High Performance Fortran

Juan Ángel Lorenzo del Castillo
Grupo de Arquitectura de Computadores
Departamento de Electrónica y Computación
Universidad de Santiago de Compostela

USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

CESGA

Introducción

- Nuevo paradigma de programación: *data parallel programming*
- Objetivo: promover el uso del paralelismo
- Lenguaje estándar: HPF
- Extiende Fortran 90
- Expresa el paralelismo de forma simple

Programa

- Programación *data-parallel*
- Introducción a Fortran 90
- Distribución y alineamiento de datos en HPF
- Directivas y construcciones paralelas en HPF
- Funciones intrínsecas y librería HPF
- Otros tópicos y desarrollos actuales

Programación data-parallel

Contenido

- Paradigmas de programación paralela
- Programación *data-parallel*
- El HPF Forum

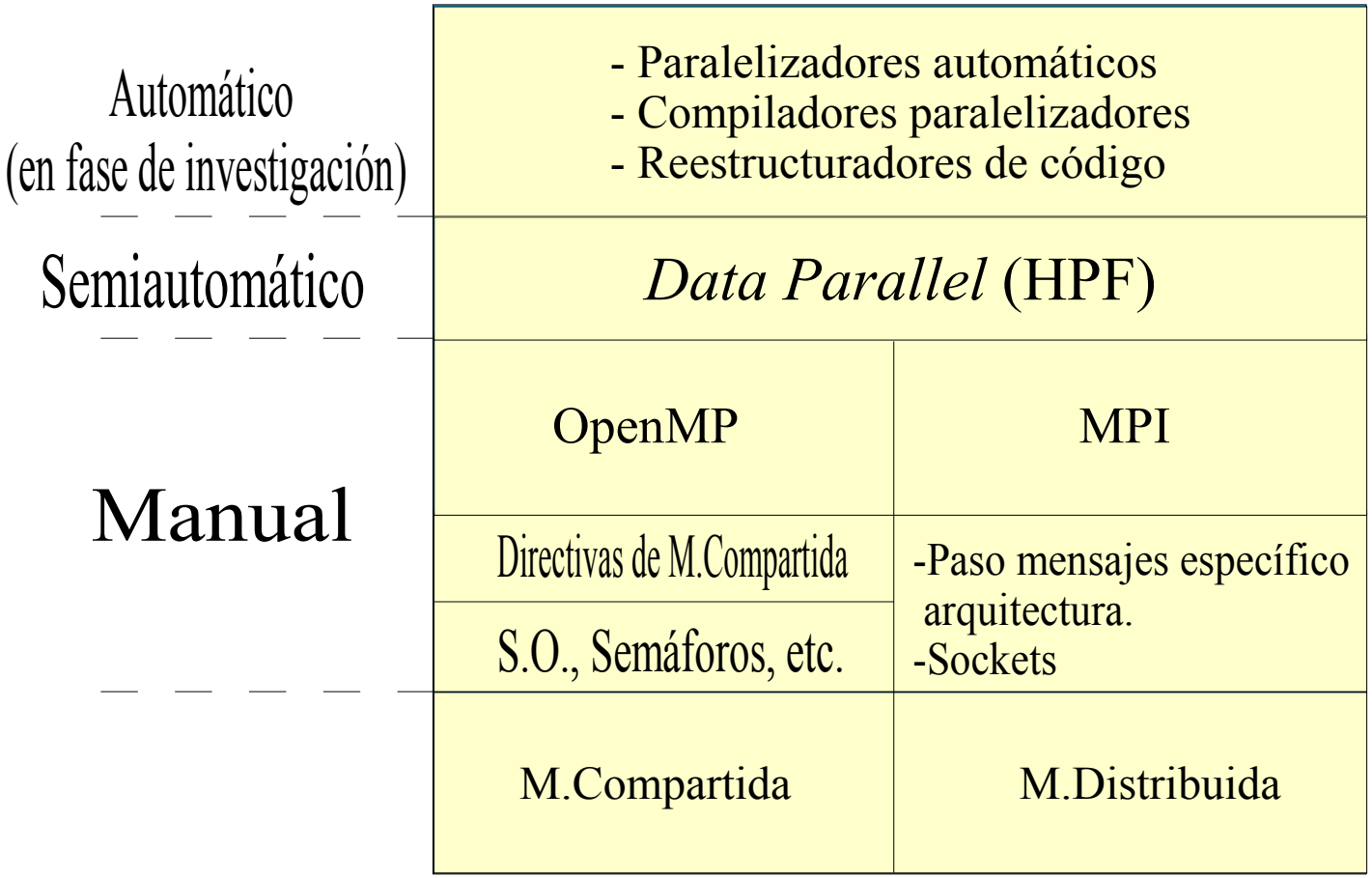
Paradigmas de programación paralela

- Sistemas de memoria compartida
  - Programa secuencial + directivas
  - Utiliza construcciones del SO (semáforos, barreras, *locks*, etc.)
  - Variables compartidas
- Sistemas de memoria distribuida
  - Lenguaje convencional + rutinas especiales
  - Variables privadas
  - Comunicación vía llamadas a subrutinas

Paradigmas de programación paralela

- Sistemas de memoria distribuida
  - Control completo del programador
  - Muy flexible pero...
    - Difícil de escribir y depurar
    - El programador es responsable de obtener alta eficiencia
  - Portabilidad mediante librerías estándar (PVM o MPI)

Programación *data-parallel*



-Comerciales: SUN, SGI

-No comerciales: Polaris, SUIF

Programación *data-parallel*

Programación en alto nivel

- Distribución de datos y comunicaciones hecha por el compilador
- Independiente de la arquitectura
- Características:
  - Control *single-threaded*
  - Espacio global de nombres
  - Sincronización débil (mayor que con MP)

Programación *data-parallel*

- Características
  - Paralelismo a través de operaciones sobre matrices de datos
  - Directivas del compilador
- Detalles de bajo nivel transferidos del programador al compilador $\Rightarrow$
  - Eficiencia: mayor dependencia del compilador
- Busca un mayor uso de sistemas paralelos

Programación *data-parallel*

Ventajas e Inconvenientes

- Ventajas: Facilidad de uso
  - Fácil de escribir (no paso de mensajes explícito)
  - Fácil de depurar (control *single-threaded*)
  - Portable: códigos viejos y nuevos
- Desventajas: Flexibilidad y control
  - Sólo adecuado a ciertos problemas
  - Control restringido: distribución de datos
  - Difícil de obtener máximo rendimiento
  - Necesidad de buenos compiladores

Programación *data-parallel*

- Útil para problemas con:
  - Datos en arrays grandes
  - Operaciones similares en cada elemento
  - Operaciones independientes
  - Problemas bien-balanceados de por sí
- Dificultades con problemas irregulares
  - Matrices dispersas
  - Problemas dinámicos

Programación *data-parallel*

Lenguajes *data-parallel*

- Muchas implementaciones
  - Fortran-Plus, MP Fortran, CM Fortran, C*, CRAFT, Fortran D, Vienna Fortran
- Problemas
  - Demasiados lenguajes y dialectos
  - Lenguajes específicos de cada máquina
  - Pérdida de portabilidad
- Necesidad de un estándar: Creación del HPF Forum.
  - Compilador *pghpf (Portland Group Compiler)*

El HPF Forum

Objetivos

- Diseñar un lenguaje que ofrezca
  - Programación *data-parallel*
  - Máximo rendimiento en todas las máquinas
- Objetivos adicionales
  - Simplicidad y portabilidad (código viejo y nuevo)
  - Fácil implementación de compiladores, rápida disponibilidad

El HPF Forum

El concepto de HPF

- Definición de HPF: Fortran 90 +
  - Directivas para distribución de datos
  - Construcciones que expresen paralelismo: **`FORALL`**
- Funcionamiento:
  - Cada procesador ejecuta el mismo programa (SPMD)
  - Cada procesador opera sobre parte del conjunto total de datos.
  - Las directivas HPF dicen qué procesador trabaja sobre qué datos

Resumen

- Computadores paralelos
- Paradigmas de programación:
  - Paso de mensajes
  - Paralelismo de datos
- High Performance Fortran
  - *High Performance Fortran Forum*
- *Portland Group Compiler:* pghpf

Introducción a Fortran 90

Introducción

- HPF incluye todo Fortran 90 (algunas características no soportadas en el *subset*).
- Fortran 90: Fortran 77 +
  - Operaciones con arrays
  - Funciones de control mejoradas
  - Tipos de datos derivados (estructuras)
  - Reserva dinámica de memoria y punteros
  - Nuevas funciones intrínsecas
  - Formato de escritura libre

Nueva sintaxis mejorada

```
PROGRAM hola_mundo
!Decimos hola
  PRINT *, "Hola mundo" !Imprime "Hola mundo"
END PROGRAM hola_mundo
```

- No restricciones posicionales
- Nombres de 31 caracteres (letras, dígitos y _)
- Comentarios comienzan por !
- No distingue mayúsculas y minúsculas
- Líneas de 132 caracteres como máximo

Nueva sintaxis mejorada

- Más de una sentencia por línea (uso de ;)

```
x = y + z; z = z - y; CALL rutina(x, y)
```

- Continuación de línea: &

```
CALL rutina(arg1, arg2, arg3, arg4, &
        arg5, arg6)
PRINT *, "Esto es un ejemplo de continua&
        &ción de una línea"
```

- 39 líneas de continuación como máximo

Tipos de datos

- Tipos intrínsecos (**INTEGER, REAL, CHARACTER, LOGICAL, COMPLEX**) y tipos derivados (definidos por el usuario)
- Nuevo estilo de definición

```
INTEGER :: a, b = 3, c
REAL :: m = 1E-3, n = 0.75
LOGICAL :: v1 = .TRUE., f1 = .FALSE.
COMPLEX :: comp1 = (5.6, 1.4E-2)
CHARACTER(LEN=5) :: r = "abcde"
REAL, PARAMETER :: pi = 3.1415926, pi2 = pi/2.0
```

- Recomendable el uso de **IMPLICIT NONE** (deshabilita declaración implícita)

Datos en punto flotante

- Precisión y rango de exponente de un real depende del procesador (problema de portabilidad)
  - Parámetro **KIND** asociado a un real que indica su precisión y rango.

```
INTEGER :: i
REAL(KIND=4) :: x, y, z
i = KIND(x)  ! Devuelve i=4
```

  - El significado de KIND=4 depende del procesador ⇒
    - Sigue el problema de portabilidad

Datos en punto flotante

- Uso con **`SELECTED_REAL_KIND(P,R)`**
- Ejemplo:

```
INTEGER, PARAMETER :: mireal = &
         SELECTED_REAL_KIND(P=8,R=30)
REAL(KIND=mireal) :: x,y
REAL, PARAMETER :: z = -1.3E-2_mireal
```

