High Performance Fortran (HPF)

A set of extensions and modifications to the established International Standard for Fortran (ISO/IEC 1539:1991(E) and ANSI X3.198-1992), informally referred to as "Fortran 90."

Goals and Scope of High Performance Fortran
To define language extensions and feature selection for Fortran supporting:

- Data parallel programming (defined as single threaded, global name space, and loosely synchronous parallel computation);
- Top performance on MIMD and SIMD computers with non-uniform memory access costs (while not impeding performance on other machines); and
- Code tuning for various architectures.

High Performance Fortran Language Specification
High Performance Fortran Forum
May 3, 1993
Version 1.0

Fortran 90

Operations for processing whole arrays and subarrays (array sections) are included in the language for two principal reasons: (1) these features provide a more concise and higher level language that will allow programmers more quickly and reliably to develop and maintain scientific/engineering applications, and (2) these features can significantly facilitate optimization of array operations on many computer architectures.
— Fortran Standard (page xiii)

Other features of Fortran 90 that improve upon the features provided in FORTRAN 77 include:

- Additional storage classes of objects:
  Allocatable

  ```
  REAL, DIMENSION (:,:), ALLOCATABLE :: A, B
  ```

  Assumed-shape objects

  ```
  SUBROUTINE S (D)
      INTEGER, DIMENSION (:, :, :) :: D
  ```

  Pointer facility

  ```
  REAL, DIMENSION (:), POINTER :: V
  ...
  V => REAL_ARRAY(4,:)
  ...
  V = 0
  ```

- Module facilities:
  User-defined data types and structures

  Defined operators on those types

  Generic procedures

  Interface blocks

- Intrinsic procedures:
  Construction and transformation of arrays
  `RESHAPE, CSHIFT, TRANSPOSE, SPREAD`

  Numerical accuracy intrinsic procedures
  `MAX-EXPONENT, PRECISION`

  Bit manipulation intrinsics
  `IOR, IAND, BTEST`

Fortran 90 Binding

HPF is an extension of Fortran 90. The array calculation and dynamic storage allocation features of Fortran 90 make it a natural base for HPF. The new HPF language features fall into four categories with respect to Fortran 90:

- New directives;
- New language syntax;
- Library routines; and
- Language restrictions.

New Features in High Performance Fortran

HPF extends Fortran 90 in several areas, including:

- Data distribution features;
  `ALIGN`: collocation of data

  `DISTRIBUTE`: partitioning of data among memory regions or abstract processors

- Data parallel execution features;
  `FORALL` construct expresses assignments to sections of arrays

  `INDEPENDENT` directive asserts that the statements in a particular section of code do not exhibit any sequentializing dependences

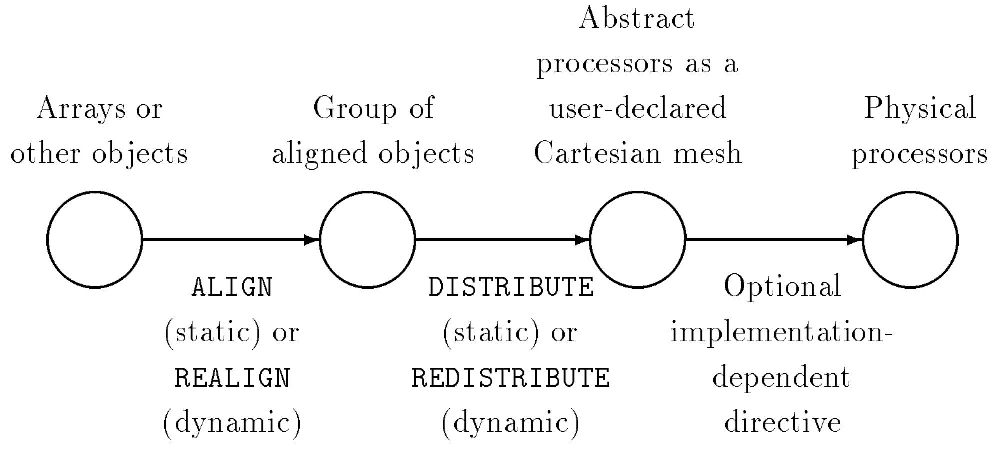
- Extended intrinsic functions and standard library;

- `EXTRINSIC` procedures;
  an explicit interface to procedures written in other paradigms

