

Resumen problemas concurrencia

Problema: ¿Hay Database condition?

Todas las variables son compartidas, niñas

V_5 que es privada por thread y $*var$ (que apunta al Heap, memoria app compartida)

`for(i=0; i<n, i++)`

$V_1[i] = V_1[i] + a[i]$ \rightarrow Lectura = 10 database

$V_2 = V_2 + a[i]$ \rightarrow Varios threads tocan a la vez V_2 , si hay database

$V_3 = V_3 + a_2 + c[i]$ \rightarrow Lo mismo que el anterior, se lee y escribe en variable compartida

$V_4 = V_4 + 1$ \rightarrow Si hay database

$V_5 = V_5 + 1$ \rightarrow NO hay, es privada por Thread

$*var = *var + 1$ \rightarrow Es privado, pero apunta a memoria compartida \rightarrow Si database

#prueba `omp atomic` \rightarrow permite que no se metan otros threads

#prueba `omp critical`

{ \rightarrow causa meter un set-lock

exclusividad

I \rightarrow set-unlock

Video 2

Idealmente: paralelizar una app tal que: Tiempo ejecución en P procesadores: $T_p = \frac{T_1}{P}$

Pero hay que añadir penalizaciones $T_p = \frac{T_1}{P} + \text{overhead}(p)$ \rightarrow Por lo que

TDG: Task Dependency Graph

Las tareas producen y consumen

$$T_p \geq T_1/P$$

↓
tiempo ideal

$$T_p \geq T_{\text{eo}}$$

Si algo producido por una tarea lo necesita otra \rightarrow dependencia (de datos)

Métricas

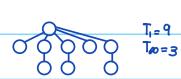
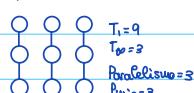
• T_1 : Tiempo que tarda en procesarse en secuencial

• T_{eo} : Tiempo que tarda en procesarse con infinitos recursos el cráneo crítico
(se tiene infinitos recursos, todo lo demás se oculta)

• Parallelismo: $\frac{T_1}{T_{\text{eo}}}$

• Puer: Número mínimo de procesadores que nos permite que nuestra estrategia de paralelización ejecutada en una máquina con ese número de CPUs tarde T_{eo}
Por lo que nuestro T con Puer es T_{eo}
Se necesita saber ordenes de Parallelismo, TDG y coste de tareas

Ejemplos:



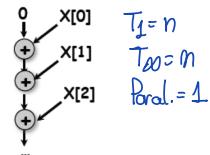
Ejemplo 1

Vector sum

Compute the sum of elements $X[0] \dots X[n-1]$ of a vector X
 $\text{sum} = 0; \text{for } (i=0; i < n; i++) \text{sum} += X[i];$

Sí: hacemos secuencial:

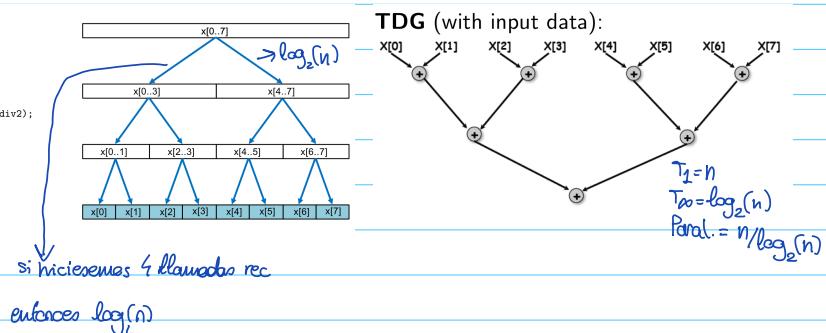
TDG (with input data)



Pero si lo hacemos recursivo:

```
int recursive_sum(int *X, int n) {
    int ndiv2 = n/2;
    int sum0;
    if (n==1) return X[0];
    sum1 = recursive_sum(X, ndiv2);
    sum2 = recursive_sum(X+ndiv2, n-ndiv2);
    return sum1+sum2;
}

void main() {
    int sum, X[N];
    ...
    sum = recursive_sum(X, N);
    ...
}
```



Granularidad y

parallelismo

Dado un problema secuencial \rightarrow Número de fases que uno puede generar y el tamaño de estas (granularidad)

