CMCP – Conceptos y Métodos de la Computación Paralela

Master en Computación en la Nube y de Altas Prestaciones

T2. Programación en Memoria Compartida

J. E. Roman

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curso 2024-2025

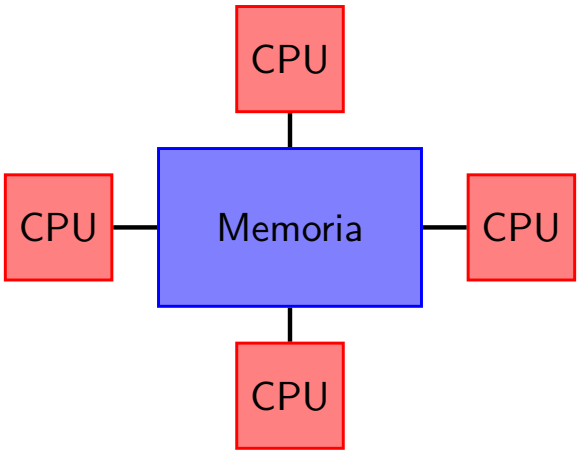
Contenido

Apartado 1

Arquitectura de Memoria Compartida

Arquitecturas de Memoria Compartida

Espacio de direcciones único para todos los procesadores



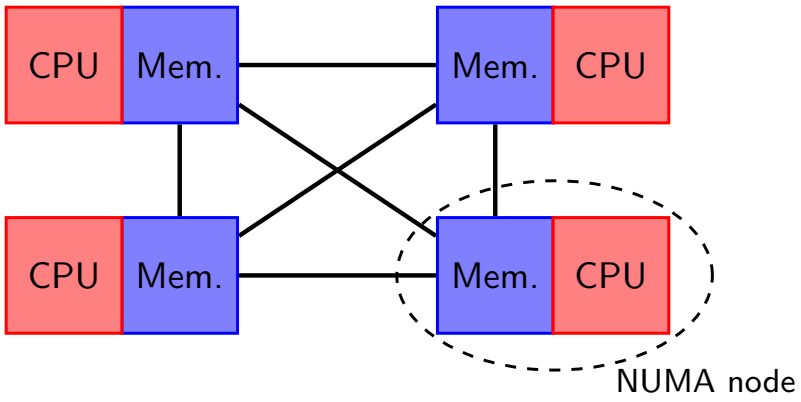
Tipos:

- UMA: Uniform Memory Access
- NUMA: Non-Uniform Memory Access
- cc-NUMA: Cache Coherent NUMA

Ventajas: fácil programación; Desventajas: escalabilidad, precio

Memoria Compartida NUMA

El tiempo de acceso a memoria depende de la posición de la memoria relativa al procesador

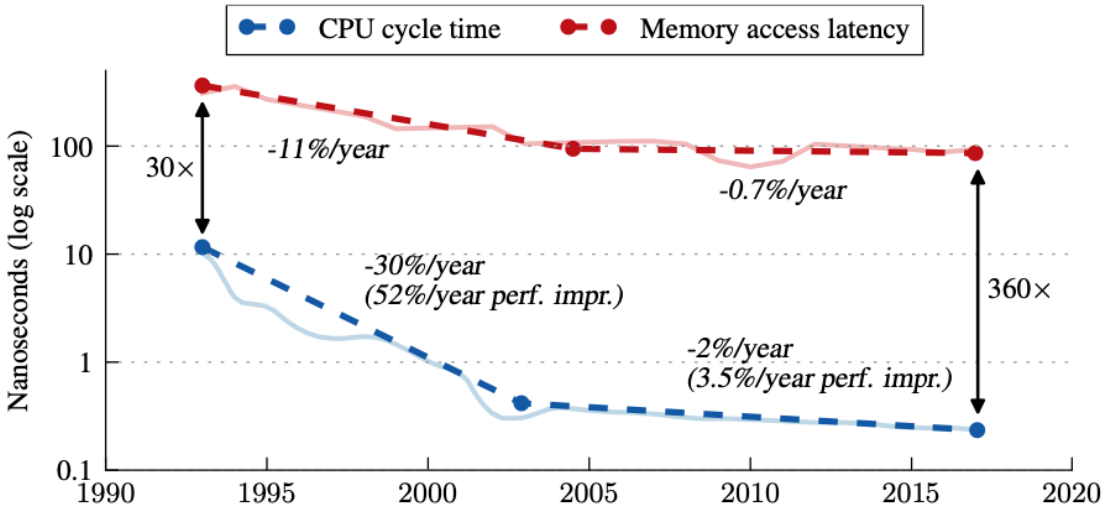


La memoria es el cuello de botella del sistema

- Procesador más rápido que la memoria
- El sistema de memoria debe servir a todos a la vez
- La potencia de pico del procesador es inalcanzable