- Changes in sequence and storage association.
  to maintain compatibility with Fortran 90

Model

- A two-level mapping of data objects to memory regions, referred to as "abstract processors."
- Data objects (typically array elements) are first *aligned* relative to one another; this group of arrays is then *distributed* onto a rectilinear arrangement of abstract processors.
- The implementation then uses the same number, or perhaps some smaller number, of physical processors to implement these abstract processors. This mapping of abstract processors to physical processors is language-processor dependent.
- Underlying assumptions: an operation on two or more data objects is likely to be carried out much faster if they all reside in the same processor, and these can be done concurrently if they are on different processors.



Scalar Communication Example 1

```
      REAL a(1000), b(1000), c(1000), x(500), y(0:501)
      INTEGER inx(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b, inx
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: c
!HPF$ ALIGN x(i) WITH y(i+1)
      ...
      a(i) = b(i)                     ! Assignment 1
      x(i) = y(i+1)                   ! Assignment 2
      a(i) = c(i)                     ! Assignment 3
      a(i) = a(i-1) + a(i) + a(i+1)   ! Assignment 4
      c(i) = c(i-1) + c(i) + c(i+1)   ! Assignment 5
      x(i) = y(i)                     ! Assignment 6
      a(i) = a(inx(i)) + b(inx(i))    ! Assignment 7
```

Scalar Communication Example 2

```
      REAL a(1000), b(1000), c(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: a, b, c
      ...
      a(i) = b(i+2)                     ! Statement 1
      b(i) = c(i+3)                     ! Statement 2
      b(i+2) = 2 * a(i+2)               ! Statement 3
      c(i) = a(i+1) + b(i+2) + c(i+3)   ! Statement 4
```

The minimum total inherent communication in this program fragment is four array elements. Some compilation strategies may produce communication for element `c(i+3)` in the last statement.

Aggregate Communication Example 1

```
      REAL a(1000), b(1000), c(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: c
      ...
      FORALL ( i = 1:1000 ) a(i) = b(i)       ! Forall 1
      FORALL ( i = 1:1000 ) a(i) = c(i)       ! Forall 2

      ! Forall 3
      FORALL ( i = 2:999 ) a(i) = a(i-1) + a(i) + a(i+1)

      ! Forall 4
      FORALL ( i = 2:999 ) c(i) = c(i-1) + c(i) + c(i+1)
```

- `FORALL` statement conceptually evaluates its right-hand side for all values of its indexes, then assigns to the left-hand side for all index values.
- In Forall 1, no communication.
- 900 array elements to be transferred. to update the values of `a`, each processor requires data from every other processor, i.e. all-to-all communication.
- In Forall 3, the array references are all mapped to the same processor except for the first and last values of `i` on each processor. Two array elements per processor (except `procs(1)` and `procs(10)`) need to be transferred, i.e. a simple shift communication pattern.
- In Forall 4, a massive shift communication pattern with wraparound.

Same Computation using Array Assignments

```
      REAL a(1000), b(1000), c(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: c
      ...
      ! Assignment 1 (equivalent to Forall 1)
      a(:) = b(:)

      ! Assignment 2 (equivalent to Forall 2)
      a(1:1000) = c(1:1000)

      ! Assignment 3 (equivalent to Forall 3)
      a(2:999) = a(1:998) + a(2:999) + a(3:1000)

      ! Assignment 4 (equivalent to Forall 4)
      c(2:999) = c(1:998) + c(2:999) + c(3:1000)
```

Aggregate Communication Example 2

Some array intrinsics have inherent communication costs.

```
      REAL a(1000), b(1000), scalar
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b
      ...
      ! Intrinsic 1
      scalar = SUM( a )

      ! Intrinsic 2
      a = SPREAD( b(1), DIM=1, NCOPIES=1000 )

      ! Intrinsic 3
      a = CSHIFT(a,-1) + a + CSHIFT(a,1)
```

Aggregate Communication Example 3

Nonaligned data generate need for communication.

```
      REAL a(1000), c(100,100), d(100,100)
!HPF$ PROCESSORS procs(10)
!HPF$ ALIGN c(i,j) WITH d(j,i)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a
!HPF$ DISTRIBUTE (BLOCK,*) ONTO procs :: d
      ...
      a(1:200) = a(1:200) + a(2:400:2)
      c = c + d
```

Communication and Parallelism Tradeoff 1

```
      REAL x(100), y(100)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs:: x, y
      ...
      DO k = 3, 98
         x(k) = y(k) * (x(k-1) + x(k) + x(k+1)) / 3.0
         y(k) = x(k) + (y(k-1) + y(k-2) + y(k+1) + y(k+2)) / 4.0
      ENDDO
```

Little inherent communication, but the computation will run sequentially.

