**Paralelismo con F95/HPF/OMP**

# 1. Preliminares

En estas notas resumimos algunas ideas básicas para la programación en computadoras paralelas mediante los lenguajes Fortran 90/95 (F90/F95) y *High Perfomance Fortran* (HPF), que están fundamentalmente orientados al paradigma del *paralelismo en los datos*. Algunas de estas ideas se pueden implementar mediante otros lenguajes que incluyan un equivalente de tal paradigma, e.g. la norma *Open Machine Parallel*[19] (OMP) tanto en C++ como en F90/F95.

*Nota: en la presente versión no incluiremos un repaso del OMP con C++/F95, remitiendo al lector interesado a los tutorials incluidos en los directorios OMP del curso.*

# 2. Fortran 90/95

Fortran 90/95 es un lenguaje estándar (norma ISO) orientado al álgebra matricial numérica pero parcialmente orientado a objetos. Algunas de sus instrucciones seriales preveen la eventualidad de hacerlo en forma automática en paralelo (e.g la instrucción FOR ALL cuando el programa es compilado en un multiprocesador) pero no provee sintaxis para descomposición de los datos en subdominios. En cambio en el HPF, que es una extensión estándar del F90/F95 aunque menos difundido, se dispone de la posibilidad de distribuir los datos sobre el conjunto de procesadores (*datos distribuidos*) y de efectuar cálculo paralelo sobre los mismos. En la práctica, en un programa fuente F95/HPF:

1. introducimos primero *directivas* en donde sugerimos al compilador una cierta distribución de los datos matriciales sobre un arreglo abstracto de procesadores en paralelo explicito o implicito;

2. luego vamos introduciendo otras directivas en donde proponemos zonas con tareas de cálculo paralelo sobre los datos matriciales distribuidos;

3. todas las directivas, como es frecuente en las extensiones de lenguaje, se las intercala como comentarios especiales en lugares apropiadas, de modo tal que son ignoradas por un compiladores serial (o escalar) F90/F95;

## 2.1. Algunos compiladores C++/F95

Entre los compiladores C++ y F95 sobre Linux que últimamente más hemos experimentado en el CIMEC, tanto en PC individuales (bajo Linux) o en nuestro *Cluster Beowulf* "Gerónimo", mencionamos:

1. `g++` (C++), `g77` (F77) y `g95`[8] (F95): los dos primeros son clásicos que ya vienen preinstalados en casi todos los paquetes de distribución del `Linux`[12], mientras que el restante es de muy reciente disponibilidad. Todos estos son compiladores seriales que *no* preveen paralelismo;

2. `KAI-Intel`[10]: C++/OMP y F95/OMP, de libre distribución [licencia tipo GLP (*General Public license*) bajo *Linux* pero con licencia comercial bajo *windows*. Su compatibildad con la norma OMP lo hace quizás interesante, por ejemplo, en una Pentium dual o en un cluster de Pentium duales. En este último caso notar que sólo podremos hacer paralelismo tipo OMP dentro de cada nodo pero obligadamente con paso de mensajes entre nodos;

3. `ADAPTOR`[2]: traductor de libre distribución bajo *Linux* y orientado específicamente a *cluster Beowulf*. Traduce y compila fuentes de HPF o de F77/F90 (con o sin OMP o MPI) apoyándose en compiladores C/C++ y F77/F95 seriales pre-instalados, por ejemplo, `g++` y `g95`;

4. `Portland PGI`[20] ofrece paquetes comerciales de compiladores de C++/F95/HPF que incluye una opción orientada específicamente a *cluster Beowulf*.

### 2.2. Bibliografía

i) Para F90/F95: Marshall and Schonfelder[15], Meissner[16], Press[21] *et al.*., Norton[17], Decyk[4] *et.al* y Marshall[13] *et al.*;

ii) Para HPF: Koebel[11] *et al.*, Ewing[6] *et al.*, Marshall[14] *et al.*, Brandes[1,2]

iii) Para OMP: normas OMP para F95/C++[19], Hermans[9];

iv) Para cálculo paralelo: Foster[7]; Cormen *et al.*[3] y Edelman[5].

## 3. Convenciones en la notación

En los fragmentos de código, por didáctica, usaremos mayúsculas para las palabras `reservadas` del lenguaje, minúsculas para los identificadores, funciones y procedimientos del usuario (notemos que F90 es insensitivo al respecto). Los programas fuente tendrán extensión ya sea `*.f`, `*.f90` o `*.hpf` (hay que leer el manual del compilador empleado). Si bien el formato `libre` en los programas fuente es el ahora recomendado a partir del F90 (equivalente al empleado en los lenguajes Pascal o C), a veces usaremos el formato `fijo` para destacar las directivas `!HPF$`. En los comentarios en los programas fuente con formato fijo usaremos tanto una `c` en la columna 1 de cada línea (de hasta 72 caracteres) para indicar que es toda una línea de comentarios, como también un signo de exclamación `!` para los comentarios intercalados a continuación de las instrucciones. Muchos de los acrónimos utilizados aquí se mantendrán en idioma inglés y se recopilan en un apéndice, junto con el significado de otras abreviaturas más frecuentes.

## 4. Operaciones matriciales en F90/F95

Fortran 90/95 es un lenguaje de programación con paralelismo en los datos. Se basa en el "viejo" Fortran 77 (F77), soporta cálculo matricial numérico, Tipos Abstractos de Datos (TAD), extensa librería intrínseca matricial, pero parcialmente orientado a objetos (defecto a superar en el Fortran 2003). F90 es un salto cualitativo con respecto del F77 mientras que F95 es una leve actualización del F90, en donde destacamos las construcciones `FORALL` y `WHERE`. Al final de esta sección, mostramos dos ejemplos de codificación matricial: el producto matriz banda vector y un simil matricial dinámico para la clasificación por incrementos decrecientes (`Shellsort`).

