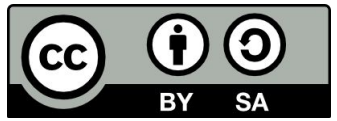
Computación de Altas Prestaciones

Modelo de programación basado en directivas con paralelismo en control

José Luis Risco Martín
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid



Índice

Bibliografía:

- The Fortran Programming Guide. Chapter 10. https://docs.oracle.com/cd/E19957-01/806-3593/806-3593.pdf.

Introducción

Problemas de la programación por medio de threads

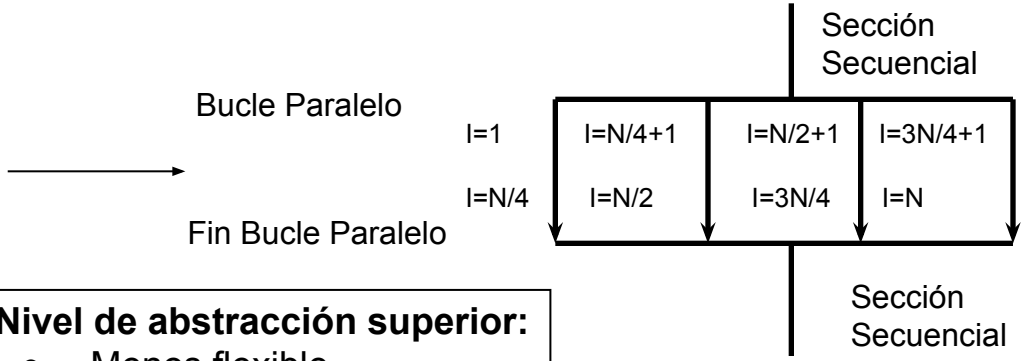
- Análisis del código
- Recodificación
- Incluir primitivas *multithreading*

No cambiamos el lenguaje pero ...

Paralelización por medio de directivas

- Ejecución de un **bucle** sobre múltiples procesos

```
C$PAR DOALL
      DO I=1, N
        A(I) = B(I)
      END DO
```



Nivel de abstracción superior:

- Menos flexible
- Más portable
- Menos eficiente

Fortran Programming Guide
https://docs.oracle.com/cd/E19957-01/806-3593/806-3593.pdf

Introducción

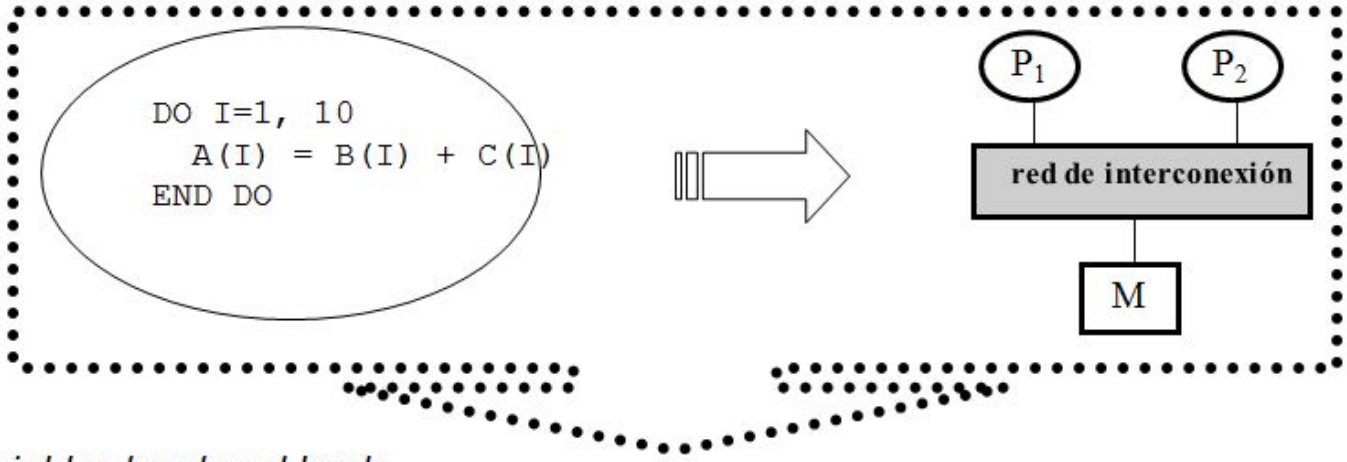
- Paralelización explícita de un programa secuencial mediante la inclusión de directivas de compilación:
  - Informan del paralelismo en la aplicación
- Se trata de un paralelismo en control
  - El trabajo se reparte entre los procesadores (iteraciones de bucle o subrutinas)
  - Los procesadores sincronizan periódicamente sus actividades

ATENCIÓN: Debemos diferenciar estas directivas de las siguientes

- **Directivas de ayuda al paralelizador automático:**
  - Guían en la paralelización automática ayudando en el análisis de dependencias
- **Directivas de optimización manual**
  - Informan de las optimizaciones que debe realizar el compilador a nivel de bucle

Ejemplo Sencillo de Paralelización por Medio de Directivas

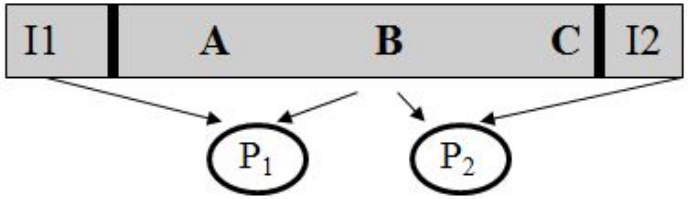
La paralelización con memoria compartida no implica que toda las variables sean compartidas (también hay locales)



Variables locales al bucle

```
DO I1=1,2,3,4,5
  A(I1) = B(I1) + C(I1)
END DO
```

```
DO I2=6,7,8,9,10
  A(I2) = B(I2) + C(I2)
END DO
```

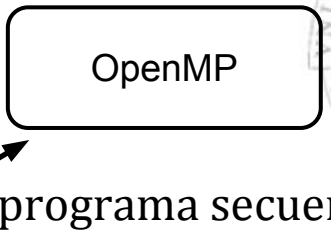


Tipos de Directivas

- Directivas de control de repartición de subrutinas (grano medio y grueso):
  - Indican los fragmentos de código, funciones o subrutinas, que se deben ejecutar concurrentemente
- Directivas de control de repartición de iteraciones de un bucle (grano fino):
  - Indican cómo distribuir las iteraciones de un bucle entre los threads

No todos los compiladores aportan ambos tipos de directivas. Por ejemplo, las directivas PAR *de Solaris o* DOACROSS *de SGI solo indican paralelismo a nivel de bucle*

Ventajas e Inconvenientes



- Ventajas:
  - Es la manera más rápida de ejecutar en paralelo un programa secuencial
  - No se modifica el programa secuencial (portabilidad)
  - Apropiado para todo tipo de grano (grueso, medio y fino)
  - En una primera aproximación se puede realizar una paralelización automática
- Inconvenientes:
  - Se obtienen eficiencias pobres si la paralelización es a grano fino
  - Solo válido para sistemas con multiprocesamiento simétrico
  - En computadores ccNUMA no se tiene en cuenta la distribución de los datos
  - Aportan directivas nuevas para distribución de datos:
    - CRAFT del Cray T3x
    - En SGI Origin 2000 directivas tipo HPF

Local y Compartida

- Una variable o array es local (privado) cuando es local cada una de las iteraciones de un bucle (thread)
  - El valor asignado a una variable privada en una iteración no se propaga al resto
  - Cada thread tiene una copia local no inicializadas de las variables y arrays especificados en la lista
    - En principio sólo existen durante la ejecución del bucle (temporal)
    - En el caso del contador del bucle, recorren todo el espacio de iteraciones asignado al thread

