- NEAREST(X,S) Nearest different machine number in the direction given by the sign of S
- RRSPACING(X) Reciprocal of the relative spacing of model numbers near X
- SCALE(X,I) X * b**I where b = RADIX(X)
- SET_EXPONENT(X,I) Model number whose sign and fractional part are those of X and whose exponent part is I
- SPACING(X) Absolute spacing of model numbers near X

Kind Functions

- KIND(X) Kind type parameter value of X.

Also see transformational intrinsic functions SELECTED_INT_KIND and SELECTED_REAL_KIND

Array Inquiry Functions

- ALLOCATED(ARRAY) True if the array is allocated.
- LBOUND(ARRAY[,DIM]) Array lower bounds
- SHAPE(SOURCE) Array (or scalar) shape
- SIZE(ARRAY[,DIM]) Array size
- UBOUND(ARRAY[,DIM]) Array upper bounds

Pointer Association Inquiry Function

- ASSOCIATED(POINTER[,TARGET]) True if pointer is associated with target

Argument Presence Inquiry Function

- PRESENT(A) True if optional argument is present

Character Inquiry Function

- LEN(STRING) Lenght of the actual argument

Bit Inquiry Function

- BIT_SIZE(I) Maximum number of bits that may be held in an integer I.

5.4 Fortran90 intrinsic subroutines

Intrinsic subroutines are new in Fortran90; Fortran77 has only intrinsic functions. These subroutines are referenced in the same way as any other subroutines. None of them may be used as an actual argument.

List of Intrinsic Subroutines

- CALL DATE_AND_TIME([DATE][,TIME][,ZONE][,VALUES]) Real time clock reading date and time.
- CALL MVBITS(FROM,FROMPOS,LEN,TO,POS) Copy bits
- CALL RANDOM_NUMBER(HARVEST) Uniform random numbers (0<=x<1)
- CALL RANDOM_SEED([SIZE][,PUT][,GET]) Initialize or restart random number generator
- CALL SYSTEM_CLOCK([COUNT][,COUNT_RATE][,COUNT_MAX]) Integer data from real-time clock

5.5 HPF Library

The Library procedures are divided into folowing categories:

- Mapping Inquiry Subroutines
- Bit Manipulation Functions
- Array Reduction Functions

- Array Combining Scatter Functions
- Array Prefix and Suffix Functions
- Array Sorting Functions

Mapping Inquiry Subroutines

The mapping inquiry subroutines HPF_ALIGNMENT, HPF_TEMPLATE and HPF_DISTRIBUTION allow the program to determine the actual mapping at run time. It may be especially important to know the exact mapping when an EXTRINSIC subprogram is invoked. To keep the number of routines small, the inquiry procedures are structured as subroutines with optional INTENT(OUT) arguments.

Bit Manipulation Functions

The HPF library includes three elemental bit-manipulation functions.LEADZ computes the number of leading zero bits in an integer's representation. POPCNT counts the number of one bits in an integer. POPPAR computes the parity of an integer.

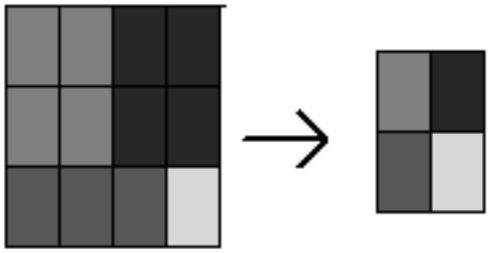
Array Reduction Functions

HPF extends Fortran90's repertoire of array reduction functions by IALL, IANY, IPARITY, and PARITY which correspond to the commutative, associative binary operatons IAND, IOR, IEOR, and .NEQV., respectively. These new functions operate in the same manner as the Fortran90 SUM and ANY which correspond respectively to + and .OR. operators.

Array Combining Scatter Functions

These are generalized array reduction functions in which arbitrary, but nonoverlaping, subset of array elements can be combined. There are twelve scatter functions: ALL_SCATTER, ANY_SCATTER, etc., each being a generalization of one of the twelve reduction functions: ALL, ANY, COPY, COUNT, IALL, IANY, IPARITY, MAXVAL, MINVAL, PARITY, PRODUCT, and SUM.

Example We want to combine array elements of array A of shape [3,4] and put them to array RESULT of shape[2,2] according to the following receipe:



that is,

```
RESULT(1,1)=A(1,1)+A(2,1)+A(1,2)+A(2,2)
RESULT(2,1)=A(3,1)+A(3,2)+A(3,3)
RESULT(1,2)=A(1,3)+A(2,3)+A(1,4)+A(2,4)
RESULT(2,2)=A(3,4)
```

We can achieve this using function SUM_SCATTER in the following way:

1. declare base array B of shape [2,2] (because it is the shape of the RESULT) and set it to 0.
2. declare two mapping arrays (because rank of B is 2), say I1,I2, each of shape [3,4] (because it is the shape of A).
3. initialize arrays I1 and I2 in the following way:

```
       1  1  1  1          1  1  2  2
I1 =   1  1  1  1    I2 =  1  1  2  2
       2  2  2  2          1  1  1  2
```

   This defines mapping of elements of A to elements of RESULT. For example, element A(1,3) goes to RESULT(1,2) = RESULT(I1(1,3),I2(1,3)).
4. function call RESULT = SUM_SCATTER(A,B,I1,I2) does specified reduction.

Array Prefix and Suffix Functions

These functions allow for a scan of a vector. For a prefix scan, each element of the result is a function of the elements of the vector that precede it, and for suffix scan, each element of the result is a function of the elements of the vector that follow it.

There are twelve prefix function ALL_PREFIX, ANY_PREFIX, etc., and twelve suffix functions ALL_SUFFIX, ANY_SUFFIX, etc., each corresponding to one of tweleve

scan functions: ALL, ANY, COPY, COUNT, IALL, IANY, IPARITY, MAXVAL, MINVAL, PARITY, PRODUCT, and SUM.

Use of the optional arguments DIM, MASK, SEGMENT and EXLUSIVE is demonstrated in the following examples:

Examples Example 1

let A = [1, 2, 3, 4] SUM_PREFIX(A) = [1, 3, 6, 10] SUM_SUFFIX(A) = [10, 9, 7, 4] *note that the results has always the same size and shape as array A*

Example 2

let A = [1, 2, 3, 4] SUM_PREFIX(A, EXCLUSIVE=.TRUE.) = [0, 1, 3, 6] SUM_SUFFIX(A, EXCLUSIVE=.FALSE.) = [10, 9, 7, 4] *default value of the optional parameter EXCLUSIVE is .FALSE.*

Example 3

let A = [1, 1, 1, 2] and M = [T, T, F, T] SUM_PREFIX(A, MASK=M) = [1, 2, 2, 4] SUM_SUFFIX(A, MASK=M) = [4, 3, 2, 2] *masked elements do not contribute to the result*

Example 4

let A = [1, 2, 3, 4, 5, 6] and S = [T, T, T, F, F, T] SUM_PREFIX(A, SEGMENT=S) = [1, 3, 6, 4, 9, 6] SUM_SUFFIX(A, SEGMENT=S) = [6, 5, 3, 9, 5, 6] *array S "divides" array A into 3 segments: (1,2,3)(4,5)(6). The scan is made in each segment independently*

Example 5

```
      1  1  1
A =   2  2  2
      3  3  3

                      1  7 13
  SUM_PREFIX(A) =     3  9 15
                      6 12 18
```

array A is processed in array element order, as if temporarily regarded as rank one array. Arrays MASK and SEGMENT are interpreted the same way, if present

Example 6

```
       1  1  1
A =    2  2  2
       3  3  3

                              1  2  3
SUM_PREFIX(A, DIM=2) =        2  4  6
                              3  6  9
```

since optional argument DIM is present, a family of independent scan operations are carried out along the selected dimension of A

Example 7

```
       1  2  3  4  5          T T T T T          T T F F F
A =    6  7  8  9 10    M =   F F T T T    S =   F T T F F
      11 12 13 14 15          T F T F F          T T T T T

                                        0  1  0  3  7
SUM_PREFIX(A, 2, M, S,.TRUE.) =         0  0  0  0  9
                                        0 11 11 24 24
```

any combination of optional parameters DIM, MASK, SECTION, EXCLUSIVE is permitted. If no elements are selcted for a given element of the result, that result element is set to a default value that is specific to the particular function. For SUM_PREFIX and SUM_SUFFIX it is 0.

Array Sorting Functions

HPF includes procedures for sorting multidimensional arrays: GRADE_DOWN and GRADE_UP. These are structured as functions that return sorting permutations. An array can be sorted along a given axis, or the whole array may be viewed as a sequence in array element order. The sorts are stable, allowing for conveniet sorting of structures by major and minor keys.

6 Extrinsic Procedures

Some algorithms may not be expressible efficiently in a data parallel paradigm. Some

computations are inherently sequential (e.g., existing commercial visualization packages) or it may be resonable to take full advantage of the machine's architecure on which the program is run at the price of nonpartability of the code. Therefore, it may desirable for an HPF program to call a procedure written in a language other than HPF and possibly supporting programming paradigm other than that of HPF. In that case the user may have to take responsibility for interprocessor communication, synchronization of processors, possibly dynamic process forking, etc.

For obvious reasons, HPF cannot specify anything about language in which the extrinsic procedure is written (such as Fortran77, C, ADA or any other existing or yet to be developed language). However, HPF does require extrinsic procedures to satisfy certain behavioral requirements: the call to an extrinsic procedure that fulfills these rules is semantically equivalent to the execution of an ordinary HPF procedure. An extrinsic procedure must be declared EXTRINSIC and must have an explicit interface.

6.1 Restrictions

HPF requires a called extrinsic procedure to satisfy the following behavioral requirements:

- The overall implementation must behave as if all actions of the caller preceding the subprogram invocation are completed before any action of the subprogram is executed; and as if all actions of the subprogram are completed before any action of the caller following the subprogram invocation is executed.
- IN/OUT intent restrictions declared in the interface for the extrinsic procedure must be obeyed.
- Replicated variables, if updated, must be updated consistently.
- No HPF variable is modified unless it could be modified by an HPF procedure with the same explicit interface.
- When a subprogram returns and the caller resumes execution, all objects accesible to the caller after the call are mapped exactly as they were before the call.
- Exactly the same set of processors are visible to the HPF environment before and after the subprogram call.

Extrinsics are not in Subset HPF

HPF_LOCAL procedures

The annex to HPF defines a mechanism for coding a specific type of extrinsic procedures: HPF_local procedures. These are single processor "node" codes written

either in single-process Fortran90 or in a single processor subset of HPF; the idea is that only data that is mapped to a given physical processor is accessible to it. This allows the programming of MIMD multiprocessor machines in a single-program multiple-data (SPMD) style. Implementation-specific libraries may be provided to facilitate communication between the physical procesors that are independently executing this code, but specification of such libraries is outside the scope of HPF.

7 Storage and Sequence Association

7.1 Storage Associations

In general, the physical storge units or storage order for data object cannot be specified. However, Fortran 90's COMMON, EQUIVALENCE, and SEQUENCE statements provide sufficient control over the order and layout of storage units to permit data to share storage units:

- COMMON statements provides the primary means of sharing data between units.
- EQUIVALENCE statement provides a means whereby two or more objects can share the same storage units.
- SEQUENCE statement defines a storage order for structures permitting structures to appear in common blocks and be equivalenced.

The model of storage association is a single linearly addressed memory, based on the traditional single address space, single memory unit architecture. This model can cause severe inefficiencies on architecures where storage for variables is mapped.

HPF modifies the model of storage associations, and, as a result, conforming Fortran77 and Fortran90 may be not conforming HPF.

On the other hand, HPF introduces SEQUENCE directive to allow a user to declare explicitely that variables or COMMON block are to be treated by the compiler as *sequential* for which Fortran90 storage association rules apply. Since some implementations may supply an optional compilation environment where SEQUENCE directives is applied by default to all variables and COMMON blocks, HPF defines also NO SEQUENCE directive.

7.2 HPF Storage Associations Rules

- COMMON blocks are either sequential or nonsequential. By default they are nonsequential. They may be declared as sequential using explicit HPF direcitve SEQUENCE. *Note that some implementations may suppply an optional compilation environment where* sequential *is the default. In such a case user may explicitly declare COMMON block as* nonsequential *using HPF directive*NO SEQUENCE.
- In essence, variables in a nonsequential COMMON block can be mapped using HPF ALIGN and DISTRIBUTE directives as long as the components of the COMMON block are the same in every scoping unit (subroutine, function, etc.) that declares the COMMON block. A nonsequential variable in any occurence of the COMMON block must be nonsequential with identical type, shape, and mapping attributes in every occurence of the COMMON block. See more precise formulation below.
- Variables involved in an EQUIVALENCE statements may be mapped only by the mechanism of declaring a rank-one array to cover exactly the aggregate variable group and mapping that array.
- A sequential COMMON block has a single common block storage sequence as defined in Fortran90 standard. All variables that appear in a sequential COMMON block are *sequential* variables. A sequential variable cannot be explicitly mapped (by HPF ALIGN or DISTRIBUTE directives) unless it is a scalar or rank 1 array that is an aggregate cover
- No explicit mapping may be given for a component of a derived type having the Fortran 90 SEQUENCE attribute.

An *aggregate variable group* is a collection of variables whose individual storage sequences are parts of a single storage sequence. Variables associated by EQUIVALENCE statements or by combination of EQUIVALENCE and COMMON statements form an *aggregate variable group*. The variables of a sequential COMMON block form a single *aggregate variable group*. The size of an *aggregate variable group* in the number of storage units in the group's storage sequence (as defined in Fortran90)

If there is a member in an agregate variable group whose storage sequence is totally associated with the storage sequence of the aggregate variable group, that variable is called an *aggregate cover*. Example:

```
COMMON /F00/ A(100), B(100), C(100)
DIMENSION, REAL(200) :: Z
EQUIVALENCE (A(1), Z(1))
```

Z is an *aggregate cover* for aggregate variable group (A,B).

A COMMON block contains a sequence of *components*. Each *component* is either anaggregate variable group, or a variable that is not a member of any aggregate variable group. Example:

```
COMMON /F00/ A(100), B(100), C(100), D(100), E(100)
DIMENSION, REAL(100) :: X, Y
EQUIVALENCE (A(51), X(1)) (C(80), Y(1))
```

COMMON block /FOO/ has two components: (A,B) and (C,D,E). There is no aggregate cover in this example.

Sequential COMMON blocks contain a single component. Nonsequential COMMON blocks may contain several components that may be nonsequential or sequential variables or aggregate variable groups.

If a COMMON is *nonsequential*, then all of the following must hold:

- Every occurence of the COMMON block has the same number of components with each corresponding component having a storage sequence of exactly the same size.
- If a component is a nonsequential variable in any occurrence of the COMMON block, then it must be nonsequential with identical type, shape, and mapping attributes in every occurence of the COMMON block.
- If a component is sequential and explicitly mapped in any occurence of the COMMON block, then it must be sequential and explicitly mapped with identical mapping attributes in every occurrence of the COMMON block. In addition, the type and shape of the explicitly mapped variable must be identical in all occurences; and
- Every occurence of the COMMON block must be nonsequential.

Variables are either *sequential* or *nonsequential*. A variable is *sequential* if and only if any of the following holds:

- it appears in a sequential COMMON block
- it is a member of an aggregate variable group
- it is an assumed-size array
- it is a component of a derived type with the Fortran90 SEQUENCE attribute
- it is declarewd to be sequential in an HPF SEQUENCE directive

7.3 Sequence Associations

Sequence association is a special form of argument association that applies to character, array, and sequence structure arguments.

For array arguments the fundamental rule in Fortran90 is that the shapes of an actual argument and its associated dummy argument must be the same. To make this simple rule viable and to make passing array sections possible, a new type of dummy aragument was introduced in Fortran90: *assumed-shape* dummy argument.

But, alas, Fortran77 does not have assumed-shape arrays and therefore does not offer the array argument simplicity they provide. Moreover, being array element sequence oriented, the Fortran77 array argument association mechanisms are geared towards associating array element sequences rather than associating array objects. Fortran90, in order to be completely upward compatible with Fortran77, provides these separete mechanisms that are natural only in systems with a linearly addressed memory.

HPF modifies Fortran90 sequence associations rules. In a nutshell, a distributed array can be passed to a subprogram only if acutual and dummy arguments are conformable, that is, they have the same shape. Otherwise both actual and dummy arguments must be declared sequential. For a more precise description see below. As a result, conforming Fortran77 and Fortran90 programs may be not conforming HPF.

7.4 HPF Sequence Associations Rules

When an array element or the name of an assumed-size array is used as an actual argument, the associatead dummy argument must be a scalar or specified to be a sequential array. An array-element designator of a nonsequinetial array must not be associated with a dummy array argument.

When an actual argument is an array or array section and the corresponding dummy argument differs from the actual argument in shape, then the dummy argument must be declared sequential and the actuall array argument must be sequential.

A variable of type character (scalar or array) is nonsequential if it conforms to the requirements of the definition of a nonsequential variable. If the lenght of an explicit-lenght character dummy argument differs from the lenght of the actual argument, then both the actual and dummy arguments must be sequentinal.

8 Subset HPF

At the time when specification of the HPF was being agreed upon (1992) there were no commercial Fortran90 implementations on the market, except for a few source-to-source, Fortran90 to Fortan77 or C, translators. In addition, some advanced HPF features did not seem to be easily implementable, either. Many vendors expressed their concern

about possibility of implemetation of HPF in a timely fashion. This inspired an idea of defining a subset of HPF. The subset is a mandatory minimum of Fortran90 and HPF features to be supported by a language processor in order to earn the name of a HPF compiler. This way it was possible to develop early implemenations of HPF that provide portable, basic set of tools for developing data parallel programs in Fortran.
(Note that implementators are allowed to support additional Fortran90 and HPF features, and these features may be not portable at this time).

8.1 Fortran 90 Features in Subset HPF

- All Fortran77 standard conforming featurs, except for storage and sequence associations.
- DO WHILE, END DO, IMPLICIT NONE, INCLUDE
- Arithmetic and logical array features:
  - array sections (subsript triple notation, vector-valued subsripts)
  - array constructors limited to one level of implied DO
  - array assignment
  - masked array assignment (WHERE and WHERE...ELSEWHERE)
  - arithmetic and logical operations on whole arrays and array sections
  - array-valued external functions
  - automatic arrays
  - ALLOCATABLE arrays and ALLOCATE and DEALOCATE statements
  - assumed-shape arrays
- Type declaration statements TYPE, TARGET and POINTER *(except for kind-selector and access-spec in TYPE declaration)*.
- Attribute specifications statements: ALLOCATABLE, INTENT, OPTIONAL, PARAMETER, SAVE.
- Procedure features: INTERFACE blocks *(with no generic-spec or module-procedure-stmt)*, optional arguments and keyword argument passing.
- All intrinsic procedures are included in Subset HPF **execpt for**:
  - character computation functions: ADJUSTL, ADJUSTR, LEN_TRIM, REPEAT, SCAN, TRIM, VERIFY
  - bit inquiry function: BIT_SIZE
  - pointer association inquiry function: ASSOCIATED

- numeric inquiry functions: DIGITS, EPSILON, EXPONENT, FRACTION, HUGE, MAXEXPONENT, MINEXPONENT, NEAREST, PRECISION, RADIX, RANGE, RRSPACING, SCALE, SETEXPONENT, SPACING, TINY, TRANSFER
- conversion functions: ACHAR, IACHAR
- kind functions: KIND, SELECTED_INT_KIND, SELECTED_REAL_KIND

The following functions are part of Subset HPF as they are defined by Fortran77 (thus Fortran90 extensions are excluded form Subset HPF): CHAR, ICHAR, INDEX, LEN, LGE, LGT, LLE, LLT, LOGICAL
The following functions are part of Subset HPF, however, with the contraint that an optional agrument expressions DIM are initialization expressions and hence deliver a known shape at compile time: ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM, LBOUND, SIZE, UBOUND, SPREAD, CSHIFT, EOSHIFT, MACLOC, MINLOC.

- Syntax improvements: long (31) character names, lower case letters, use of "_" and "!" initiated comments, both full line and trailing.

8.2 HPF Features Not in Subset HPF

- REALIGN, REDISTRIBUTE, and DYNAMIC directives
- INHERIT directive used with transcriptive form of the DISTRIBUTE directive such as "DISTRIBUTE A * ONTO *"
- PURE function attribute
- FORALL...ENDFORALL construct
- The HPF library and he HPF_LIBRARY module
- Actual argument expressions corresponding to optional DIM arguments to the Fortran90 MAXLOC and MINLOC intrinsic functions that are not initialization expressions

9 More About HPF

It is impossible to touch all aspects of HPF in a such short tutorial. Hopefully, it provides enough information to get started with HPF, or just learn what HPF is all about. An HPF programmer definitely must refer to **the language specification** (href=ftp://ftp.npac.syr.edu/HPFF/hpf-v10-final.ps.Z or

href=http://www.netlib.org/hpf/index.html) and vendor's manuals for necessary details. A book "The High Performance Fortran Handbook" by C.H.Koebel, D.B.Loveman, R.S.Schreiber, G.L.Steele, and M.E.Zosel, MIT Press 1994 is recommended as well.

HPF is a superset of Fortran90, therefore a knowledge of this language is necessary to understand HPF. In this tutorial only selected Fortran90 features are discussed. There are several books describing Fortran90, my favorite is "Fortran 90 Handbook, Complete ANSI/ISO Reference" by J.C.Adams, W.S.Brainerd, J.T.Martin, B.T.Smith and J.L.Wagener, McGraw_Hill Book Company, 1992.

The current version of HPF, sometimes referred to as HPF-1, is just the first step to define a comprehensive language for parallel machines. The HPF Forum continues its effort, and the current status can be found in a web page http://www.erc.msstate.edu/hpff/home.html .

There are ongoing efforts to build a suite of applications written in HPF. One of those is available now, see http://www.npac.syr.edu/NPAC1/PUB/haupt/parkbench.html. The Low Level HPF Compiler Benchmark Suite which is a part of the PARKBENCH suite (http://www.epm.ornl.gov/~walker/parkbench).

Last but not least, it is also worth mentioning the HPF Application Evaluation project carried on at NPAC at Syracuse University (href=http://www.npac.syr.edu/hpfa) .