### 4.1. Diferencias entre las instrucciones DO y FORALL

Consideremos el vector a=[11 22 33 44 55] y las siguientes tareas

```
DO       i=2,5  ; a (i) = a (i-1)  ; END DO
FORALL  (i=2:5)   a (i) = a (i-1)
```

El orden de cómputo del lazo `DO` es *secuencial* y lo resumimos en la Tabla 1. En cambio, en el lazo `FORALL` se procede con las siguientes etapas:

| i | a |
|---|---|
| 0 | [11 22 33 44 55] |
| 2 | [11 11 33 44 55] |
| 3 | [11 11 11 44 55] |
| 4 | [11 11 11 11 55] |
| 5 | [11 11 11 11 11] |

Cuadro 1: Ejemplo de la ejecución secuencial del DO.

1. se determina el conjunto de indices *válido* $i_v$ definido por el lazo, en este caso $i_v = (2\ 3\ 4\ 5)$;

2. se determina el conjunto de indices *activo* $i_a$ debido a eventual máscaras booleanas, pero que en este caso coincide con el conjunto válido: $i_a = (2\ 3\ 4\ 5)$;

3. se evalúa *simultáneamente toda* la expresión del miembro derecho para los índices activos (en cualquier orden): `a (i-1)` con `i=2:5` o sea $(a_1\ a_2\ a_3\ a_4)$=(11 22 33 44);

4. se asigna *toda* la expresión evaluada al miembro izquierdo (en cualquier orden): `a (i)` con `i=2:5` o sea $(a_2\ a_3\ a_4\ a_5)$=(11 22 33 44).

De este modo, no siempre un lazo `FORALL` es quivalente a un lazo `DO` pero, por otra parte, nos permite asignamientos más generales, e.g.

```
FOR ALL (i=1:m, j=1:n)           x (i,j) = i + j
FOR ALL (i=1:n, j=1:n, i < m)    y (i,j) = 0.0
FOR ALL (i=1:n)                  z (i,i) = 1.0
```

en donde se asignan a cada elemento de la matriz `x`, la suma de sus índices, cero a la parte triangular superior de la matriz `y`, y un 1 a la diagonal principal de la matriz `z`, respectivamente.

## 4.2. Sintaxis matricial para el producto matriz banda-vector

Por ejemplo, consideremos el siguiente producto matriz-vector,

$$b = ax = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 \\ a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 \\ a_{43}x_3 + a_{44}x_4 + a_{45}x_5 \\ a_{54}x_4 + a_{55}x_5 \end{bmatrix} . \quad (1)$$

Guardemos la matriz banda `a` con el formato comprimido

$$c = \begin{bmatrix} 0 & a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} & 0 \\ a_{54} & a_{55} & 0 & 0 \end{bmatrix} ; \quad (2)$$

de $n$ filas y $m = m_1 + 1 + m_2$ columnas, donde $m_1$ es el número de sub-diagonales, $m_2$ es el número de supra-diagonales, y el 1 tiene en cuenta la columna de la diagonal principal. En este

ejemplo tendremos $m_1 = 1$, $m_2 = 2$, $m = 4$. Ahora hagamos la siguiente operación matricial:

$$y = \text{SPREAD (x, DIM=2, NCOPIES = m)} = \begin{bmatrix} x_1 & x_1 & x_1 & x_1 \\ x_2 & x_2 & x_2 & x_2 \\ x_3 & x_3 & x_3 & x_3 \\ x_4 & x_4 & x_4 & x_4 \\ x_5 & x_5 & x_5 & x_5 \end{bmatrix} ; \tag{3}$$

e introduzcamos el vector auxiliar

$$u = \text{prog\_arit (-m1, 1, m)} = [\ -1 \quad 0 \quad 1 \quad 2\ ] ; \tag{4}$$

donde `prog_arit` es una cierta función del usuario que nos devuelve, en este caso, los $m$ primeros elementos de la progresión artimética que empieza en $-m1$ y que va con incremento 1. A continuación hacemos,

$$y = \text{EOSHIFT (y, DIM=1, SHIFT = u)} = \begin{bmatrix} 0 & x_1 & x_2 & x_3 \\ x_1 & x_2 & x_3 & x_4 \\ x_2 & x_3 & x_4 & x_5 \\ x_3 & x_4 & x_5 & 0 \\ x_4 & x_5 & 0 & 0 \end{bmatrix} ; \tag{5}$$

por lo que el producto "punto" `z = c y` será igual a

$$z = cy = \begin{bmatrix} 0 & a_{11}x_1 & a_{12}x_2 & a_{13}x_3 \\ a_{21}x_1 & a_{22}x_2 & a_{23}x_3 & a_{24}x_4 \\ a_{32}x_2 & a_{33}x_3 & a_{34}x_4 & a_{35}x_5 \\ a_{43}x_3 & a_{44}x_4 & a_{45}x_5 & 0 \\ a_{54}x_4 & a_{55}x_5 & 0 & 0 \end{bmatrix} ; \tag{6}$$

y, finalmente, la suma

$$b = \text{SUM (z, DIM=2)} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 \\ a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 \\ a_{43}x_3 + a_{44}x_4 + a_{45}x_5 \\ a_{54}x_4 + a_{55}x_5 \end{bmatrix} ; \tag{7}$$

nos da el producto buscado. Todo lo anterior puede resumirse en las dos líneas de código:

```
n = SIZE (c, DIM=1) ; m = SIZE (c, DIM=2)
b = SUM (c * EOSHIFT ( SPREAD (x, DIM=2, NCOPIES=m), DIM=1,
                       SHIFT = prog_arit (-m1,1,m)), DIM=2)
```

Tal "estilo" de codificación suele encontrarse en las librerías de cómputo científico.

