Universidad Autónoma de Madrid

excelencia Campus Internacional UAM CSIC

Escuela Politécnica Superior

Unit 2: Parallel Programming Models

Francisco J. Gómez Arribas

Agenda

2.1 Formas de paralelismo: ILP, SIMD, Multicore, Multithread, coprocesamiento con GPU, distribución de procesos, paralelismo de datos. Conceptos de latencia y ancho de banda

2.2 Modelos de programación paralela con su correspondiente implementación HW/SW.

- 2.2.1 Paradigma SIMD. Conjunto de instrucciones SSE y AVX.
- 2.2.2 Vectorización, dependencias de datos y control, criterios de paralelización de bucles.
- 2.2.3 Paralelismo a nivel de Thread. Memoria compartida.
  - 2.2.3.1 OpenMP: Programación paralela basada en directivas.
  - 2.2.3.2 Limitaciones por compartir memoria. Efectos de False Sharing y Data race.
  - 2.2.3.3 Procesamiento multithread con coprocesador GPU
- 2.2.4 Modelo de paso de mensajes. Memoria distribuida.
- 2.2.5 Programación paralela a nivel de datos (data-parallel programming).
- 2.2.6 Programación heterogénea con hardware específico.
  - 2.2.6.1 Aceleradores basados en FPGA.
  - 2.2.6.2 Plataformas ACAP (Adaptive Compute Acceleration Platform).

Contenidos

- Modelos de Programación Paralela en Sistemas multiprocesador.
  - Características distintivas de cada modelo
  - Ejecución en Threads o Procesos
- Paralelización de bucles
  - Análisis de dependencias.
  - Criterios de paralelización.
  - Principales optimizaciones

Computación Vectorial

```
do i = 0, N-1
   C(i) = A(i) + B(i)
enddo
```

Vectores:

- Dirección de comienzo
- longitud
- paso (*stride*)

Código vectorial

```
LV V1,A(R1)
LV V2,B(R1)
ADDV V3,V2,V1
SV C(R1),V3
```

2000 – 2008 – 2016 – 2024 – … – 2116

dir. com. = 2000 / **longitud** = 16 / **paso** = 8

Computación Vectorial

- **Problemas**

Programas: ¿Todo operaciones vectoriales?

¿Se pueden vectorizar siempre?

```
do i = 0, N-1
    A(i) = A(i) + 1
enddo
```

¡Hay que **desordenar** el código original!

escalarmente: $L_0 +_0 S_0 / L_1 +_1 S_1 / \ldots / L_{N-1} +_{N-1} S_{N-1}$

vectorialmente: $L_0 L_1 \ldots L_{N-1} / +_0 +_1 \ldots +_{N-1} / S_0 S_1 \ldots S_{N-1}$

Computación Vectorial: dependencias estructurales

- **Longitud de los registros** (`Lmax`)
  - **¿Qué hacemos si los vectores son más largos?**

```
do i = 0, N-1
    A(i) = A(i) + 1
enddo
```

strip mining

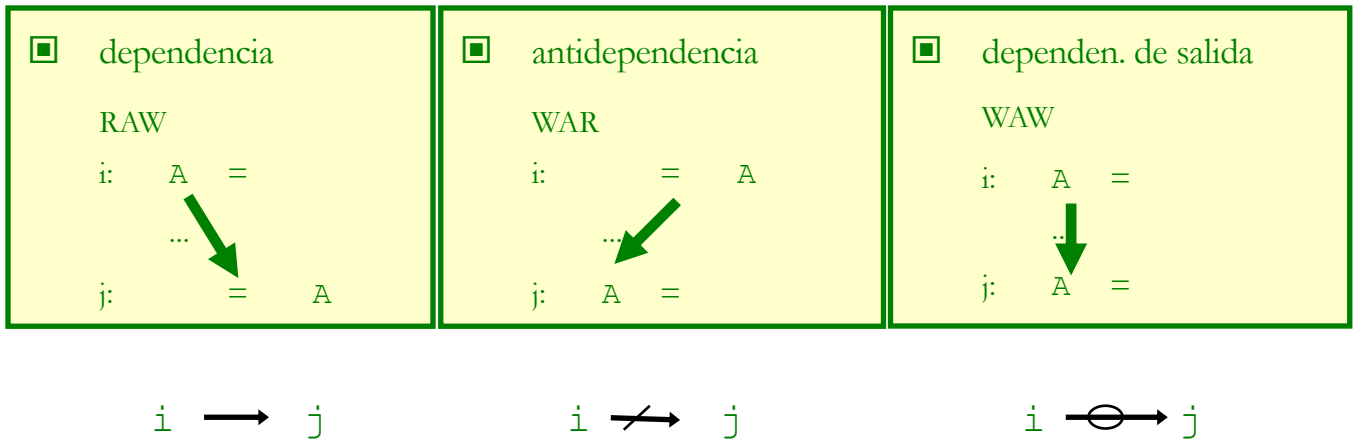
$$T_v = \lceil N/L_{max} \rceil (t_i + t_{buc}) + t_v N$$

```
        MOVI VS,#1    ;sride
        MOVI R1,#N
mas:    MOV VL,R1
        LV V1,A(R2)
        ADDVI V2,V1,#1
        SV A(R2),V2
        ADDI R2,R2,#Lmax
        SUBI R1,R1,#Lmax
        BGTZ R1,mas
```