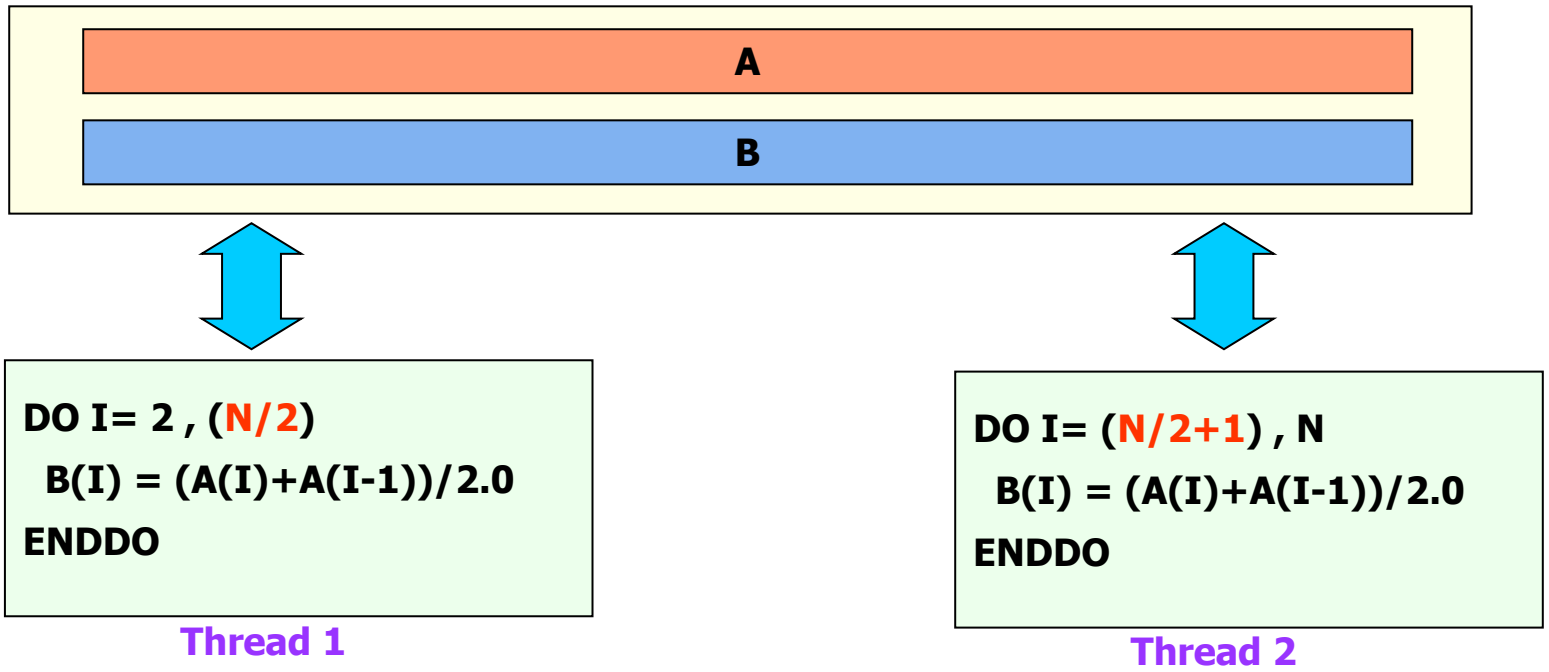
"Una variable debe ser local si no depende de otra iteración del bucle y su valor se usa únicamente dentro de la iteración"

- Una variable o array compartida es una variable compartida por todas las iteraciones (thread)
  - El valor asignado a una variable compartida puede ser visto por otras iteraciones del bucle
  - Todos los threads comparten la única copia de las variables y arrays especificados en la lista
  - No garantiza la exclusión mutua en el acceso a los datos compartidos

Local y Compartida

```
C$ Declaración de I como local y A y B como compartidas
     DO 3 I=1, 1000
          A(I) =B(I)
3    CONTINUE
```

- El ejecutable da lugar a un número de threads determinado por una variable de entorno, las variables A y B están compartidas, pero I es local a cada *thread* y cogerá un valor inicial y final dependiendo del número de *threads*



Local y Compartida

Roja = local

```
. . .
DO i = 1, n
  a(1) = b(i)
  DO j = 2, n
    a(j) = a(j-1) + b(j)*c(j)
  END DO
  x(i) = f(a)
END DO
. . .
```

```
. . .
DO i = 1, n
  a(i) = y
END DO
. . .
```

Última Local

Última local

- Especifica variables o arrays que son locales y cuyo valor en la última iteración del bucle DO se usará tras la terminación del bucle

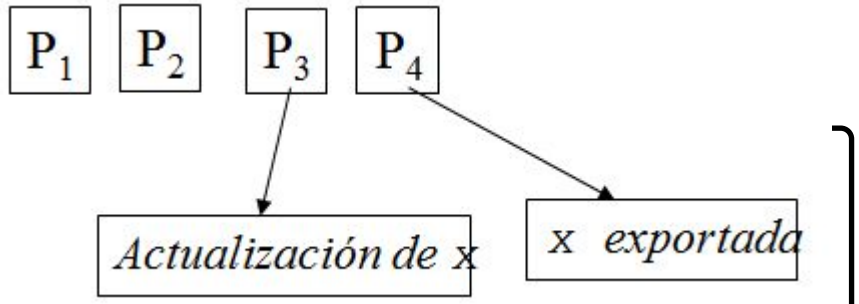
```
C$ Declara que i es última local
    DO i = 1, n
       . . .
    END DO
    . . . = i
```

Última Local

Última local

- Puede producir resultados diferentes a la ejecución secuencial ya que se queda con los valores de las variables en la última iteración

```
C$Declara que x es última local
    DO i = 1, n
      if (. . .) then
        x = . . .
      END IF
    END DO
    PRINT *,x
```



- El procesador que realiza la última iteración no tiene porque coincidir con el procesador que contiene el último valor actualizado de x.

Reducción

Variable de reducción

- Una variable de reducción es aquella cuyo valor parcial se puede calcular localmente en cada thread , y cuyo valor final se obtiene operando con estos valores parciales

```
C$ Declara que x es de reducción
    DO i = 1, n
      x = x + a(i)
    END DO
```

Reducción

Precisión numérica en las operaciones de reducción

- Los números en punto flotante se almacenan en formato mantisa/exponente con un número finito de dígitos:
  - Errores de desbordamiento: El rango de representación está acotado por el exponente
  - Errores de redondeo: Los dígitos de la mantisa son finitos

IEEE 754: 53 bits mantisa y 11 exponente

- $2^{1023} = 8.9\ 10^{307}$; $2^{1024} = Inf$
- $1.0 + 2^{-53} = 1$

- Los errores de redondeo eliminan la propiedad conmutativa de las operaciones
  - Acumulación
  - Anulación catastrófica

```
(-100.0+100.0+1.0e-15)*1.0e+32 = 1.0e+17
(-100.0+ 1.0e-15+100.0)*1.0e+32 = 0.0
```