Prestaciones de la Memoria

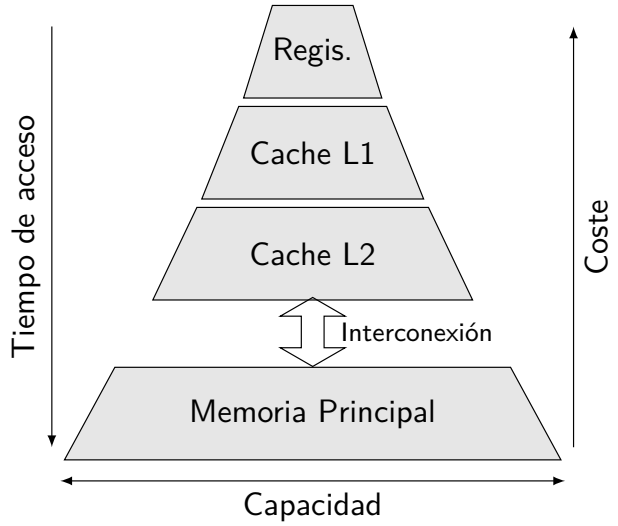
En las últimas décadas, la mejora de velocidad de acceso a la memoria ha sido menor que la del procesador (*memory wall*)



Fuente: M. Radulović, "Memory bandwidth and latency in HPC: system requirements and performance impact", PhD thesis, UPC (2019)

Mejoras recientes: DRAM apiladas en 3D - *High Bandwidth Memory* (HBM), *multi-channel DRAM* (MCDRAM)

Jerarquía de Memorias



Objetivo: situar cerca del procesador los datos que se van a usar de forma inmediata (localidad de referencia, temporal y espacial)

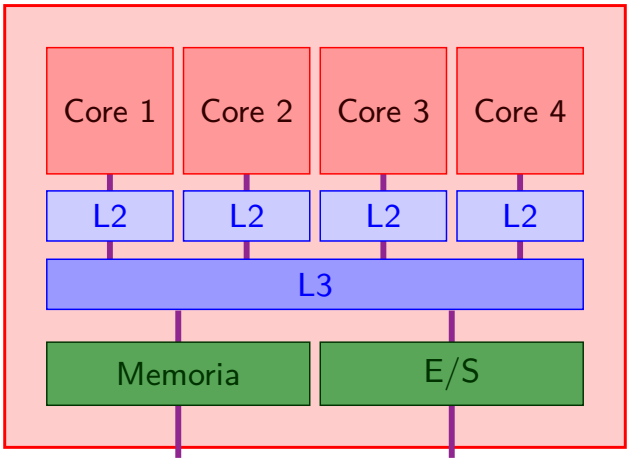
Otro nivel: almac. persistente (SSD, HDD), algoritmos *out-of-core*

Memoria cache en multiprocesadores

- Juega un papel crucial en las prestaciones
- Política de gestión de coherencia de la cache