### 4.3. Clasificación matricial por incrementos decrecientes

Por ejemplo, supongamos que queremos ordenar de menor a mayor el vector $a = [\ 14 \quad 7 \quad 12 \quad 5 \quad 10 \quad 3 \quad 8 \quad 6 \quad 13 \quad 4 \quad 11 \quad 2 \quad 9 \quad 1\ ]$, de longitud $n = |a| = 14$, utilizando el método de los incrementos decrecientes (`Shellsort`) en una versión matricial *á* la Numerical Recipes[21]:

```
SUBROUTINE  qshell (a)
  IMPLICIT NONE
  INTEGER, DIMENSION   (:), INTENT (INOUT) :: a
  INTEGER, DIMENSION (:,:), ALLOCATABLE    :: t
  INTEGER, DIMENSION (2)                   :: forma
  INTEGER, DIMENSION ( SIZE (a) ) :: g
  INTEGER                          :: i, k, n, p
  !begin
  n = SIZE (a)
  g = HUGE (a)
  i = n / 2
  i = 2 * i
  DO WHILE (i > 1)
    i = i / 2
    p = (n + i - 1) / i
    forma = (/ i , p /)
    ALLOCATE ( t (i,p) )
    t = RESHAPE (SOURCE = a, SHAPE = forma, PAD = g)
    DO WHILE ( ANY (t (:,1:p-1  ) > t (:,2:p  ) ) )
       CALL swap (  t (:,1:p-1:2) , t (:,2:p:2)
     #            , t (:,1:p-1:2) > t (:,2:p:2) )
       CALL swap (  t (:,2:p-1:2) , t (:,3:p:2)
     #            , t (:,2:p-1:2) > t (:,3:p:2) )
    END DO
    a = RESHAPE (t, SHAPE (a) )
    DEALLOCATE (t)
  END DO
END SUBROUTINE
```

en donde `swap` (intercambia) es

```
SUBROUTINE  swap (a,b,m)
  IMPLICIT NONE
  INTEGER, DIMENSION (:,:), INTENT (INOUT) :: a, b
  LOGICAL, DIMENSION (:,:), INTENT (IN)    :: m
  INTEGER, DIMENSION (SIZE(a,1),SIZE(a,2)) :: t
  !begin
  WHERE (m) ; t = a ; a = b ; b = t ; END WHERE
END SUBROUTINE
```

Una “prueba de escritorio” nos conduce a las siguientes etapas:

$$a = \begin{bmatrix} 14 \\ 7 \\ 12 \\ 5 \\ 10 \\ 3 \\ 8 \\ 6 \\ 13 \\ 4 \\ 11 \\ 2 \\ 9 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 14 & 6 \\ 7 & 13 \\ 12 & 4 \\ 5 & 11 \\ 10 & 2 \\ 3 & 9 \\ 8 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 14 \\ 7 & 13 \\ 4 & 12 \\ 5 & 11 \\ 2 & 10 \\ 3 & 9 \\ 1 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 6 \\ 7 \\ 4 \\ 5 \\ 2 \\ 3 \\ 1 \\ 14 \\ 13 \\ 12 \\ 11 \\ 10 \\ 9 \\ 8 \end{bmatrix} \quad ; \tag{8}$$

en donde clasificamos por filas (eventualmente en forma reiterada). Volvemos a hacer otra pasada

pero ahora con otro formato. En la siguiente iteración

$$t = \begin{bmatrix} 6 & 5 & 1 & 12 & 9 \\ 7 & 2 & 14 & 11 & 8 \\ 4 & 3 & 13 & 10 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} 5 & 6 & 1 & 12 & 9 \\ 2 & 7 & 11 & 14 & 8 \\ 3 & 4 & 10 & 13 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} 5 & 1 & 6 & 9 & 12 \\ 2 & 7 & 11 & 8 & 14 \\ 3 & 4 & 10 & 13 & \infty \end{bmatrix} ; \qquad (9)$$

(notar el efecto del "relleno" $\infty$ hecho con el argumento opcional `PAD`). Como `ANY (...)` sigue siendo `true` entonces sigue iterando el lazo `WHILE` más interno

$$t = \begin{bmatrix} 5 & 1 & 6 & 9 & 12 \\ 2 & 7 & 11 & 8 & 14 \\ 3 & 4 & 10 & 13 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 5 & 6 & 12 & 9 \\ 2 & 7 & 8 & 11 & 14 \\ 3 & 4 & 10 & 13 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 5 & 6 & 9 & 12 \\ 2 & 7 & 8 & 11 & 14 \\ 3 & 4 & 10 & 13 & \infty \end{bmatrix} . \qquad (10)$$

y sale del lazo `WHILE` más interno. En la última pasada tendremos

$$\begin{aligned} t &= \begin{bmatrix} 1 & 2 & 3 & 5 & 7 & 4 & 6 & 8 & 10 & 9 & 11 & 13 & 12 & 14 \end{bmatrix} \\ &\rightarrow \begin{bmatrix} 1 & 2 & 3 & 5 & 4 & 7 & 6 & 8 & 9 & 10 & 11 & 13 & 12 & 14 \end{bmatrix} \\ &\rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \end{bmatrix} . \end{aligned} \qquad (11)$$

### 4.4. Instrucciones de asignación matricial

F90/F95 permite que una variedad de operaciones escalares (definidas sobre identificadores simples) puedan ser aplicadas también a arreglos completos. Esto hace que la operación escalar se hace en cada elemento del arreglo. Si una operación involucra varios valores, entonces deben ser arreglos *conformables*, esto es, arreglos del mismo tamaño y forma. Por ejemplo, consideremos la suma escalar y matricial

```
INTEGER                    :: a1, b1, c1
INTEGER, DIMENSION (10,40) :: a2, b2, c2
...
a1 = b1 + c1   ! suma escalar
a2 = b2 + c2   ! suma matricial
...
```