\uparrow parallelismo \equiv \uparrow granularidad

Se relacionan

Fine vs coarse

Δ NO tiene por qué bajar el tiempo paralelo haciendo descomposición más pequeña \rightarrow Aumento de overhead por ejemplo Haciendo una descomposición más pequeña: Parallelización y granularidad \uparrow

! No necesariamente baja el tiempo

⚠️ Mandelbrot → embarrassing parallel
 No hay dependencias, por lo que es todo paralelo. No es perfecto ya que se producen **local imbalances**

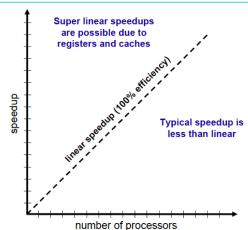
• Speedup: Reducción relativa del tiempo de ejecución secuencial usando P procesador

$$Sp = \frac{T_1}{T_p}$$

⚠️ $Sp \geq 1$ si está bien

• Escalabilidad: Cómo el Speedup evoluciona en función de los procesadores

$$Ep = \frac{Sp}{P}$$



Escalabilidad

1) Fuerte: Incrementar el número de procesadores P , manteniendo el tamaño del problema (N) pero ahora repartiendo la carga de trabajo entre N/p partes del problema
 Se reduce el $T_{calculo}$

2) Débil: Manteniendo el $T_{calculo}$ aumentar la carga de trabajo siendo ahora de $N \cdot P$, y por lo tanto aumentar también el número de procesadores

Ley de Amdahl

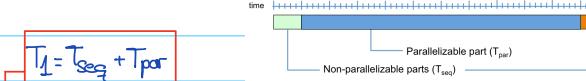
La mejora de rendimiento se limita por la fracción de tiempo secuencial \rightarrow Se puede calcular:

Además se puede calcular la Eficiencia \rightarrow $E_p = \frac{T_{seq,par}}{T_{seq}}$

$$S_{parallel} = \frac{T_{seq}}{T_{par}}$$

$$S_{sequential} = \frac{T_{seq}}{T_{par}}$$

Para 1 procesador:



$$T_1 = T_{seq} + T_{par}$$

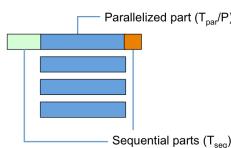
$$\varphi = \frac{T_{par}}{T_1} = \frac{T_{par}}{T_{seq} + T_{par}}$$

$$T_{par} = \varphi \cdot T_1$$

$$T_{seq} = (1-\varphi) \cdot T_1$$

Para p procesadores

$$T_p = T_{seq} + T_{par}/p$$



$$T_p = (1-\varphi)T_1 + (\varphi \cdot T_1/p)$$

Speedup

$$Sp = \frac{T_1}{T_p} = \frac{T_1}{(1-\varphi)T_1 + (\varphi \cdot T_1/p)} = \frac{1}{(1-\varphi) + \varphi/p}$$

$$\Delta Sp > \infty = \frac{1}{1-\varphi}$$

Pero no es ideal, hay overheads por lo que:

$$T_p = (1-\varphi) \cdot T_1 + \varphi \cdot T_1/p + \text{overhead}(p) \quad (\leftarrow T_p = T_{seq} + T_{par} + \text{overhead})$$

Fuentes de overhead

Data sharing: Por mensajes (explicits) o por cache (implícitos)

Idleness: El thread no encuentra trabajo a ejecutar (por dependencias, load unbalance...)

Computation: Trabajo extra para obtener el algoritmo paralelo

Memoria: Memoria extra usada para obtener el algoritmo paralelo

Contention: Competición para acceder a recursos compartidos

Ejemplo 2

Computación

Stencil - Jacobi

Gauss

Computación Stencil: Para calcular mandelbrot, expansión de color...