Procesadores *Multi-Core*

Múltiples núcleos: tendencia actual en diseño de procesadores



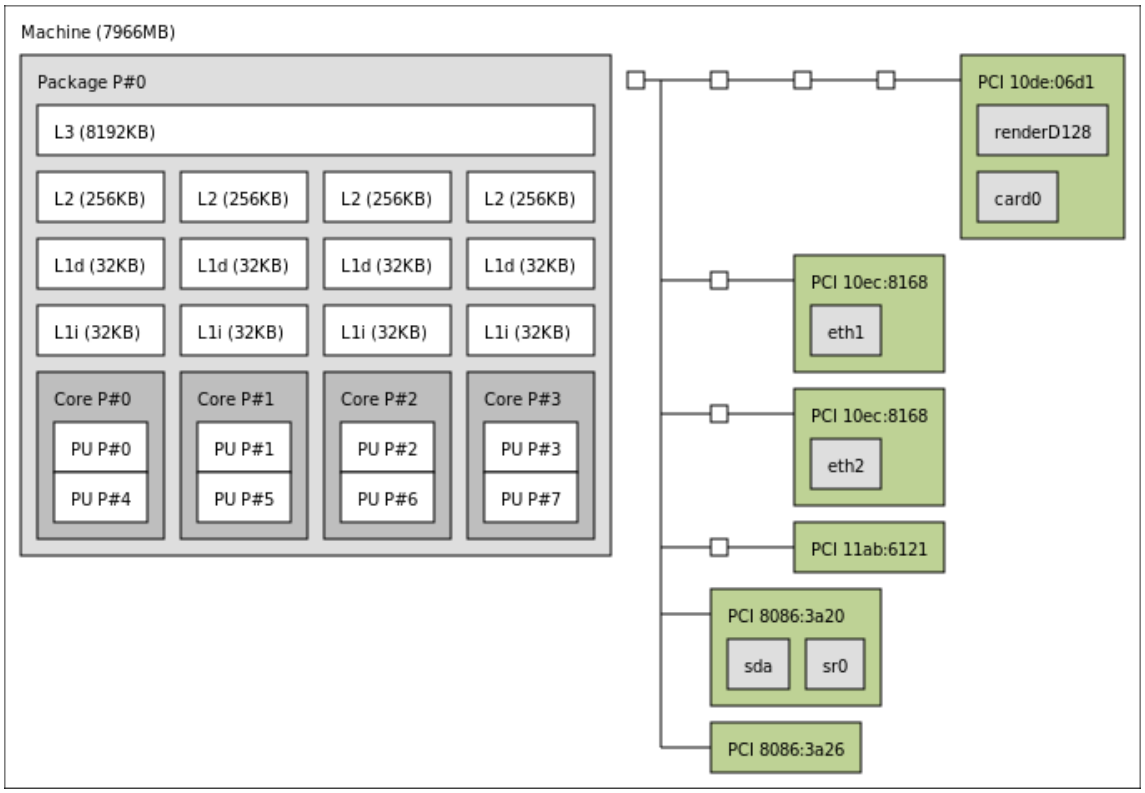
Características

- Multiprocesador (no) simétrico en un solo chip
- Varios niveles de cache en el propio chip

Ventajas: precio; Desventajas: baja eficiencia (ancho de banda)

`lstopo`

El comando `lstopo` (del software `hwloc`) muestra la topología



Apartado 2

Introducción a OpenMP

- Creación de Hilos
- Entorno de Datos

Modelo de Hilos

Para arquitecturas de memoria compartida

- Espacio de direcciones de memoria común
- Programación bastante similar al caso secuencial
  - Variables accesibles por todos
  - No es necesario intercambiar datos explícitamente
- Peligros
  - El acceso concurrente a memoria puede ser problemático
  - Mala eficiencia si no se usa la memoria caché con cuidado

Procesos multi-hilo

- Hilo: flujo de instrucciones independiente que puede ser planificado para ejecución por el sistema operativo
- Un proceso puede tener múltiples hilos de ejecución
- Todos los hilos comparten espacio de direcciones del proceso (código, datos) y recursos (p.e., ficheros abiertos)

OpenMP

`http://www.openmp.org`

Una especificación para programación paralela basada en hilos

- Implementado en el compilador (con directivas)
- Disponible en C/C++ y Fortran
- Portable/multi-plataforma (Unix, Windows)
- Creación/terminación de hilos implícita (una de las directivas)
- Mejora continua: versión actual es 5.2

Componentes

- Directivas de compilador
- Biblioteca OpenMP: unas pocas funciones simples
- Variables de entorno

Sintaxis

Directivas:

```
#pragma omp <directive> [clause [...]]
```

Uso de funciones:

```
#include <omp.h>
...
iam = omp_get_thread_num();
```

Opciones de compilador:

```
   (gnu)$ gcc -fopenmp prg-omp.c
 (intel)$ icc -qopenmp prg-omp.c
(nvidia)$ nvc -mp prg-omp.c
```

Compilación condicional: comprobar si OpenMP está activo

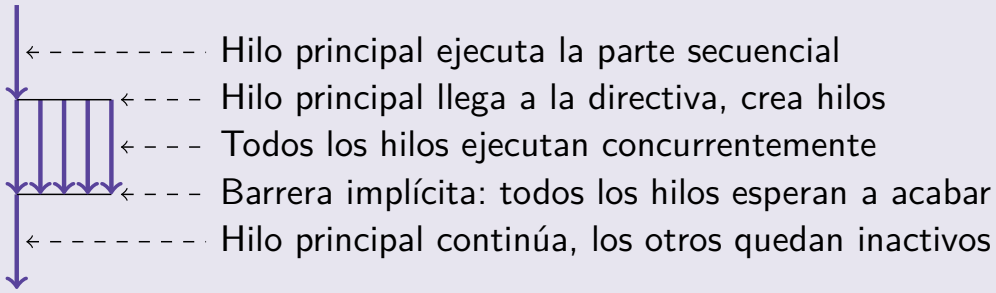
```
#if defined(_OPENMP)
iam = omp_get_thread_num();
#endif
```