Además, todas las operaciones unarias y binarias las podemos aplicar sobre arreglos, por ejemplo,

```
LOGICAL, DIMENSION (10,20) :: l
REAL   , DIMENSION (10,20) :: a, b
a = a + 1.0    ! suma 1 a cada elemento de A
a = SQRT (a)   ! raiz cuadrada de cada elemento
l = (a == b)   ! asigna T o F para cada elemento
```

También es posible operaciones matriciales sobre intervalos de los índices [como en *Octave*[18] (simil de *Matlab* 3.2 en *Linux* y de libre distribución)]. Una `sección` de un arreglo se representa por el triplete: `valor_inicial :valor_final :paso`. Si omitimos `paso` (stride), entonces el compilador supone `paso=1`, e.g. ver Fig. 1. Cuando las operaciones son realizadas sobre secciones de arreglos, los elementos son seleccionados por su posición, no por su índice, e.g., ver Fig. 2. Finalmente, la construcción `WHERE` permite usar máscaras matriciales. Por ejemplo (sólo en el F95 estándar)

```
REAL, DIMENSION (10,10) :: x
WHERE (x .NE. 0)
  x = 1.0 / x
ELSEWHERE
  x = 0.0
END WHERE
```

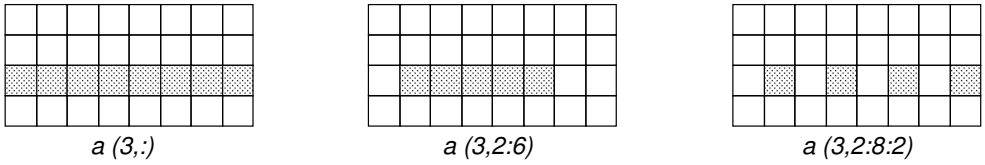


Figura 1: Secciones matriciales en F90.

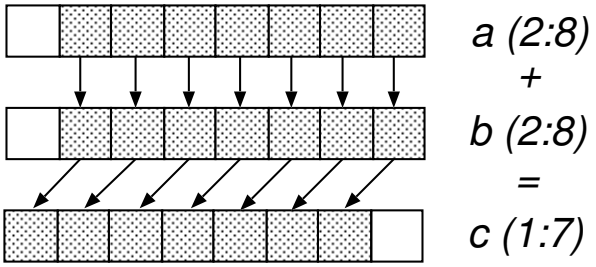


Figura 2: Uso de secciones matriciales en F90.

### 4.5. Funciones matriciales intrínsecas

Todas las funciones intrínsecas en F90 pueden aplicarse tanto a escalares como a arreglos. En el segundo caso, la función se aplica a cada elemento del arreglo. Por ejemplo, si `a` es un arreglo, entonces `ABS (a)` retorna otro arreglo con los valores absolutos de cada elemento de `a`. Además, existen funciones de transformación para pasar de arreglos a escalar y viceversa. Por ejemplo, las intrínsecas `MAXVAL` y `SUM` realizan una operacion de *reducción* sobre un arreglo, devolviendo un valor escalar para el máximo y la suma de los elementos del mismo, respectivamente. El desplazamiento circular `CSHIFT` es frecuente en operaciones con simetría circular, e.g. las transformadas rápidas de Fourier (FFT) o de Wavelet (WFT). El desplazmiento lineal `EOSHIFT` lo es en, por ejemplo, el tratamiento de matrices ralas (e.g. el producto matricial de una matriz banda en formato comprimido por un vector), e.g. ver Fig. 3. Por ejemplo, consideremos el siguiente fragmento en F77,

```
REAL, DIMENSION (0:99) :: x, b
INTEGER                :: i
DO  i = 0, 99
  b (i) = (x ( MOD (i+99, 100) ) + x ( MOD (i+1, 100) ) ) / 2
END DO
```

donde `MOD (a,b)` es la operación `a MOD b`, y la cual puede re-escribirse en F90,

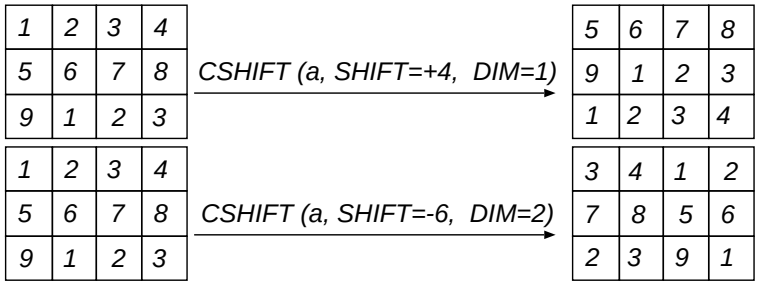


Figura 3: Ejemplo en el uso de la función de desplazamiento circular `CSHIFT`. Izq.: un desplazamiento `SHIFT = +4` se aplica al índice `DIM=1` (por omisión, se asume este índice). Der.: un desplazamiento `SHIFT = -6` se aplica al índice `DIM=2`.