- **`x,y,z`** reales de al menos 8 cifras decimales y un exponente entre 10^{-30} y 10^{30}
- **`P`** y **`R`** opcionales (uno de ellos obligatorio)
- Si la precisión o rango exigidos no está disponible, se devuelve un valor negativo

Tipos de datos derivados

- Se construyen a partir de tipos intrínsecos o de otros tipos derivados ya definidos:

```
!Definición de tipo
TYPE datos_personales
  CHARACTER(LEN=12) :: nombre, apellido
  INTEGER :: edad
END TYPE datos_personales
!
!Declaración de variables
TYPE(datos_personales) :: p1, p2
!Asignación de valores
p1 = datos_personales("José","Pérez",35)
!Referencia a componentes
p2%apellido = p1%apellido
```

Vectores y matrices

- Declaración:

```
REAL, DIMENSION(10) :: vec1, vec2
REAL, DIMENSION(4,3) :: mat1
REAL, DIMENSION(21:40) :: vect3
REAL, DIMENSION(5:9,-7:12) :: mat2
REAL :: vec4(20), mat3(10,2)
```

- Inicialización:

```
INTEGER, DIMENSION(5) :: V=(/1,2,3,4,5/), V2
V2 = (/(i/2,i=2,10,2)/)       !v2=(1,2,3,4,5)
V2 = (/6,4,(i-2,i=3,4),15/) !v2=(6,4,1,2,15)
V2 = 0
```

Vectores y matrices

- Expresiones con arrays completos

Fortran 77

```
REAL a(100),b(100)
DO 1 i=1,100
   a(i) = 2.0
1  b(i) = b(i)*a(i)
```

Fortran 90

```
REAL, DIMENSION(100)::a,b

a = 2.0
b = b * a
```

- Código compacto y fácil de leer

Fortran 77

```
REAL a(20),b(20),c(20)
DO 1 i=1,20
1  a(i-1) = b(i+9)+c(i-10)
```

Fortran 90

```
REAL::a(0:19),b(10:29),&
     c(-9:10)
a = b + c
```

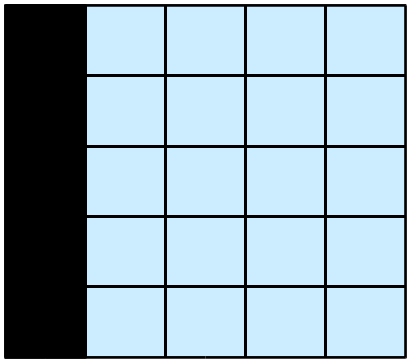
Secciones de arrays

- Podemos referirnos a secciones de arrays a través de la notación de triplete:
  - *índice_inicial:índice_final:incremento*
- Ejemplos:

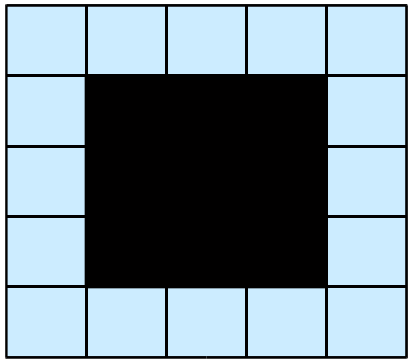
```
REAL, DIMENSION(20) :: a
a(2:8:3)      !a(2), a(5), a(8)
a(8:2:-3)     !a(8), a(5), a(2) en este orden
a(6:)         !a(6),..., a(20)
a(:9)         !a(1),..., a(9)
a(::5)        !a(1), a(6), a(11), a(16)
media=SUM(a(2:20:2))/10   !media de los elementos
                           pares
```

Secciones de arrays

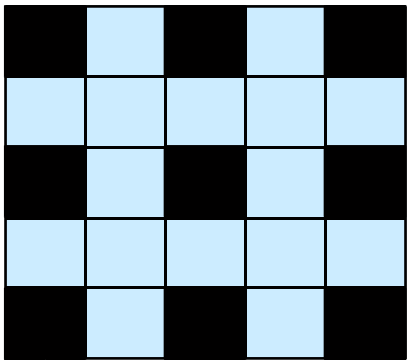
- Ejemplos de secciones:



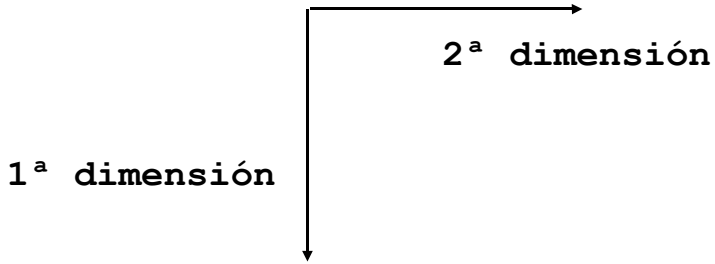
`a(:,1)`



`a(2:4,2:4)`



`a(1::2,1::2)`



Arrays conformables

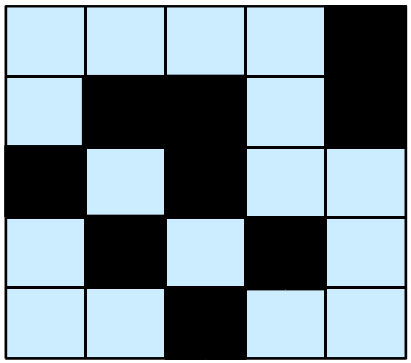
- Los arrays que aparecen en expresiones deben ser conformables:
  - Tener el mismo rango (nº de dimensiones), y
  - La misma extensión en cada dimensión
- La secuencia de extensiones define la *shape*

```
REAL :: a(1024), b(512)
a(513:) = b          !Esto es correcto
a(2:1024:2) = b      !Esto es correcto
a = b                !Esto no compila
```

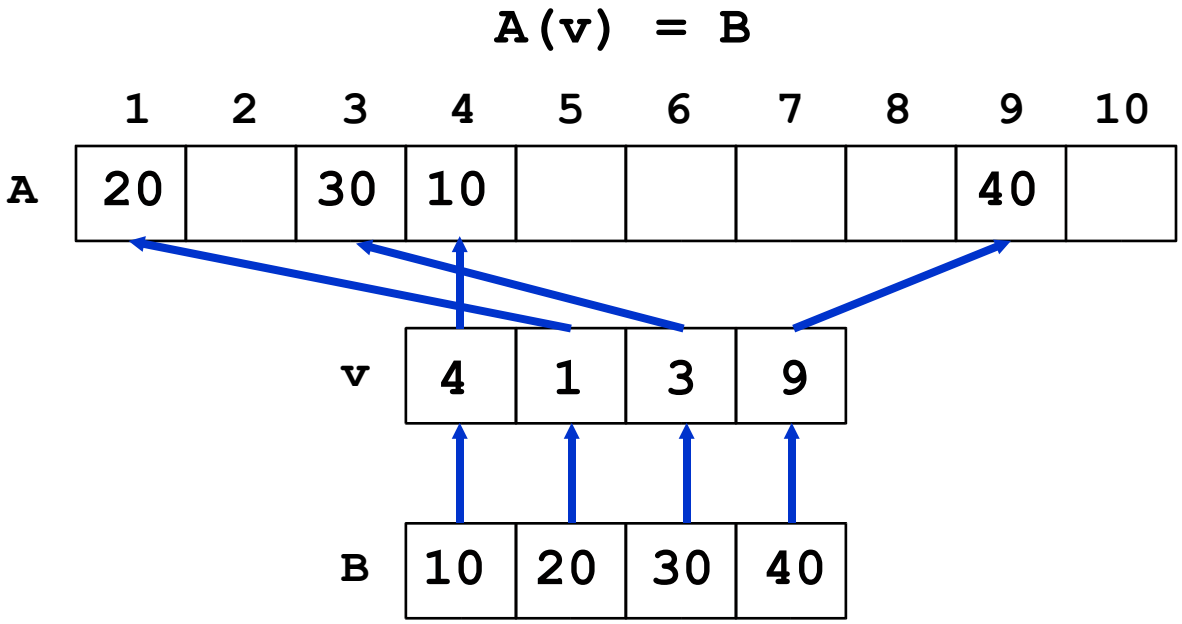
WHERE

- Uso de máscaras lógicas sobre un array:
  - Equivale a un **IF** en cada elemento
  - Uso de ==,/=,>,>=,<,<= en vez de **.EQ.**, **.NEQ.**, etc.

```
REAL, DIMENSION(5,5) :: a,b
....
WHERE(a /= 0.0) b = 1/a
...
WHERE(a > 2.0)
   b = a*9.0
ELSE WHERE
   b = a/3.0
END WHERE
```



Indirecciones



- **`B(i)`** se copia en **`A(v(i))`** para todo **`i`**

Arrays y procedimientos

- *Explicit shape arrays*
  - La forma es declarada explícitamente

```
SUBROUTINE s1(a,b,c,m,n)
REAL :: a(100,100)              !Estático
REAL, DIMENSION(m,n) :: b       !Ajustable
REAL :: c(-10:20,n:m)           !Ajustable
```

- *Assumed shape arrays*
  - Sólo se declara el rango

```
SUBROUTINE s2(a,b,n)
REAL :: a(:,:,:)
REAL, DIMENSION(0:,n:) :: b
```

Control de flujo

- Bloque IF:

```
IF(condicion1) THEN
    sentencias_fortran
ELSE IF(condicion2) THEN
    sentencias_fortran
ELSE
    sentencias_fortran
END IF
```

- IF lógico:

```
IF(condición) sentencia_fortran
```

  - ***sentencia_fortran*** no puede ser IF, CASE o DO

Control de flujo

- Bloque CASE:

```
INTEGER :: numero
SELECT CASE(numero)
CASE(:3)        !numero<=3
        ........
CASE(5)         !numero==5
        ........
CASE(4,6:9)     !numero==4 o 6 <= numero <= 9
        ........
CASE DEFAULT    !En otro caso
  ........
END SELECT
```

Lazos DO

```
DO contador=inic, final [,inc]
   sentencias_Fortran
   if (X>10) EXIT
END DO
```

```
ext: DO
   sentencias_Fortran
   int: DO
     if(X>10)EXIT int
END DO
```

- No se recomienda usar **`DO WHILE(`*cond.*`)`**. En su lugar, utilizar **`EXIT`** (**s**ale del lazo)
- **`CYCLE:`** Vuelve al inicio del lazo, incrementando el contador
- Bucles anidados: **`EXIT`** o **`CYCLE`** del bucle más interno

Funciones y subrutinas

- Funciones: proporcionan un único resultado

```
REAL FUNCTION raizcubica(x)
   IMPLICIT NONE
   REAL, INTENT(IN) :: x
   raizcubica = EXP(LOG(x)/3.0)
END FUNCTION raizcubica
```

- También pueden devolver un array

```
FUNCTION raizcubica(x)
   IMPLICIT NONE
   REAL, DIMENSION(:), INTENT(IN) :: x
   REAL, DIMENSION(SIZE(x)) :: raizcubica
   raizcubica = EXP(LOG(x)/3.0)
END FUNCTION raizcubica
```