Modelo de Ejecución

El modelo de ejecución de OpenMP sigue un esquema *fork-join*

La directiva `parallel` crea los hilos

Modelo fork-join



Una región paralela es la parte del código ejecutado por un equipo de hilos

- Un programa OpenMP es una sucesión de regiones secuenciales y paralelas
- Los hilos se "reciclan" de una región paralela a la siguiente

Ejemplo Básico

Ejemplo Hola Mundo en OpenMP

```
#include <stdio.h>
int main() {
  #pragma omp parallel
  {
    printf("Hello world\n");
  }
  return 0;
}
```

- Cuando se alcanza la directiva, se crean los hilos
- Cada hilo imprimirá el mensaje: ejecución replicada
- La ejecución espera a que todos los hilos hayan acabado
- Podemos omitir las llaves `{...}` si solo hay una sentencia
- La llave de abertura `{` debe ir DEBAJO de la directiva

Establecer el Número de Hilos

El número de hilos se puede especificar de varias formas

1. Cláusula `num_threads`
   - Un modificador de la directiva
   - Solo afecta a esa directiva
2. Función `omp_set_num_threads()`
   - Se debe llamar en la parte secuencial
   - Afecta a todas las regiones paralelas posteriores
3. Variable de entorno `OMP_NUM_THREADS`
   - Todas las regiones paralelas del programa
   - Típicamente se asigna el valor al ejecutar el programa
   - Método más común

Si usamos varios métodos: `num_threads` tiene la precedencia más alta, `OMP_NUM_THREADS` la menor

Número de hilos por defecto: normalmente igual al número de núcleos

Funciones de Hilos

- `omp_get_num_threads()`: devuelve el número de hilos
- `omp_get_thread_num()`: devuelve el identificador de hilo (empieza en 0, hilo principal es siempre 0)

```
#include <omp.h>
#include <stdio.h>
int main()
{
  omp_set_num_threads(3);
  #pragma omp parallel
  {
    printf("threads = %d\n",omp_get_num_threads());
    printf("I am %d\n",omp_get_thread_num());
  }
  return 0;
}
```

Atención: `omp_get_num_threads()`$\rightarrow$1 si llamada fuera de `parallel`

Ejecución Concurrente

Posible salida del programa anterior

```
threads = 3
I am 0
threads = 3
I am 1
threads = 3
I am 2
```

```
threads = 3
I am 0
threads = 3
I am 2
threads = 3
I am 1
```

```
threads = 3
threads = 3
I am 1
I am 0
threads = 3
I am 2
```

- Los hilos se ejecutan independientemente
- No se puede predecir qué hilo acabará antes
- Al imprimir en pantalla, la salida de los hilos se intercala en un orden impredecible

Hacer que los Hilos Hagan Cosas Distintas

La ejecución replicada es inútil para computación paralela

- Cada hilo debería realizar un cálculo distinto
- Usar un identificador de hilo para decidir quién hace qué

```
#include <omp.h>
#include <stdio.h>
int main()
{
  #pragma omp parallel
  {
    if (omp_get_thread_num()==0)
      printf("threads = %d\n",omp_get_num_threads());
    printf("I am %d\n",omp_get_thread_num());
  }
  return 0;
}
```

En este ejemplo el número de hilos se mostrará solo una vez

Sincronización con Barreras

¿Cómo asegurarse de que `threads = ...` aparece primero?
$\longrightarrow$ usar una barrera explícita (espera a que todos lleguen)

```
#pragma omp parallel
{
  if (omp_get_thread_num()==0)
    printf("threads = %d\n",omp_get_num_threads());
  #pragma omp barrier
  printf("I am %d\n",omp_get_thread_num());
}
```

La sincronización de hilos es muy importante

- Controlar el progreso de los hilos
- Algunas directivas tienen una barrera implícita
- Garantizar la corrección de un programa
- Particularmente para evitar condiciones de carrera

Variables Compartidas

Vamos a almacenar los valores en variables

```
int nthr, myid;
#pragma omp parallel
{
  nthr = omp_get_num_threads();
  myid = omp_get_thread_num();
  if (myid==0)
    printf("threads = %d\n",nthr);
  #pragma omp barrier
  printf("I am %d\n",myid);
}
```

```
threads = 4
I am 3
I am 3
I am 3
I am 3
```

La salida es incorrecta, todos los hilos muestran el mismo id. Por qué?