Communication and Parallelism Tradeoff 2

In contrast:

```
      REAL x(100), y(100), z(100)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs:: x, y, z
      ...
!HPF$ INDEPENDENT
      DO k = 3, 98
         x(k) = y(k) * (z(k-1) + z(k) + z(k+1)) / 3.0
         y(k) = x(k) + (z(k-1) + z(k-2) + z(k+1) + z(k+2)) / 4.0
      ENDDO
```

Communication and Parallelism Tradeoff 3

Assignment of work to processors may itself require communication.

```
      INTEGER indx(1000), inv(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: indx, inv
      ...
      FORALL ( j = 1:1000 ) inv(indx(j)) = j**2
```

`indx` must be a permutation of the integers from 1 to 1000 in order for the FORALL to be well-defined.

Where will the results be stored? The processor owning element `inv(indx(j))` depends on the values stored in `indx`. Two possible implementations of this are:

- Each processor calculates the squares for elements of `indx` that it owns and performs a scatter operation to communicate those values to the elements of `inv` where the final results are stored.
- Each processor determines the owner of `inv(indx(j))` for all elements of `indx` that it owns and notifies those processors. Each processor then computes the right-hand side for all elements for which it received notification.

Communication and Parallelism Tradeoff 4

Parallelism conflicts with the distribution of data.

```
      REAL a(1000,1000), b(1000,1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK,*) ONTO procs :: a, b
      ...
      DO i = 2, 1000
         a(i,:) = a(i,:) - (b(i,:)**2)/a(i-1,:)
      ENDDO
```

Exploitation of any of the potential parallelism will require scattering the data to other processors.

- Redistribute `a` and `b` before the `DO` loop to achieve the effect of

  ```
  !HPF$ DISTRIBUTE (*,BLOCK) ONTO procs :: a, b
  ```

  Redistribute back to the original distributions after the `DO` loop.
- Group the columns of `a` into blocks, then operate on the blocks separately.
- Execute the vector operations sequentially.

Communication and Parallelism Tradeoff 5

Ordering of array operations.

```
      REAL, DIMENSION(100,100) :: x, y, z
!HPF$ ALIGN WITH x :: y, z
      ...
      x = TRANSPOSE(y) + TRANSPOSE(z) + x
```

Despite two occurrences of the `TRANSPOSE` intrinsic, an optimizing compiler might implement this as:

```
      REAL, DIMENSION(100,100) :: x, y, z, t1
!HPF$ ALIGN WITH x :: y, z, t1
      ...
      t1 = y + z
      x = TRANSPOSE(t1) + x
```

with only one use of transposition.

Communication and Parallelism Tradeoff 6

Where will the result of an intermediate array expression be stored?

```
      REAL a(1000), b(1000), c(1000), d(1000)
      INTEGER ix(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs:: a, b, c, d, ix
      ...
      a = b(ix) + c(ix) + d(ix)
```

Two alternative implementation strategies:

- Evaluate each element of the right-hand side on the processor where it will be stored. This strategy potentially requires fetching three values (the elements of `b`, `c`, and `d`) for each element computed. It always uses the maximum parallelism of the machine.
- Evaluate each element of the right-hand side on the processor where the corresponding elements of `b(ix)`, `c(ix)`, and `d(ix)` are stored. Ignoring set-up costs, this potentially communicates one result for each element computed. If the values of `ix` are evenly distributed, then it also uses the maximum machine parallelism.

DISTRIBUTE Directives

The `DISTRIBUTE` directive specifies a mapping of data objects to abstract processors in a processor arrangement.

```
      REAL SALAMI(10000)
!HPF$ DISTRIBUTE SALAMI(BLOCK)
```

The block size may be specified explicitly:

```
      REAL WEISSWURST(10000)
!HPF$ DISTRIBUTE WEISSWURST(BLOCK(256))
```

There must be at least $\lceil 10000/256 \rceil = 40$ abstract processors.
The fortieth processor will contain a partial block of only 16 elements: `WEISSWURST(9985:10000)`.)