Paralelización : Análisis de dependencias

dependencias verdaderas

dependencias de nombre

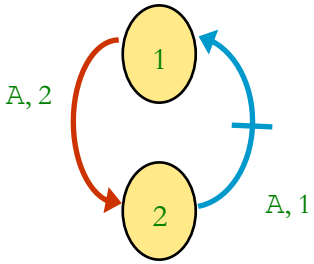


Paralelización : Grafo de dependencias

Bucles

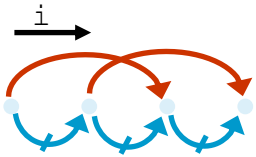
- Grafo de dependencias
- Distancia de la dependencia

```
do i = 2, N-2
1   A(i) = B(i) + 2
2   C(i) = A(i-2) + A(i+1)
enddo
```




```
A(2) = B(2) + 2
C(2) = A(0) + A(3)

A(3) = B(3) + 2
C(3) = A(1) + A(4)

A(4) = B(4) + 2
C(4) = A(2) + A(5)

A(5) = B(5) + 2
C(5) = A(3) + A(6)
```

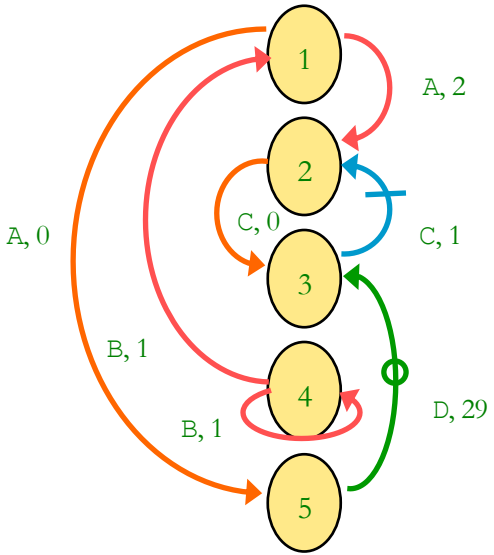
Espacio de iteraciones



Paralelización de Bucles: Análisis de dependencias

- Analice las dependencias del siguiente bucle:

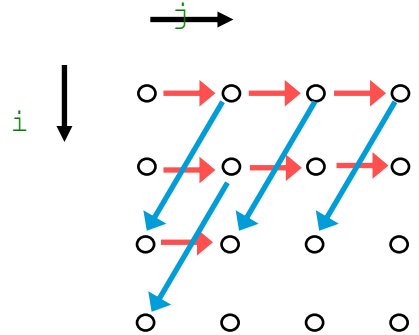
```
do i = 2, N-31
1   A(i) = B(i) + 2
2   C(i) = A(i-2)
3   D(i+1) = C(i) + C(i+1)
4   B(i+1) = B(i) + 1
5   D(i+30) = A(i)
enddo
```

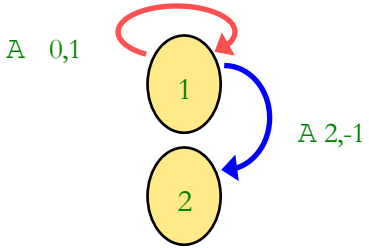


Paralelización de Bucles: Análisis de dependencias

- Espacio de iteraciones

```
do i = 2, N-1
  do j = 1, N-2
  1  A(i,j) = A(i,j-1) * 2
  2  C(i,j) = A(i-2,j+1) + 1
  enddo
enddo
```





Dependencias de datos

- Dado que trabajaremos con bucles las dependencias de datos se pueden dar entre instrucciones de cualquier iteración.
- Si hay una dependencia entre las instrucciones de las iteraciones i1 e i2, diremos que hay una dependencia a distancia i2 – i1.

- Las dependencias se representan mediante dos grafos: **grafo de dependencias** y **espacio de iteraciones.**

Vectorización

- Veamos mediante unos ejemplos cómo generar el código vectorial

  - Ejemplo 1

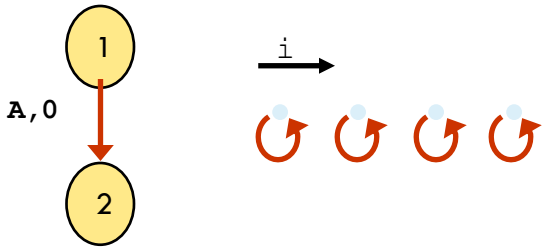
```
do i = 0, N-1
  A(i) = B(i) + C(i)
enddo
```

```
MOVI    VL,#N
MOVI    VS,#1

LV      V1,B(R1)
LV      V2,C(R1)
ADDV    V3,V1,V2
SV      A(R1),V3
```

Vectorización

- Ejemplo 2

```
do i = 0, N-1
  A(i) = B(i) + C(i)
  D(i) = A(i)
enddo
```

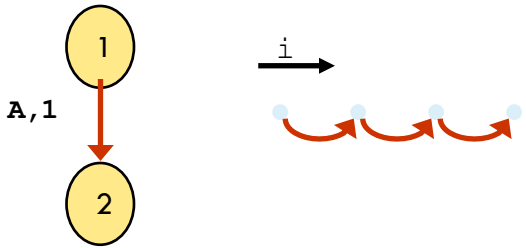



```
MOVI    VL,#N
MOVI    VS,#1

LV      V1,B(R1)
LV      V2,C(R1)
ADDV    V3,V1,V2
SV      A(R1),V3


SV      D(R1),V3
```

Vectorización

- Ejemplo 3

```
do i = 1, N-1
  A(i) = B(i) + C(i)
  D(i) = A(i-1)
enddo
```



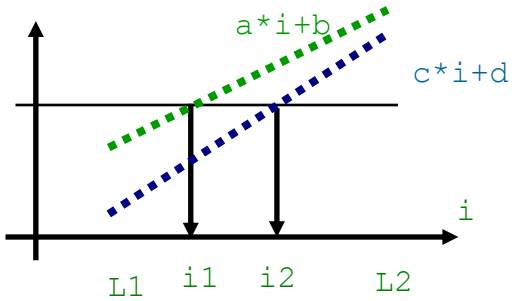

```
MOVI    VL,#N-1
MOVI    VS,#1

LV      V1,B+1(R1)
LV      V2,C+1(R1)
ADDV    V3,V1,V2
SV      A+1(R1),V3

LV      V4,A(R1)
SV      D+1(R1),V4
```