Funciones y subrutinas

- Subrutinas:

```
SUBROUTINE raices(x,raiz2,raiz3,raiz4)
  IMPLICIT NONE
  REAL, INTENT(IN)  :: x
  REAL, INTENT(OUT) :: raiz2,raiz3,raiz4
  REAL :: lnx
  raiz2 = SQRT(x)
  lnx = LOG(x)
  raiz3 = EXP(lnx/3.0)
  raiz4 = EXP(lnx/4.0)
END SUBROUTINE raices
```

- **RETURN**: devuelve el control al programa principal

Funciones y subrutinas

- Llamada a función:

```
PROGRAM ejemplo1
  IMPLICIT NONE
  REAL, EXTERNAL :: raizcubica
  REAL :: resul, z
  resul = raizcubica(z) + z
END PROGRAM ejemplo1
```

- Llamada a subrutina:

```
PROGRAM ejemplo2
  IMPLICIT NONE
  REAL :: z, r2, r3, r4
  CALL raices(z,r2,r3,r4)
END PROGRAM ejemplo2
```

Funciones y subrutinas

- Atributo opcional **`INTENT`**: informa sobre los argumentos de los procedimientos
  - **`INTENT(IN)`**:
    - Argumento de entrada
    - No se puede modificar su valor en el procedimiento
  - **`INTENT(OUT)`**:
    - Argumento de salida
    - Valor indefinido al entrar en el procedimiento
  - **`INTENT(INOUT)`**:
    - Argumento de entrada y salida
- Argumentos de función: **`INTENT(IN)`**

Funciones y subrutinas

- Si una variable local se inicializa en su declaración, su valor se mantiene al salir del procedimiento

```
SUBROUTINE ejemplo
  IMPLICIT NONE
  INTEGER :: x=0
  x = x + 1
END SUBROUTINE ejemplo
```

- Si no se quiere dar valor inicial, usar **SAVE**

```
SUBROUTINE ejemplo
  IMPLICIT NONE
  INTEGER, SAVE :: x
  ....
END SUBROUTINE ejemplo
```

Funciones intrínsecas

- Funciones básicas
  - Funciones de F77 admiten arrays como argumentos: **`SIN`**, **`MAX`**, **`ABS`**, **`LOG`**, **`INT`**, **`REAL`**,...
  - Se aplica la función a cada elemento del array
- Funciones de manipulación de bits
  - **`ISHFT`**, **`BTEST`**, **`IBITS`**, **`MVBITS`**, ...
- Manejo de cadenas de caracteres
  - **`LEN`**, **`LGT`**, **`TRIM`**, **`VERIFY`**, **`REPEAT`**, **`SCAN`**,...
- Obtención de información de arrays
- Funciones de transformación de arrays

Información de arrays

- **`SIZE(ARRAY [, DIM])`**: Número de elementos del array, total o sobre la dimensión DIM
- **`SHAPE(SOURCE)`**: Devuelve un array de enteros de tamaño el rango de **`SOURCE`** y elementos las dimensiones del mismo
- **`LBOUND, UBOUND(ARRAY [, DIM])`**: Límite inferior y superior de array.

```
REAL, DIMENSION(-1:7,4:20) :: a
SIZE(a)          !Devuelve 153
SIZE(a,DIM=2)    !Devuelve 17
SHAPE(a)         !Devuelve (9,17)
LBOUND(a)        !Devuelve (-1,4)
UBOUND(a,DIM=1)  !Devuelve 7
```

Transformación de arrays

- Funciones de construcción de arrays:
  - **`SPREAD`**, **`PACK`**, **`RESHAPE`**, ...
- Producto de vectores y matrices
  - **`DOT_PRODUCT`**, **`MATMUL`**
- Funciones de reducción
  - **`SUM`**, **`MAXVAL`**, **`PRODUCT`**, **`ANY`**, ...
- Funciones de localización geométrica
  - **`MAXLOC`**, **`MINLOC`**
- Funciones de manipulación de arrays
  - **`CSHIFT`**, **`EOSHIFT`**, **`TRANSPOSE`**

Funciones de reducción

- Normalmente operan sobre todos los elementos

```
REAL :: a(1024), b(4, 1024), c
c = SUM(a)          !Suma de todos los elementos
c = COUNT(a==0)     !Número de elementos cero
c = MAXVAL(a, MASK=a<0)  !Máximo de los negativos
```

- Reducciones lógicas: **ANY**, **ALL**

```
LOGICAL a(n)
REAL, DIMENSION(n) :: b, c
IF( ALL(a) )          ···· !AND global
IF( ALL(b == c) )     ···· !Si son todos iguales
IF( ANY(a) )          ···· !OR global
IF( ANY(b < 0.0) )    ···· !Si algún elemento < 0
```

Funciones de localización geométrica

- **`MINLOC`**, **`MAXLOC(SOURCE)`**: Posición del mínimo o máximo en el array

```
P = MINLOC(a)
```

`a`

1	5	0	9
3	1	7	2
8	9	4	1

`P`

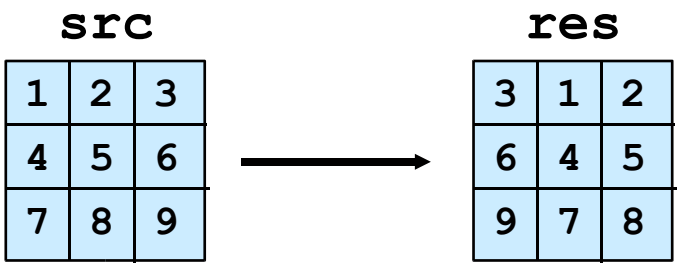
3	1

- El array resultado tiene un tamaño igual al rango de **`SOURCE`**
- Si hay más de un valor extremo, se coge el primero

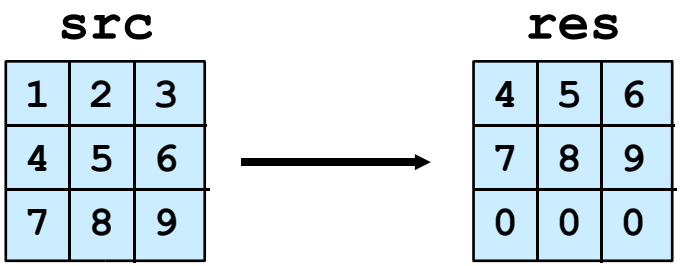
Desplazamientos

- **`CSHIFT(SOURCE,SHIFT,[, DIM])`**: Desplazamiento circular sobre la dimensión **`DIM`** (**`SHIFT>0`**, izquierda, **`SHIFT<0`**, derecha)

```
res = CSHIFT(src, SHIFT=-1, DIM=1)
```




```
res = EOSHIFT(src, SHIFT=1, DIM=2)
```



Distribución de datos en HPF

Distribución de datos

- Data parallel ⇒ procesamiento y manipulación de grandes arrays:
  - PEs ejecutan similares operaciones sobre diferentes elementos de los arrays
- Distribución de los arrays:
  - Diferentes distribuciones, según problema
  - Buscar la mejor distribución

Maximizar computación/comunicación

Distribución de datos

- Los compiladores actuales no son capaces de establecer las distribuciones:
  - Necesidad de directivas para controlar la distribución y ayudar al compilador
- Objetivo: reducir el *overhead* de comunicaciones y obtener código eficiente
- Clave en el desarrollo de programas HPF
- Directivas HPF:

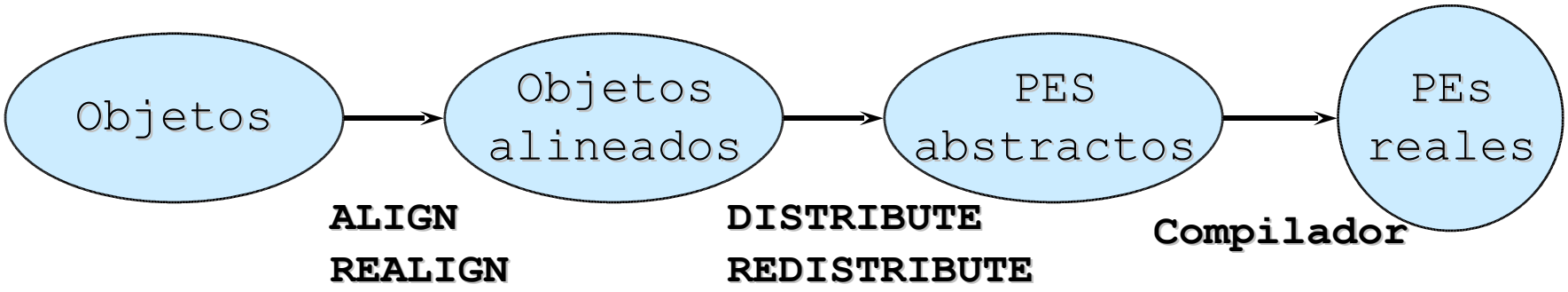
  `!HPF$` ⇒ Comentarios Fortran 90

Distribución de datos en HPF

- Directivas:
  - **`PROCESSORS, DISTRIBUTE, ALIGN, TEMPLATE, REDISTRIBUTE, REALIGN`**
- Tres etapas:
  - **Alineamiento:** diferentes objetos son alineados unos respecto a otros o con templates (usuario)
  - **Distribución:** los objetos alineados se distribuyen sobre una malla abstracta de PEs (usuario)
  - **Proyección:** de la malla abstracta sobre la real (compilador)

Distribución de datos en HPF

- Tres etapas:



Directiva PROCESSORS

- Especifica la forma de la malla de PEs abstractos:

```
!HPF$ PROCESSORS malla(6,2)
```

  - Define una malla de 6 filas y 2 columnas

- La proyección a PEs físicos no está definida

Directiva PROCESSORS

- Reglas:
  - Dos objetos proyectados sobre el mismo PE abstracto están en el mismo PE físico
  - Mallas con la misma forma son equivalentes
  - El nº de PEs en la malla abstracta debe ser menor o igual que el nº de PEs físicos

Directiva PROCESSORS

- Ejemplos

```
!HPF$ PROCESSORS P1(8) !P1(i) y P2(i) son el
!HPF$ PROCESSORS P2(8) !mismo PE abstracto
!HPF$ PROCESSORS P3(2,4)
```

Directiva PROCESSORS

- Funciones relacionadas:
  - **NUMBER_OF_PROCESSORS**
    - Nº de procesadores físicos

```
!HPF$ PROCESSORS p(NUMBER_OF_PROCESSORS()/3,3)
```

  - **PROCESSORS_SHAPE**
    - Forma de la malla física
    - Ejemplo: malla de 4*9 PEs físicos

```
PRINT *, PROCESSORS_SHAPE(malla) !Imprime (4,9)
```