HPF also provides a cyclic distribution format:

```
      REAL DECK_OF_CARDS(52)
!HPF$ DISTRIBUTE DECK_OF_CARDS(CYCLIC)
```

Size of block and cyclic distribution

Define the ceiling division function `CD(J,K) = (J+K-1)/K` (using Fortran integer arithmetic with truncation toward zero.)
Define the ceiling remainder function `CR(J,K) = J-K*CD(J,K)`.
Let d be the size of a *distributee* in a certain dimension.
let p be the size of the processor arrangement in the corresponding dimension.
For simplicity, assume all dimensions have a lower bound of 1.

BLOCK(m) means that a *distributee* position whose index along that dimension is j is mapped to an abstract processor whose index along the corresponding dimension of the processor arrangement is CD(j,m) (note that $m \times p \geq d$ must be true), and is position number m+CR(j,m) among positions mapped to that abstract processor. The first *distributee* position in abstract processor k along that axis is position number 1+m*(k-1).
BLOCK by definition means the same as BLOCK(CD(d,p)).

`CYCLIC(`m`)` means that a *distributee* position whose index along that dimension is j is mapped to an abstract processor whose index along the corresponding dimension of the processor arrangement is `1+MODULO(CD(`j`,`m`)-1,`p`)`. The first *distributee* position in abstract processor k along that axis is position number `1+`m`*(`k`-1)`.

`CYCLIC` by definition means the same as `CYCLIC(1)`.

`CYCLIC(`m`)` and `BLOCK(`m`)` imply the same distribution when $m \times p \geq d$, but `BLOCK(`m`)` additionally asserts that the distribution will not wrap around in a cyclic manner, which a compiler cannot determine at compile time if m is not constant.

Note that `CYCLIC` and `BLOCK` (without argument expressions) do not imply the same distribution unless $p \geq d$, a degenerate case in which the block size is 1 and the distribution does not wrap around.

Suppose that we have 16 abstract processors and an array of length 100:

```
!HPF$ PROCESSORS SEDECIM(16)
      REAL CENTURY(100)
```

Distributing the array `BLOCK` (which in this case would mean the same as `BLOCK(7)`):

```
!HPF$ DISTRIBUTE CENTURY(BLOCK) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	8	15	22	29	36	43	50	57	64	71	78	85	92	99	
2	9	16	23	30	37	44	51	58	65	72	79	86	93	100	
3	10	17	24	31	38	45	52	59	66	73	80	87	94		
4	11	18	25	32	39	46	53	60	67	74	81	88	95		
5	12	19	26	33	40	47	54	61	68	75	82	89	96		
6	13	20	27	34	41	48	55	62	69	76	83	90	97		
7	14	21	28	35	42	49	56	63	70	77	84	91	98		

Distributing the array `BLOCK(8)`:

```
!HPF$ DISTRIBUTE CENTURY(BLOCK(8)) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	9	17	25	33	41	49	57	65	73	81	89	97			
2	10	18	26	34	42	50	58	66	74	82	90	98			
3	11	19	27	35	43	51	59	67	75	83	91	99			
4	12	20	28	36	44	52	60	68	76	84	92	100			
5	13	21	29	37	45	53	61	69	77	85	93				
6	14	22	30	38	46	54	62	70	78	86	94				
7	15	23	31	39	47	55	63	71	79	87	95				
8	16	24	32	40	48	56	64	72	80	88	96				

Distributing the array `BLOCK(6)` is not HPF-conforming because $6 \times 16 < 100$.