Paralelización de Bucles: Test de dependencias

- Test de dependencias: únicamente para funciones lineales del índice del bucle.

```
do i = L1, L2
  X(a*i+b) =
           = X(c*i+d)
enddo
```

?



$$\frac{d-b}{MCD(c,a)} \notin Z \rightarrow \text{no hay depend.}$$

Paralelización de Bucles: Repartir iteraciones

- Paralelizar el código significa repartir las iteraciones de un bucle a threads o procesadores.
  - Pero hay que respetar las dependencias de datos.
  - El problema principal son las dependencias de distancia > 0, y sobre todo aquellas que forman ciclos en el grafo de dependencias.
- Siempre es posible ejecutar un bucle en P threads o procesadores, añadiendo sincronización para asegurar que se respetan las dependencias de datos…

… lo que no significa que siempre sea sensato hacerlo, pues hay que tener en cuenta el coste global: cálculo + sincronización (comunicación).

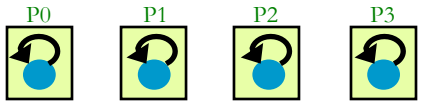
Paralelización de Bucles: Repartir iteracciones

- Objetivos:
  - **Repartir las iteraciones de un bucle entre los procesadores, para que se ejecuten "simultaneamente".**
  - **Siempre que se pueda, que se ejecuten de manera independiente, sin necesidad de sincronizar (dependencias de distancia 0).**
  - **En función de las características del sistema (comunicación, reparto de tareas…) hay que intentar que las tareas tengan un cierto tamaño.**

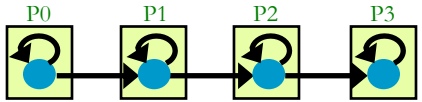
Limitaciones :Tal vez sólo se pueda utilizar un número limitado de procesadores.

Paralelización de Bucles: Casos habituales

```
do i = 0, N-1
    A(i) = A(i) + 1
    B(i) = A(i) * 2
enddo
```

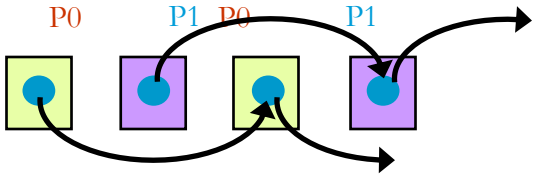



```
do i = 0, N-2
    A(i) = B(i) + 1
    B(i+1) = A(i) * 2
enddo
```



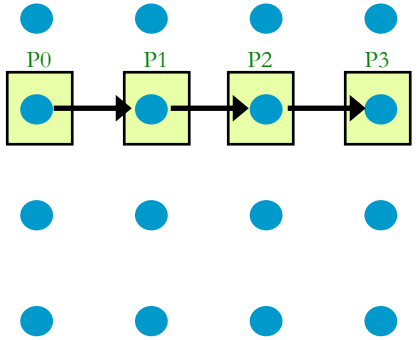
?

```
do i = 0, N-3
    A(i+2) = A(i) + 1
enddo
```

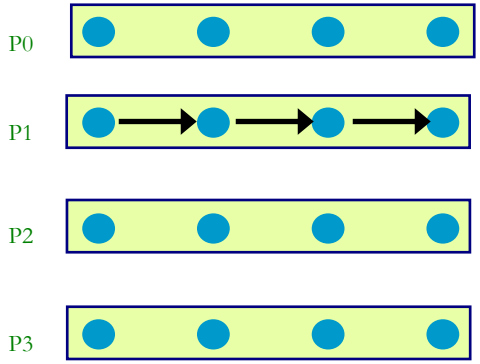


Paralelización de Bucles: Casos habituales

```
do i = 0, N-1
 do j = 1, M-1
   A(i,j) = A(i,j-1) + 1
  enddo
enddo
```




```
do i = 0, N-1
 do j = 1, M-1
   A(i,j) = A(i,j-1) + 1
  enddo
enddo
```



Paralelización de Bucles: Criterios

- Si todas las dependencias son de distancia 0 (las iteraciones son independientes), éstas se pueden repartir como se quiera entre los procesadores, sin tener que sincronizarlas: doall.

- Si hay dependencias entre iteraciones, pero todas van hacia adelante, se pueden sincronizar mediante barreras: forall (o doall + barrier).

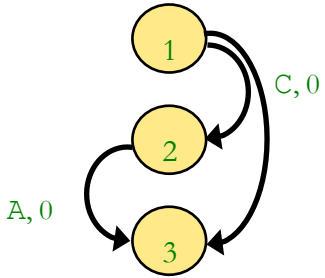
- Si las dependencias forman ciclos, hay que utilizar sincronización punto a punto: doacross.

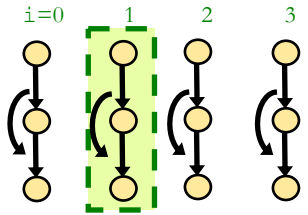
Paralelización de Bucles: iteraciones independientes

- Las iteraciones son independientes, por lo que el reparto puede hacerse como se quiera: doall.

```
do i = 0, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
  D(i) = C(i) / A(i)
enddo
```

```
doall i = 0, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
  D(i) = C(i) / A(i)
enddoall
```





Paralelización de Bucles: dependencia forward

- Hay dependencias entre iteraciones, pero todas van "hacia adelante": forall.