Directiva `DISTRIBUTE`

- Especifica como un objeto (array o plantilla) debe ser distribuido sobre la malla abstracta
- Ejemplo:

```
!HPF$ PROCESSORS p(6,3)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO p
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO p :: A
```

  - Cada dimensión de A es distribuida BLOCK sobre la dimensión correspondiente de la malla p
  - ONTO puede omitirse. El compilador distribuye

Directiva `DISTRIBUTE`

- Tipos de distribución:
  - **`BLOCK`**
  - **`CYCLIC`**
  - **`BLOCK(size)`**
  - **`CYCLIC(size)`**
  - **`*`**

Directiva `DISTRIBUTE`

- El número de entradas **`BLOCK`** o **`CYCLIC`** debe ser igual al número de dimensiones de la malla de PEs:
  - Incorrecto (A array bidimensional):

```
!HPF$ PROCESSORS p(6)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO p :: A
```

  - Correcto:

```
!HPF$ PROCESSORS p(6)
!HPF$ DISTRIBUTE (BLOCK,*) ONTO p :: A
```

    - A se distribuye por filas (**`BLOCK`**) sobre la malla unidimensional

Tipos de distribución

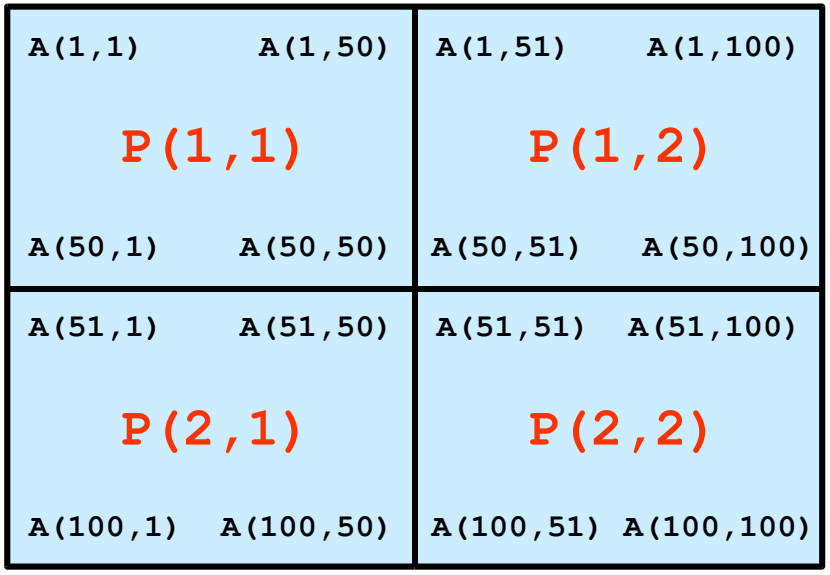
- **`BLOCK(b)`**
  - Distribución por bloques de tamaño b
  - **`BLOCK(b)`** $\Rightarrow \lceil N/b \rceil \leq P$
  - **`BLOCK`** $\Leftrightarrow$ **`BLOCK(`**$\lceil$**`N/P`**$\rceil$**`)`**
  - Problemas:
    - Error si : b*P < N
    - Ineficiencia si: N << b*P

N: Tamaño del vector
P: Número de procesadores

Tipos de distribución

- Ejemplo:

```
!HPF$ PROCESSORS p(2,2)
      INTEGER, DIMENSION(100,100):: A
!HPF$ DISTRIBUTE(BLOCK,BLOCK) ONTO p:: A
```



Tipos de distribución

- **BLOCK(b)**
  - Ejemplo: N=100, P=6
    - BLOCK(17) ⇒ Correcto, 1 PE parcialmente vacío

PE 1	PE 2	PE 3	PE 4	PE 5	PE 6
17	17	17	17	17	15

    - BLOCK(25) ⇒ Ineficiente, 2 PEs vacíos

PE 1	PE 2	PE 3	PE 4	PE 5	PE 6
25	25	25	25	0	0

    - BLOCK(16) ⇒ Incorrecto, no caben 4 elementos

PE 1	PE 2	PE 3	PE 4	PE 5	PE 6
16	16	16	16	16	16

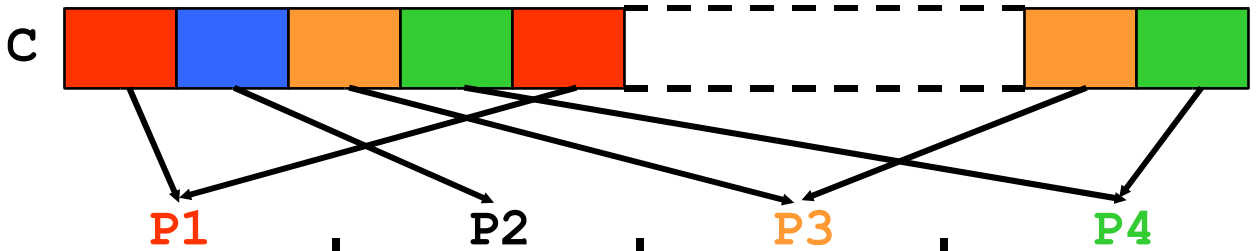
Tipos de distribución

- **`CYCLIC(b)`**
  - Distribución cíclica por bloques
  - Bloques de tamaño b se distribuyen cíclicamente
  - No restricción en el tamaño del bloque
  - **`CYCLIC(⌈N/P⌉)`** ⇔ **`BLOCK`**
  - **`CYCLIC`** ⇔ **`CYCLIC(1)`**
- **`CYCLIC(b)`** y **`BLOCK(b)`** son equivalentes
  - BLOCK(b) podría simplificar cálculos de direcciones

Tipos de distribución

- Ejemplo:

```
!HPF$ PROCESSORS p(4)
      INTEGER, DIMENSION(1000):: C
!HPF$ DISTRIBUTE(CYCLIC) ONTO p:: C
```

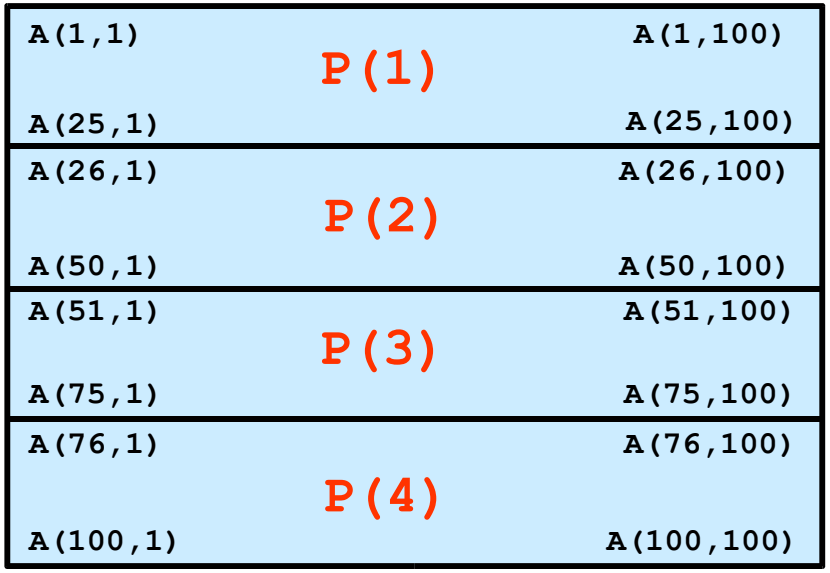


P1	P2	P3	P4
C(1)	C(2)	C(3)	C(4)
C(5)	C(6)	C(7)	C(8)
.	.	.	.
.	.	.	.
C(997)	C(998)	C(999)	C(1000)
C(:997:4)	C(2:998:4)	C(3:999:4)	C(4:1000:4)

Tipos de distribución

- * : La dimensión correspondiente de la matriz no se distribuye

```
!HPF$ PROCESSORS p(4)
      INTEGER, DIMENSION(100,100):: A
!HPF$ DISTRIBUTE(BLOCK,*) ONTO p:: A
```




```
P(1): A(1:25,:)

P(2): A(26:50,:)

P(3): A(51:75,:)

P(4): A(76:100,:)
```

Ejemplo de distribución

```
INTEGER, DIMENSION(100, 100) :: A
!HPF$ PROCESSORS p(4)
!HPF$ DISTRIBUTE (CYCLIC(10),*) ONTO p :: A
```

A(1,j) : A(10,j)	A(41,j) : A(50,j)	A(81,j) : A(90,j)	PE 1
A(11,j) : A(20,j)	A(51,j) : A(60,j)	A(91,j) : A(100,j)	PE 2
A(21,j) : A(30,j)	A(61,j) : A(70,j)		PE 3
A(31,j) : A(40,j)	A(71,j) : A(80,j)		PE 4

Directiva ALIGN

- Alinea unos objetos respecto de otros
  - Objetos alineados se proyectan sobre los mismos PEs
- Ejemplo: A y B en los mismos procesadores

```
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO p :: A
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO p :: B
```

  - Más claro

```
!HPF$ ALIGN A(:,:) WITH B(:,:)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO p :: B
```

- Un objeto puede estar alineado o distribuido, pero no ambos

Directiva ALIGN

- Sintaxis múltiple:

```
!HPF$ ALIGN A(:,:) WITH B(:,:)
!HPF$ ALIGN A(i,j) WITH B(i,j)
!HPF$ ALIGN (i,j)  WITH B(i,j) :: A
!HPF$ ALIGN WITH B :: A
```

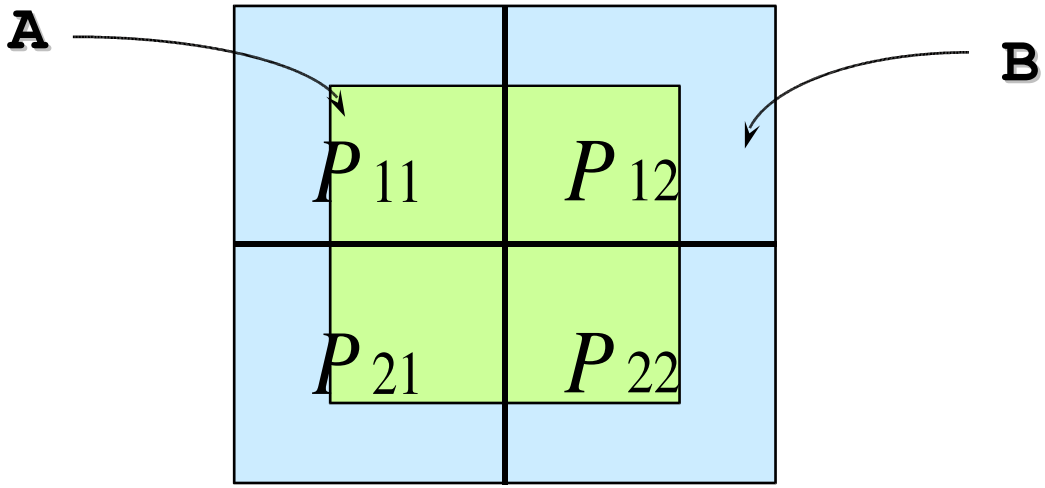
- Alineamientos complejos

```
!HPF$ ALIGN c(i) WITH d(3*i+21)
!HPF$ ALIGN A(i,j) WITH B(j,i)
```