Distributing the array `CYCLIC` (which means exactly the same as `CYCLIC(1)`):

```
!HPF$ DISTRIBUTE CENTURY(CYCLIC) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100												

Distributing the array `CYCLIC(3)`:

```
!HPF$ DISTRIBUTE CENTURY(CYCLIC(3)) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	4	7	10	13	16	19	22	25	28	31	34	37	40	43	46
2	5	8	11	14	17	20	23	26	29	32	35	38	41	44	47
3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48
49	52	55	58	61	64	67	70	73	76	79	82	85	88	91	94
50	53	56	59	62	65	68	71	74	77	80	83	86	89	92	95
51	54	57	60	63	66	69	72	75	78	81	84	87	90	93	96
97	100														
98															
99															

ALIGN Directive

```
!HPF$ ALIGN A(:,*,K,:,:,*) WITH B(31:,:,K+3,20:100:3)
```

may be transformed into its equivalent

```
!HPF$ ALIGN A(I,J,K,L,M,N) WITH B(I-LBOUND(A,1)+31,       &
!HPF$         L-LBOUND(A,4)+LBOUND(B,2),K+3,(M-LBOUND(A,5))*3+20)
```

with the attached requirements

```
SIZE(A,1) .EQ. UBOUND(B,1)-30
  SIZE(A,4) .EQ. SIZE(B,2)
 SIZE(A,5) .EQ. (100-20+3)/3
```

Equivalent ALIGN Examples

```
!Second axis of X is collapsed
!HPF$ ALIGN X(:,*) WITH D1(:)
!HPF$ ALIGN X(J,*) WITH D1(J)
!HPF$ ALIGN X(J,K) WITH D1(J)

!Replicated representation along second axis of D3
!HPF$ ALIGN X(:,:) WITH D3(:,*,:)
!HPF$ ALIGN X(J,K) WITH D3(J,*,K)

!Transposing two axes
!HPF$ ALIGN X(J,K) WITH D2(K,J)
!HPF$ ALIGN X(J,:) WITH D2(:,J)
!HPF$ ALIGN X(:,K) WITH D2(K,:)
!But there isn't any way to get rid of *both* index variables;
! the subscript-triplet syntax alone cannot express transposition.
```

```
!Reversing both axes
!HPF$ ALIGN X(J,K) WITH D2(M-J+1,N-K+1)
!HPF$ ALIGN X(:,:) WITH D2(M:1:-1,N:1:-1)

!Simple case
!HPF$ ALIGN X(J,K) WITH D2(J,K)
!HPF$ ALIGN X(:,:) WITH D2(:,:)
!HPF$ ALIGN (J,K) WITH D2(J,K):: X
!HPF$ ALIGN (:,:) WITH D2(:,:):: X
!HPF$ ALIGN WITH D2:: X
```

PROCESSORS Directive

```
!HPF$ PROCESSORS P(N)
!HPF$ PROCESSORS Q(NUMBER_OF_PROCESSORS()),       &
!HPF$            R(8,NUMBER_OF_PROCESSORS()/8)
!HPF$ PROCESSORS BIZARRO(1972:1997,-20:17)
!HPF$ PROCESSORS SCALARPROC
```

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

The shape may also be specified with the DIMENSION attribute:

```
!HPF$ PROCESSORS :: RUBIK(3,3,3)
!HPF$ PROCESSORS, DIMENSION(3,3,3) :: RUBIK

!HPF$ PROCESSORS, DIMENSION(3,3,3) ::      &
!HPF$            RUBIK, RUBIKS_REVENGE(4,4,4), SOMA
```

Here RUBIKS_REVENGE is $4 \times 4 \times 4$ while RUBIK and SOMA are each $3 \times 3 \times 3$. (By the rules enunciated above, however, such a statement may not be completely portable because no HPF language processor is required to handle shapes of total sizes 27 and 64 simultaneously.)

Suggestions for Machine-Dependent Directives

```
!Declaration for multiprocessor by ABC Corporation
!ABC$ PHYSICAL PROCESSORS(8)
!Declaration for mpp by XYZ Incorporated
!XYZ$ PHYSICAL PROCESSORS(65536)
!Declaration for hypercube machine by PDQ Limited
!PDQ$ PHYSICAL PROCESSORS(2,2,2,2,2,2,2,2,2,2)
!Declaration for two-dimensional grid machine by TLA GmbH
!TLA$ PHYSICAL PROCESSORS(128,64)
!One of the preceding might affect the following
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
```

User specified Mapping

```
!PDQ$ PHYSICAL PROCESSORS(2,2,2,2,2,2,2,2,2,2,2,2,2)
!HPF$ PROCESSORS G(8,64,16)
!PDQ$ MACHINE LAYOUT G(:GRAY(0:2),:GRAY(6:11),:BINARY(3:5,12))
```

TEMPLATE Directive

```
!HPF$ TEMPLATE A(N)
!HPF$ TEMPLATE B(N,N), C(N,2*N)
!HPF$ TEMPLATE DOPEY(100,100),SNEEZY(24),GRUMPY(17,3,5)

!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) ::    &
!HPF$                              WHINEY(64,64),MOPEY(128,128)
!HPF$ TEMPLATE, DIMENSION(91,91) :: BORED,WHEEZY,PERKY
```

Templates are useful in the particular situation where 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. For example, one might want four $N \times N$ arrays aligned to the four corners of a template of size $(N+1) \times (N+1)$:

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) :: EARTH(N+1,N+1)
      REAL, DIMENSION(N,N) :: NW, NE, SW, SE
!HPF$ ALIGN NW(I,J) WITH EARTH( I , J )
!HPF$ ALIGN NE(I,J) WITH EARTH( I ,J+1)
!HPF$ ALIGN SW(I,J) WITH EARTH(I+1, J )
!HPF$ ALIGN SE(I,J) WITH EARTH(I+1,J+1)
```

Templates may also be useful in making assertions about the mapping of dummy arguments.

Subprogram Interfaces

```
      LOGICAL FRUG(128),TWIST(128)
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ DISTRIBUTE (BLOCK) ONTO DANCE_FLOOR::FRUG,TWIST
      CALL TERPSICHORE(FRUG(1:40:3),TWIST(1:40:3))
```

The two array sections `FRUG(1:40:3)` and `TWIST(1:40:3)` are mapped onto abstract processors in the same manner:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1			25												
	10			34											
		19													
4			28												
	13			37											
		22													
7			31												
	16			40											

INHERIT Directive

Subroutine `TERPSICHORE` inherits the template for the second dummy but not the first:

```
      SUBROUTINE TERPSICHORE(FOXTROT,TANGO)
      LOGICAL FOXTROT(:),TANGO(:)
!HPF$ INHERIT TANGO
```

Therefore the template of `TANGO` is a copy of the 128 element template of the whole array `TWIST`. The template is mapped like this:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	9	17	25	33	41	49	57	65	73	81	89	97	105	113	121
2	10	18	26	34	42	50	58	66	74	82	90	98	106	114	122
3	11	19	27	35	43	51	59	67	75	83	91	99	107	115	123
4	12	20	28	36	44	52	60	68	76	84	92	100	108	116	124
5	13	21	29	37	45	53	61	69	77	85	93	101	109	117	125
6	14	22	30	38	46	54	62	70	78	86	94	102	110	118	126
7	15	23	31	39	47	55	63	71	79	87	95	103	111	119	127
8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128

`TANGO(I)` is aligned with element 3*I-2 of the template. But the template of `FOXTROT` has the same size 14 as `FOXTROT` itself. The actual argument, `FRUG(1:40:3)` is mapped to the 16 processors in this manner:

Abstract processor	Elements of FRUG
1	1, 2, 3
2	4, 5, 6
3	7, 8
4	9, 10, 11
5	12, 13, 14
6–16	none

It would be reasonable to understand the mapping of the template of `FOXTROT` to coincide with the layout of the array section:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1			9												
	4			12											
		7													
2			10												
	5			13											
		8													
3			11												
	6			14											

Not permitted in HPF!

Within subroutine TERPSICHORE it would be correct to make the descriptive assertion

```
!HPF$ DISTRIBUTE TANGO *(BLOCK)
```

but it would not be correct to declare

```
!HPF$ DISTRIBUTE FOXTROT *(BLOCK)                 !Nonconforming
```

```
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) ONTO DANCE_FLOOR::GURF(128)
!HPF$ ALIGN FOXTROT(J) WITH *GURF(3*J-2)
```

could be correctly included in TERPSICHORE to describe the layout of FOXTROT on entry to the subroutine without using an inherited template.

Pure Procedures

Consequence of Data Parallel Statements
Pure functions may be used in expressions in `FORALL` statements and constructs (but general functions cannot).