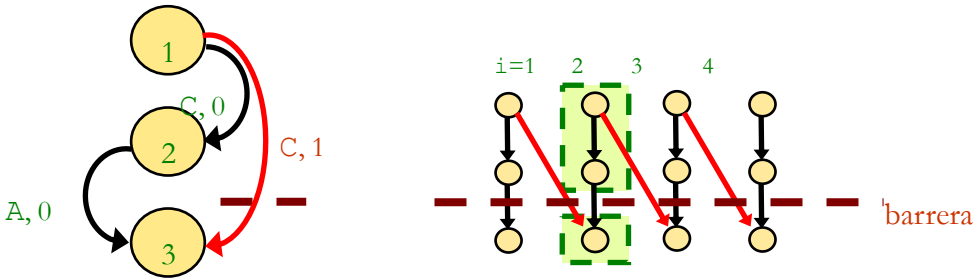
- Cada dependencia puede sincronizarse con una barrera: los procesos esperan a que todos hayan ejecutado una determinada instrucción antes de pasar a ejecutar la siguiente.

- Hay más sincronización que la estrictamente necesaria, pero es sencillo de implementar.

Paralelización de Bucles: dependendencia forward

```
do i = 1, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
  D(i) = C(i-1) / A(i)
enddo
```

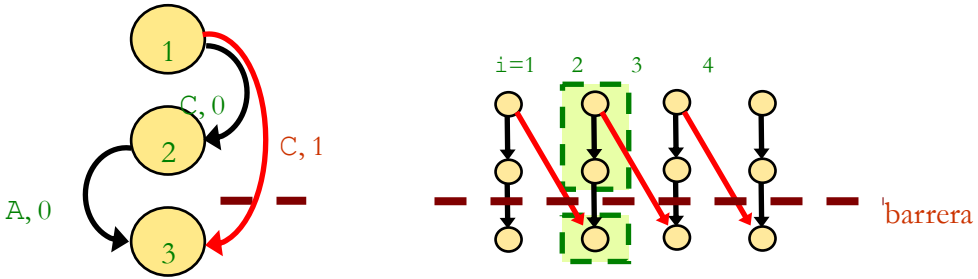
```
forall i = 1, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
  BARRERA (...)
  D(i) = C(i-1) / A(i)
endforall
```



Paralelización de Bucles: dependencia forward

```
do i = 1, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
  D(i) = C(i-1) / A(i)
enddo
```

```
doall i = 1, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
enddoall
[ BARERRA (...) ]
doall i = 1, N-1
  D(i) = C(i-1) / A(i)
enddoall
```



Paralelización de Bucles: dependencia con ciclos

- Las dependencias forman ciclos: doacross, sincronización punto a punto (productor / consumidor).

  - Sincronización de las dependencias mediante vectores de eventos:

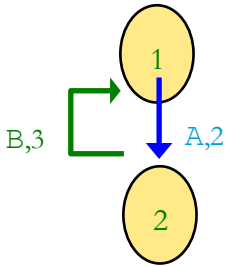
    `post(vA,i)` → `vA(i) := 1`

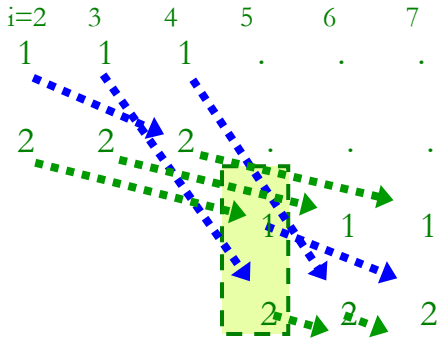
    `wait(vA,i)` → esperar a que `vA(i) = 1`

  - Si `vA(i) = 0`, aún no se ha ejecutado la iteración i de la instrucción que se está sincronizando.

Paralelización de Bucles: dependencia doacross

```
do i = 2, N-2
  A(i) = B(i-2) + 1
  B(i+1) = A(i-2) * 2
enddo
```

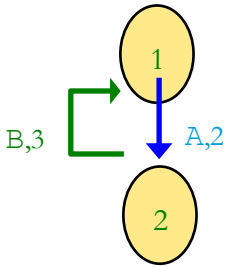


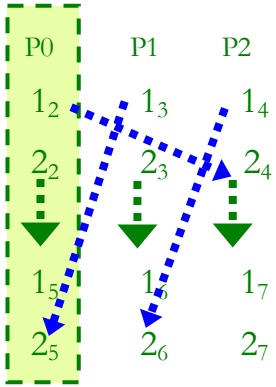



```
doacross i = 2, N-2

  wait (vB,i-3)
  A(i) = B(i-2) + 1
  post (vA,i)

  wait (vA,i-2)
  B(i+1) = A(i-2) * 2
  post (vB,i)

enddoacross
```

Paralelización de Bucles: dependencia doacross

```
do i = 2, N-2
  A(i) = B(i-2) + 1
  B(i+1) = A(i-2) * 2
enddo
```



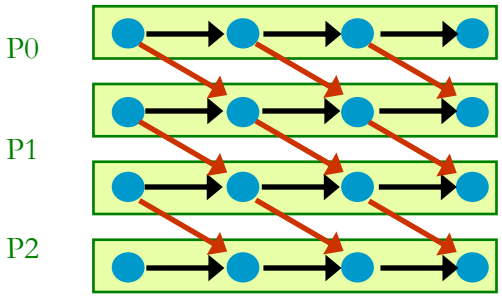



```
doacross i = 2, N-2, 3
  wait (vB,i-3)
  A(i) = B(i-2) + 1
  post (vA,i)
  wait (vA,i-2)
  B(i+1) = A(i-2) *
  post (vB,i)
enddoacross
```

Paralelización de Bucles: dependencia doacross

- Vectores de eventos en dos dimensiones

```
do i = 0, N-2
 do j = 0, N-2
   A(i+1,j+1) = A(i+1,j) + 1
   B(i,j) = A(i,j)
 enddo
enddo
```