Directiva ALIGN

- Es posible alinear un objeto con un subconjunto de otro:

```
 REAL :: A(n,n), B(n+2,n+2)
!HPF$ PROCESSORS p(2,2)
!HPF$ ALIGN A(i,j) WITH B(i+1,j+1)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO p :: B
```

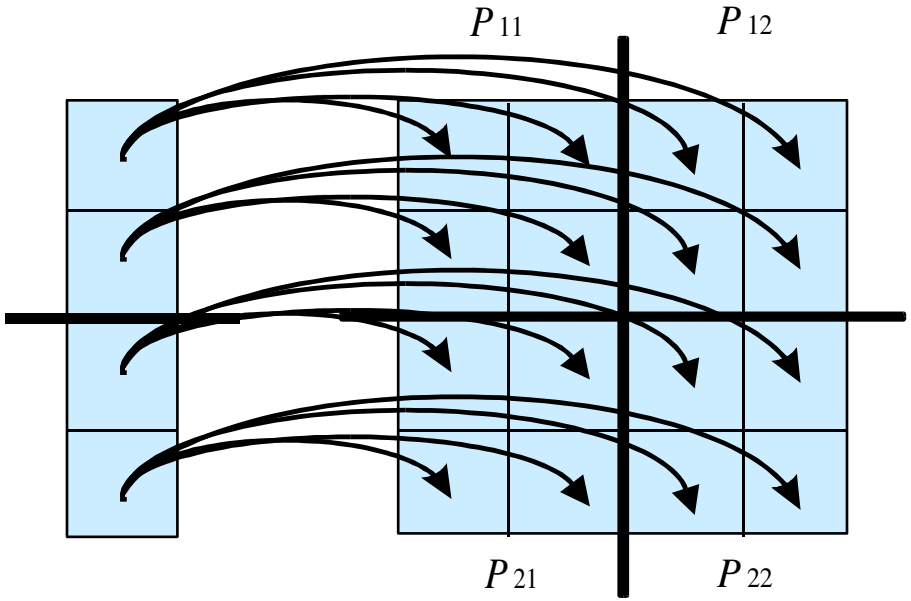


Directiva ALIGN

- Replicación:

```
!HPF$ PROCESSORS p(2,2)
!HPF$ ALIGN col(:) WITH matriz(:,*)
!HPF$ DISTRIBUTE matriz(BLOCK,BLOCK) ONTO p
```

  - Cada elemento de col se alinea con una fila entera de matriz

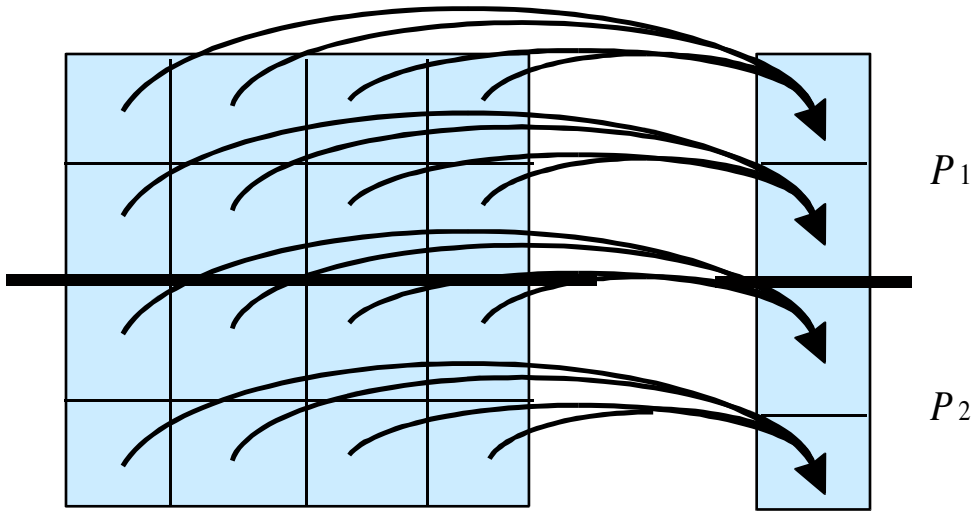


Directiva ALIGN

- Colapso:

```
!HPF$ PROCESSORS p(2)
!HPF$ ALIGN matriz(:,*) WITH col(:)
!HPF$ DISTRIBUTE col(BLOCK) ONTO p
```

  - Cada fila entera de matriz se alinea con cada elemento de col

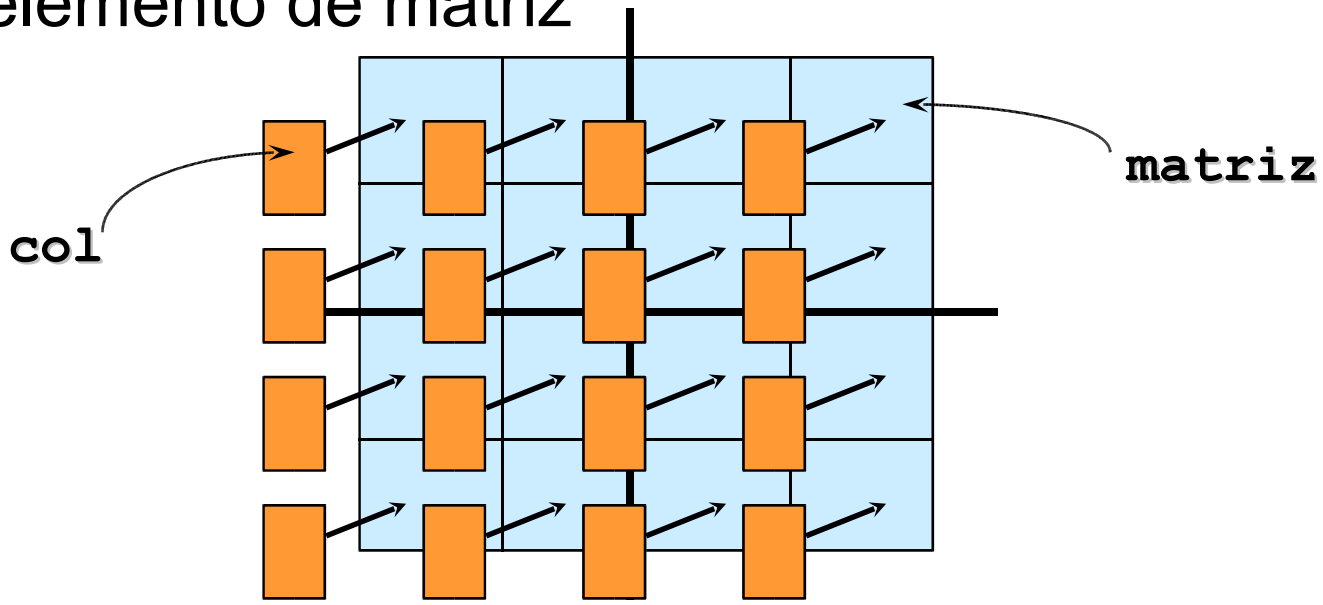


Directiva ALIGN

- Replicación a todos

```
!HPF$ PROCESSORS p(2,2)
!HPF$ ALIGN col(*) WITH matriz(*,*)
!HPF$ DISTRIBUTE matriz(BLOCK,BLOCK) ONTO p
```

  - Cada elemento de col se alinea con cada elemento de matriz



Directiva ALIGN

- Ejemplo: producto matricial C = A*B
- A replicado por filas y B por columnas

```
REAL :: C(M,N), A(M,L), B(L,N)
!HPF$ PROCESSORS malla(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO malla :: C
!HPF$ ALIGN A(:,*) WITH C(:,*)
!HPF$ ALIGN B(*,:) WITH C(*,:)
      FORALL(i=1:M,j=1:N)  &
         C(i,j) = DOT_PRODUCT(A(i,:),B(:,j))
```

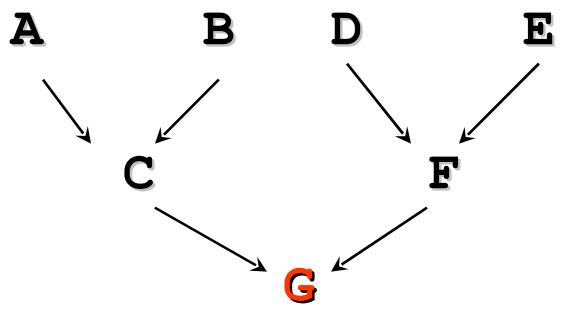
- No requiere comunicación

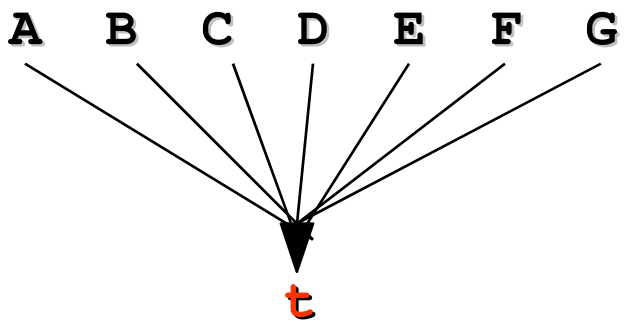
Directiva TEMPLATE

- Proporciona un espacio de índices para alinear sin necesidad de reservar memoria

```
!HPF$ TEMPLATE t(100)
```

- Simplifica el alineamiento:
  - Evita situaciones confusas





Directiva TEMPLATE

- Imprescindible en algunos casos
  - Vectores replicados por filas o columnas

```
REAL :: row(N), col(M)
!HPF$ PROCESSORS p(2,2)
!HPF$ TEMPLATE t(M,N)
!HPF$ ALIGN col(:) WITH t(:,*)
!HPF$ ALIGN row(:) WITH t(*,:)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO p :: t
```

  - t no ocupa memoria

Redistribución dinámica

- Directivas para redistribuir o realinear en ejecución
  - **`REALIGN`**: cambia el alineamiento de un objeto
  - **`REDISTRIBUTE`**: cambia la distribución de un objeto y de todos los alineados con el
  - Un objeto alineado no puede ser redistribuido y viceversa
- Se usan igual que las estáticas, pero:
  - Aparecen en la parte ejecutable del programa
  - Los arrays realineados o redistribuidos deben declararse como **`DYNAMIC`**

Redistribución dinámica

- Ejemplo:

```
REAL a(m,n)
!HPF$ DYNAMIC, DISTRIBUTE(BLOCK,BLOCK):: a
                .....
!!En ejecución
!HPF$ REDISTRIBUTE a(CYCLIC, CYCLIC)
```