- Por defecto, las variables son compartidas: todos los hilos pueden leer y escribir en la misma posición de memoria
- `myid` no puede ser una variable compartida porque hilos distintos escriben un valor distinto
- `nthr` está bien porque todos escriben el mismo valor

Variables Privadas

Una variable privada es aquella que permite tener valores diferentes en cada hilo

- Es como tener un array con tantos elementos como hilos

```
int nthr;
#pragma omp parallel
{
  int myid;
  nthr = omp_get_num_threads();
  myid = omp_get_thread_num();
  if (myid==0)
    printf("threads = %d\n",nthr);
  #pragma omp barrier
  printf("I am %d\n",myid);
}
```

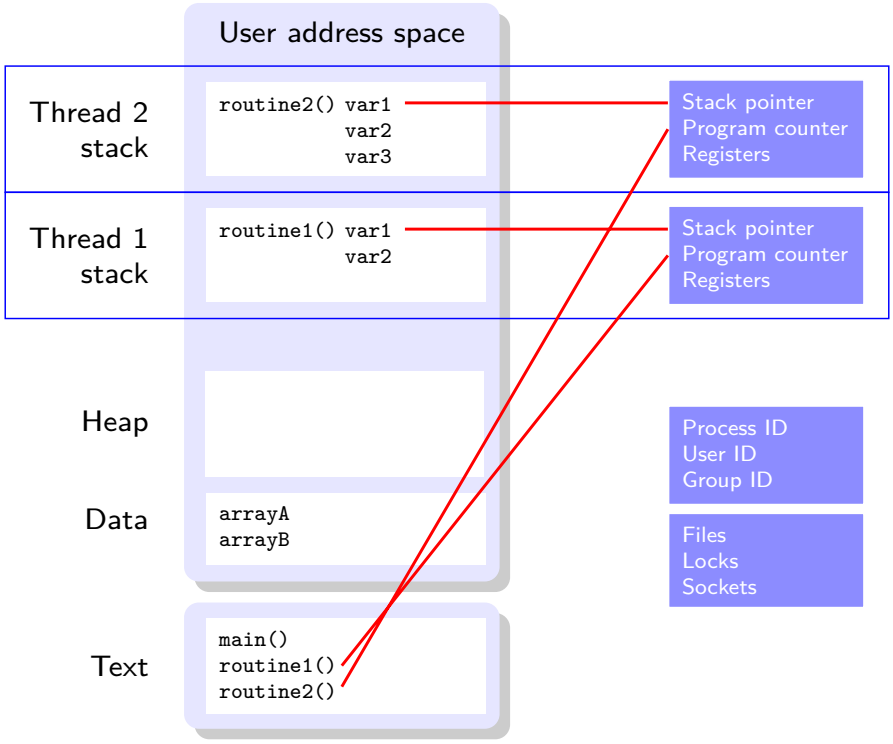
```
threads = 4
I am 2
I am 3
I am 0
I am 1
```

Funciona otra vez. Por qué?

Variables declaradas dentro de la región paralela son privadas: todos los hilos crean su propia copia de la variable (distinta posición de memoria)

Modelo de Ejecución - Memoria

Cada hilo tiene su propio contexto de ejecución



Atributos de Compartición de Datos

Hay cláusulas para declarar si una variable debe ser compartida o privada

```
int nthr, myid;
#pragma omp parallel private(myid)
{
  nthr = omp_get_num_threads();
  myid = omp_get_thread_num();
  if (myid==0)
    printf("threads = %d\n",nthr);
  #pragma omp barrier
  printf("I am %d\n",myid);
}
```

```
threads = 4
I am 2
I am 3
I am 0
I am 1
```

- `shared`, `private` para modificar el uso por defecto
- Cláusulas especiales: `reduction`, `firstprivate`, `lastprivate`

Apartado 3

Paralelismo de Bucles

- Parallelizar un Bucle Manualmente
- Construcción de Bucle

Tipos de Paralelismo

Los esquemas de paralelización más comunes son:

Paralelismo de tareas

- Los hilos ejecutan diferentes fragmentos de código (*tareas*) que han sido definidas por el programador

Paralelismo de bucles

- Los hilos ejecutan diferentes iteraciones de un bucle $\longrightarrow$ las iteraciones del bucle se *distribuyen*
- Cada iteración constituye una tarea
- También llamado paralelismo de datos porque los bucles suelen usarse para recorrer estructuras de datos de gran dimensión (una matriz, una malla)

En todos los casos, es importante analizar las dependencias de datos (sección siguiente)

Distribución por Bloques de las Iteraciones

Suponemos que queremos paralelizar un bucle sencillo

```
for (i=0; i<N; i++) {
  a[i] = a[i] + b[i];
}
```

Estrategia general: hacer que `i` sea una variable privada con valores disjuntos para cada hilo