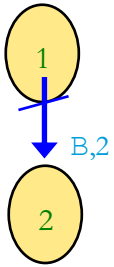



```
doacross i = 0, N-2
 do j = 0, N-2
   A(i+1,j+1) = A(i+1,j) + 1
   post (vA,i,j)
   wait (vA,i-1,j-1)
   B(i,j) = A(i,j)
 enddo
enddoacross
```

Paralelización de Bucles: dependencia doacross

- Antidependencias / Dependencias de salida
- Si son entre procesos, hay que sincronizarlas.

```
do i = 0, N-3
  A(i) = B(i+2) / A(i)
  B(i) = B(i) + C(i)
enddo
```



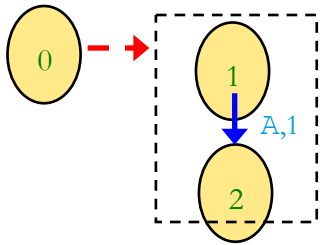

```
doacross i = 0, N-3
  A(i) = B(i+2) / A(i)
  post (vB,i)
  wait (vB,i-2)
  B(i) = B(i) + C(i)
enddoacross
```

```
LD B(i+2)
post
...
wait
ST B(i)
```

Paralelización de Bucles: dependencia doacross

- Instrucciones de tipo if

```
do i = 1, N-1
  if (B(i) > 0) then
    A(i) = A(i) + B(i)
    C(i) = A(i-1) / 2
  endif
enddo
```




```
doacross i = 1, N-1

  if (B(i) > 0) then
     A(i) = A(i) + B(i)
     post (vA,i)
     wait (vA,i-1)
     C(i) = A(i-1) / 2

   else post (vA,i)

   endif
enddoacross
```

Paralelización de Bucles: Optimizaciones

- 0. Deshacer la definición de constantes y las variables de inducción.

- 1. Eliminar todas las dependencias que no son intrínsecas al bucle. Por ejemplo:

```
do i = 0, N-1
  X = A(i) * B(i)
  C(i) = SQRT(X)
  D(i) = X - 1
enddo
```

convertir X en una variable privada

```
doall i = 0, N-1
  X(i) = A(i) * B(i)
  C(i) = SQRT(X(i))
  D(i) = X(i) - 1
enddoall
```

Paralelización de Bucles: Optimizaciones

❖ 2. Fisión del bucle.

Si una parte del bucle hay que ejecutarla en serie, romper el bucle convenientemente y ejecutar lo que sea posible en paralelo.

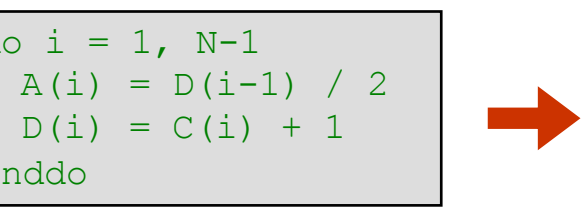
```
do i = 1, N-1
  A(i) = A(i-1) / 2
  D(i) = D(i) + 1
enddo
```

```
do i = 1, N-1
  A(i) = A(i-1) / 2
enddo

doall i = 1, N-1
  D(i) = D(i) + 1
enddoall
```

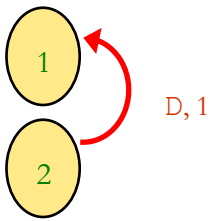
Paralelización de Bucles: Optimizaciones

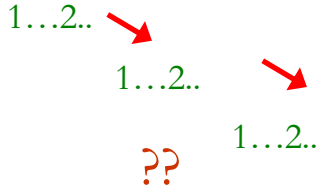
- 3. Ordenar las dependencias (hacia adelante).

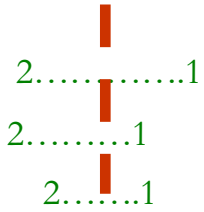
```
do i = 1, N-1
  A(i) = D(i-1) / 2
  D(i) = C(i) + 1
enddo
```



```
forall i = 1, N-1
  D(i) = C(i) + 1
  BARRERA (...)
  A(i) = D(i-1) / 2
endforall
```



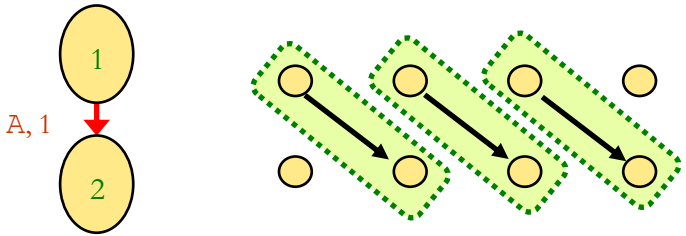




Paralelización de Bucles: Optimizaciones

- 4. Alinear las dependencias: *peeling*.

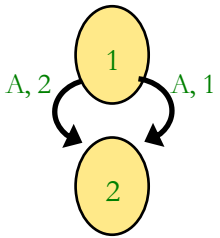
```
do i = 1, N-1
  A(i) = B(i)
  C(i) = A(i-1) + 2
enddo
```

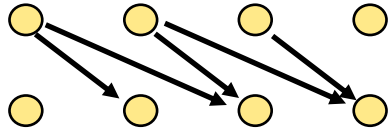



```
C(1) = A(0) + 2
doall i = 1, N-2
  A(i) = B(i)
  C(i+1) = A(i) + 2
enddoall
A(N-1) = B(N-1)
```

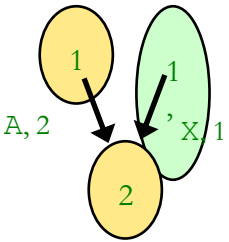
Paralelización de Bucles: Optimizaciones