| función matricial | valor retornado |
|---|---|
| `TRANSPOSE (a)` | matriz traspuesta de $a$ |
| `MATMUL (a,b)` | producto matricial $ab$ |
| `DOT_PRODUCT (a,b)` | producto escalar $a^T b$ |
| `SPREAD (x, NCOPIES=m, DIM=d)` | vector $x$ replicado $m$ veces en la dimensión $d$ |
| `MERGE (TSOURCE=a, FSOURCE=b, MASK=m)` | queda $a$ donde $m$ es verdadero sino usa $b$ |
| `PACK (ARRAY=a, MASK=m, VECTOR=v)` | reduce a vector el arreglo $a$ sólo en donde $m$ es verdadero, mientras que $v$ es un vector opcional de relleno |
| `UNPACK (VECTOR=v, MASK=m, FIELD=f)` | expande el vector $v$, el cual debe tener tantos elementos como `True` haya en la máscara $m$ $f$ es conformable con $m$ |
| `CSHIFT (ARRAY=a, SHIFT=s, DIM=d)` | desplazamiento circular de $a$ en $s$ veces, en la dimensión $d$ |
| `EOSHIFT (ARRAY=a, SHIFT=s, DIM=d)` | desplazamiento lineal de $a$ en $s$ veces, en la dimensión $d$ |
| `MAXVAL(ARRAY=a, DIM=d, MASK=m)` | máximo valor de $a$ en la dimensión $d$ cuando $m$ es `True` |
| `SUM (ARRAY=a, DIM=d, MASK=m)` | suma de $a$ en la dirección $d$ cuando $m$ es `True` |
| `RESHAPE (SOURCE=a, SHAPE=s, PAD=p, ORDER=o)` | reconstruye el arreglo $a$ a la forma especificada por $s$ con relleno $p$ y en el orden $o$ opcionales |

Cuadro 2: Algunas funciones matriciales intrínsecas en F95 (lista no exhaustiva).

```
REAL, DIMENSION (1:100) :: x, b, izquierda, derecha
izquierda = CSHIFT (x,+1)
derecha   = CSHIFT (x,-1)
b = (izquierda + derecha) / 2
```

o, más brevemente,

```
REAL, DIMENSION (100) :: x, b
b = ( CSHIFT (x,+1) + CSHIFT (x,-1) ) / 2
```

en todos estos casos `b` es la suma de dos arreglos: `x` desplazado a la izquierda una vez y `x` desplazado a la derecha una vez. Otro ejemplo, sólo válido en F95 (y no en F90 estándar), para construir la matriz identidad `u`, podemos hacer

```
INTEGER, PARAMETER :: n = 100   ! define una constante
INTEGER i                       !
REAL u (n,n)                    ! reserva memoria estatica
u = 0.0                         ! inicia matriz
FOR ALL (i=1:n) u (i,i) = 1.0   ! solo la diagonal principal
```

## 5. HPF para el paralelismo en los datos

El término *paralelismo en los datos* se refiere a la simultaneidad (o concurrencia) lograda cuando la misma operación se hace a la vez con algunos o todos los elementos de un conjunto de datos. Un programa con paralelismo en los datos es una implementación de una secuencia de tales operaciones. A partir de un programa con paralelismo en los datos podemos obtener un algoritmo

paralelo, introduciendo las técnicas de descomposición de dominio a las estructuras de datos. Las operaciones son entonces particionadas frecuentemente, pero no siempre, de acuerdo a la regla del *cómputo local* [el que hará la asignación (owner computes rule)], esto es, aquel procesador que tiene asignado el elemento del miembro izquierdo será el responsable de conseguir todos los datos del miembro derecho, efectuando las operaciones de comunicación necesarias. Típicamente, el programador será el responsable en especificar el modo de la descomposición empleado para el dominio, pero el compilador se encargará de los detalles y de la comunicación. Un objetivo en cálculo paralelo busca "saturar" a todos los procesadores con, aproximadamente, la misma cuota de trabajo (balance de carga), y con la más baja la comunicación posible entre procesadores.
Siguiendo a Press *et al.*,[21] distinguiremos las máquinas paralelas de pequeña escala [Small-Scale Parallel machines (SSP)], donde el número de procesadores $p$ disponibles es mucho menor que el tamaño del problema $n$, de aquellas máquinas masivamente paralelas [Massively Multiprocesador machines (MMP)] en las cuales es mucho mayor $p \gg n$ (o al menos $p \approx O(n^2)$) y, en consecuencia, la memoria disponible será usualmente muy grande. La razón se debe a que se pueden concebir algoritmos paralelos eficientes sobre las MMP pero en la práctica inviables sobre una SSP, e.g. nuestro cluster Gerónimo. Por ejemplo, una triangularización de Gauss en paralelo al estilo del Numerical Recipes, sobre un Sistema de Ecuaciones Algebraicas Lineales (SEAL), en donde se emplea reiteradamente el producto exterior de vectores, lo cual conduce a arreglos temporarios "al vuelo" (mediante `SPREAD`) no explicítamente declarados por el programador, los cuales, al tener un tamaño comparable al de la matriz a triangularizar, enseguida "saturar la memoria disponible. Así, los algoritmos paralelos orientados a una máquina SSP o bien a una MMP pueden ser muy diferentes en cuanto a su concepción y practicidad.
El paradigma de programación con datos en paralelo es (i) de un nivel abstracto mayor, en el sentido de que no se le exige al programador especificar explícitamente las estructuras de comunicación; y (ii) es más restrictivo, porque no todos los algoritmos paralelos pueden implementarse en términos de paralelismo en los datos. Por eso, aunque útil, no es un paradigma universal de programación en paralelo. Aquí sólo nos concentramos en un particionamiento de los datos, mediante construcciones explícitas e implícitas, tales que mejoren la simultaneidad y el balance de carga. Lo interesante de este paradigma es que el programador *sugiere* al compilador, mediante directivas, como debería distribuir y alinear los arreglos sobre los procesadores, mientras que el compilador se encarga de definir todas las operaciones de comunicación.