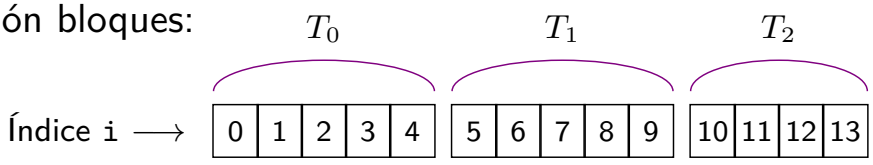
Distribución por bloques

```
#pragma omp parallel private(myid, i, nloc, istart, iend)
{
  nthr = omp_get_num_threads();
  myid = omp_get_thread_num();
  nloc = (N+nthr-1)/nthr;
  istart = myid*nloc;
  iend = (myid+1)*nloc;
  if (myid == nthr-1) iend = N;
  for (i=istart; i<iend; i++) {
    a[i] = a[i] + b[i];
  }
}
```

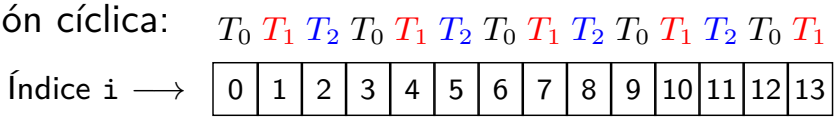
cada hilo ejecuta un rango contiguo de iteraciones

Distribución Cíclica de las Iteraciones

Distribución bloques:



Distribución cíclica:



Distribución cíclica

```
#pragma omp parallel private(myid, i)
{
  nthr = omp_get_num_threads();
  myid = omp_get_thread_num();
  for (i=myid; i<N; i+=nthr) {
    a[i] = a[i] + b[i];
  }
}
```

cada hilo empieza en un índice diferente e incrementa en una cantidad fija igual al número de hilos

Construcción de Bucle

La directiva `for` distribuye automáticamente las iteraciones del bucle que va a continuación y hace privada la variable del bucle

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<N; i++) {
    a[i] = a[i] + b[i];
  }
}
```

Atajo: directivas `parallel` y `for` en la misma línea

```
#pragma omp parallel for
for (i=0; i<N; i++) {
  a[i] = a[i] + b[i];
}
```

OpenMP impone restricciones al bucle

- El número de iteraciones debe ser conocido

```
for (i=0; i<n && !found; i++)
  if (x[i]==elem) found=1;
```

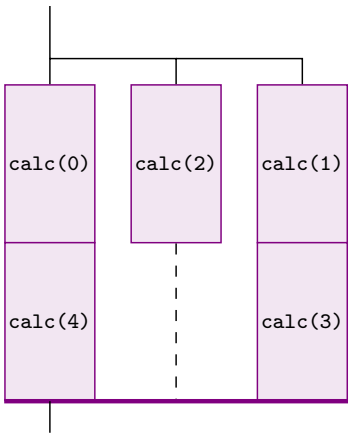
- El cuerpo del bucle no puede tener `break` o `goto`

Ejemplo de `parallel for`

Veamos una posible ejecución con 3 hilos

Bucle sencillo

```
#pragma omp parallel for
for (i=0; i<5; i++) {
  a[i] = calc(i);
}
```



Barrera implícita al finalizar la construcción `parallel for`

Variables:

- `a`: acceso concurrente, pero no hay más de un hilo accediendo a la misma posición
- `i`: distinto valor en cada hilo → necesitan una copia privada

Construcción de Bucle - Cláusula `nowait`

Cuando hay varios bucles independientes dentro de una región paralela, `nowait` evita la barrera implícita

Bucles sin barrera

```
void a8(int n, int m, float *a, float *b, float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;

        #pragma omp for
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```


Planificación

Idealmente, todas las iteraciones cuestan lo mismo y a cada hilo se le asigna aproximadamente el mismo número de iteraciones

En la realidad, se puede producir desequilibrio de la carga con la consiguiente pérdida de prestaciones

---

En OpenMP es posible especificar la planificación

Puede ser de dos tipos:

- Estática: las iteraciones se asignan a hilos a priori
- Dinámica: la asignación se adapta a la ejecución actual

La planificación se realiza a nivel de rangos contiguos de iteraciones (*chunks*)

Planificación - Cláusula `schedule`

Sintaxis de la cláusula de planificación:

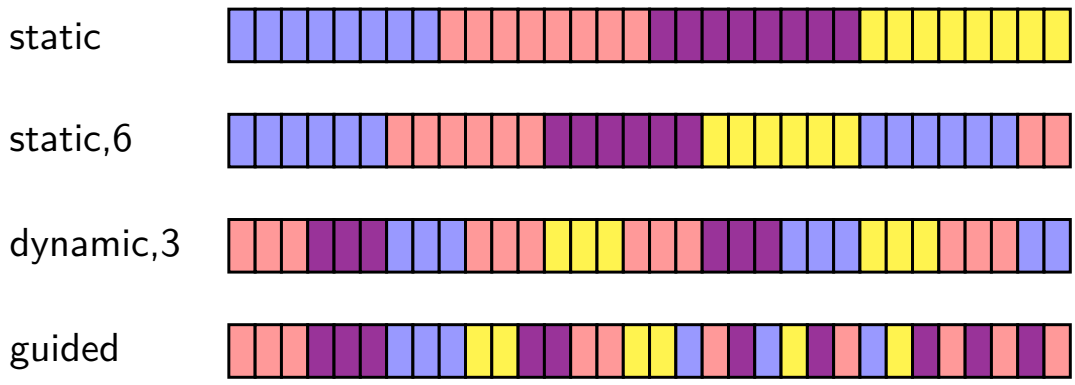
```
schedule(type[,chunk])
```

- `static` (sin `chunk`): a cada hilo se le asigna estáticamente un rango aproximadamente igual
- `static` (con `chunk`): asignación cíclica (*round-robin*) de rangos de tamaño `chunk`
- `dynamic` (`chunk` opcional, por defecto 1): se van asignando según se piden (*first-come, first-served*)
- `guided` (`chunk` mínimo opcional): como `dynamic` pero el tamaño del rango va decreciendo exponencialmente ($\propto n_{rest}/n_{hilos}$)
- `runtime`: se especifica en tiempo de ejecución con la variable de entorno `OMP_SCHEDULE`

Planificación - Ejemplo

Ejemplo: bucle de 32 iteraciones ejecutado con 4 hilos

```
$ OMP_NUM_THREADS=4 OMP_SCHEDULE=guided ./prog
```



Trabajando con Bucles

Enfoque básico

- Encontrar bucles intensivos computacionalmente
- Hacer que las iteraciones del bucle sean independientes
  - Para que puedan ser ejecutadas en cualquier orden de forma segura sin dependencias con iteraciones previas
- Colocar la directiva OpenMP apropiada y probar
  - Poner especial atención a las variables `private`

```
int i, j;
double A[N];

j = 5;
for (i=0; i<N; i++) {
  j += 2;
  A[i] = big(j);
}
```

Eliminar dependencia asociada al bucle

```
int i;
double A[N];

#pragma omp parallel for
for (i=0; i<N; i++) {
  int j = 5 + 2*(i+1);
  A[i] = big(j);
}
```


Bucles Anidados

Hay que poner la directiva antes del bucle a paralelizar

Caso 1

```
#pragma omp parallel for \
        private(j)
for (i=0; i<n; i++) {
    for (j=0; j<m; j++) {
        // cuerpo del bucle
    }
}
```

Caso 2

```
for (i=0; i<n; i++) {
    #pragma omp parallel for
    for (j=0; j<m; j++) {
        // cuerpo del bucle
    }
}
```

- En el primer caso, las iteraciones de `i` se reparten; el mismo hilo ejecuta el bucle `j` completo
- En el segundo caso, en cada iteración de `i` se activan y desactivan los hilos; hay `n` sincronizaciones

Bucles con Acumulador

shared

```
sum = 0;
#pragma omp parallel for shared(sum)
for (i=0; i<n; i++) {
    sum = sum + x[i]*x[i];
}
```

Incorrecto: condición de carrera al leer/escribir `sum`

private

```
sum = 0;
#pragma omp parallel for private(sum)
for (i=0; i<n; i++) {
    sum = sum + x[i]*x[i];
}
```

Incorrecto: tras el bucle, solo la suma del hilo principal queda disponible; además, las copias de los otros hilos no se inicializan

`reduction`

Para realizar reducciones con operadores conmutativos y asociativos (`+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`, `max`, `min`)

`reduction(redn_oper: var_list)`

```
suma = 0;
#pragma omp parallel for reduction(+:suma)
for (i=0; i<n; i++) {
    suma = suma + x[i]*x[i];
}
```

Cada hilo realiza una porción de la suma, al final se combinan en la suma total

Es como una variable privada, pero:

- Al final, los valores privados se combinan
- Se inicializa correctamente (al elemento neutro de la operación)

Medida de Tiempos

```
double omp_get_wtime()
```

Método portable para medir el tiempo "de pared"