- Operación muy costosa
- Poco recomendable
- No en el *subset*

Distribución de arrays dinámicos

- Un array dinámico (ALLOCATABLE o POINTER) puede ser alineado o distribuido.
- La distribución se realiza cuando se invoca a allocate

```
SUBROUTINE PEPITO(N,M)
      REAL, ALLOCATABLE, DIMENSION(:) :: A,B
!HPF$ ALIGN B(I) WITH A(I+N)
!HPF$ DISTRIBUTE A(BLOCK(M*2))
      N = 43
      M = 91
      ALLOCATE(A(27))
      ALLOCATE(B(13))
      ....
```

  - La distribución usa los valores iniciales de N y M

Distribución de arrays dinámicos

- Para poder alinear, el objeto con el que se alinea debe existir:

```
      SUBROUTINE JUANITO(N,M)
        REAL, ALLOCATABLE, DIMENSION(:) :: A,B
!HPF$ ALIGN B(I) WITH A(I+N)
!HPF$ DISTRIBUTE A(BLOCK(M*2))
        N = 43
        M = 91
        ALLOCATE(B(13))        !*** Error
        ALLOCATE(A(27))
        ....
```

- B no puede crearse antes que A

Resumen

- Dos procesos para distribuir datos:
  - **`ALIGN`**: Relación entre objetos
  - **`DISTRIBUTE`**: Distribución sobre PEs abstractos
- Método flexible y potente de describir una distribución eficiente
- La proyección a PEs físicos depende del compilador

Construcciones Paralelas en HPF

Construcciones paralelas

- Formas de expresar paralelismo en HPF
  - Operaciones con arrays (ya visto)
  - Nuevas construcciones: **`FORALL`**, **`PURE`**
  - Nuevas directivas: **`INDEPENDENT`**
  - Nuevas funciones intrínsecas

- Orientadas a paralelismo de grano fino

Sentencia FORALL

- Generaliza la asignación de arrays de F90
- No es un lazo en sentido estricto: un **`FORALL`** no itera en un orden definido
- Útil para:
  - Asignaciones basadas en índices de arrays

```
FORALL (I=1:N,J=1:N) A(I,J) = 1.0/REAL(I+J)
```

  - Expresiones de movimiento irregular de datos

```
FORALL (I=1:N) B(I) = A(I,I)
```

  - Operaciones sobre arrays con procedimientos definidas por el usuario (funciones PURE)

```
FORALL(I=1:N,J=1:N) A(I,J) = Fu(B(I,J),C(I,J))
```

Sentencia FORALL

- Condiciones para un FORALL
  - Orden de ejecución indefinido ⇒ ”deben” ser independientes
  - Un índice no puede depender de otro, o del paso

```
!Non-conforming HPF
FORALL (I=1:N, J=1:I) A(I,J) = ...
!Correcto
FORALL (I=1:N, J=1:N, J.LE.I) A(I,J) = ...
```

  - No múltiples asignaciones al mismo elemento

```
FORALL (I=1:N) S = X(I) !Non-conforming
```

    - El valor de S no estaría definido

Bloque FORALL

- Permite a un único **FORALL** controlar varias asignaciones

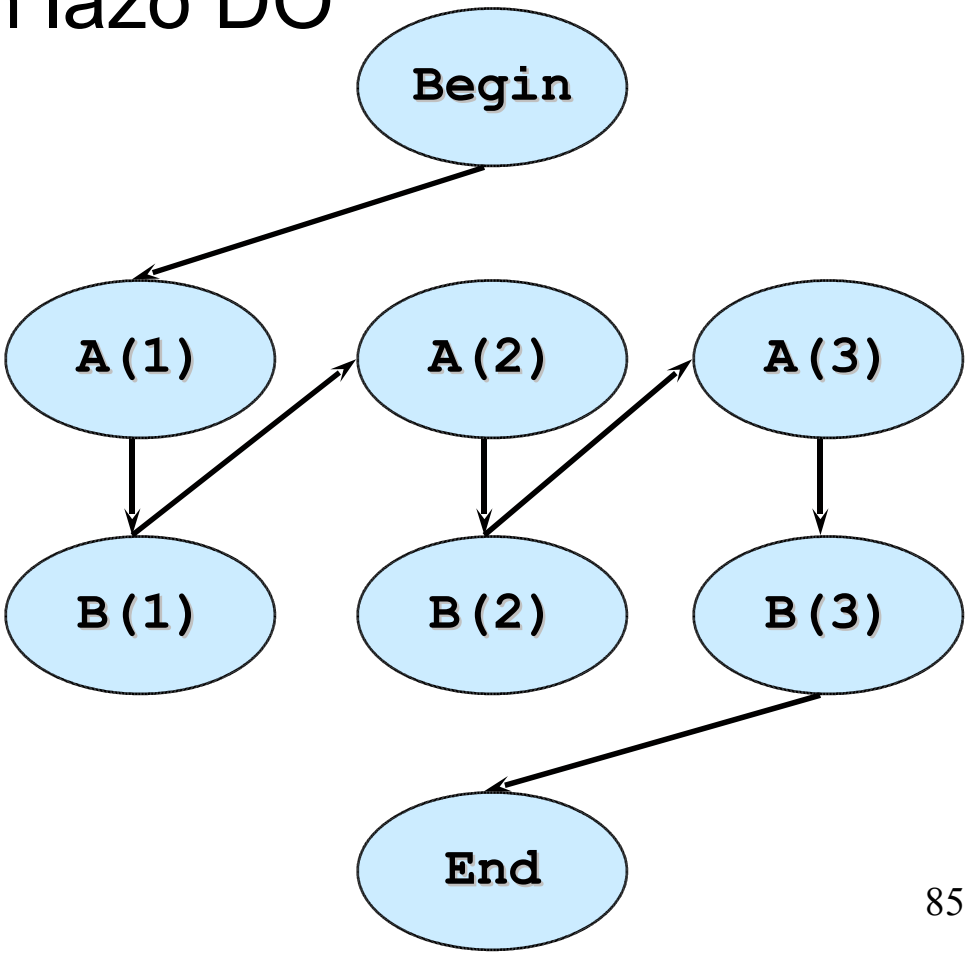
```
FORALL (I=1:N)
   A(I,J) = 1.0 / REAL(I+J)
   B(I) = I
END FORALL
```

- En el cuerpo puede haber asignaciones, otros **FORALL** o **WHERE**, o funciones **PURE**
- Un **FORALL** interno no puede modificar los índices de uno externo (puede usarlos)