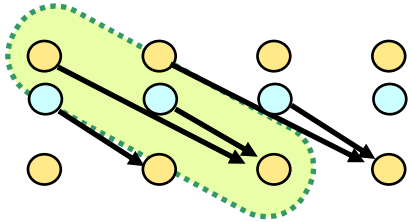
```
do i = 2, N-1
  A(i) = B(i)
  C(i) = A(i-1) + A(i-2)
enddo
```





```
do i = 2, N-1
  A(i) = B(i)
  X(i) = B(i)
  C(i) = X(i-1) + A(i-2)
enddo
```



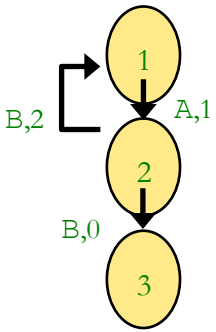


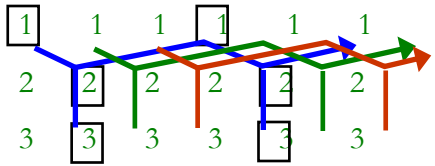
```
C(2) = A(1) + A(0)
C(3) = B(2) + A(1)

doall i = 2, N-3
  A(i) = B(i)
  X(i+1) = B(i+1)
  C(i+2) = X(i+1) + A(i)
enddoall

A(N-2) = B(N-2)
A(N-1) = B(N-1)
```

Paralelización de Bucles: Optimizaciones

5. Generar hilos independientes

```
do i = 0, N-3
  A(i+1) = B(i) + 1
  B(i+2) = A(i) * 3
  C(i) = B(i+2) - 2
enddo
```



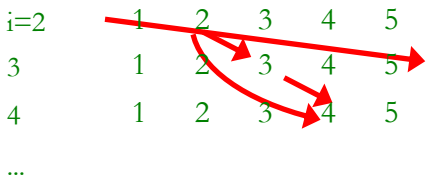


```
B(2) = A(0) * 3
C(0) = B(2) - 2

doall k = 0, 2
 do i = pid, N-4, 3
   A(i+1) = B(i) + 1
   B(i+3) = A(i+1) * 3
   C(i+1) = B(i+3) - 2
 enddo
enddoall

A(N-2) = B(N-3) + 1
```

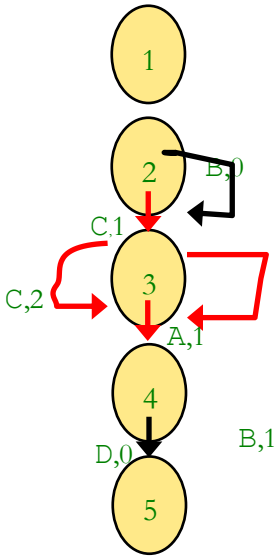
Paralelización de Bucles: Optimizaciones

6. Minimizar la sincronización

```
do i = 2, N-1
  B(i) = B(i) + 1
  C(i) = C(i) / 3
  A(i) = B(i) + C(i-1)
  D(i) = A(i-1) * C(i-2)
  E(i) = D(i) + B(i-1)
enddo
```

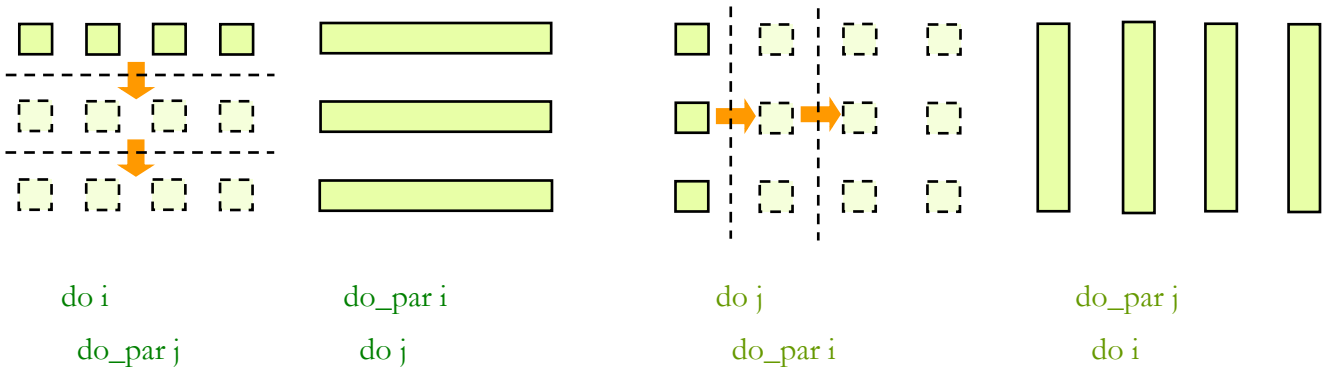



```
doacross i = 2, N-1
  B(i) = B(i) + 1
  C(i) = C(i) / 3
  post (vC,i)

  wait (vC,i-1)
  A(i) = B(i) + C(i-1)
  post (vA,i)

  wait (vA,i-1)
  D(i) = A(i-1) * C(i-2)
  E(i) = D(i) + B(i-1)
enddoacross
```



Paralelización de Bucles: Optimizaciones

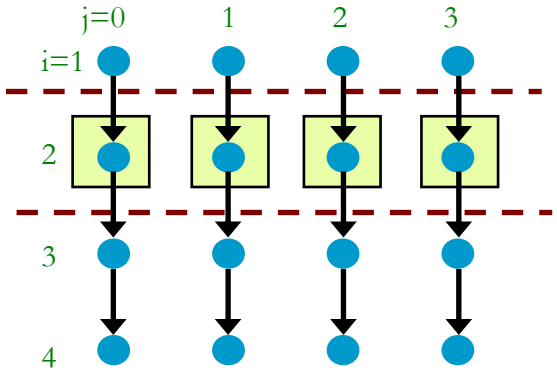
- 7. Tratamiento de bucles: intercambio.



Paralelización de Bucles: Optimizaciones

- Ejemplo:

```
do i = 1, N-1
  do j = 0, N-1
    A(i,j) = A(i-1,j) + 1
  enddo
enddo
```

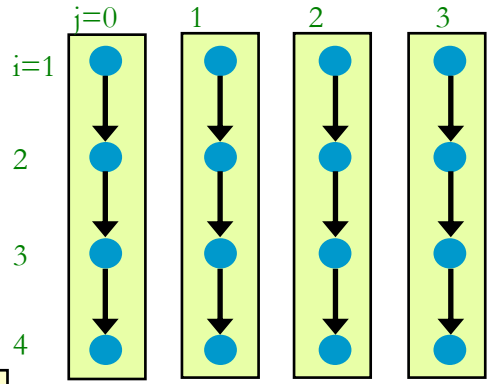



```
do i = 1, N-1
  doall j = 0, N-1
    A(i,j) = A(i-1,j) + 1
  enddoall
enddo
```

Paralelización de Bucles: Optimizaciones

- Ejemplo:

```
do j = 0, N-1
  do i = 1, N-1
     A(i,j) = A(i-1,j) + 1
  enddo
enddo
```



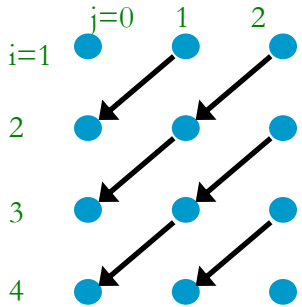

```
doall j = 0, N-1
  do i = 1, N-1
    A(i,j) = A(i-1,j) + 1
  enddo
enddoall
```

Paralelización de Bucles: Optimizaciones

- 8. **Tratamiento de bucles:** cambio de sentido.

```
do i = 1, 100
 do j = 0, 2
   A(i,j) = A(i,j) - 1
   D(i) = A(i-1,j+1) * 3
 enddo
enddo
```

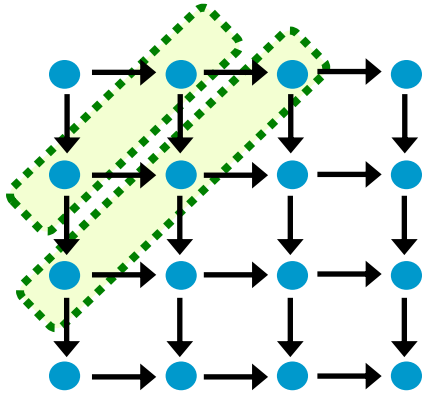
```
do j = 2, 0, -1
 doall i = 1, 100
   A(i,j) = A(i,j) - 1
   D(i) = A(i-1,j+1) * 3
 enddoall
enddo
```



Paralelización de Bucles: Optimizaciones

9. *Skew*

```
do i = 1, N
 do j = 1, N
   A(i,j) = A(i-1,j) + A(i,j-1)
 enddo
enddo
```



```
do j = 1, 2N-1
 doall i = max(1,j-N+1), min(j,N)
   A(i,j-(i-1)) = A(i-1,j-(i-1)) + A(i,j-1-(i-1))
 enddoall
enddo
```

Paralelización de Bucles: Optimizaciones

10. Otras optimizaciones típicas

Aumento del tamaño de grano y reducción del *overhead* de la paralelización.

- Juntar dos bucles en uno (¡manteniendo la semántica!): fusión.
- Convertir un bucle de dos dimensiones en otro de una sola dimensión: colapso o coalescencia.

…

Paralelización de Bucles: Optimizaciones

- ¿Cómo se reparten las iteraciones de un bucle entre los procesadores?
- Si hay tantos procesadores como iteraciones, tal vez una por procesador.

- Pero si hay menos (lo normal), hay que repartir.
- El reparto puede ser:

  - estático: en tiempo de compilación.
  - dinámico: en ejecución.

Paralelización de Bucles: reparto de iteraciones

- El objetivo: intentar que el tiempo de ejecución de los trozos que se reparten a cada procesador sea similar, para evitar tiempos muertos (*load balancing*).

  - En todo caso, hay que terner en cuenta las dependencias (sincronización), el tamaño de grano, la localidad de los accesos y el coste del propio reparto.

Paralelización de Bucles: reparto de iteraciones

- Lo que se ejecuta en cada procesador se decide en tiempo de compilación. Es por tanto una decisión prefijada.
  - Planificación estática

- Cada proceso tiene una variable local que lo identifica, pid [0..P-1].
- Dos opciones básicas: reparto consecutivo y reparto entrelazado.

Paralelización de Bucles: reparto de iteraciones

- Consecutivo

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
principio = pid * N/P
fin = (pid+1) * N/P - 1
do i = principio, fin
   ...
enddo
```

- Entrelazado

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

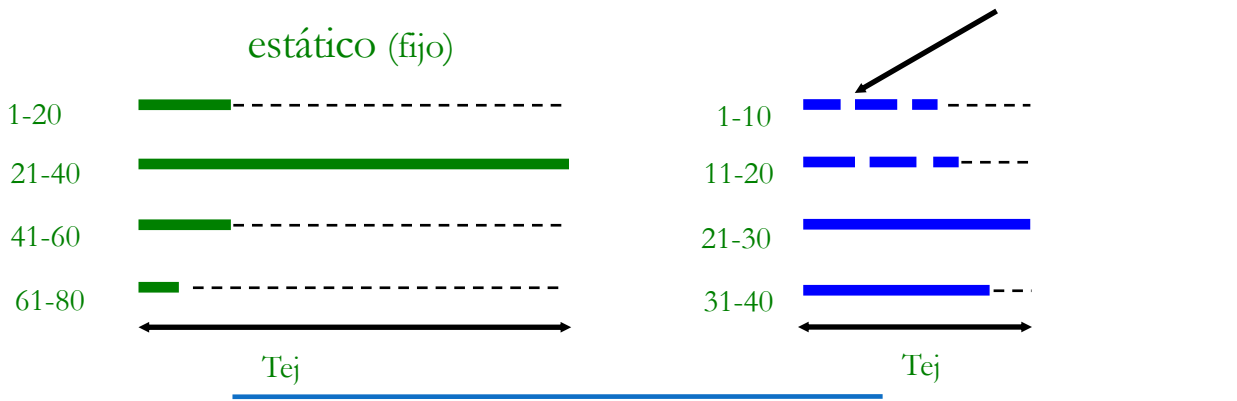
```
do i = pid, N, P
   ...
enddo
```

- No añade carga a la ejecución de los *threads*.
- Pero no asegura el equilibrio de la carga entre los procesos.
- Permite cierto control sobre la localidad de los accesos a cache.

Paralelización de Bucles: reparto de iteraciones

- Equilibrio en el reparto de carga

```
do i = 1, 80
  if (A(i) > 0) calcular()
enddo
```



Paralelización de Bucles: reparto de iteraciones

- Para intentar mantener la carga equilibrada, las tareas se van escogiendo en tiempo de ejecución de un cola de tareas. Cuando un proceso acaba con una tarea (un trozo del bucle) se asigna un nuevo trozo.
  - Planificación dinámica
- Dos opciones básicas: los trozos que se van repartiendo son de tamaño constante o son cada vez más pequeños.

Paralelización de Bucles: reparto de iteraciones

- *Self / Chunk scheduling*

Las iteraciones se reparten una a una, o por trozos de tamaño Z.

Añade carga a la ejecución de los *threads*.
Hay que comparar ejecución y reparto.

```
LOCK (C);
  mia = i;
  i = i + Z;            Z = 1 → self
UNLOCK (C);
while (mia <= N-1)
  do j = mia, min(mia+Z-1, N-1)
    ...
  enddo
  LOCK (C)
   mia = i;
   i = i + Z;
  UNLOCK (C)
endwhile
```

Paralelización de Bucles: reparto de iteraciones

- *Guided / Trapezoidal*

  Los trozos de bucle que se reparten son cada vez más pequeños según nos acercamos al final.

  - *Guided* : parte proporcional de lo que queda por ejecutar:

$$Z_s = (N - i) / P \qquad \text{(entero superior)}$$

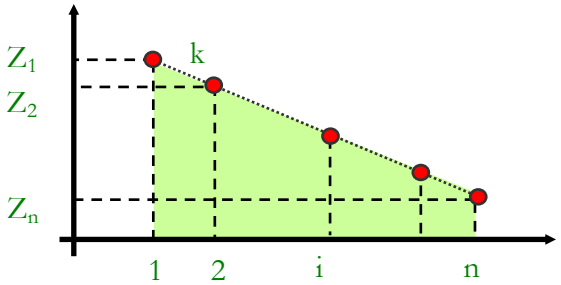
que equivale a:

$$Z_i = Z_{i-1} (1 - 1/P)$$

Paralelización de Bucles: reparto de iteraciones

- *Trapezoidal*: reduciendo el trozo anterior en una constante:

$Z_i = Z_{i-1} - k$



op. de planificación

$$\sum_{s=1}^{n} Z_s = \frac{Z_1 + Z_n}{2} n = \frac{Z_1 + Z_n}{2}\left(\frac{Z_1 - Z_n}{k} + 1\right) = N \quad \rightarrow \quad k = \frac{Z_1^2 - Z_n^2}{2N - (Z_1 + Z_n)}$$

Paralelización de Bucles: reparto de iteraciones

- En general, el reparto dinámico busca un mejor equilibrio en el reparto de carga, pero:
  - hay que considerar la carga que se añade (*overhead*), en relación al coste de las tareas que se asignan.
  - hay que considerar la localidad en los accesos a los datos y los posibles problemas de falsa compartición.

Paralelización de Bucles: reparto de iteraciones

- **Ejemplo de reparto** (1.000 iteraciones, 4 procesadores):
  - *chunk* (Z = 100)

    100 100 100 100 100 100 100 100 100 100 (10)

  - *guided*

    quedan: 1000 750 562 421 … 17 12 9 6 4 3 2 1

    se reparten: 250 188 141 106 … 5 3 3 2 1 1 1 1 (22)

  - *trapezoidal* ($Z_1 = 76, Z_n = 4 \gg k = 3$)

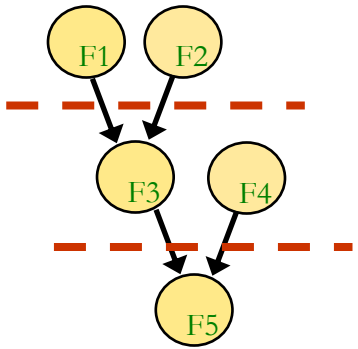
    76 73 70 67 … 16 13 10 7 4 (25)

Paralelización de Bucles: secciones paralelas

- Paralelismo a nivel de procedimiento o función:

  - modelo Fork / Join

  - *Parallel sections*




```
Fork
   F1
   F2
Join
Fork
   F3
   F4
Join
F5
```

```
doall k = 0, 1
   if (pid = 0) then F1
   if (pid = 1) then F2
endoall
[barrera]
doall k = 0, 1
   if (pid = 0) then F3
   if (pid = 1) then F4
endoall
F5
```