## 5.1. Concurrencia (o simultaneidad)

Dependiendo del lenguaje empleado, los datos operados en un programa con datos en paralelo pueden ser regulares (e.g. una matriz) o bien irregulares (e.g. una matriz rala o un árbol). En F90 y en HPF estándar las estructuras de datos operan sólo sobre arreglos. En constraste, HPF-extendido y pC++ permiten estructuras de datos adicionales (e.g. árboles, conjuntos). La concurrencia (simultaneidad) puede ser o bien implícita o bien expresada mediante construcciones paralelas explícitas. Por ejemplo, la instrucción de multiplicación `A = B * C` en F90, es una construcción paralela implícita, donde `A`, `B` y `C` son arreglos y es equivalente al producto punto `.*` en `Octave`. Esta instrucción supone también que los arreglos son *conformables*, esto es, los tres arreglos tienen el mismo tamaño y forma. Por otra parte,

```
DO  i = 1, p  ;  DO j = 1, q   ;  DO  k = 1, r
  a (i,j,k) = b (i,j,k) * c (i,j,k)
END DO ; END DO ; END DO
```

es una construcción implícitamente paralela, por lo que el compilador detectará que las iteraciones de los lazos son *independientes* entre si, esto es, en cada iteración del lazo no se usa una variable leída/escrita en otra iteración. Por lo tanto, pueden hacerse en paralelo.

Un programa con datos en paralelo es una cierta secuencia de intrucciones paralelas explícitas e implícitas. La compilación en una computadora paralela con memoria distribuida produce un modelo SPMD (Single Program Multiple Data), en el cual cada procesador ejecuta el mismo código sobre un subconjunto de datos. En muchos casos, el compilador primero particiona los datos en dominios *disjuntos* sobre cada procesador y, a continuación, usa la regla del cómputo local para determinar cúal es el procesador que ejecutará las operaciones aritméticas/lógicas. Cuando el cálculo un procesador requiere los datos que posee otro, entonces el compilador introduce las operaciones de comunicación necesarias. Por ejemplo,

```
REAL                  :: y, s                    ! y, s: escalares
REAL, DIMENSION (100) :: x                       ! x     vector
x = 1.0                                          ! inicia vector
CALL RANDOM (y)                                  ! escalar random
x = y * x                                        ! escalar por vector
DO i = 2, 99 ; x (i) = x (i) + x (i+1) ; ENDDO   ! comunicacion
s = SUM (x)                                      ! comunicacion
```

aquí hay comunicación en el lazo y en la suma y depende de cómo están distribuidos sobre los procesadores los identificadores `x`, `y` y `s`. Por ejemplo, si `x` está distribuido mientras que `y` y `s` están "replicados", entonces la primera y la tercera asignación no requieren comunicación, mientras que la segunda si.

## 5.2. Localidad de los datos

La disposición de los datos es esencial en un programa con paralelismo en los datos, desde que su mapeo sobre los procesadores define la localidad o no de los datos y, por tanto, el rendimiento del código. Por ejemplo, la asignación matricial `A=B*C` puede ser con o sin comunicación, dependiendo si las entradas de cada matriz están en el mismo procesador o no. La identificación de la mejor distribución de los datos en un programa con paralelismo en los datos, es un problema de optimización global y no generalmente tratable por el compilador. Por eso, en los lenguajes con paralelismo en los datos, el programador propone como distribuirlos y para eso en HPF utilizamos la directiva `DISTRIBUTE`. Por ejemplo,

```
!HPF$  PROCESSORS, DIMENSION (16) :: p
       REAL, DIMENSION (1024)     :: x
!HPF$  DISTRIBUTE x (BLOCK) ON TO p
```

aquí el arreglo `x` se distribuye por bloques sobre los 16 procesadores, i.e., el procesador 0 tiene los primeros 1024/16 elementos, el procesador 2 los segundos 1024/16 y asi siguiendo.

## 5.3. Distribución de datos en HPF

Para identificar concurrencia (simultaneidad) en HPF, el compilador analiza, entre otros eleemntos, las operaciones matriciales involucradas, la presencia de la construcción `FORALL` y las directivas `INDEPENDENT`. Aunque HPF es una extensión estándar de F90/F95 es, de todas maneras, un lenguaje complejo, por lo que veremos sólo algunas de sus construcciones más simples. Las asignaciones matriciales especifican concurrencia pero no localidad, esto es, son oportunidades para un cálculo paralelo pero no indican como deberían ser aprovechadas para reducir los costos de comunicación. Por eso, en el HPF se introducen directivas para la distribución de los datos, dando al programador cierto control sobre la locación de los datos. Las *directivas* son *recomendaciones* para el compilador HPF pero **NO** son intrucciones. El compilador puede (y suele) ignorarlas en caso de problemas (e.g. dependencia en los datos). Notemos que las directivas de distribución de los datos tienen fuerte peso en el rendimiento del programa, pero no afectan los

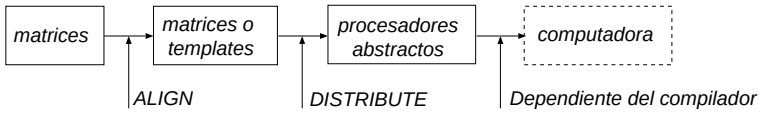


Figura 4: Modelo de alocación de datos en HPF.

resultados obtenidos. El modelo de alocación de datos sobre los procesadores abstractos en HPF comprende dos etapas: con `ALIGN` definimos una relación entre los identificadores matriciales y con `DISTRIBUTE` particionamos un identificador dado (y todos los alineados con éste) sobre el arreglo abstracto de procesadores. Finalmente, el mapeo de los procesadores abstractos sobre los procesadores físicos es dependiente del compilador y no es especificado en el HPF. Todo este proceso lo esquematizamos en la Fig. 4. En particular, mencionemos las siguientes directivas:

1. `PROCESSORS`: sugiere la forma y el tamaño de un arreglo abstracto de procesadores, declarando un identificador para un arreglo abstracto de procesadores. Por ejemplo,

```
!HPF$  PROCESSORS, DIMENSION (9)   :: p, r
!HPF$  PROCESSORS, DIMENSION (3,3) :: q, s
```

   en donde, en la primera línea, declaramos dos arreglos de procesadores abstractos `p` y `r`, mientras que, en la segunda línea, las matrices `q` y `s`. Como hemos empleado números fijos, todas son declaraciones estáticas, pero puede optarse también por una declaración dinámica. No es muy recomendable usarla en la práctica ya que le quita flexibilidad al código. Es mejor que el compilador defina un arreglo abstracto y que el usuario use una bandera en tiempo de ejecución para indicar cuántos procesadores debe usar.