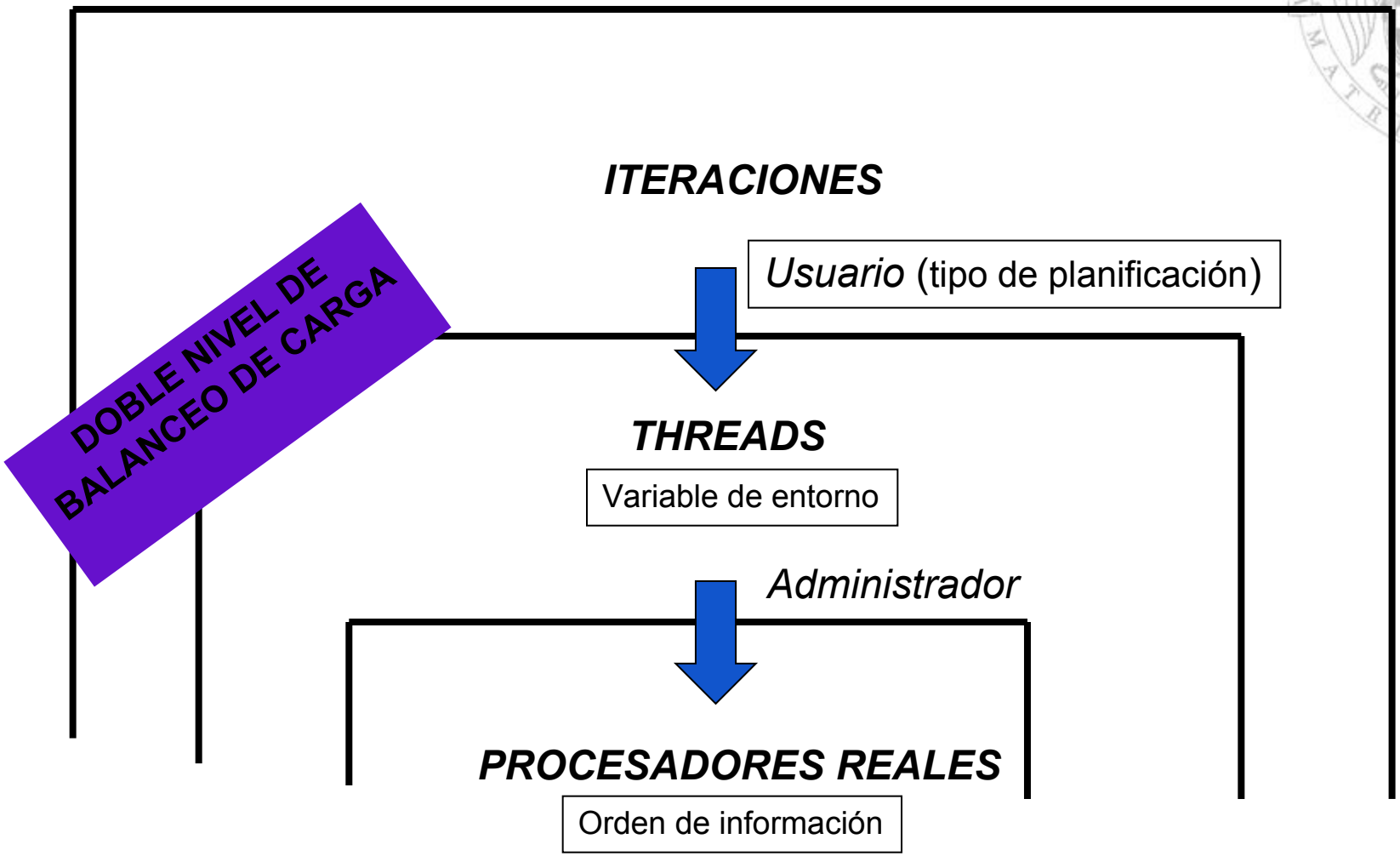
Solo Lectura

Sólo lectura

- Especifica variables y arrays que son solo de lectura dentro del bucle (válido sólo para variables compartidas)
- Este tipo de variables es un conjunto especial de variables compartidas que no se modifica durante la ejecución del bucle, facilita la labor del compilador
- Posiblemente el compilador replica el dato de modo que cada thread tenga su copia y así se optimiza el acceso a memoria

```
    x = 3
C$ Declara que x es compartida de solo lectura
    DO i = 1, n
      a(1) = x + 1
    END DO
    . . .
```

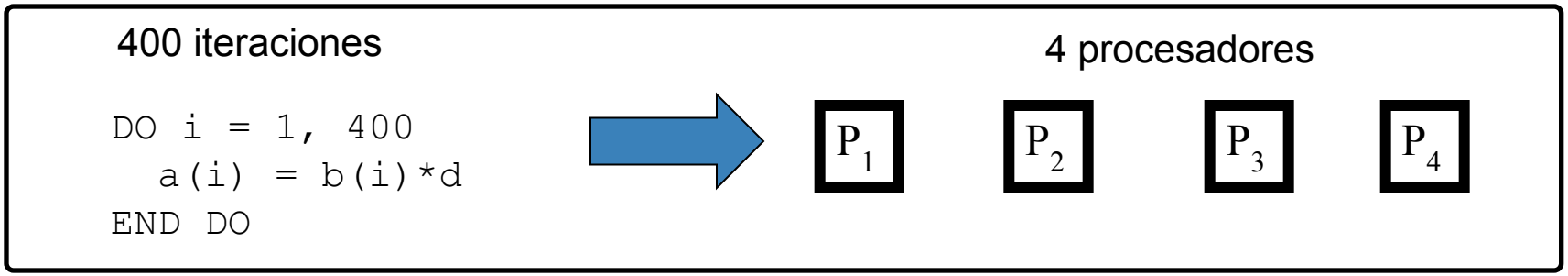
Planificación de Iteraciones



Competencia con otros usuarios ⇒ Precaución al interpretar el tiempo de ejecución

Tipos

- La estrategia de planificación especifica la distribución de las iteraciones de un bucle sobre los procesadores (threads)
- Muy importante para balancear de la carga sobre los threads

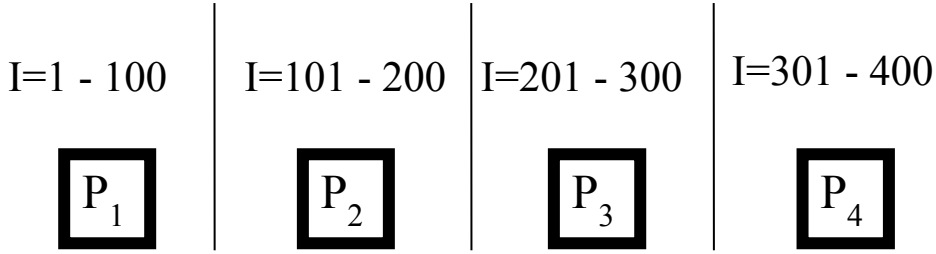


- Chunk: Tamaño mínimo a repartir
- **Tipos**:
  - Estáticas: Reparto de iteraciones al comienzo del bucle y fijo durante su ejecución
  - Dinámicas: Reparto de iteraciones durante la ejecución del bucle
    - :-) Mejor balanceo de carga (en tiempo de ejecución)
    - :-(Para que reciban la siguiente asignación de iteraciones deben pasar a una sección crítica

Planificación Estática

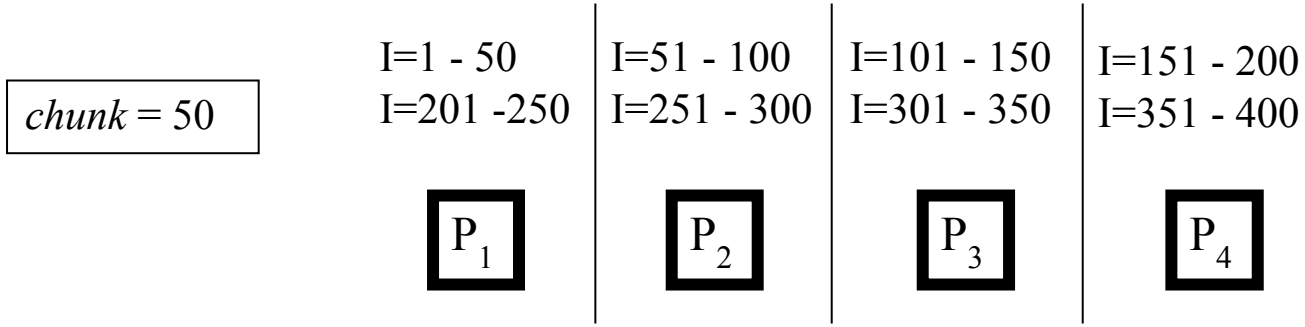
Bloques

- Distribución uniforme de las iteraciones sobre los threads



Cíclica estática

- Distribuye un subconjunto de iteraciones en cada thread de modo circular
- Existe un valor por defecto para el tamaño de subconjunto (*chunksize*)



Planificación Dinámica

CÍCLICA DINÁMICA

- Distribuye dinámicamente un subconjunto de iteraciones en cada *thread* de modo circular
- Mejor balanceo cuando la carga es diferente en cada iteración

chunk = 50

P_1	P_2	P_3	P_4
I=1 - 50 I=201 -250	I=51 - 100 I=251 - 300	I=101 - 150 I=301 - 350	I=151 - 200 I=351 - 400

FACTORIZADA

- Con *n* iteraciones y *k* threads distribuye dinámicamente *n*/2*k* iteraciones en cada *thread*
- Existe un valor por defecto para *m* (mínimo por subconjunto)