Tiene forma de: while(error < 0){

Jacobi / Gauss

I

Por lo que se puede calcular por Jacobi o por Gauss

Jacobi usa dos matrices: U y V-temp sin dependencias

Gauss usa una matriz: U, con dependencias

Jacobi

Usa más iteraciones para el error, pero tiene menos dependencia que Gauss

```

void compute(int n, double *u, double *utmp) {
    int i, j;
    double tmp;

    for (i = 1; i < n-1; i++) {
        for (j = 1; j < n-1; j++) {
            tmp = u[n*(i+1) + j] + u[n*(i-1) + j] + // elements u[i+1][j] and u[i-1][j]
                  u[n*i + (j+1)] + u[n*i + (j-1)] - // elements u[i][j+1] and u[i][j-1]
                  4 * u[n*i + j];
            utmp[n*i + j] = tmp / 4;           // element utmp[i][j]
        }
    }
}

```

→ $n-1$ iteraciones en i } $(n-1)^2$ iteraciones } $n-1$ iteraciones en j } $(n-1)^2$ iteraciones }
 body tarda t_{body}

Task is ... (granularity)	Num. tasks	Task cost	T_1	T_{∞}	Parallelism
All iterations of i and j loops	1	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	1
Each iteration of i loop	n	$n \cdot t_{body}$	$n^2 \cdot t_{body}$	$n \cdot t_{body}$	n
Each iteration of j loop	n^2	t_{body}	$n^2 \cdot t_{body}$	t_{body}	n^2
r consecutive iterations of i loop	$n \div r$	$r \cdot r \cdot t_{body}$	$n^2 \cdot t_{body}$	$n \cdot r \cdot t_{body}$	$n \div r$
c consecutive iterations of j loop	$n^2 \div c$	$c \cdot t_{body}$	$n^2 \cdot t_{body}$	$c \cdot t_{body}$	$n^2 \div c$
A block of $r \times c$ iterations of i and j, respectively	$n^2 \div (r \cdot c)$	$r \cdot c \cdot t_{body}$	$n^2 \cdot t_{body}$	$r \cdot c \cdot t_{body}$	$n^2 \div (r \cdot c)$

Task is ... (granularity)	Num. tasks	Task cost	Task creation ovh
All iterations of i and j loops	1	$n^2 \cdot t_{body}$	t_{create}
Each iteration of i loop	n	$n \cdot t_{body}$	$n \cdot t_{create}$
Each iteration of j loop	n^2	t_{body}	$n^2 \cdot t_{create}$
r consecutive iterations of i loop	$n \div r$	$n \cdot r \cdot t_{body}$	$(n \div r) \cdot t_{create}$
c consecutive iterations of j loop	$n^2 \div c$	$c \cdot t_{body}$	$(n^2 \div c) \cdot t_{create}$
A block of $r \times c$ iterations of i and j, respectively	$n^2 \div (r \cdot c)$	$r \cdot c \cdot t_{body}$	$(n^2 \div (r \cdot c)) \cdot t_{create}$

→ teniendo en cuenta

ahead de creación

Gauss

Menos iteraciones, más dependencia

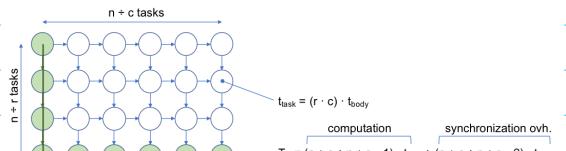


```
void compute(int n, double *u, double *utmp) {
    int i, j;
    double tmp;

    for (i = 1; i < n-1; i++) {
        for (j = 1; j < n-1; j++) {
            tmp = u[n*(i+1) + j] + u[n*(i-1) + j] + // elements u[i+1][j] and u[i-1][j]
                  u[n*i + (j+1)] + u[n*i + (j-1)] - // elements u[i][j+1] and u[i][j-1]
                  4 * u[n*i + j];
            u[n*i + j] = tmp / 4; // element u[i][j]
        }
    }
}
```

1) each task computes a block of $r \times c$ iterations of the i and j loops

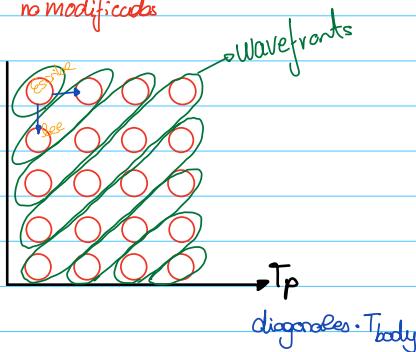
2) Cada tarea de sincronización tarda t_{synch}



Tareas:

$P_1 a, b, c$

$P_3 a, b$



Data Sharing

Nos basaremos en arquitectura de memoria no compartida

- Los procesadores si acceden a su memoria local NO tienen overhead
- Pueden acceder a otras memorias para realizar loads (remote load) → genera overhead

Para calcular el Taccess

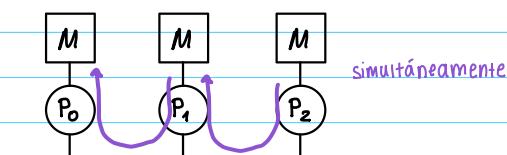
- Startup: Tiempo en preparar el acceso remoto (t_s)
- Transfer: Tiempo en transferir el mensaje (m Bytes, T_m tiempo por Byte) ($m \cdot T_m$)

$$\text{Por lo que } T_{\text{access}} = t_s + m \cdot T_m$$



- Un procesador P_i solo puede ejecutar UN acceso remoto
 - Un procesador P_i solo puede servir UN acceso remoto
- a la vez se puede

Modelar el coste de compartición de datos.



1. LD/ST = coste 0 (acc. a mem. → local)

2. Coste inicialización del msj.: $t_s + N \cdot w$ (coste com. x elemento)
↳ Reglas:

1. Remotamente, "yo" sólo puedo acceder a 1 a la vez

2. Remotamente, "a mí" sólo me pueden acceder 1 a la vez

3. Remotamente, "yo" puedo acceder a 1 mientras 1 me accede a mí.

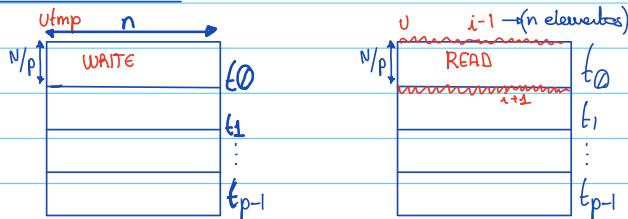
Jacobi

Definiremos la fase como: $\frac{n}{p}$ iteraciones del loop i (filas)

```
void compute(int n, double *u, double *utmp) {
    int i, j;
    double tmp;

    for (i = 1; i < n-1; i++) {
        for (j = 1; j < n-1; j++) {
            tmp = u[n*(i+1) + j] + u[n*i - 1 + j] + // elements u[i+1][j] and u[i-1][j]
                  u[n*i + (j+1)] + u[n*i + (j-1)] - // elements u[i][j+1] and u[i][j-1]
                  4 * u[n*i + j]; // element u[i][j]
            utmp[n*i + j] = tmp / 4; // element utmp[i][j]
        }
    }
}
```

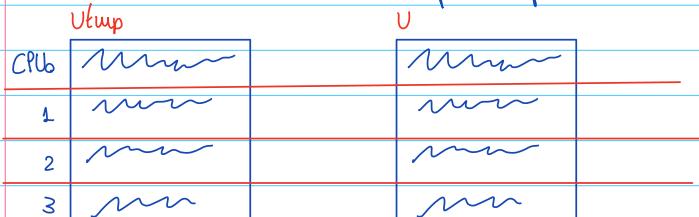
Paso 1 - Ver Tareas



Paso 2 - TDG



Paso 3 - Distribución de memoria (Suponemos $p=4$)



Por lo que en los momentos que $i+1$ y $i-1$ salten de región de memoria \rightarrow overhead
ya que una CPU necesita datos de otra.

Paso 4 - Cronograma y cálculos

CPU 0: L tarea

1: L U m

2: L U m

3: L U m

no se puede ya que
se hace L y U a la vez

$$\cdot T_1 = n^2 \cdot t_{body} \xrightarrow{\text{carga del bucle inter}}$$

$$\cdot T_{\text{task}} = \frac{N}{p} \cdot N \cdot t_{body} = \frac{N^2}{p} \cdot t_{body}$$

$\frac{N}{p} \uparrow$ [Tarea] $\leftrightarrow N$

$$\cdot T_{\text{over}} = 2 \cdot (t_0 + n \cdot t_w)$$

$$\cdot T_p = T_{\text{task}} + T_{\text{over}}$$

$$\cdot Sp = \frac{T_1}{T_p}$$