2. `DISTRIBUTE`: sugiere el particionamiento de los arreglos (y de todos los alineadas con estos) sobre un arreglo abstracto de procesadores. En el actual estándar son posibles tres distribuciones. Sea `n` el número total de elementos de un vector `x`, `m` el número de los elementos de `x` asignados a cada procesador, y `p` el número de procesadores. Entonces, podemos tener

```
*          ! sin distribucion
BLOCK (n)  ! distribucion en bloques, por omision: m=piso(n/p)
CYCLIC (n) ! distribucion ciclica   , por omision: m=1)
```

   Esta directiva, opcionalmente, incluye la especificación `ONTO` para definir una distribución particular sobre el arreglo de procesadores abstractos. En caso contrario, la distribución efectiva es definida por el compilador, e.g.

```
!HPF$  PROCESSORES, DIMENSION (16)   :: p
       REAL       , DIMENSION (1024) :: d, e
!HPF$  DISTRIBUTE  d (BLOCK)
!HPF$  DISTRIBUTE  e (BLOCK) ONTO p
```

3. `ALIGN`: sugiere alinear los elementos de las diferentes identificadores matriciales de acuerdo a algún esquema admitido por el compilador. Sugiere cómo deberían colocarse los elementos los cuales, en lo posible, deberían estar en mismo procesador ya que, de esa manera, las operaciones entre datos alineados no requieren comunicación y la eficiencia aumenta. Su forma general es

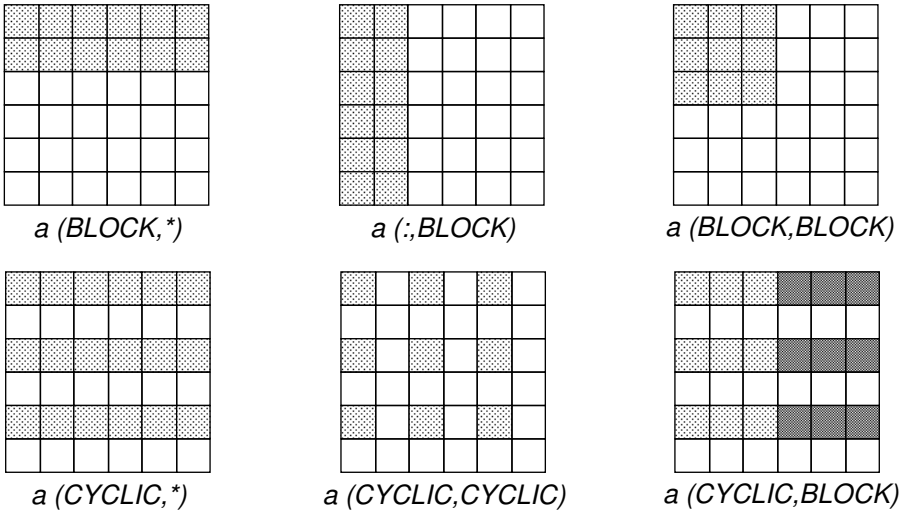


Figura 5: Ejemplos de la directiva DISTRIBUTE en HPF.

```
!HPF$  ALIGN matriz WITH blanco
```

en la cual la `matriz` debería alinearse con el `blanco`, e.g. ver la Fig. 6. Existen varias posibilidades para indicar los alineamientos, incluyendo todo (:), índices mudos (i) o colapso de índice (*), e.g.

```
      REAL, DIMENSION (50,70) :: a
      REAL, DIMENSION (50)    :: b, c
      INTEGER                 :: i
!HPF$  ALIGN c (:) WITH b (:)
!HPF$  ALIGN c (i) WITH b (i)
!HPF$  ALIGN c (i) WITH b (i,*)
```

Por otra parte, en el uso combinado de las directivas `ALIGN` y `DISTRIBUTE`, hay que tener en cuenta que `DISTRIBUTE` se aplica tanto al identificador matricial alcanzado por la directiva, como también a todos aquellos alineados con éste (mediante `ALIGN`). Por eso, no podemos usar `DISTRIBUTE` sobre un arreglo que ya fue alineado con otro. Por ejemplo,

```
!HPF$  PROCESSORES, DIMENSION (16)        :: p
       REAL       , DIMENSION (1024,1024) :: a, b
       REAL       , DIMENSION (1024)      :: c
       INTEGER                            :: i, j
!HPF$  DISTRIBUTE a (BLOCK,*) ON TO p
!HPF$  ALIGN  b (:,:) WITH a (:,:)
!HPF$  ALIGN  c (i,j) WITH a (j,i)
```

4. `INDEPENDENT`: esta directiva sugiere que las iteraciones de un lazo `DO` pueden ser hechas en forma independiente (en cualquier orden) sin que el resultado final se altere. Una forma práctica de darse cuenta si un dado lazo `DO` es independiente o no, es verificar si el resultado final fuera el mismo que cuando se ejecuta el lazo al revés. Esta directiva debe preceder inmediatamente al lazo `DO` al cual se aplica, sugiriéndole al compilador que el lazo es seguro de paralelizar. En su forma más simple, no necesita argumentos adicionales. Por ejemplo,

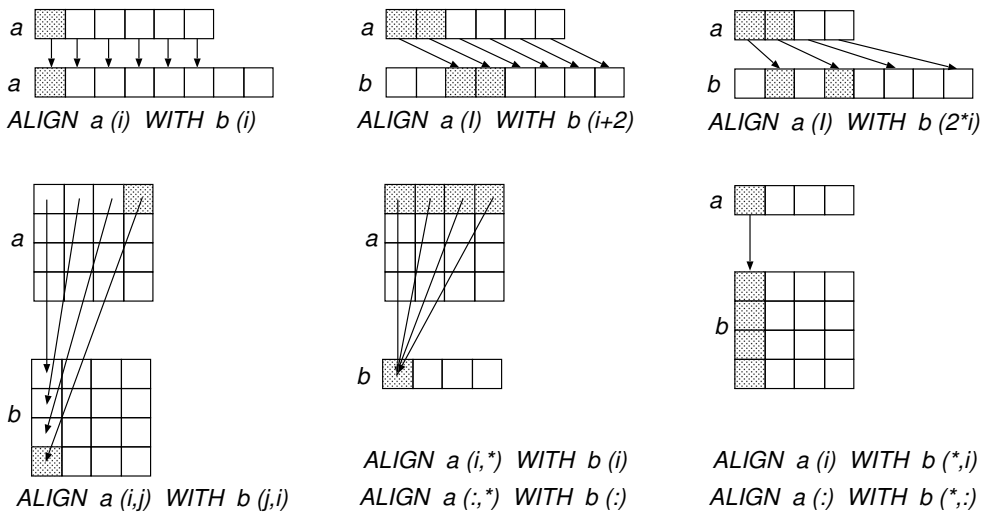


Figura 6: Ejemplos de la directiva `ALIGN` en HPF.

```
!HPF$  INDEPENDENT
       DO i = 1, n
         a ( index (i) ) = b (i)
       END DO
```

aquí le "prometemos" al compilador que (i) `index` no contiene índices repetidos y (ii) que `a` y `b` no comparten memoria. En el siguiente ejemplo, prometemos que los lazos `i, j` son independientes entre si, mientras que el restante `k` no,

```
!HPF$  INDEPENDENT
       DO  i = 1, n1
!HPF$    INDEPENDENT
         DO  j = 1, n2
           DO  k = 1, n3
             ...
           END DO ! k
         END DO ! j
       END DO ! i
```

# Referencias


[1] T. Brandes. Adaptor: Parallel fortran compilation system. http://www.scai.fraunhofer.de/EP-CACHE/adaptor/www/adaptor_home.html.

[2] T. Brandes. Adaptor: Parallel fortran compilation system: Installation and users guides, OpenMP and HPF programmers guides and HPF language reference manual. http://www.scai.fraunhofer.de/EP-CACHE/adaptor/www/adaptor_home.html.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1997.

[4] V.K. Decyk, Norton C.D., and B.K. Szymanski. How to Support Inheritance and Run-Time Polymorphism in Fortran 90. *Computer Physics Communications*, 115:9–17, 1998.

[5] A. Edelman. Parallel scientific computing. Technical report, MIT, 1998.

[6] A.K. Ewing, R.J. Hare, H. Richardson, and A.D. Simpson. Writing Data Parallel Programmes with High Perfomance Fortran. Technical report, The University of Edinburgh, 1999.

| Acrónimo | Significado |
|---|---|
| HPF | High Perfomance Fortran |
| OMP | Open Machine Parallel |
| MPI | Message Passing Interface |
| PVM | Parallel Virtual Machine |
| F77 | Fortran 77 |
| F90 | Fortran 90 |
| F95 | Fortran 95 |
| SPMD | Single Program Multiple Data |
| SSP | Small-Scale Parallel machine |
| MMP | Massively Multiprocesador machine |
| TAD | Tipos Abstractos de Datos |
| PBS | Portable Batch queue System |
| TORQUE | Tera-scale Open-source Resource and QUEue manager |
| **Abreviatura** | **Significado** |
| i.e. | es decir (o esto es), del latín "id est" |
| e.g. | por ejemplo, del latín "exempli gratia" |
| 1D | 1 dimensión |
| 2D | 2 dimensiones |

Cuadro 3: Acrónimos y abreviaturas empleadas.


[7] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[8] G95: the GNU Fortran 95 compiler. http://g95.sourceforge.net/.

[9] M. Hermans. Parallel programming in fortran 95 using OpenMP. Univ. Polit. de Madrid, 2002.

[10] KAI-Intel: C++ and Fortran 95 compilers. http://www.kai.com/.

[11] C.H. Koebel, D.B. Loveman, G.L. Steele, and M.E. Zosel. *The High Perfomance Fortran Handbook*. The MIT Press, 1997.

[12] The Linux Documentation Project, http://sunsite.unc.edu/mdw/linux.html.

[13] A.C. Marshall, J.S. Morgan, and J.L. Schonfelder. Fortran 90 Course Notes. Technical report, Liverpool University, 1997.

[14] A.C. Marshall, J.S. Morgan, and J.L. Schonfelder. HPF Programming Course Notes. Technical report, Liverpool University, 1997.

[15] A.C. Marshall and J.L. Schonfelder. Programming in Fortran 90/95. Technical report, Liverpool University, 2000.

[16] L. P. Meissner. Fortran 90 and 95. Array and Pointer Techniques. Objects, Data Structures and Algorithms. Technical report, Computer Science Department. Univesity of San Francisco, 1998.

[17] C.D. Norton. *Object Oriented Programming Paradigms in Scientific Computing*. PhD thesis, Rensselaer Polytechnic Institute, 1996.

[18] Octave 2.1.33 General Public License, GNU project, i386-redhat-linux-gnu, 2001.

[19] Open Machine Parallel: Fortran (v. 2.0, november 2000) and c++ (v. 2.0, march 2002). http://www.openmp.org/.

[20] Portland Cluster Kit: C++/F95/HPF compilers. http://www.pgroup.com/, file:/usr/local/pgi/doc/index.htm.

[21] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes*. Cambridge University Press, 2nd edition, 1992.