Sentencia FORALL

- Diferencia con un lazo DO

```
DO I = 1, 3
  B(I) = 1.0 / A(I)
END DO
```

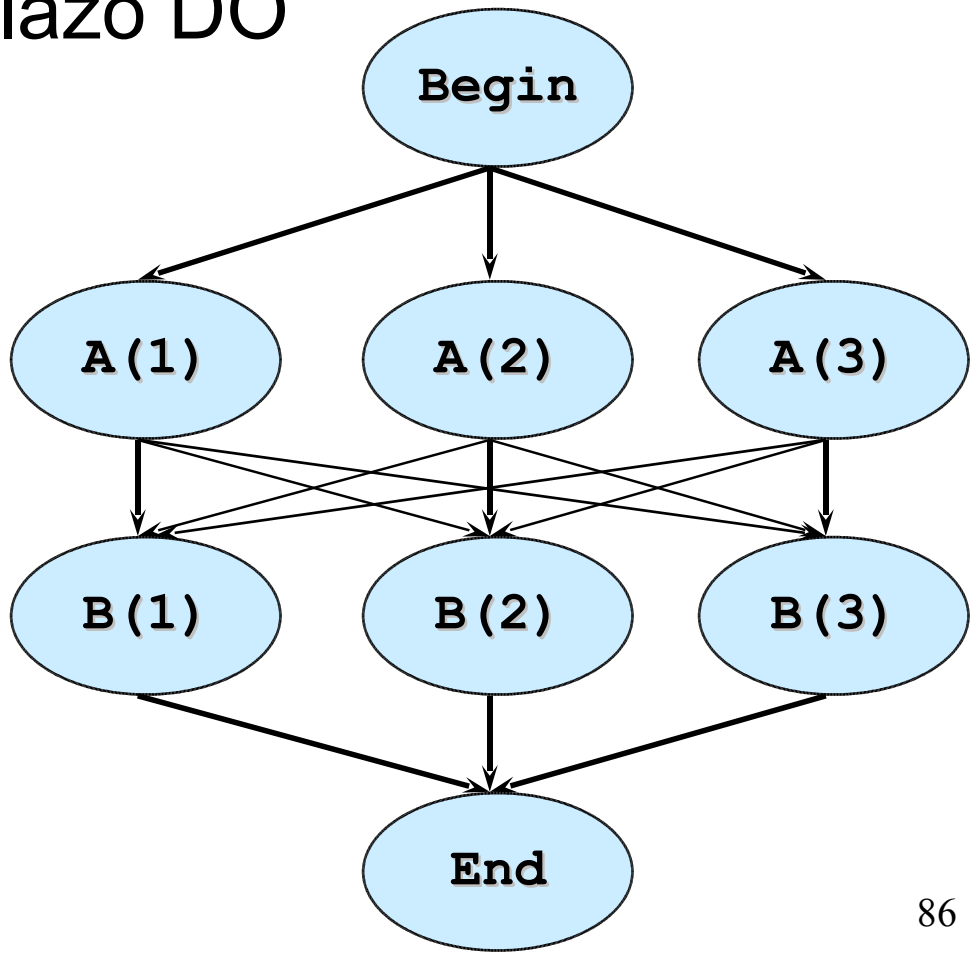


Sentencia FORALL

- Diferencia con un lazo DO

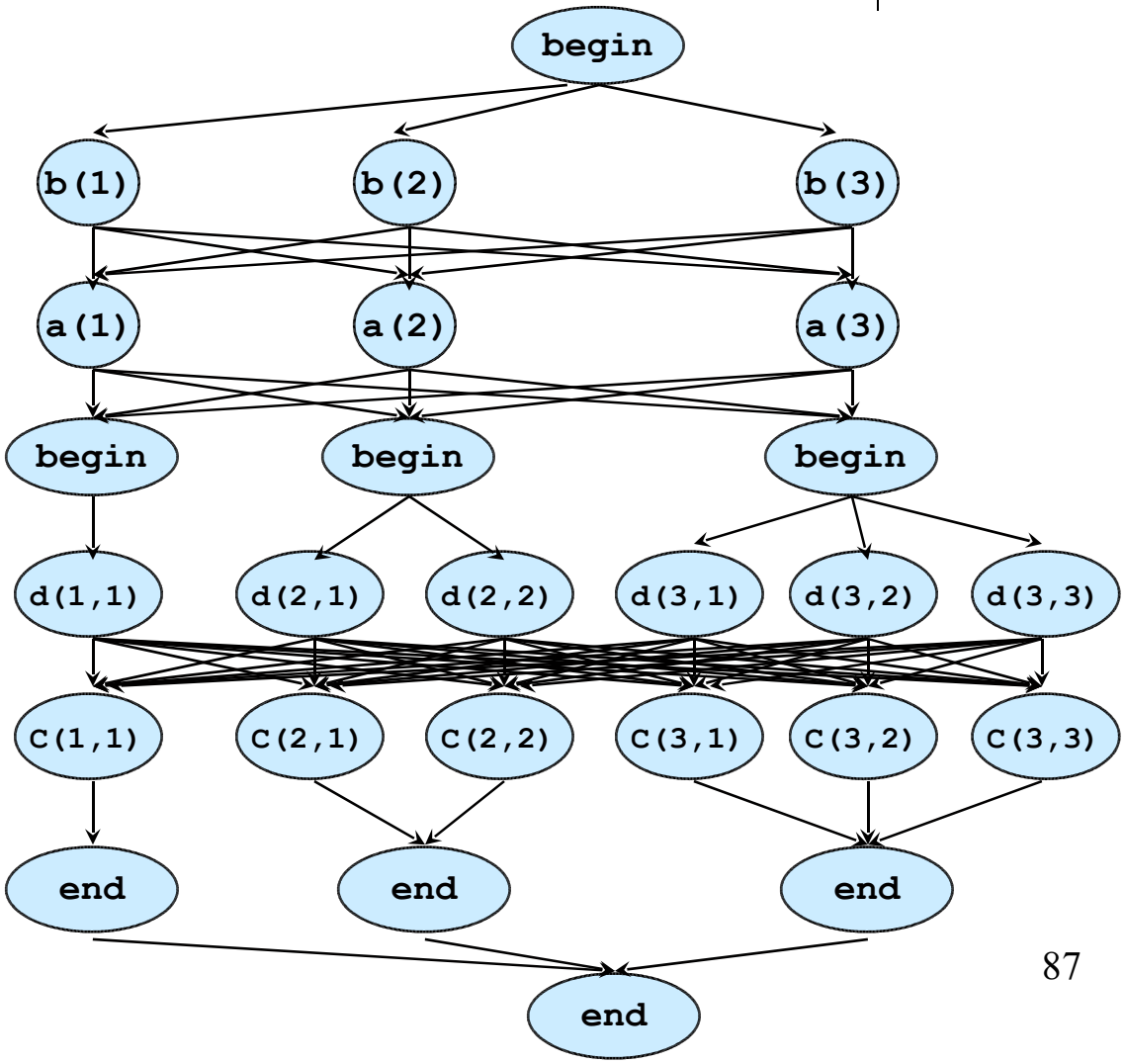
```
FORALL (I=1:3) &
    B(I) = 1.0/A(I)
```

Sincronización implícita entre etapas



Bloque FORALL

```
FORALL (I=1:3)
  a(I) = b(I)
  FORALL (J=1:I)
    c(I,J) = d(I,J)
  END FORALL
END FORALL
```



Ejemplo sentencia FORALL

- Lazo DO

```
INTEGER, DIMENSION(5):: A = (/ (I,I=1,5) /)
         ........
DO I = 2, 5
  A(I) = A(I-1)
END DO
```

  - Resultado A = [1,1,1,1,1]
- Con FORALL

```
FORALL (I=2:5) A(I) = A(I-1)
```

  - Resultado A = [1,1,2,3,4]

Ejemplos de FORALL

- Ejemplos de FORALL

```
FORALL (i=1:10) A(index(i)) = B(i)
```

  - **index(i)** no puede contener valores repetidos

```
FORALL (i=1:10, A(i) > 0.0)  &
    A(i) = 1.0/A(i)

FORALL (i=1:m, j=1:n) &
    C(i,j)=DOT_PRODUCT(A(i,:),B(:,j))
```

  - Producto de matrices A*B

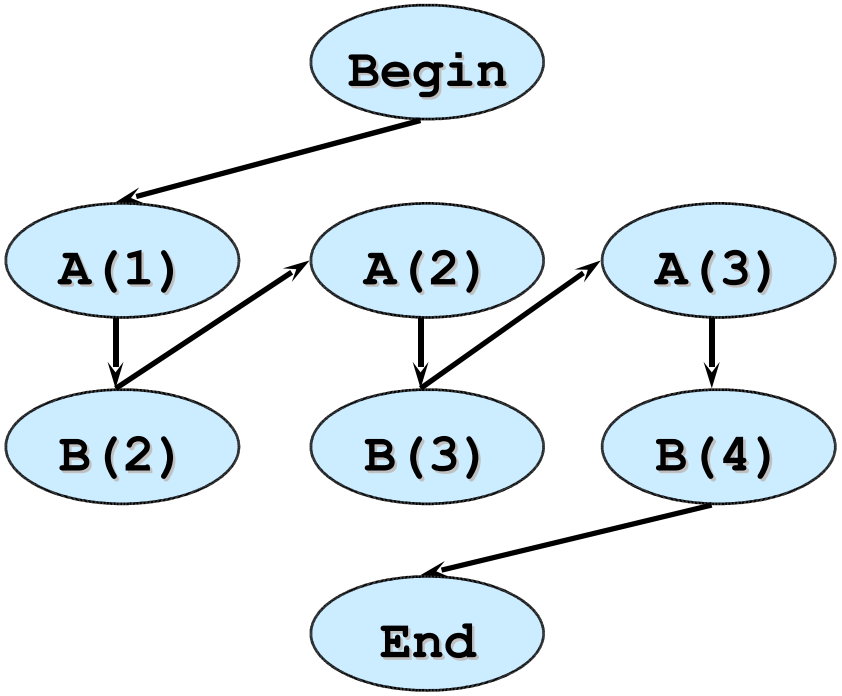
Directiva INDEPENDENT

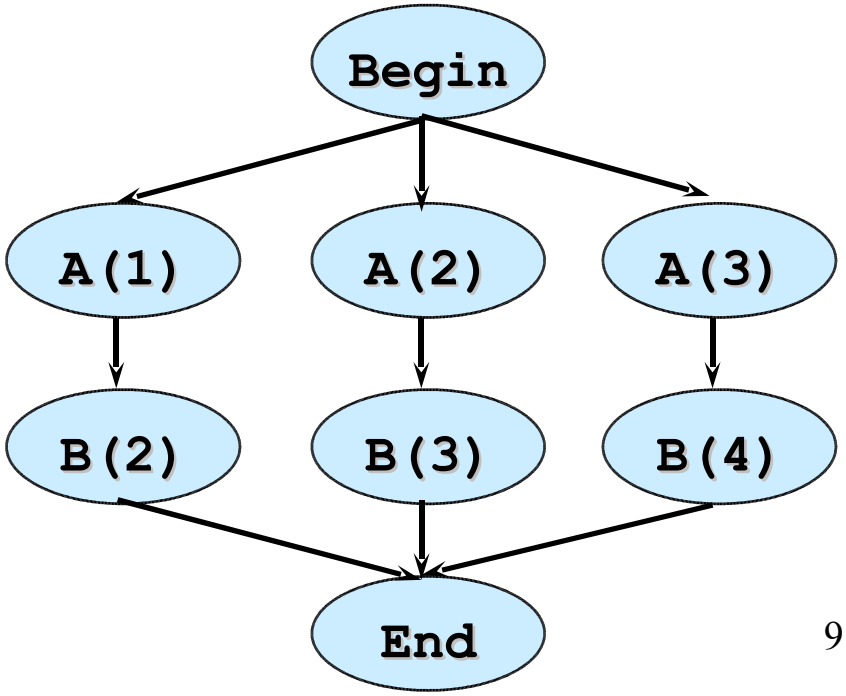
- Para **`FORALL`** y **`DO`**
- Elimina puntos de sincronización en los **`FORALL`**
- El programador debe asegurar que el resultado del lazo será el mismo que si se ejecutara en serie
- Si esto no se cumple el resultado es indefinido

INDEPENDENT y DO

```
DO i=2,4
   B(i) = A(i-1)
END DO
```

```
!HPF$ INDEPENDENT
DO i=2,4
   B(i) = A(i-1)
END DO
```





INDEPENDENT y FORALL

- Permite una mayor eficiencia en problemas no bien balanceados
- Uso: igual que con el **DO**

```
!HPF$ INDEPENDENT
      FORALL(I=1:N) ....
```

Directiva INDEPENDENT

- Restricciones:
  - Si un objeto se escribe en una iteración, no puede volver a leerse o escribirse

```
DO i=2, N                    DO i=1, N
  A(i) = A(i-1)                 S = S + A(i)*B(i)
END DO                       END DO
```

  - No puede haber dependencias de control

```
DO i=1, N
  IF(A(i) .EQ. 10) EXIT
END DO
```

Directiva INDEPENDENT

- Restricciones (pghpf)
  - Sólo se permiten instrucciones Fortran 77 en el cuerpo de un **DO** independiente
  - **DO** independientes pueden anidarse: no puede haber más de uno en un nivel

```
!HPF$ INDEPENDENT
      DO i=1, n
!HPF$ INDEPENDENT
        DO j=1, m
          ...
        END DO
!HPF$ INDEPENDENT
        DO j=m, 1, -1
          ...
        END DO
      END DO
```

La cláusula NEW

- Permite definir variables temporales, privadas al lazo
- Una nueva variable es creada en cada iteración

```
!HPF$ INDEPENDENT, NEW (S)
      DO i=1, n
         S = SRQT(A(i)**2 + B(i)**2)
         C(i) = S
      END DO
```

- Fuera del lazo, S indefinida

Problemas con DO independientes

- Excepto en casos simples, un **DO** independiente funcionará bien sólo en máquinas de memoria compartida (referencias muy complejas)

NO DEBERÍAN USARSE EN SISTEMAS DE MEMORIA DISTRIBUIDA
PROCURAR USAR ARRAYS O `FORALL`

Procedimientos PURE

- Permite llamar a procedimientos dentro de un **`FORALL`**
- Una función **`PURE`**:
  - No debe alterar los datos de entrada (necesario **`INTENT(IN)`**)
  - No debe tener efectos colaterales (p.e. modificación de datos globales, I/O, remapping, etc.)
  - Deben tener un interfaz explícito (INTERFACE)

Procedimientos PURE

- Ejemplo

```
PURE REAL FUNCTION MY_EXP(X)
      REAL, INTENT(IN) :: X
      MY_EXP = 1+X+X*X/2.0+X**3/6.0
END FUNCTION MY_EXP

PURE REAL FUNCTION MY_SINH(X)
      REAL, INTENT(IN) :: X
      MY_SINH = (MY_EXP(X)-MY_EXP(-X))/2.0
END FUNCTION MY_SINH

        ..............

FORALL(i=1:N, j=1:N) S(i,j) = MY_SINH(A(i,j))
```