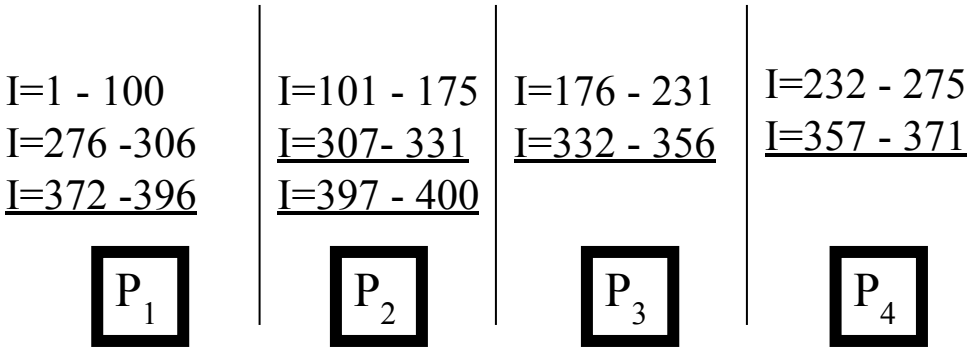
m = 25

	P_1	P_2	P_3	P_4
50	I=1 - 50	I=51 - 100	I=101 - 150	I=151 - 200
25	I=201 -225	I=226 - 250	I=251 - 275	I=276 - 300
25	I=301 -325	I=326 - 350	I=351 - 375	I=376 - 400

Planificación Dinámica

Auto-planificación guiada

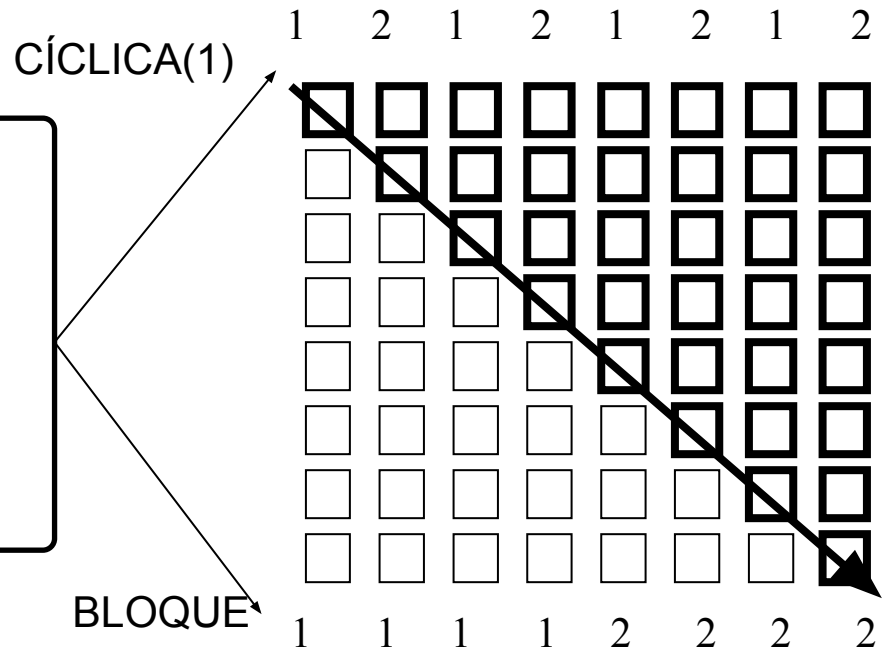
- Con n iteraciones y *k* threads distribuye *n/k* iteraciones al primer thread
- Asigna el resto de iteraciones dividido por *k* al siguiente thread, y así . .
- Buen balanceo y se reduce el número de accesos a sección crítica
- Se suele poder determinar, *m*, es el número de iteraciones mínimo para un thread



Impacto en el rendimiento

- Es muy importante encontrar la planificación óptima para los bucles más costosos
- La ganancia viene limitada por el thread más lento
- Búsqueda de equilibro de carga entre los procesadores

```
DO j = 1, jmax
  DO i = j+1, imax
    tmp = a(i,j)
    a(i,j) = a(j,i)
    a(j,i) = tmp
  END DO
END DO
```



Especificación del Número de Procesadores

- Por medio de la variable de entorno CPUS se controla el máximo número de procesadores disponible para el programa

```
demo% setenv CPUS 4
```

- Indica el número threads que se crearán (siempre menor que el número de procesadores del computador)
  - Si coincide con el número de procesadores disponibles, cada uno ejecuta un thread
  - De lo contrario competirán por los procesadores (no suele ser eficiente)
    - Si no hay mas usuarios probar con N-1
    - Incluso menor número (el tiempo de respuesta es menor si no hay competencia)

```
DO I=1, 1000
  A(I) = B(I)
END DO
```

```
demo% setenv CPUS 4
```

→ T ~ 250

4 procesadores

```
demo% setenv CPUS 5
```

→ T ~ 400

No hace falta recompilar el código para ejecutarlo en diferentes arquitecturas

- Desarrollo y depuración en una estación ⇒ Resultados en el multiprocesador

Pila del Programa

- El programa en ejecución mantiene una pila para el programa principal y distintas pilas para cada thread
  - La pila es un espacio de direcciones temporal usado para almacenar argumentos y las variables automáticas (petición de memoria dinámica en tiempo de ejecución) al invocar un subprograma
- Depende del tipo de compilador, por defecto puede crear estáticas o dinámicas
- En la mayoría de los códigos es necesario ampliar la pila (prueba y error ⇒ segmentation fault)
- **Para cambiar el tamaño (en Kbytes):**

```
demo% limit
cputime         unlimited
filesize        unlimited
datasize        2097148 kbytes
stacksize       8192 kbytes
coredumpsize    unlimited
vmemoryuse      unlimited
descriptors     64
demo% limit stacksize 65536
```

64 Mbytes

Pila del Programa

Bucles con llamadas a subrutina

- Cuando se realizan múltiples llamadas a la misma rutina desde diferentes threads se crean interferencias al acceder todos a las mismas variables locales estáticas
  - Es necesario hacer todas las variables automáticas
  - De este modo cada thread usará una copia local almacenada en su pila

Responsabilidad del Programador

- El compilador genera código multithreading a pesar de que haya dependencias
- Es responsabilidad del programador intentar que esto no ocurra
- El compilador no asegura el acceso mutuamente exclusivo a las variables compartidas
- El programador debe asegurar que la compartición no causa problemas

Algunos Ejemplos

Ejemplos:

- Opción -mp en los compiladores de Silicon Graphics (Power Challenge y Origin 2000)
- Opción -xexplicitpar en los compiladores de Sun (HPC 450, 4500, 5500, 6500, 10000)
- BBN Fortran del computador BBN
- Directivas del CRAFT para el SGI Cray T3E
- **El estándar OpenMP**

Las correspondientes en C suelen ser menos eficientes y potentes debido a la gestión de memoria dinámica

¿Qué Bucles Pueden/Deben ser Paralelizados?

- Un bucle es susceptible de ser paralelizado si:
  - Es un bucle DO (no se pueden paralelizar los DO WHILE)
  - Los valores de las variables array en cada iteración no dependen de valores variables array en cualquier otra iteración del bucle
  - Si el bucle modifica un escalar, éste no debe ser usado al terminar el bucle ya que no se garantiza su valor final
  - Para cada iteración, cualquier subprograma invocado dentro del bucle no referencia o modifica valores de variables array para otra iteración
  - El índice contador del bucle debe ser un entero

¿Qué Bucles Pueden/Deben ser Paralelizados?

- **Bucles con poca computación (pocas iteraciones):**
  - Sobrecarga en la iniciación y sincronización de los threads

```
DO I=1, 10
  A(I) = B(I)
END DO
```

- **Bucles con dependencias:**
  - La ejecución simultánea de sus iteraciones produce resultados diferentes a la ejecución secuencial

```
DO I=2, 1000
  A(I) = A(I-1)+B(I)
END DO
```

- En un anidamiento de bucles sólo se paraleliza un nivel (desde el exterior al interior)

Dependencias de Datos (Inhibidores de la Paralelización)

¿La ejecución simultánea de las iteraciones de un bucle produce el mismo resultado que la ejecución secuencial?

Las iteraciones son independientes

Recurrencia
Reducción
Direccionamiento indirecto

El bucle es reversible

Un Bucle No Tiene Dependencia de Datos si:

1. Ninguna iteración escribe en una posición que es leída o escrita por otra iteración
2. Las iteraciones pueden leer de una misma posición siempre que ninguna escriba sobre la posición
3. Una misma iteración puede leer y escribir en una posición repetidamente

Dependencias de Datos (Inhibidores de la Paralelización)

Si un código se ejecutaba eficientemente sobre un computador vectorial tiene mucha probabilidad de que se ejecute de un modo igualmente eficiente sobre un computador con varios procesadores paralelizado por medio de directivas

La idea es la misma:
Realizar simultáneamente la misma operación sobre diferentes componentes del mismo vector

Si el código presenta una dependencia muchas veces se puede reestructurar para eliminar la dependencia

Lo mejor es realizar una paralelización automática con PFA (Power Fortran Accelerator)

- Incluye directivas explícitas automáticamente
- Si no puede paralelizar un bucle informa de la razón

Dependencias de Datos (Inhibidores de la Paralelización)

- **Tipos de bucles:**
  - Imposibles de paralelizar (secuenciales)
  - Paralelizables (sin dependencias)
  - Paralelizables en función de una variable (indexaciones indirectas)
  - Paralelizables si reestructuramos el código
- **Si una dependencia no se puede eliminar reestructurando debemos:**
  - Sacar del bucle: fisión de bucles
  - Si hay anidamiento intentar paralelizar el otro bucle

Lo mejor es realizar primero una paralelización automática con PFA (*Power Fortran Accelerator*)
- Incluye directivas explícitas automáticamente
- Si no puede paralelizar un bucle informa de la razón

Dependencias de Datos (Inhibidores de la Paralelización)

Negrita = local

SIN DEPENDENCIA

```
      DO 10 I = 1,N
 10     A(I) = X +
B(I)*C(I)
```

paralelo

DEPENDENCIA DE DATOS

```
      DO 20 I = 2,N
 20     A(I) = B(I) -
A(I-1)
```

secuencial

STRIDE MAYOR QUE UNO

```
      DO 20 I = 2,N,2
 20     A(I) = B(I) -
A(I-1)
```

paralelo

VARIABLE LOCAL

```
DO I = 1, N
   X = A(I)*A(I) +
B(I)
   B(I) = X + B(I)*X
END DO
```

paralelo

LLAMADA A FUNCIÓN

```
 DO 10 I = 1, N
       X = SQRT(A(I))
       B(I) = X*C(I) +
X*D(I)
 10 CONTINUE
```

depende de la función

DEPENDENCIA SERIE

```
INDX = 0
DO I = 1, N
   INDX = INDX + I
   A(I) = B(I) +
C(INDX)
END DO
```

reestructurar

Dependencias de Datos (Inhibidores de la Paralelización)

Negrita = local

SALTO INCONDICIONAL

```
 DO I = 1, N
        IF (A(I) .LT. EPSILON) GOTO 320
        A(I) = A(I) * B(I)
     END DO
 320 CONTINUE
```

secuencial

DIRECCIONAMIENTO INDIRECTO

```
INDEX = SELECT(N)
DO I = 1, N
   A(I) = A(INDEX)
END DO
```

depende de INDEX

ANÁLISIS COMPLICADO

```
DO I = K+1, 2*K
   W(I) = W(I) + B(I,K) * W(I-K)
END DO
```

paralelo

ARRAY LOCAL

```
DO I = 1, N
   D(1) = A(I,1) - A(J,1)
   D(2) = A(I,2) - A(J,2)
   D(3) = A(I,3) - A(J,3)
   TOTAL_DISTANCE(I,J) = SQRT(D(1)**2 + D(2)**2 +
D(3)**2)
END DO
```

paralelo

Dependencia Serie

Una variable escalar se acumula de iteración en iteración

- Exige una ejecución en orden de las iteraciones

```
INDX = 0
DO I = 1, N
     INDX = INDX + I
     A(I) = B(I) + C(INDX)
END DO
```

SE DEBE REESTRUCTURAR

- Se puede calcular el escalar en cada iteración sin referirse a valores anteriores

```
C$DOACROSS LOCAL (I, INDX)
  DO I  = 1, N
     INDX = (I*(I+1))/2
     A(I) = B(I) + C(INDX)
  END DO
```

Recurrencia

Actualización en una iteración depende de valores calculados en otras iteraciones

- Exige una ejecución en orden de las iteraciones

1. Recurrencia hacia adelante

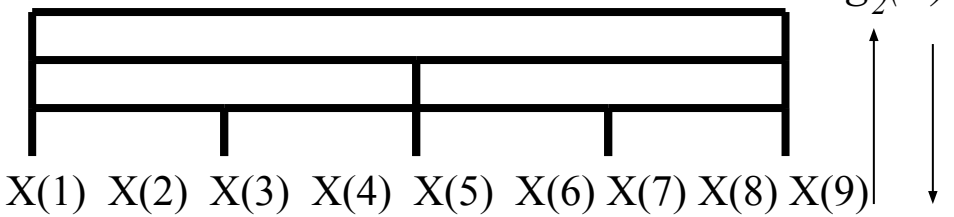
```
DO I=2, 1000
  X(I) = X(I-1)*B(I)+A(I)
END DO
```

SE DEBE REESTRUCTURAR

- La técnica de reducción cíclica suaviza el efecto de este tipo de recurrencia

```
X(1) = A(1)
X(2) = A(2)+B(2)X(1)
X(3) = A(3)+B(3)X(2) = [A(3)+B(3)A(2)]+[B(3)B(2)]X(1)
. . .
```

A´(3) B´(3)



Recurrencia

2. Recurrencia hacia atrás

```
DO I=1, 999
  X(I) = X(I+1)*B(I)+A(I)
END DO
```

SE DEBE REESTRUCTURAR

- Se resuelve de modo sencillo creando un nuevo array

```
DO I=1, 1000
  X1(I) = X(I)
END DO
DO I=1, 999
  X(I) = X1(I+1)*B(I)+A(I)
END DO
```

Reducción

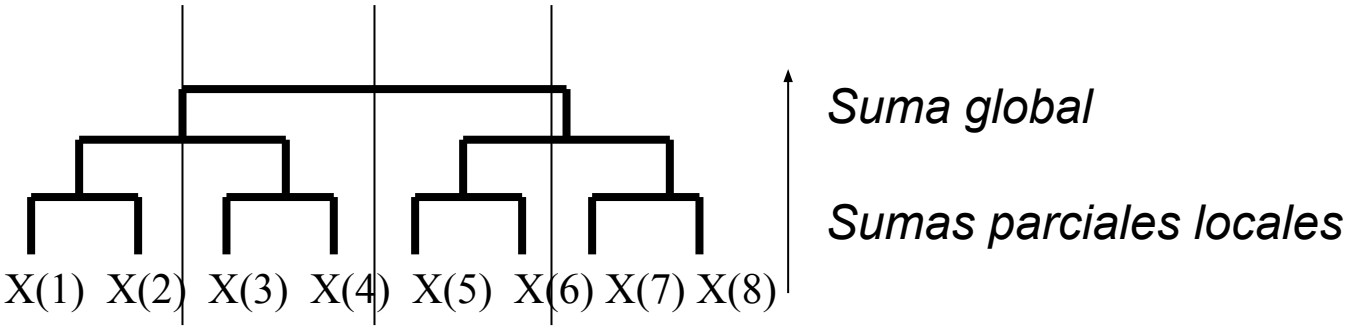
- **Reducen arrays de dimensión d a d-1** (i.e. reducción de un vector a un escalar)
  - Exige una ejecución en orden de las iteraciones

```
SUM = 0.0
DO I=1, 1000
  SUM = SUM + A(I)*B(I)
END DO
```

RESUELTO POR EL COMPILADOR

SE DEBE REESTRUCTURAR

- **Problemas:**
  - Grado de paralelismo
  - Actualización de variable compartida
  - Las reducciones más comunes las resuelve el compilador con reduction



Reducción

```
SUM  = 0.0
DO I = 1,N
   SUM = SUM + A(I)
END DO
```

```
 NUM_THREADS = MP_NUMTHREADS()
C
C  IPIECE_SIZE = N/NUM_THREADS ROUNDED UP
C
   IPIECE_SIZE = (N + (NUM_THREADS -1)) / NUM_THREADS
   DO K = 1, NUM_THREADS
      PARTIAL_SUM(K) = 0.0
C
C  THE FIRST THREAD DOES 1 THROUGH IPIECE_SIZE, THE
C  SECOND DOES IPIECE_SIZE + 1 THROUGH 2*IPIECE_SIZE,
C  ETC. IF N IS NOT EVENLY DIVISIBLE BY NUM_THREADS,
C  THE LAST PIECE NEEDS TO TAKE THIS INTO ACCOUNT,
C  HENCE THE "MIN" EXPRESSION.
C
   DO I =K*IPIECE_SIZE -IPIECE_SIZE +1, MIN(K*IPIECE_SIZE,N)
      PARTIAL_SUM(K) = PARTIAL_SUM(K) + A(I)
   END DO
   END DO
C
C  NOW ADD UP THE PARTIAL SUMS
   SUM = 0.0
   DO I = 1, NUM_THREADS
      SUM = SUM + PARTIAL_SUM(I)
   END DO
```

```
   SUM = 0.0
C$DOACROSS LOCAL (I), REDUCTION (SUM)
   DO 10 I = 1, N
      SUM = SUM + A(I)
10 CONTINUE
```

Direccionamiento Indirecto

El uso de vectores de índices puede producir dependencias en tiempo de ejecución

- Puede exigir una ejecución en orden de las iteraciones

```
DO L=1, 1000
  A(ID(L)) = A(L)*B(L)
END DO
```

RESPONSABILIDAD DEL PROGRAMADOR

- Si hay valores repetidos en ID(L) el resultado depende del orden

```
DO L=1, 1000
  A(ID1(L)) = A(ID2(L))*B(L)
END DO
```

- Recurrencia lineal si ID2(L) es menor que ID1(L)

Ejemplos Típicos

```
      DO I = 1, N
         TOTAL = 0.0
         DO J = 1, M
            TOTAL = TOTAL +
A(J)
         END DO
         B(I) = C(I) * TOTAL
      END DO
```

No hay dependencia en el bucle J

```
DO J=1, 1000
  A(I) = A(I) + B(I-1)
END DO
```

No hay dependencia

```
DO J=1, 1000
  DO I=1, 1000
    A(I,J) = A(I-1,J)+B(I)
  END DO
END DO
```

No hay dependencia en el bucle J

Búsqueda de Eficiencia

Por lo general es siempre mejor paralelizar el bucle más externo. Excepto si su número de iteraciones es muy pequeño

```
    . . .
C$DOACROSS
    DO j = 1, 4
      DO i = 1, 1000
       b(i,j) = a(i,j)
      END DO
    END DO
    . . .
```

- Solo 5 threads
  - Problemas con más de 4 procesadores
  - Peor balanceo de la carga
- Menos gestión de threads

```
    . . .
    DO j = 1, 4
C$DOACROSS
      DO i = 1, 1000
           b(i,j) = a(i,j)
      END DO
    END DO
    . . .
```

- Permite más threads
- Más gestión de threads

Compromiso entre Memoria cache y Paralelismo

Resumen de norma básica para mejorar la localidad espacial de un programa

- Para optimizar la cache el bucle debe recorrer los arrays según están almacenados en memoria
- En Fortran significa que el índice más a la izquierda (fila) debe variar más rápido

No es paralelizable en K

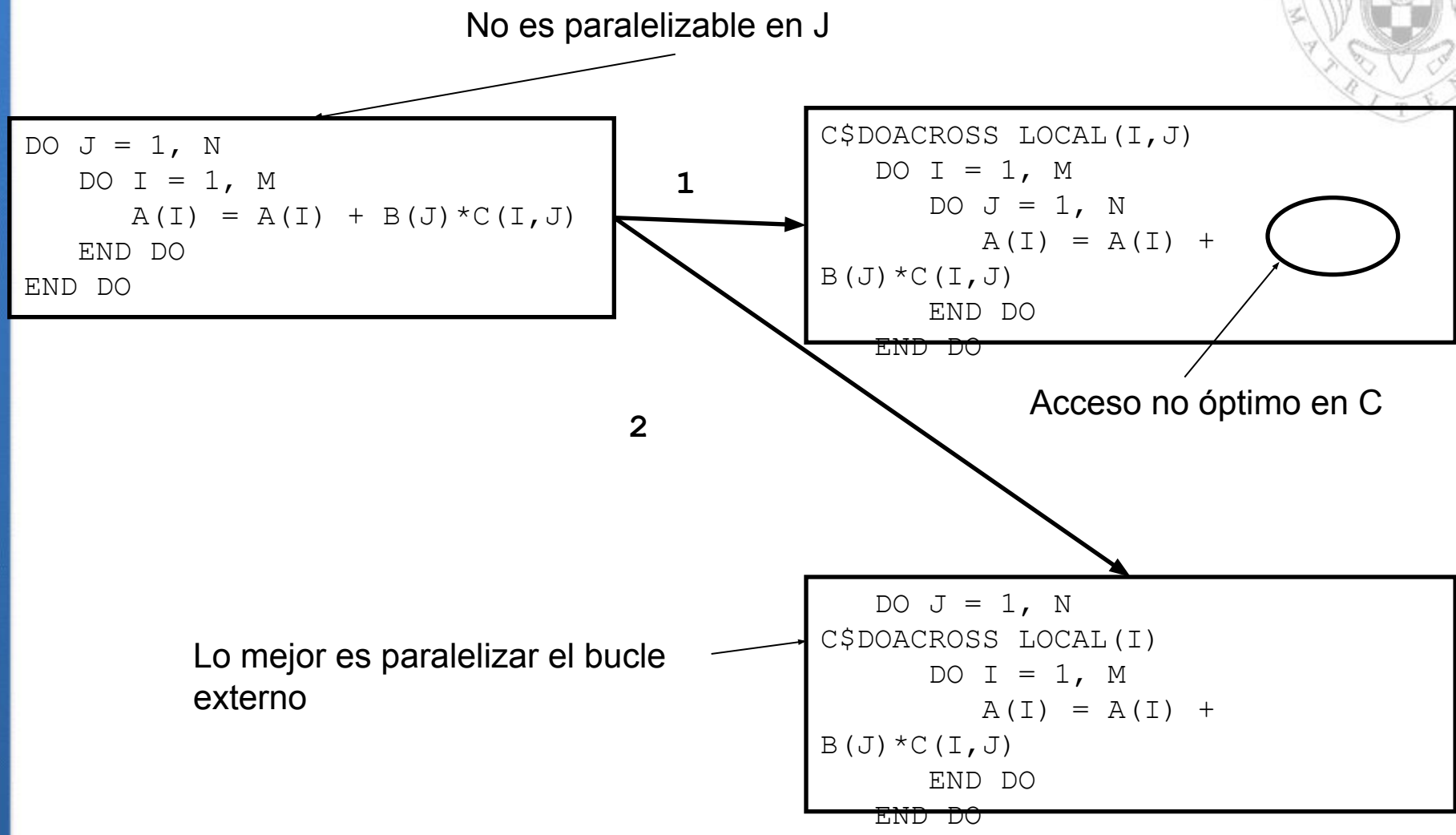
```
DO I = 1, N
   DO K = 1, N
      DO J = 1, N
         A(I,J) = A(I,J) + B(I,K) * C(K,J)
      END DO
   END DO
END DO
```

Multiplicación de matrices

Lo mejor es paralelizar el bucle externo

```
C$DOACROSS LOCAL(I, J, K)
      DO J = 1, N
         DO K = 1, N
            DO I = 1, N
               A(I,J) = A(I,J) + B(I,K) * C(K,J)
            END DO
         END DO
      END DO
```

Compromiso entre Memoria cache y Paralelismo



Compromiso entre Memoria cache y Paralelismo

```
DO J = 1, N
   DO I = 1, M
      A(I) = A(I) + B(J)*C(I,J)
   END DO
END DO
```

La mejor solución depende de:

- M
- N
- parámetros hardware

Solución compromiso

3

```
 NUM = MP_NUMTHREADS()
   IPIECE = (N + (NUM-1)) / NUM
C$DOACROSS LOCAL(K,J,I)
   DO K = 1, NUM
      DO J = K*IPIECE - IPIECE + 1, MIN(N, K*IPIECE)
         DO I = 1, M
            PARTIAL_A(I,K) = PARTIAL_A(I,K) +
B(J)*C(I,J)
         END DO
      END DO
   END DO
C$DOACROSS LOCAL (I,K)
   DO I = 1, M
      DO K = 1, NUM
         A(I) = A(I) + PARTIAL_A(I,K)
      END DO
   END DO
```

El Problema de la Compartición Falsa

El problema no reduce el tiempo de ejecución con el número de procesadores

```
DO I1=1,3,5,7,9
  A(I1) = B(I1) + C(I1)
END DO
```

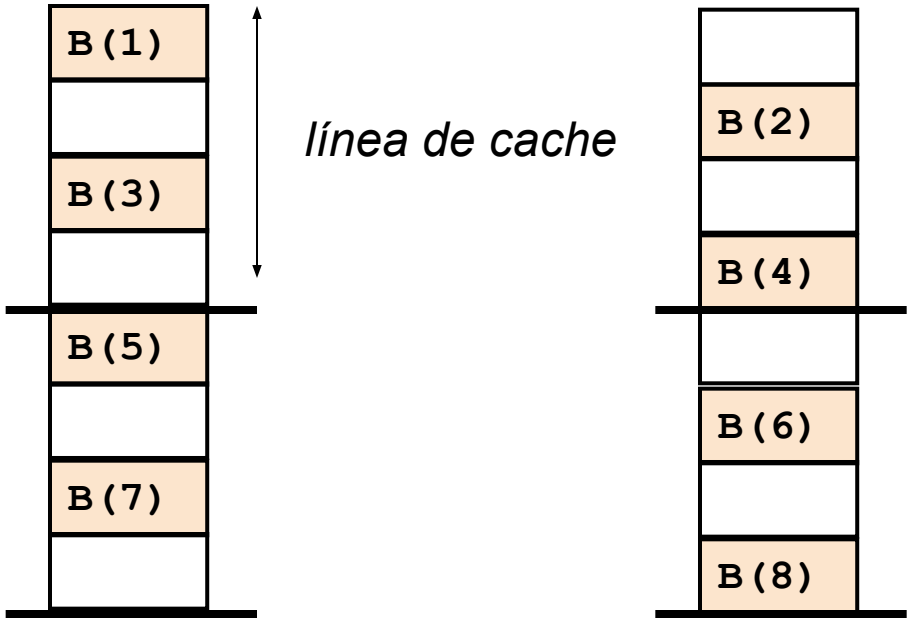
```
DO I2=2,4,6,8,10
  A(I2) = B(I2) + C(I2)
END DO
```

Problemas

- Con los arrays B y C usamos solo la mitad de los datos que llevamos a cache
- Existe compartición falsa sobre el array A

El Problema de la Compartición Falsa

Uso Ineficiente de la Memoria Cache



Ambos procesadores leen la misma línea de cache y además no la utilizan al completo

El Problema de la Compartición Falsa

Compartición falsa

Procesador 1

A(1) = B(1) + C(1)

- Trae línea que contiene A(1)
- Escribe A(1) y marca la línea como “modificada”

TIEMPO

El Problema de la Compartición Falsa

Compartición falsa

Procesador 1

A(1) = B(1) + C(1)

- Trae línea que contiene A(1)
- Escribe A(1) y marca la línea como “modificada”

Procesador 2

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

- Detecta que la línea que contiene A(2)está modificada
- Coge de 1 copia actualizada
- Escribe A(2) y marca la línea como “modificada”

TIEMPO

El Problema de la Compartición Falsa

Compartición falsa

Procesador 1

A(1) = B(1) + C(1)

- Trae línea que contiene A(1)
- Escribe A(1) y marca la línea como “modificada”

A(3) = B(3) + C(3)

- Detecta que la línea que contiene A(3)está modificada
- Coge de 2 copia actualizada
- Escribe A(3) y marca la línea como “modificada”

Procesador 2

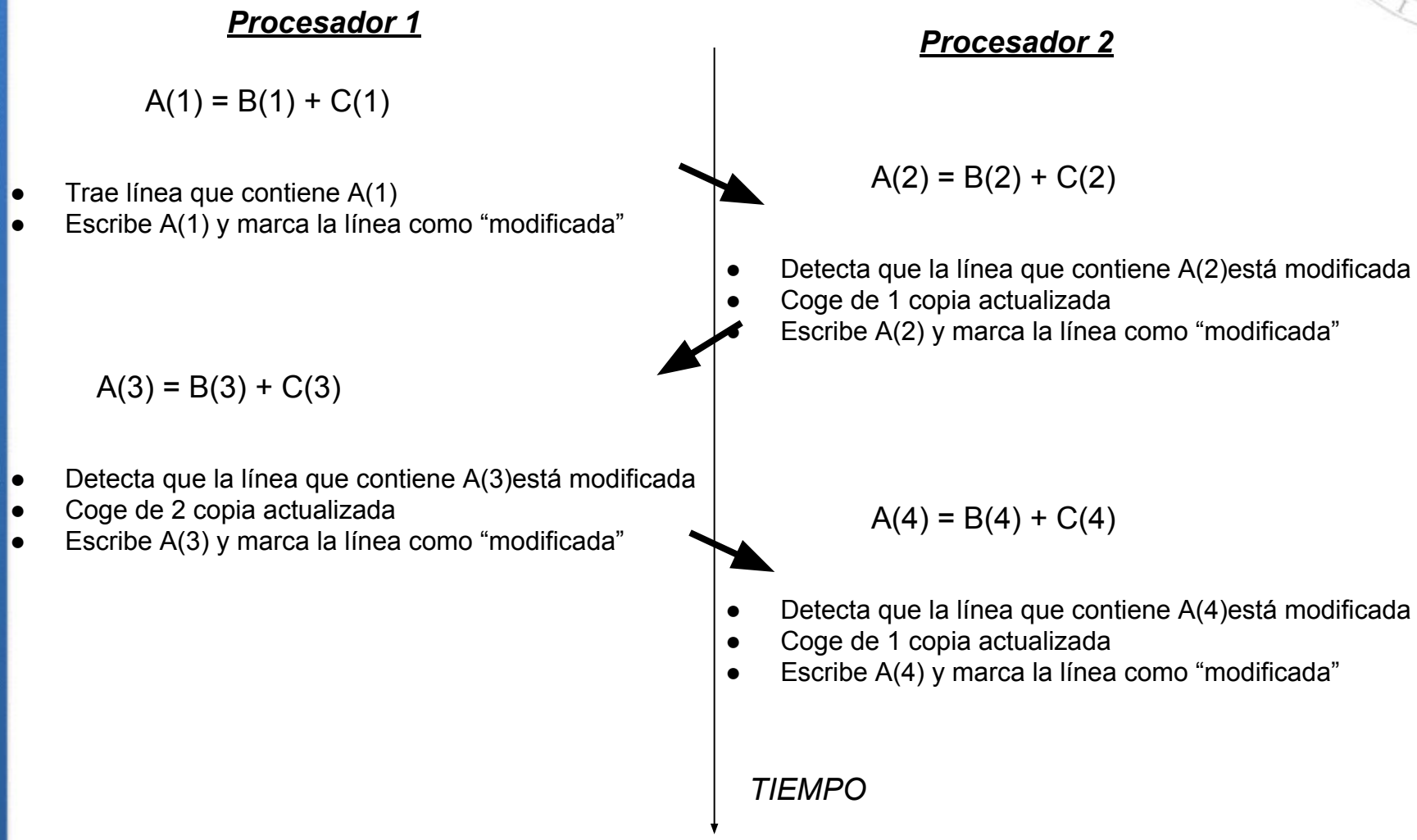
A(2) = B(2) + C(2)

- Detecta que la línea que contiene A(2)está modificada
- Coge de 1 copia actualizada
- Escribe A(2) y marca la línea como “modificada”

TIEMPO

El Problema de la Compartición Falsa

Compartición falsa



El Problema de la Compartición Falsa

Conclusiones

- Siempre que sea posible se debe usar la línea completa
  - Precaución con strides de acceso mayores que la unidad
- Evitar la compartición falsa
  - Puede ocurrir con matrices compartidas que se modifican en el bucle
  - Probar con un tamaño de subconjunto mayor que uno (4 o 8)
    - Compromiso entre localidad espacial para cada subconjunto y buen balanceo de carga

Pasos en la Depuración

Pueden producir resultados diferentes si usamos fuera del bucle una variable privada

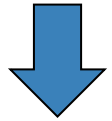
Queremos depurar un código con ambos tipos de paralelización

- **1. Desactiva la paralelización**
  - Quita las opciones de paralelización, o
  - Ejecuta el código con la variable de entorno #procesadores igual a 1
    - Si el problema desaparece es que es debido a la paralelización (pasa a 2)
    - Si no depura el código serie
- **2. Compila con paralelización automática**
  - Activa las opciones de paralelización automática
    - Si el problema continúa puede ser la reducción (pasa a 3)
    - Si no, el problema es de la paralelización explícita (siguiente trasparencia)
- **3. Compila sin activar la reducción automática**
  - Activa solo la opción de paralelización automática
    - Si el problema continua, el compilador paraleliza un bucle que no debería (siguiente transparencia)
    - Si no el error es debido a los errores de redondeo introducidos por la reducción

Pasos en la Depuración

Aislemos la función problemática

1. Usar fsplit para formar un fichero por función
2. Crear un Makefile para compilar cada función de modo independiente
3. Se van haciendo pruebas compilando y ejecutando, dejando siempre una sin la opción de paralelización
    - Enlazar con la opción de paralelización todas todas



Detectemos el bucle problemático

1. Compilar con la opción de paralelización la función problemática para conocer que bucles han sido paralelizados
2. Forzar la no paralelización uno a uno de estos bucles hasta obtener los resultados correctos

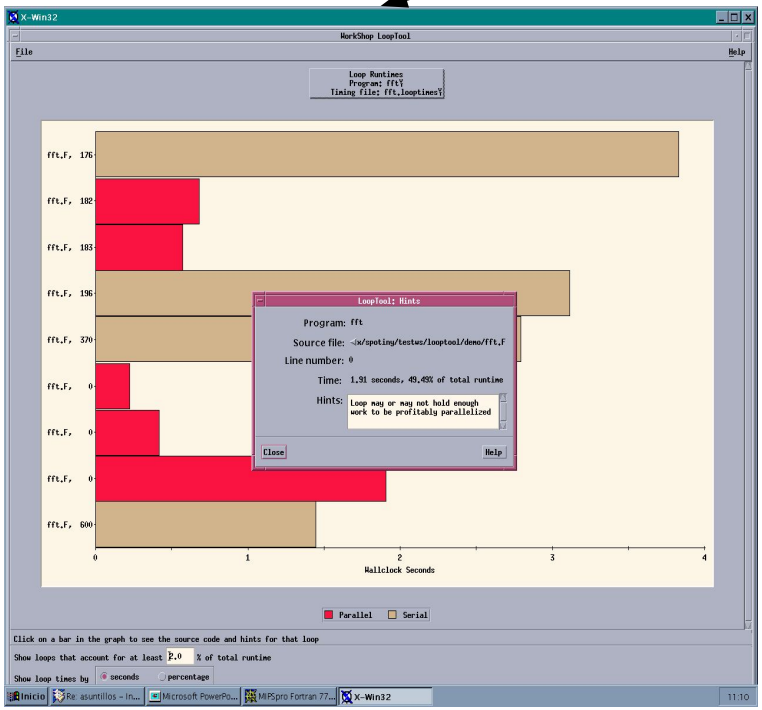
Estudiar la causa del error

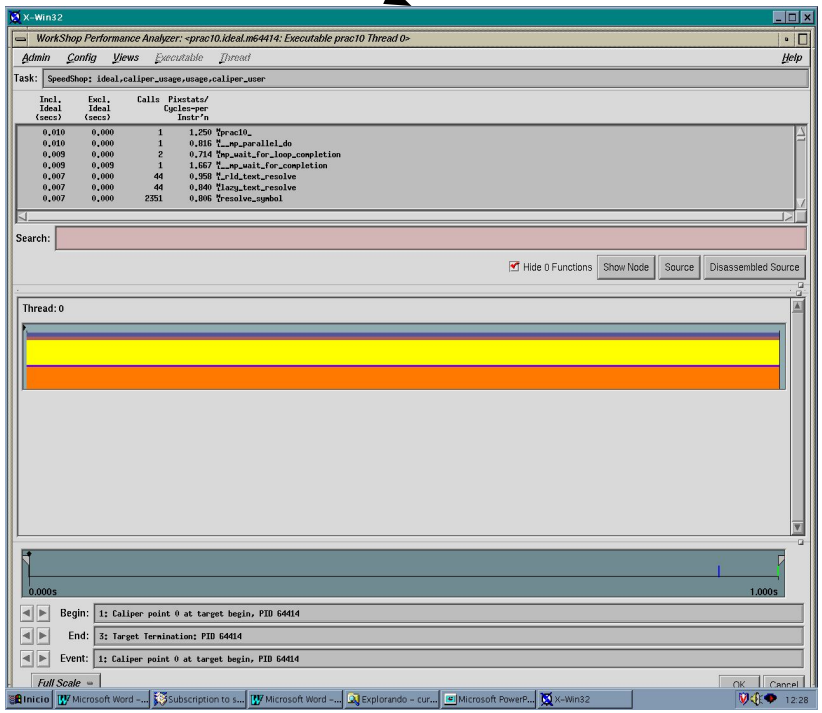
Pasos en la Paralelización de un Programa

1. Partir del programa serie optimizado ⇒ resultados para validar
2. Emplear la paralelización automática
3. Verificar que los resultados son = que el código original (si no depurar)
4. Paralelización explícita
   a. Usar paralelización explícita en los bucles más costosos (con profiling)
      i. Estudiando que ha hecho el paralelizador automático sobre estos bucles
   b. Verificar que los resultados son = que en la versión serie (si no depurar)
      i. Desde 1 procesador (errores debidos a locales usadas después del DO) y subiendo => E.M. en S.C.
   c. Comprobar el speedup para un número creciente de procesadores con tamaño de problema constante y creciente
      i. Cuidado con el planificador empleado (SCHEDTYPE)
   d. Repetir hasta que los resultados sean los requeridos (tiempo y speedup)
5. Modificación del código
   a. Reestructurar los bucles para mejorar su grado de paralelismo y disminuir dependencias de datos
   b. Cambiar el algoritmo numérico
   c. Usar un lenguaje de programación paralelo para probar la paralelización con grano grueso

Herramientas de Optimización de Códigos Paralelos

- Los sistemas suelen aportar herramientas para analizar el rendimiento de las aplicaciones paralelizadas con threads
- Las mismas que en optimización secuencial pero dando información para cada thread
- Interesante para conocer el balanceo de la carga (looptool o perfex -mp en SGI incluida en cvd)

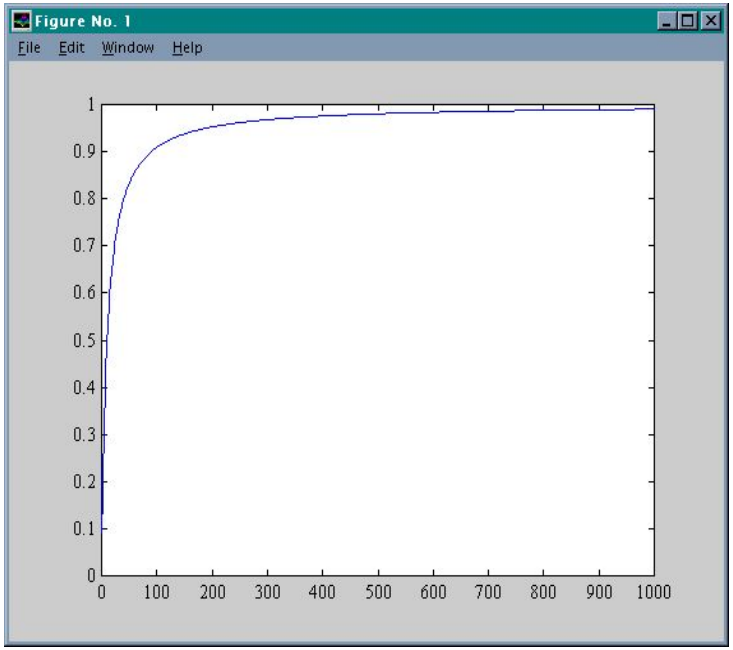




Algunos Consejos Prácticos

- Siempre centrarse en los bucles más costosos
- Las primeros pasos son los más eficientes

ganancia



esfuerzo