```
! This statement function is pure since it does not reference
! any other functions
REAL myexp
myexp(x) = 1 + x + x*x/2.0 + x*x*x/6.0
FORALL ( i = 1:n ) a(i) = myexp( a(i+1) )
...
! Intrinsic functions are always pure
FORALL ( i = 1:n ) a(i,i) = log( abs( a(i,i) ) )
```

Row-wise or column-wise operations

```
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))
```

Limited form of MIMD parallelism

```
REAL PURE FUNCTION f (x, i)
  REAL, INTENT(IN) :: x       ! associated with array element
  INTEGER, INTENT(IN) :: i    ! associated with array subscript
  IF (x > 0.0) THEN              ! content-based conditional
    f = x*x
  ELSE IF (i==1 .OR. i==n) THEN  ! subscript-based conditional
    f = 0.0
  ELSE
    f = x
  ENDIF
END FUNCTION

...

REAL a(n)
INTEGER i
  ...
FORALL (i=1:n)  a(i) = f( a(i), i)
```

Encapsulating Complex Operations

```
PURE INTEGER FUNCTION iter(x)
  COMPLEX, INTENT(IN) :: x
  COMPLEX xtmp
  INTEGER i
  i = 0
  xtmp = -x
  DO WHILE (ABS(xtmp).LT.2.0 .AND. i.LT.1000)
    xtmp = xtmp * xtmp - x
    i = i + 1
  END DO
  iter = i
END FUNCTION

  ...
FORALL (i=1:n, j=1:m)  ix(i,j) = iter(CMPLX(a+i*da,b+j*db))
```

INDEPENDENT Directive

`INDEPENDENT` can precede a `DO` loop or `FORALL` statement or construct.

```
!HPF$ INDEPENDENT
DO i = 2, 99
  a(i) = b(i-1) + b(i) + b(i+1)
END DO
```

The following example is equivalent in all respects to the first example.

```
!HPF$ INDEPENDENT
FORALL ( i=2:n ) a(i) = b(i-1) + b(i) + b(i+1)
```

Assertion by INDEPENDENT

```
!HPF$ INDEPENDENT
DO i=1, 100
  a(p(i)) = b(i)
END DO
```

This `INDEPENDENT` directive asserts that the array p does not have any repeated entries, therefore equivalent to the Fortran 90 statement

```
a(p(1:100)) = b(1:100)
```

NEW option

```
!HPF$ INDEPENDENT, NEW (j)
DO i = 2, 100, 2
  !HPF$ INDEPENDENT, NEW(vl, vr, ul, ur)
  DO j = 2 , 100, 2
    vl = p(i,j) - p(i-1,j)
    vr = p(i+1,j) - p(i,j)
    ul = p(i,j) - p(i,j-1)
    ur = p(i,j+1) - p(i,j)
    p(i,j) = f(i,j) + p(i,j) + 0.25 * (vr - vl + ur - ul)
  END DO
END DO
```

Without the `NEW` option on the j loop, neither loop would be independent, because an interleaved execution of loop iterations might cause other values of vl, vr, ul, and ur to be used in the assignment of $p(i,j)$ than those computed in the same iteration of the loop. The `NEW` option, however, specifies that this is not true if distinct storage units are used in each iteration of the loop. Using this implementation makes iterations of the loops independent of each other.

Visualizing INDEPENDENT

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

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

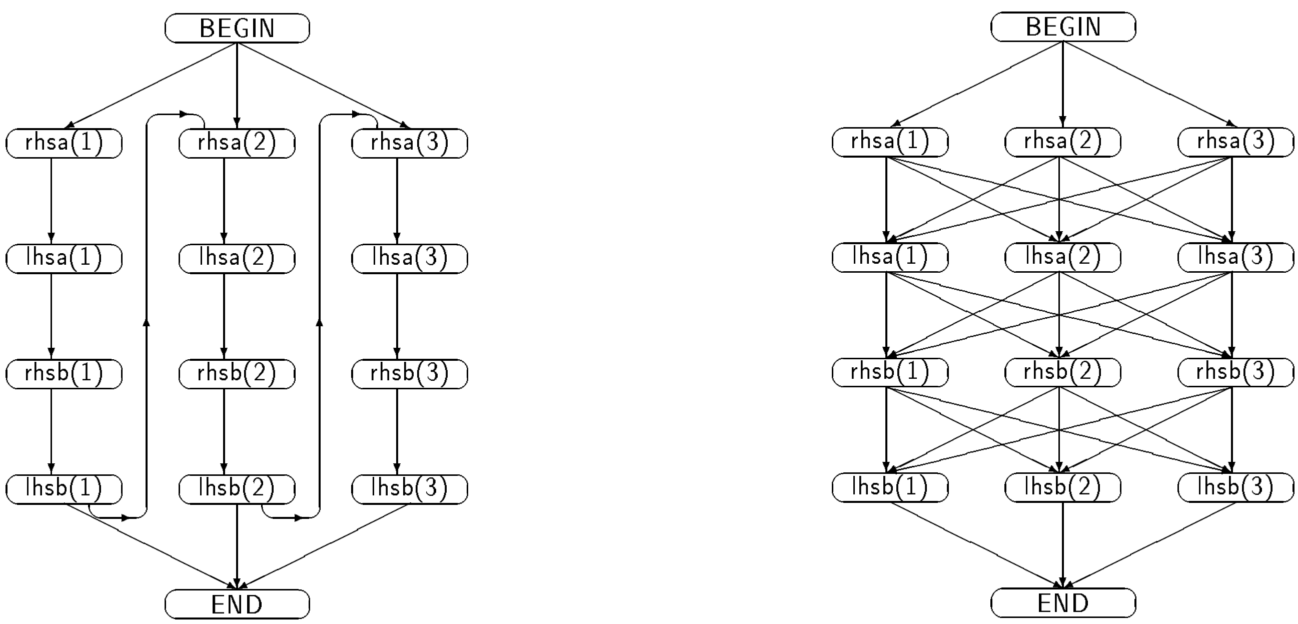


Figure 0.1: Dependences in `DO` and `FORALL` without `INDEPENDENT` assertions

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

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

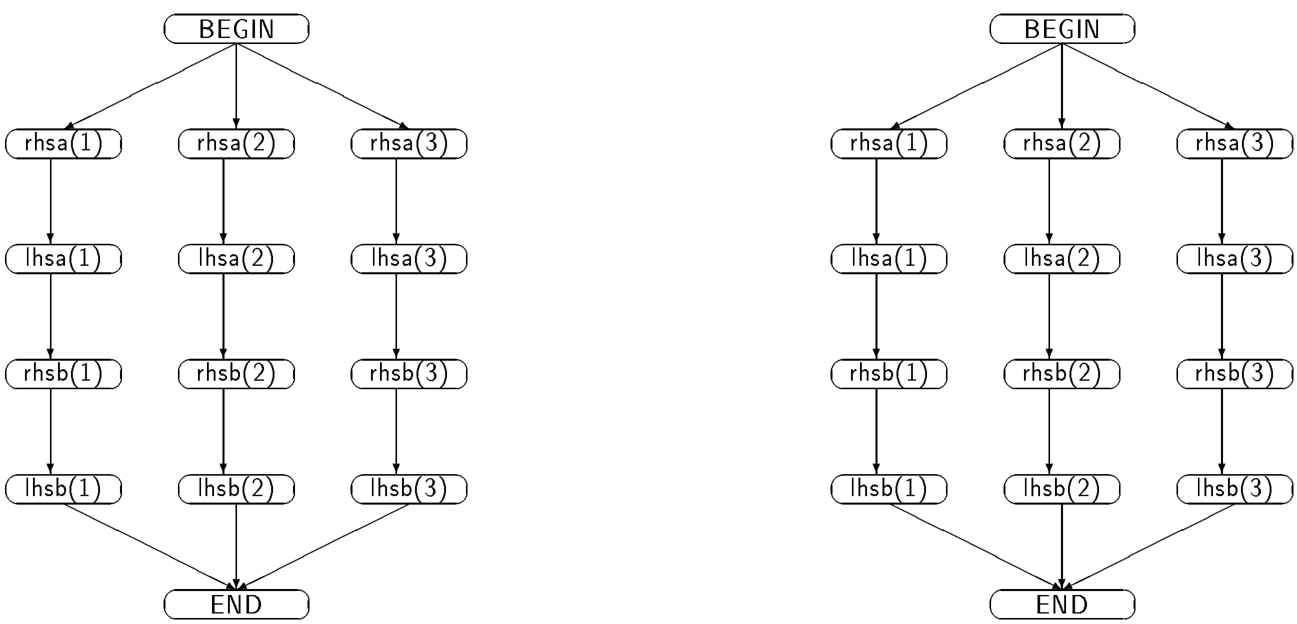


Figure 0.2: Dependences in `DO` and `FORALL` with `INDEPENDENT` assertions