Resumen

- HPF incluye constructores para expresar paralelismo (grano fino)
  - **`FORALL`**: generaliza las operaciones de arrays
  - **`INDEPENDENT`**: garantiza la ejecución sin interferencia entre iteraciones
  - **`NEW`**: permite variables temporales en **`FORALL`**
  - **`PURE`**: permite usar funciones en **`FORALL`**

Funciones Intrínsecas y Librería HPF

Funciones intrínsecas

- HPF añade a las numerosas funciones intrínsecas de Fortran 90:
  - Extensiones a **`MAXLOC`**, **`MINLOC`**, posición del máximo (mínimo) en un array: parámetro **`DIM`**
  - Dos funciones intrínsecas de información del sistema
  - Nueva intrínseca **`ILEN`**: número de bits necesarios para almacenar un valor entero
- Librería HPF: necesario **`USE HPF_library`**

Nuevas funciones intrínsecas

- Funciones de información del sistema:
  - **`NUMBER_OF_PROCESSORS([DIM])`**: Devuelve el número total de PEs físicos disponibles, o el número a lo largo de una dimensión de la malla física
  - **`PROCESSORS_SHAPE( )`**: Devuelve la forma de la malla física de PEs
- Estos valores permanecen constantes a lo largo de la ejecución del programa
- No se ven afectados por la directiva **`PROCESSORS`**

Nuevas funciones intrínsecas

- Ejemplos:
  - Malla de 128x64 PEs:

    NUMBER_OF_PROCESSORS() ⇒ 8192

    NUMBER_OF_PROCESSORS(DIM=2) ⇒ 64

    PROCESSORS_SHAPE() ⇒ (182,64)
  - Uso en un programa:

```
      INTEGER, DIMENSION(SIZE(PROCESSORS_SHAPE())):: P
      REAL, DIMENSION(3*NUMBER_OF_PROCESSORS())    :: A
!HPF$ TEMPLATE, DIMENSION(NUMBER_OF_PROCESSORS()) :: T
```

Librería HPF

- Funciones de tipo:
  - Información sobre la distribución
  - Nuevas funciones de manipulación de bits: **`ILEN`**, **`LEADZ`**, **`POPCNT`**, **`POPPAR`**
  - Nuevas funciones de reducción de arrays: **`IALL`**, **`IANY`**, **`IPARITY`**, **`PARITY`**
  - Funciones de scatter
  - Ordenación de arrays
  - Barrido: prefijo y sufijo

Información sobre la distribución

- Permiten determinar la distribución actual de un array:
  - **`HPF_ALIGNMENT(...)`**: Información sobre el alineamiento
  - **`HPF_TEMPLATE(...)`**: Información de la plantilla o matriz con la que esta alineado
  - **`HPF_DISTRIBUTION(...)`**: Información de la distribución de la plantilla o matriz con la que está alineado

Funciones de scatter

- Generaliza la asignación: **`Y(v) = X`**, aún cuando **`v`** tenga elementos repetidos, o sea multidimensional
- Formato

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

donde:

XXX → **AND, ANY, COPY, COUNT, IALL, IANY, IPARITY, MAXVAL, MINVAL, PARITY, PRODUCT, SUM**

- El resultado tiene la forma de **`BASE`**. El número de **`INDX`** debe ser el rango de **`BASE`**

Funciones de scatter

- Ejemplo

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad B = \begin{bmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & -8 & -9 \end{bmatrix}$$

$$I1 = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 3 & 2 & 1 \end{bmatrix} \quad I2 = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

$$R = SUM\_SCATTER(A,B,I1,I2) = \begin{bmatrix} 14 & 6 & 0 \\ 8 & -5 & -6 \\ 0 & -8 & -9 \end{bmatrix}$$

$$R = SUM\_SCATTER(A,B,2,I2) = \begin{bmatrix} -1 & -2 & -3 \\ 30 & 3 & -3 \\ -7 & -8 & -9 \end{bmatrix}$$

Ordenación de arrays

- Dos funciones:

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

- Ejemplo:

```
A = [2 9 4] ⇒ S = GRADE_UP(A)
⇒ S = [1 3 2]

X = A(S) ⇒ X = [2 4 9]
```

Otros tópicos y desarrollos actuales

Desarrollos actuales

- Trabajo del HPFF II $\Rightarrow$ mejorar aspectos del HPF:
  - Depuración de errores y ambigüedades
  - Mejor control de paralelismo de grano grueso (múltiples procesos, control del trabajo, etc.)
  - Distribuciones de datos para problemas irregulares
  - Entrada salida paralela
- Objetivo: HPF v2.0 (Enero 1997)

HPF subset

- Algunas características de F90 no están en el *subset* de HPF:
  - Formato de escritura libre
  - **`POINTER`**, **`TARGET`**
  - **`SELECT CASE`**
  - **`MODULE`**
  - **`KIND`**
  - Tipos de datos derivados y operadores
  - Algunas características de entrada/salida

Portland Group Compiler: pghpf

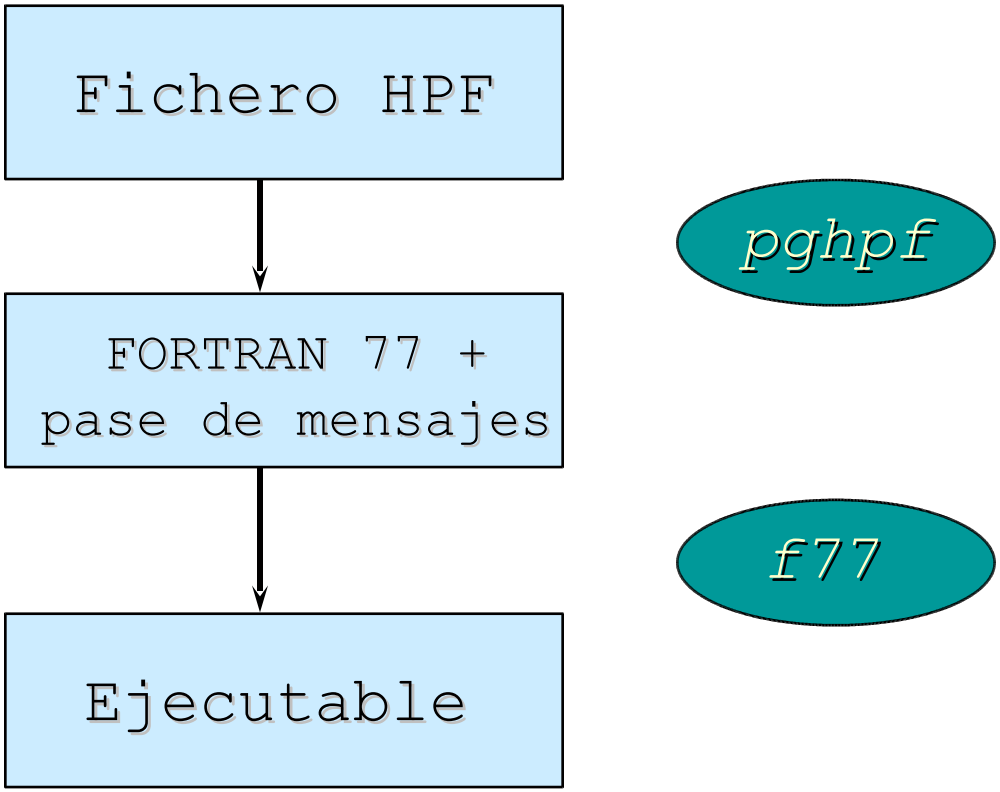
- Pre-compilador paralelizador
- Compila HPF a Fortran 77 + pase de mensajes
- Gran variedad de hosts: Sun, Cray T3D, SGI, SP2, Paramid, AP3000
- Diferentes protocolos de comunicación: PVM, MPI, RPM (PVM optimizado), SHMEM, ...
- Es un *SUPER SUBSET*

Compilador pghpf

- No admite:
  - Procedimientos internos
  - Recursividad
  - Punteros en tipos de datos derivados
  - Ciertas funciones intrínsecas no van con datos derivados: **`CSHIFT`**, **`LBOUND`**, **`PACK`**, **`SHAPE`**, **`SIZE`**, **`SPREAD`**, **`TRANSPOSE`**, ...
- Ver *pghpf Release Notes* para más información
  - http://www.cesga.es/manuales/PGHPF/release_notes/relnot.htm
  - /usr/pgi/release_notes.ps

Portland Group Compiler: pghpf

- Etapas de compilación



Portland Group Compiler: pghpf

- Compilación:

```
pghpf [-Mopt] [-o out] file.hpf[.f90] [libs]
```

Ejemplo, un solo nodo: `pghpf -Mf90 -o hola hola.hpf`

Ejemplo, varios nodos:

```
pghpf -Mmpi -o hola hola.hpf
      -L/opt/FSUNaprun/lib -lmpl -lemi
```

- Ejecución:

Ejemplo, un solo nodo: `hola -pghpf -stat all`

Ejemplo, varios nodos:

```
aprun -nproc 4 hola -- -pghpf -stat alls
```

Portland Group Compiler: pghpf

- Algunas opciones de compilación
  - **`-Mmpi[pvm,rpm]`** → Librería de pase de mensajes
  - **`-Mf90`** → Compila para un solo procesador
  - **`-Mstats`** → Estadísticas de comunicación
  - **`-Mautopar`** → Autoparaleliza lazos DO
  - **`-Mfreeform`** → Formato de escritura libre
  - **`-Mftn`** → Obtiene el código F77
  - **`-Mnoindependent`** → No aplica la directiva **`INDEPENDENT`** a lazos **`DO`**
  - **`-Minfo`** → Información de la compilación (**`-Minfo=all[,autopar,loop]`**)

Portland Group Compiler: pghpf

- Algunas opciones de ejecución: -pghpf
  - **`-np p`** → Nº de procesadores (si se usa **`aprun`**, este p debe ser ≤ que el indicado en **`-nproc`**)
  - **`-stat all[cpu,mem,...][s]`** → Muestra estadísticas de tiempo, memoria, comunicación,...

- Para más opciones, ver *pghpf User's guide*
  - http://www.cesga.es/ga/CalcIntensivo/Manuais.html

Más información

- High Performance Fortran Forum (HPFF):
  http://www.vcpc.univie.ac.at/information/mirror/HPFF/
- The HPF User Group (HUG)
  http://www.vcpc.univie.ac.at/information/HUG/
- “The High Performance Fortran Handbook”, C. H. Koebel et. al., The MIT Press, 1994
- “Fortran 90 Programming”, T.M.R. Ellis, I.R. Philips y T.M.Lahey, Addison-Wesley, 1994

Más información

- “Fortran 95 Handbook”, J.C. Adams et. al. The MIT Press, 1997
- High Performance Computing Projects at Liverpool University:
  http://www.liv.ac.uk/HPC/HPCpage.html
- Grado de penetración de HPF
  http://dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm