CPA – Computación Paralela

Grado en Ingeniería Informática

T2. Memoria Compartida. Diseño Básico de Algoritmos Paralelos

J. M. Alonso, P. Alonso, F. Alvarruiz, I. Blanquer,
D. Guerrero, J. Ibáñez, E. Ramos, J. E. Román

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curso 2021/22

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Contenido

Apartado 1

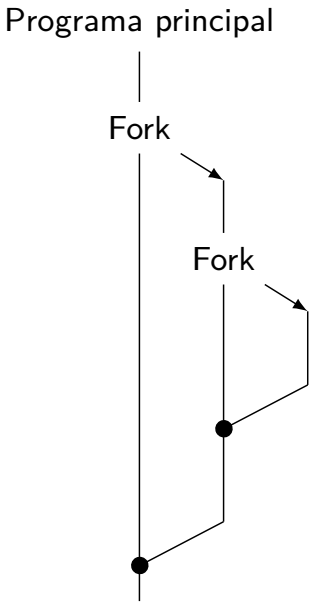
Modelo de Memoria Compartida

- Modelo
- Detalles

Procesos Concurrentes

Para especificar procesos concurrentes es habitual utilizar construcciones de tipo *fork-join*

- *Fork* crea una nueva tarea concurrente que empieza a ejecutar en el mismo punto en que estaba la tarea padre
- *Join* espera a que termine la tarea
- Ejemplo: llamada al sistema `fork()` en Unix



Este esquema se puede implementar a nivel de:

- Procesos del sistema operativo (*procesos pesados*)
- Hilos (*procesos ligeros*)

Modelo de Memoria Compartida

Características:

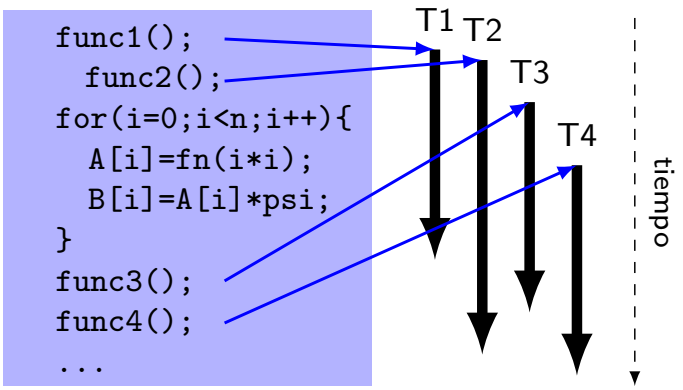
- Espacio de direcciones de memoria único para todos
- Programación bastante similar al caso secuencial
  - Cualquier dato es accesible por cualquiera
  - No hay que intercambiar datos explícitamente
- Inconvenientes
  - El acceso concurrente a memoria puede dar problemas
    - Se ha de coordinar: semáforos, monitores, ...
    - Resultado impredecible si no se protegen bien los accesos a memoria
  - Difícil controlar la localidad de datos (memorias cache)

Modelo de Hilos

Este modelo está muy ligado al de memoria compartida

Hilo (*thread*): flujo de instrucciones independiente que puede ser planificado para ejecución por el sistema operativo

- Un proceso puede tener múltiples hilos de ejecución
- Cada hilo tiene datos "privados"
- Comparten recursos/memoria del proceso
- Se requiere sincronización



Hilos Java

Modelo orientado a objetos

```
public class HelloThread extends Thread {
  public void run() {
    System.out.println("Hello from a thread!");
  }
  public static void main(String args[]) {
    (new HelloThread()).start();
  }
}
```

Métodos sincronizados

```
public class SynchronizedCounter {
  private int c = 0;
  public synchronized void increment() {
    c++;
  }
  public synchronized int value() {
    return c;
  }
}
```


Hilos POSIX (pthreads)

Estandarización de hilos en sistemas Unix (estándar IEEE POSIX 1003.1c, 1995)

- Basado en librería (API de llamadas al S.O.)
- Sólo lenguaje C
- Paralelismo explícito: bastante esfuerzo de programación

Algunas operaciones

- Creación: `pthread_create`, `pthread_join`
- Semáforos: `sem_wait`, `sem_post`
- Exclusión mutua: `mutex_lock`, `mutex_unlock`
- Variables condición: `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`

Inconvenientes:

- Portabilidad (Windows tiene sus propios hilos)
- Más orientado a paralelismo de tareas (no de datos)

OpenMP

Estandarización de hilos portable

- Basado en directivas de compilador
- Disponible en C/C++ y Fortran
- Portable/multi-plataforma (Unix, Windows)
- Fácil de usar: paralelización incremental

Algunas directivas y funciones

- `#pragma omp parallel for`
- `omp_get_thread_num()`

La creación y finalización de hilos está implícita en algunas directivas

- El programador no se ha de preocupar de hacer *fork/join*

Procesos Unix

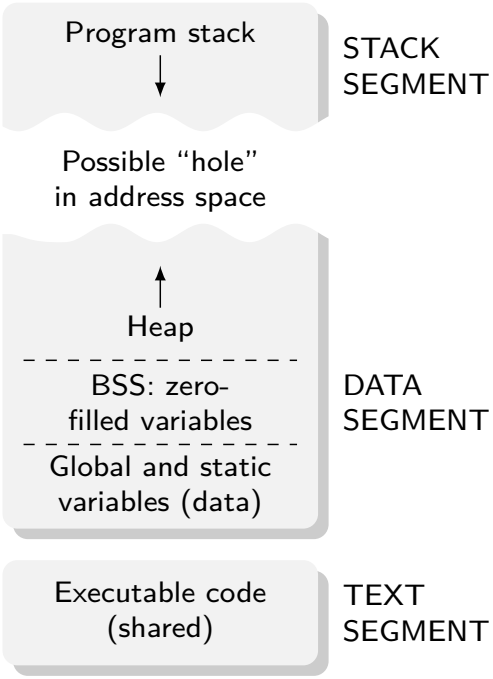
Cada proceso contiene información sobre recursos y estado de ejecución

- Código del programa (solo lectura, puede ser compartido)
- Variables (globales, *heap* y *stack*)
- Contexto de ejecución: registros, puntero de pila, etc.
- Recursos del sistema (solo accesible a través del S.O.)
  - Identificadores (proceso, usuario, grupo)
  - Entorno, directorio de trabajo, señales
  - Descriptores de ficheros

En procesos multi-hilo

- Cada hilo tiene su propio contexto de ejecución
- Cada hilo tiene una pila de llamadas independiente
- Se comparten los recursos del sistema

Modelo de Memoria en Procesos Unix



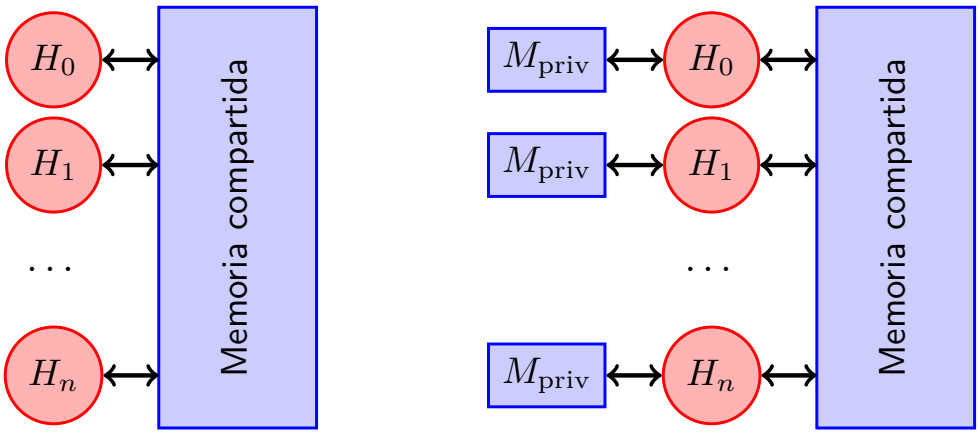
Información en el núcleo del sistema operativo (PCB: *process control block*)

- Program counter
- Stack pointer
- Registers
- Process state
- Process ID
- User ID
- Group ID
- Memory limits
- Open files, sockets
- ...

Modelo de Memoria con Hilos

Modelo simple: espacio único de direcciones

Modelo más realista: espacio único de direcciones, con variables privadas por cada hilo



Una pila de llamadas por cada hilo

- Algunas variables se crean en la pila (locales)
- Un hilo no puede saber si la pila de otro hilo está activa

Coordinación de Accesos a Memoria

El intercambio de información entre hilos se hace mediante lectura/escritura de variables en memoria compartida

Acceso simultáneo puede producir una condición de carrera

- El resultado final puede ser incorrecto
- Es de naturaleza no determinista

Ejemplo: dos hilos quieren incrementar la variable `i`

Secuencia con resultado correcto:

H0 carga `i` en un registro: 0
H0 incrementa registro: 1
H0 almacena el valor en `i`: 1
H1 carga `i` en un registro: 1
H1 incrementa registro: 2
H1 almacena el valor en `i`: 2

Secuencia con resultado incorrecto:

H0 carga `i` en un registro: 0
H1 carga `i` en un registro: 0
H0 incrementa registro: 1
H1 incrementa registro: 1
H0 almacena el valor en `i`: 1
H1 almacena el valor en `i`: 1

Exclusión Mutua y Sincronización

¿Cómo solucionar la condición de carrera?

Operaciones atómicas

- Forzar a que las operaciones problemáticas se realicen de forma atómica (sin interrupción)
- Instrucciones especiales del procesador: *test-and-set* o *compare-and-exchange* (CMPXCHG en Intel)

Sección crítica

- Fragmentos de código con más de una instrucción
- No permitir que haya más de un hilo ejecutándola
- Requiere mecanismos de sincronización: semáforos, etc.
- Puede aparecer riesgo de interbloqueo

Otro tipo de sincronización

- Barrera: esperan en un punto a que lleguen todos
- Ejecución ordenada

Apartado 2

Fundamentos del Diseño de Algoritmos Paralelos

- Análisis de Dependencias
- Grafo de Dependencias

Paralelización de Algoritmos

Paralelizar un algoritmo implica encontrar tareas (partes del algoritmo) concurrentes (se pueden ejecutar en paralelo)

Casi siempre, hay dependencias entre tareas

- Si una tarea solo puede empezar cuando otra ha finalizado

```
a = 0
PARA i=0 HASTA n-1
  a = a + x[i]
FPARA
b = 0
PARA i=0 HASTA n-1
  b = b + y[i]
FPARA
PARA i=0 HASTA n-1
  z[i] = x[i]/b + y[i]/a
FPARA
PARA i=0 HASTA n-1
  y[i] = (a+b)*y[i]
FPARA
```

Ejemplo:

- Los dos primeros bucles son independientes entre sí
- El tercer bucle utiliza los valores de a y b, que se calculan en los dos bucles anteriores

Dependencias de Datos

Se puede determinar si existen dependencias entre dos tareas a partir de los datos de entrada/salida de cada tarea

Condiciones de Bernstein:
Dos tareas T_i y T_j (T_i precede a T_j en secuencial) son independientes si

1. $I_j \cap O_i = \varnothing$
2. $I_i \cap O_j = \varnothing$
3. $O_i \cap O_j = \varnothing$

I_i y O_i representan el conjunto de variables leídas y escritas por T_i

Tipos de dependencias:

- Dependencia de flujo (se viola la condición 1)
- Anti-dependencia (se viola la condición 2)
- Dependencia de salida (se viola la condición 3)

Dependencias de Datos: Ejemplos

Dependencia de flujo

```
double a=3,b=5,c,d;
c = T1(a,b);
d = T2(a,b,c);
```

T_2 no puede empezar hasta que finalice T_1, ya que lee la variable c, que es escrita por T_1

Anti-dependencia

```
// T1,T2 modifican 3er argumento
double a[10],b[10],c[10],y;
T1(a,b,&y);
T2(b,c,a);
```

T_2 no puede empezar hasta que acabe T_1, de lo contrario T_2 machacaría el contenido de a que es entrada de T_1

Dependencia de salida

```
// T1,T2 modifican 3er argumento
double a[10],b[10],c[10],x[5];
T1(a,b,x);
T2(c,b,x);
```

Ambas tareas modifican el array x

Dependencias de Datos en Bucles

Algunas se pueden eliminar modificando el algoritmo

Código con dependencia de flujo

```
for (i=1; i<n; i++) {
  b[i] = b[i] + a[i-1];
  a[i] = a[i] + c[i];
}
```

La iteración `i` modifica `a[i]` que es leída por la iteración `i+1`

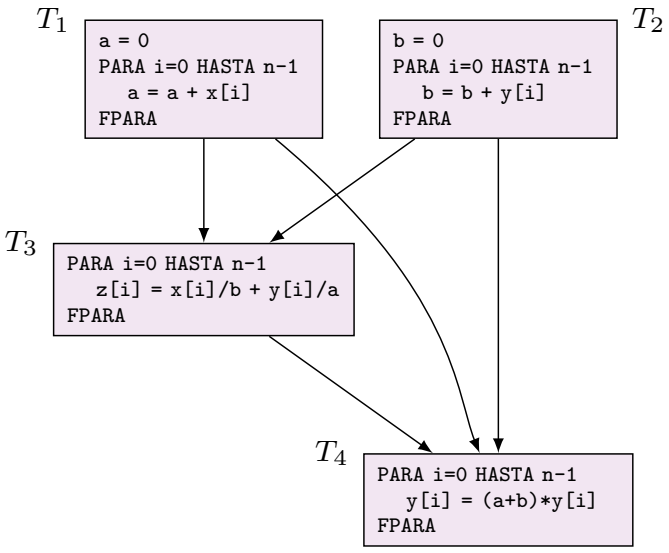
Eliminación de la dependencia mediante *sesgado del bucle*:

Código sin dependencias

```
b[1] = b[1] + a[0];
for (i=1; i<n-1; i++) {
  a[i] = a[i] + c[i];
  b[i+1] = b[i+1] + a[i];
}
a[n-1] = a[n-1] + c[n-1];
```


Paralelización de Algoritmos: Ejemplo

```
a = 0
PARA i=0 HASTA n-1
  a = a + x[i]
FPARA
b = 0
PARA i=0 HASTA n-1
  b = b + y[i]
FPARA
PARA i=0 HASTA n-1
  z[i] = x[i]/b + y[i]/a
FPARA
PARA i=0 HASTA n-1
  y[i] = (a+b)*y[i]
FPARA
```



Dependencias de flujo: $T_1 \to T_3$, $T_2 \to T_3$, $T_1 \to T_4$, $T_2 \to T_4$
Anti-dependencias: $T_2 \to T_4$, $T_3 \to T_4$

Diseño de Algoritmos Paralelos: Idea General

Básicamente dos fases:

1. Descomposición en tareas
   - Requiere un análisis detallado del problema
     $\longrightarrow$ Grafo de Dependencia de Tareas

2. Asignación de tareas
   - Qué hilo/proceso ejecuta cada tarea
   - A veces implica agrupar varias tareas

Habitualmente hay varias estrategias posibles de paralelización

- Usar una descomposición u otra puede tener gran impacto en las prestaciones
- Hay que intentar maximizar el grado de concurrencia

Grafo de Dependencias de Tareas

Abstracción utilizada para expresar las dependencias entre las tareas y su relativo orden de ejecución

- Se trata de un grafo acíclico dirigido (GAD)
- Los nodos representan tareas (pueden tener asociado un coste)
- Las aristas representan las dependencias entre tareas

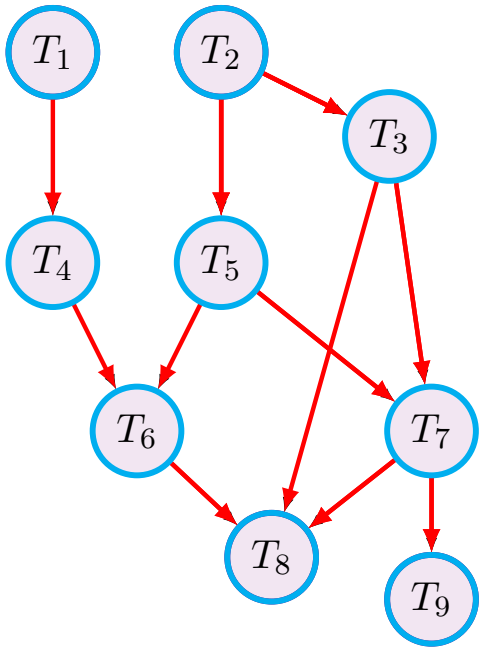
Definiciones:

- Longitud de un camino: suma de los costes c_i de los nodos que lo componen
- Camino crítico: el más largo entre un nodo inicial y uno final
- Máximo grado de concurrencia: mayor número de tareas que pueden ejecutarse al mismo tiempo
- Grado medio de concurrencia: $M = \sum_{i=1}^{N} \frac{c_i}{L}$

  (N = nodos totales, L = longitud del camino crítico)

Grafos de Dependencias de Tareas: Ejemplo

Grafo con $N = 9$ tareas (suponemos que todas tienen coste $c_i = 1$)

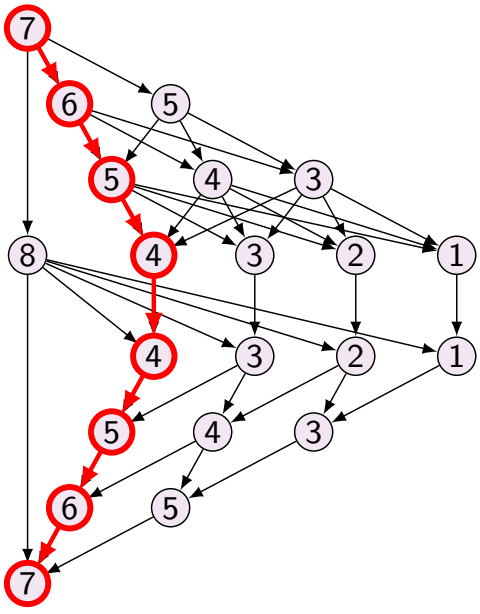


Nodos iniciales: T_1, T_2
Nodos finales: T_8, T_9
Caminos:
- $T_1 - T_4 - T_6 - T_8$ (longitud 4)
- $T_2 - T_5 - T_6 - T_8$ (longitud 4)
- $T_2 - T_5 - T_7 - T_8$ (longitud 4)
- $T_2 - T_3 - T_8$ (longitud 3)
- $T_2 - T_3 - T_7 - T_8$ (longitud 4)
- $T_2 - T_5 - T_7 - T_9$ (longitud 4)
- $T_2 - T_3 - T_7 - T_9$ (longitud 4)

Camino crítico: $L = 4$
Concurrencia:
- Grado máximo: 3
- Grado medio: $M = \sum_{i=1}^{9} \frac{1}{4} = 2{,}25$

Grafos de Dependencias de Tareas: Ejemplo

Grafo con $N = 21$ tareas (se indica el coste c_i en cada tarea)



Camino crítico
$L = 7 + 6 + 5 + 4 + 4 + 5 + 6 + 7 = 44$

$$M = \sum_{i=1}^{N} \frac{c_i}{L} = \frac{7 + 6 + 5 + 5 + \cdots}{44} = 2$$

Ejemplo de Descomposición en Tareas

Dados m polinomios

$$P_i(x) = a_{i,0} + a_{i,1}x + a_{i,2}x^2 + \cdots + a_{i,n}x^n, \;\; i = 0 : m-1$$

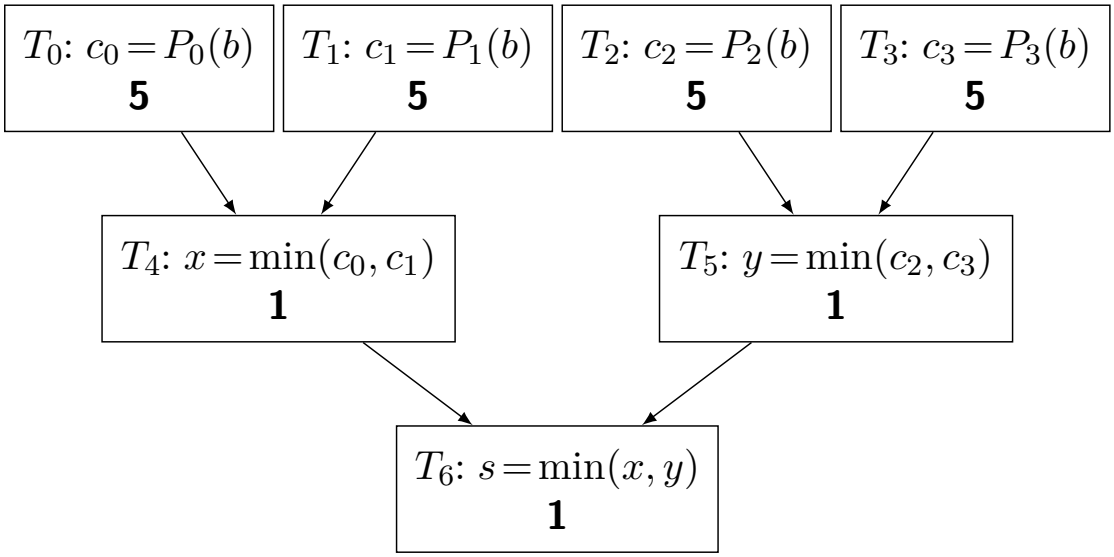
y un valor b, calcular

$$s = \min_{i=0:m-1} \{P_i(b)\},$$

Posible descomposición en tareas:

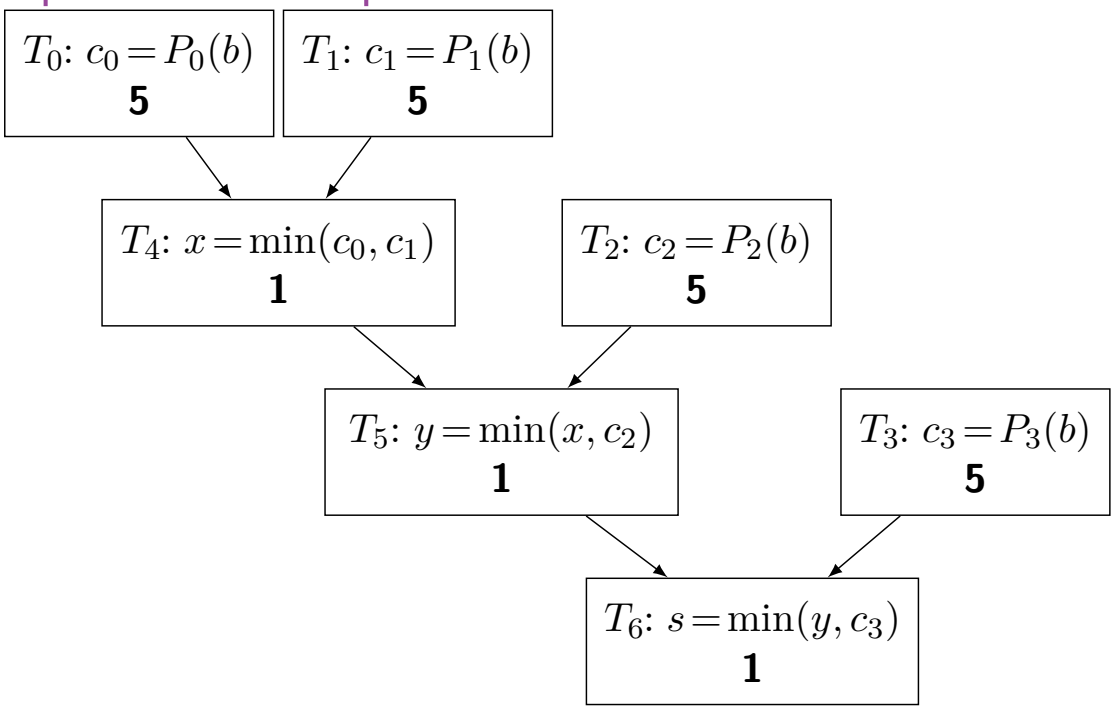
- Una tarea por cada evaluación de polinomio
  $\rightarrow$ independientes entre sí
- Varias tareas para calcular valores mínimos de dos en dos (recursivamente)

Ejemplo de Descomposición en Tareas: Grafo 1



$$L = 7, \quad M = \frac{5+5+5+5+1+1+1}{7} = 3{,}28$$

Ejemplo de Descomposición en Tareas: Grafo 2

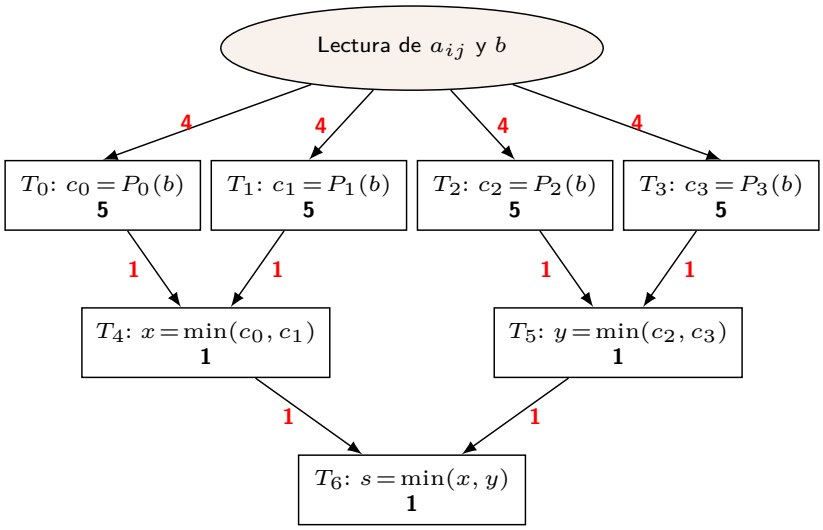


$$L = 8, \quad M = \frac{5+5+5+5+1+1+1}{8} = 2{,}875$$

Grafo con Comunicación

A veces el grafo incorpora información relativa a la comunicación

- Posibilidad de añadir nodos auxiliares (sin coste)
- Aristas con peso: denotan la comunicación entre tareas (valor proporcional a la cantidad de datos)



Apartado 3

Evaluación de Prestaciones (I)

- Parámetros Absolutos
- Prestaciones en Memoria Compartida

Evaluación de Prestaciones

El principal objetivo en la computación paralela es aumentar las prestaciones

- Es fundamental conocer cómo se comportan las diferentes partes de un programa paralelo
- Es también importante saber cómo se comportará ante cambios en el número de procesadores y el tamaño del problema

En este apartado se presentan diversas medidas y técnicas para detectar dónde un programa paralelo baja su rendimiento y compararlo con implementaciones secuenciales y otras configuraciones

Tipos de Análisis

Análisis a priori

- Realizado antes de la implementación del programa sobre el pseudocódigo o el diseño del programa
- Independiente de la máquina en la que se ejecuta
- Permite identificar la mejor opción a la hora de implementar un programa
- Permite determinar el tamaño de los problemas adecuado y las características del hardware adecuado

Análisis a posteriori

- Realizado sobre una implementación y máquina específica y utilizando un conjunto de datos de entrada
- Permite analizar cuellos de botella y detectar condiciones no observadas en el diseño

Análisis Teórico

El coste se analiza en función del tamaño del problema: n

En muchos casos el coste sólo depende de n: $t(n)$

Pero en ocasiones, ante un mismo n, puede haber un comportamiento diferente en función de los datos de entrada

- Coste del caso más favorable
- Coste del caso más desfavorable
- Coste promedio
  Promediando los tiempos de cada una de las posibles entradas por la probabilidad de que éstas aparezcan

En la práctica, se usan cotas asintóticas (inferior y superior)

Concepto de Flop

Flop: *floating point operation* - unidad de medida para:

- Coste de los algoritmos
- Rendimiento de los computadores (flop/s)

1 flop = coste de una operación elemental en coma flotante (producto, suma, división, resta)

- Consideramos despreciable el coste de las operaciones en aritmética entera
- El coste de otras operaciones en coma flotante se evaluará en base al Flop
  $\rightarrow$ por ejemplo, una raíz cuadrada igual a 8 flops

Supone una unidad de medida del coste independiente de la máquina (el tiempo que tarda un flop varía de un procesador a otro)

Notación Asintótica

Notación $\mathcal{O}$

- Permite acotar superiormente, salvo constantes y asintóticamente, la forma en que crece una función
- En la práctica se corresponde con el término de orden superior de la expresión del coste sin considerar su coeficiente
  - Ejemplo: la multiplicación matriz por vector es $\mathcal{O}(n^2)$

Notación o (o-pequeña)

- Tiene en cuenta además el coeficiente de mayor orden
- Adecuado cuando comparamos dos algoritmos que tienen el mismo orden $\mathcal{O}$
  - Ejemplo: el producto de matriz triangular por vector puede realizarse mediante el algoritmo convencional con coste $o(2n^2)$ o mediante un algoritmo optimizado $o(n^2)$

Parámetros para Evaluar las Prestaciones

Parámetros Absolutos

- Permiten conocer el coste real que tienen los algoritmos paralelos
- Suponen la base para el cálculo de los parámetros relativos que permiten comparar algoritmos
- Son los más importantes en problemas de tiempo real

Parámetros Relativos

- Permiten comparar los algoritmos paralelos entre sí y con respecto a las versiones secuenciales
- Proporcionan información del grado de aprovechamiento de los procesadores

Parámetros Absolutos

- Tiempo de ejecución de un algoritmo secuencial: $t(n)$
- Tiempo de ejecución de un algoritmo paralelo: $t(n, p)$
  - Tiempo aritmético: $t_a(n, p)$
  - Tiempo de comunicaciones: $t_c(n, p)$
- Coste total: $C(n, p)$
- *Overhead*: $t_o(n, p)$

Notación:

- Cuando la talla del problema es siempre n, sin ambigüedad, se omitirá, por ejemplo: $t(p)$
- A veces usaremos subíndices en vez de funciones: t_p, C_p

Tiempo de Ejecución

Tiempo que tarda en ejecutarse el algoritmo secuencial (en un solo procesador, $t(n)$) o el algoritmo paralelo (en p procesadores, $t(n,p)$)

- El coste a priori se medirá en flops
  - Sólo se tendrá en consideración el número de operaciones en coma flotante
- Experimentalmente el coste se medirá en segundos

Expresiones útiles para el cálculo del coste computacional:

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} i \approx \frac{n^2}{2} \qquad \sum_{i=1}^{n} i^2 \approx \frac{n^3}{3}$$

Coste Computacional: Ejemplos

```
PARA i=1 HASTA n
  PARA j=1 HASTA n
    x = x + a[i,j]
  FPARA
FPARA
```

$$t(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2 \text{ flops}$$

```
PARA i=1 HASTA n
  PARA j=i HASTA n
    x = x + 3.0*a[i,j]
  FPARA
FPARA
```

$$t(n) = \sum_{i=1}^{n}\sum_{j=i}^{n} 2 \approx \sum_{i=1}^{n} 2(n-i) =$$
$$2n^2 - 2\sum_{i=1}^{n} i \approx 2n^2 - 2\frac{n^2}{2} = n^2 \text{ flops}$$

```
PARA i=1 HASTA n
  PARA j=i HASTA n
    PARA k=i HASTA n
      x = x + a[i,k]
    FPARA
  FPARA
FPARA
```

$$t(n) = \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{n} 1 \approx \sum_{i=1}^{n}\sum_{j=i}^{n}(n-i) \approx$$
$$\sum_{i=1}^{n}(n^2 - 2ni + i^2) = \sum_{i=1}^{n} n^2 - 2n\sum_{i=1}^{n} i +$$
$$\sum_{i=1}^{n} i^2 \approx n^3 - \frac{2n^3}{2} + \frac{n^3}{3} = \frac{n^3}{3} \text{ flops}$$

Coste Total y *Overhead*

La ejecución de un algoritmo paralelo suele implicar un tiempo extra con respecto del algoritmo secuencial

El coste total paralelo contabiliza el total de tiempo empleado en un algoritmo paralelo

$$C(n,p) = p \cdot t(n,p)$$

El *overhead* indica cuál es el coste añadido con respecto del algoritmo secuencial

$$t_o(n,p) = C(n,p) - t(n)$$

Speedup y Eficiencia

El *speedup* indica la ganancia de velocidad que consigue el algoritmo paralelo con respecto a un algoritmo secuencial

$$S(n,p) = \frac{t(n)}{t(n,p)}$$

Hay que indicar a qué se refiere $t(n)$

- Puede ser el mejor algoritmo secuencial conocido
- Puede ser el algoritmo paralelo ejecutado en 1 procesador

La eficiencia mide el grado de aprovechamiento que un algoritmo paralelo hace de un computador paralelo

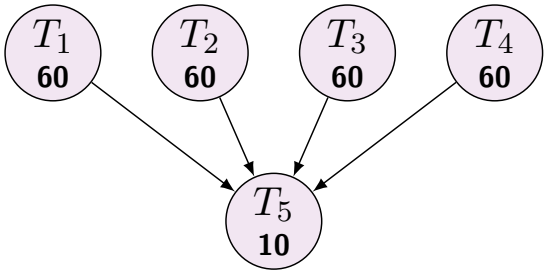
$$E(n,p) = \frac{S(n,p)}{p}$$

Suele expresarse en tanto por cien (o tanto por 1)

Ejemplo de Análisis de Prestaciones Básico

Supongamos este grafo de dependencias

(en este ejemplo, el coste no depende de n)



Suponemos que el alg. secuencial realiza T_1, T_2, T_3, T_4, T_5

Tiempo secuencial: $t_1 = 60 + 60 + 60 + 60 + 10 = 250$

Tiempo paralelo para $p = 4$, donde T_1, T_2, T_3, T_4 se ejecutan concurrentemente: $t_p = 60 + 10 = 70$

Speedup y eficiencia:

$$S_p = \frac{t_1}{t_p} = \frac{250}{70} = 3{,}57 \qquad E_p = \frac{S_p}{p} = \frac{3{,}57}{4} = 0{,}89$$

¿Cuál será el speedup para $p = 2$, $p = 3$ y $p > 4$?

Cómo Obtener Buenas Prestaciones

Idealmente, para p procesadores tenemos un *speedup* igual a p (eficiencia igual a 1)

¿De qué depende que nos acerquemos más o menos?

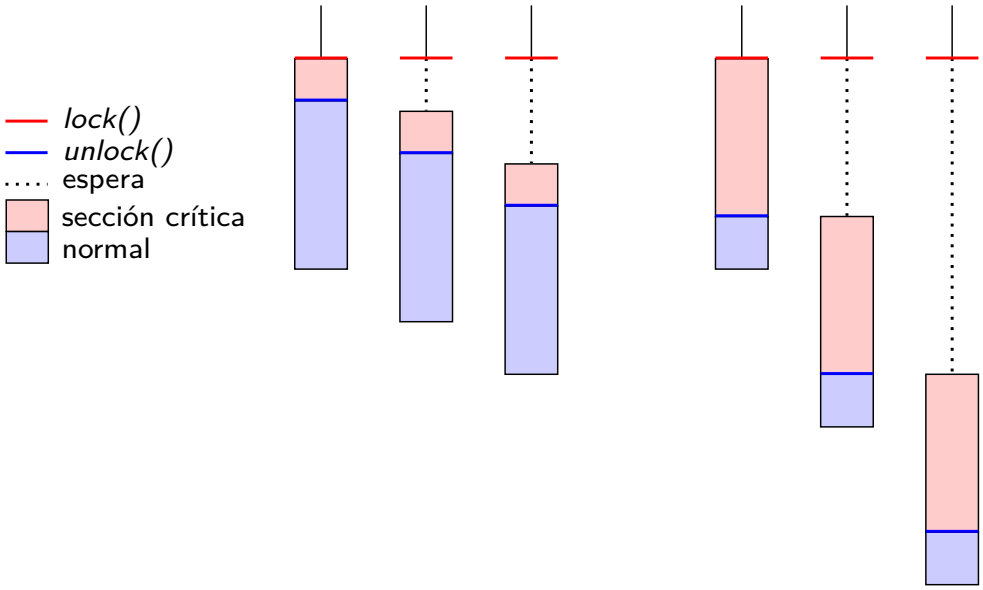
- Diseño de la paralelización apropiado
  - Reparto de la carga equilibrado
  - Minimizar tiempo en que los procesadores están ociosos
  - *Overhead* mínimo posible
- Aspectos específicos de la arquitectura donde se ejecuta
  - Distintos en memoria compartida o paso de mensajes
  - El tiempo de acceso a los datos no se considera en el análisis teórico del coste pero es muy importante en las arquitecturas actuales

Sincronización: Eficiencia

La sincronización puede tener impacto negativo en la eficiencia

La sección crítica debe ser lo más pequeña posible

- De lo contrario se produce una "serialización"



De igual forma, hay que evitar las barreras en lo posible

Apartado 4

Diseño de Algoritmos: Descomposición en Tareas

- Descomposición de Dominio
- Otras Descomposiciones

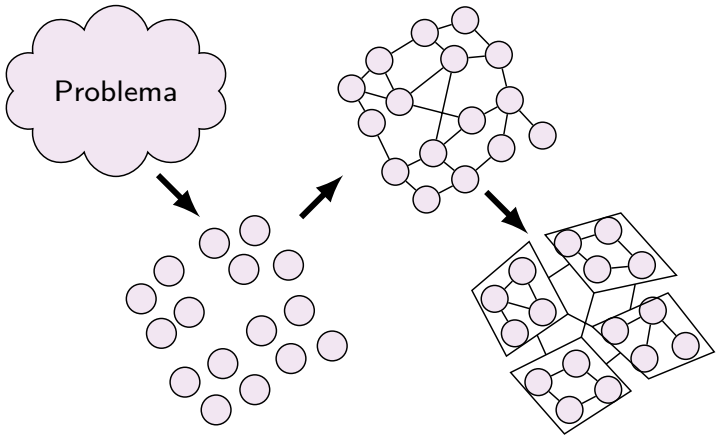
Diseño de Algoritmos Paralelos

El diseño de algoritmos paralelos presenta una complejidad mucho mayor que en el caso secuencial

- Concurrencia (implica comunicación y sincronización)
- Asignación de datos y código a procesadores
- Acceso simultáneo a datos compartidos
- Escalabilidad, para un número creciente de procesadores

Los principales pasos en el diseño son:

- Descomposición en tareas
- Asignación de tareas



Descomposición en Tareas

Tarea: cada una de las unidades de computación definidas por el programador que potencialmente pueden ser ejecutadas en paralelo

- El proceso de dividir un cálculo/programa en tareas se denomina descomposición

Granularidad

- La descomposición puede ser de grano fino o grano grueso
- Normalmente se hace una descomposición de grano fino y posteriormente se agrupan en tareas más grandes

Técnicas de Descomposición

- Descomposición del dominio
- Descomposición funcional dirigida por el flujo de datos
- Descomposición recursiva
- Otras: descomposición exploratoria, descomposición especulativa, enfoques mixtos

Descomposición del Dominio

En el caso de grandes estructuras de datos regulares

- Se dividen los datos en partes de tamaño similar (subdominios)
- A cada subdominio se le asocia una tarea, la cual realizará las operaciones necesarias sobre los datos del subdominio

Suele utilizarse cuando es posible aplicar el mismo conjunto de operaciones sobre los datos de cada subdominio

La descomposición puede ser:

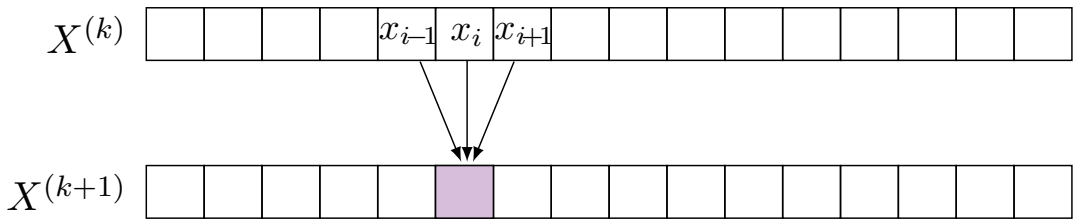
- Centrada en los datos de salida
- Centrada en los datos de entrada
- Centrada en los datos intermedios
- Descomposición basada en bloques (algoritmos matriciales)

D. D. Centrada en los Datos de Salida

Cada componente de los datos de salida se puede calcular de forma independiente del resto

Ejemplo: diseñar un algoritmo paralelo iterativo que calcule la sucesión de vectores $X^{(0)}, X^{(1)}, \ldots, X^{(k)}, X^{(k+1)}, \ldots \in \mathbb{R}^n$, donde $X^{(0)}$ es un vector conocido y el resto se obtienen así:

$$x_i^{(k+1)} = \frac{x_{i-1}^{(k)} - x_i^{(k)} + x_{i+1}^{(k)}}{2}, \quad i = 0, \ldots, n-1$$
$$x_{-1}^{(k)} = x_{n-1}^{(k)}, \quad x_n^{(k)} = x_0^{(k)}$$



D. D. Centrada en los Datos de Entrada

Ejemplo: Producto escalar de vectores

$$\left.\begin{array}{l} x = [x_0, x_1, \ldots, x_{n-1}] \\ y = [y_0, y_1, \ldots, y_{n-1}] \end{array}\right\} \quad \Rightarrow \quad x \cdot y = \sum_{i=0}^{n-1} x_i y_i$$

Suponiendo p tareas y n divisible entre p, entonces la tarea i-ésima $(i = 0, \ldots, p-1)$ calcularía

$$\sum_{j=i\frac{n}{p}}^{(i+1)\frac{n}{p}-1} x_j y_j$$

Finalmente, habría tareas adicionales para acumular las sumas parciales en la suma global

Descomposición Funcional

La descomposición funcional dirigida por el flujo de datos se utiliza cuando

- La resolución del problema se puede descomponer en fases
- En cada fase se ejecuta un algoritmo distinto

Se suelen seguir los siguientes pasos:

1. Se identifican las diferentes fases
2. A cada fase se le asigna una tarea
3. Se analizan los requisitos de datos para cada una de las tareas
   - Si el solapamiento de datos entre distintas tareas es mínimo y el flujo de datos entre ellas es relativamente pequeño, la descomposición estará completa
   - Si no ocurre lo anterior, sería necesario analizar otro tipo de descomposición

Descomposición Recursiva

Es un método para obtener concurrencia en problemas que pueden resolverse mediante la técnica de divide y vencerás

1. Dividir el problema original en dos o más subproblemas
2. A su vez estos subproblemas se dividen en dos o más subproblemas y así sucesivamente hasta finalizar el proceso usando cierto criterio de parada
3. Los datos obtenidos se combinan adecuadamente para obtener el resultado final

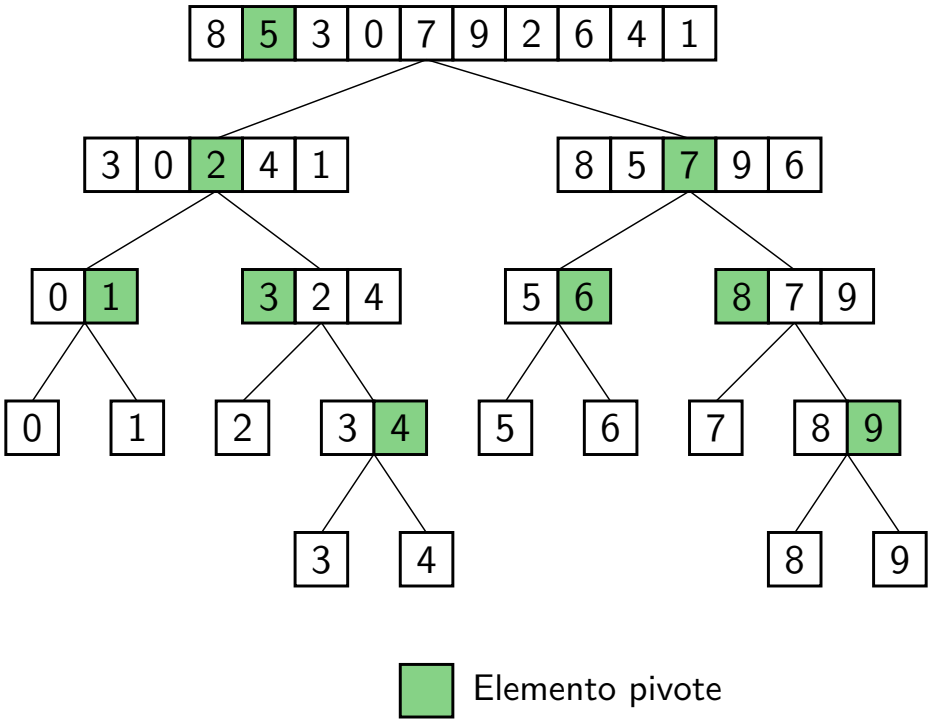
Puede implementarse de diferentes formas:

- Trabajadores replicados con bolsa de tareas
- Algoritmo recursivo

Veremos estas opciones en el apartado Esquemas Algorítmicos

Descomposición Recursiva

Ejemplo: Ordenación Quicksort



Apartado 5

Esquemas Algorítmicos (I)

- Trabajadores Replicados
- Divide y Vencerás

Esquemas Algorítmicos

Los esquemas algorítmicos son modelos de soluciones habituales de paralelización

- Un esquema sirve para resolver una amplia gama de problemas
- Un problema suele requerir combinar varios esquemas

Algunos esquemas:

- Paralelismo de datos / particionado de datos
- Paralelismo de tareas (maestro-trabajadores, trabajadores replicados)
- Esquemas en árbol y grafo (divide y vencerás)
- Paralelismo segmentado (*pipelining*)
- Paralelismo síncrono

Trabajadores Replicados con Bolsa de Tareas

Bolsa de tareas: estructura de datos compartida que contiene las tareas pendientes

```
int get_next_task() {
    static int index = 0;
    int result;
    #pragma omp critical
    {   if (index==MAXIDX) result=-1;
        else { index++; result=index; }
    }
    return result;
}
...
    int myindex;
    #pragma omp parallel private(myindex)
    {   myindex = get_next_task();
        while (myindex>-1) {
            process_task(myindex);
            myindex = get_next_task();
        }
    }
```

En el ejemplo hay un número fijo de tareas (`MAXIDX`)

Divide y Vencerás

Este método consiste en resolver un problema descomponiéndolo en una serie de subproblemas similares, resolviendo los subproblemas y combinando las soluciones

$\rightarrow$ Suele implementarse de forma recursiva (árbol)

Hay varios tipos de tareas:

- Dividir el problema: se realiza en los nodos internos para crear los nodos hijos
- Resolver el caso base: sólo en las hojas del árbol
- Combinar los resultados: se realiza en los nodos internos, colapsando el sub-árbol asociado

Ejemplos:

- *Quicksort* tiene el mayor coste en la fase de división
- *Mergesort* concentra el trabajo en la combinación

Divide y Vencerás: Ejemplo

Solución recursiva con tareas: en cada llamada se realizan dos llamadas recursivas que generan una nueva tarea cada una

Mergesort Paralelo

```
void mergesortpar(double *a, int n)
{
  int k;
  if (n<=nsmall)
    mergesortseq(a,n);
  else {
    k = n/2;
    Crear dos tareas:
      1. llamada recursiva mergesortpar(a,k)
      2. llamada recursiva mergesortpar(a+k,n-k)
    Esperar finalización de tareas
    mezclar(a,k,n-k);
  }
}
```