- Tiempo transcurrido (en segundos) desde un cierto momento pasado

Ejemplo – medición de tiempos

```
#include <omp.h>

double t1, t2;
t1 = omp_get_wtime();
/*
    ...
*/
t2 = omp_get_wtime();
printf("Tiempo transcurrido: %f s.\n", t2-t1);
```

Incluir solo la parte del algoritmo que se quiere evaluar

Apartado 4

Paralelismo de Tareas

- Dependencias de Datos
- Construcción de Secciones

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

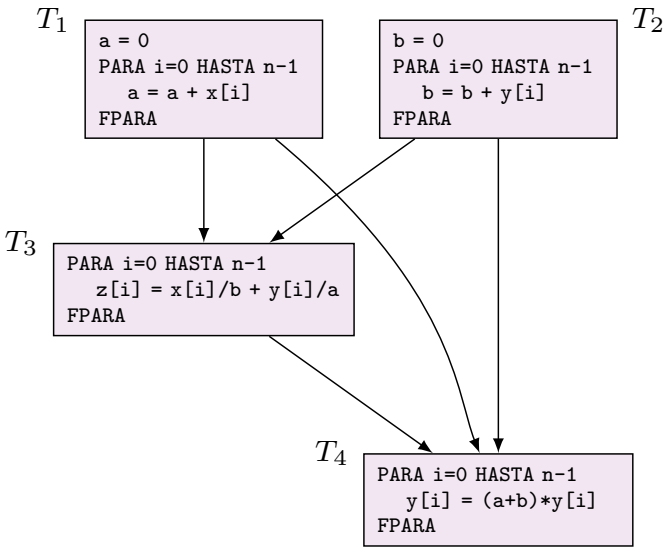
Dependencia de salida

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

Ambas tareas escriben en el array x

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: $T_1 \rightarrow T_3$, $T_2 \rightarrow T_3$, $T_1 \rightarrow T_4$, $T_2 \rightarrow T_4$
Anti-dependencias: $T_2 \rightarrow T_4$, $T_3 \rightarrow T_4$

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
- Las aristas representan las dependencias entre tareas

El grafo es una representación del algoritmo paralelo

- Un grafo ancho y corto implica gran potencial de paralelismo
- Un grafo estrecho y alto significa que hay demasiadas dependencias que secuencializan la ejecución

Una buena descomposición de tareas intenta maximizar el grado de concurrencia

Grafo de Dependencias de Tareas: Coste

El grafo se suele anotar con información de coste

- Las tareas tienen asociado un coste computacional c_i
  - Normalmente dado en *flops: floating-point operations*
  - En función del tamaño del problema, p.e., $2n^2$ flops
- Las aristas pueden tener coste de comunicación

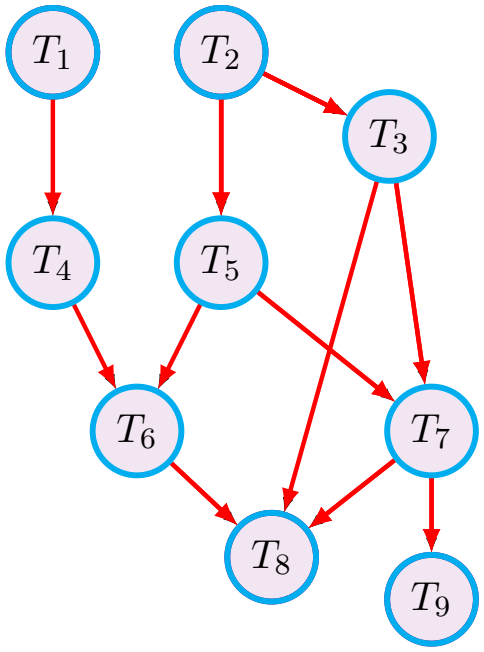
Definiciones:

- Longitud de un camino: suma de costes de nodos visitados
  $\longrightarrow$ Camino crítico es el de mayor longitud
- Grado medio de concurrencia: número de tareas que se ejecutan de media concurrentemente
  $\longrightarrow \sum_{i=1}^{N} \frac{c_i}{L}$ (N = total nodos, L = longitud camino crítico)

Si hay suficientes procesadores disponibles, el tiempo para completar el algoritmo paralelo es igual a L (más la comunicación)

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: $M = \sum_{i=1}^{9} \frac{1}{4} = 2{,}25$

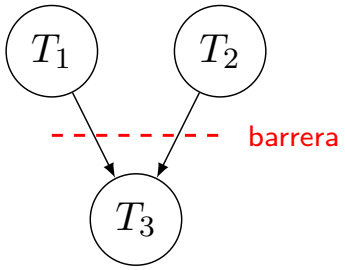
Paralelismo de Tareas con Identificador de Hilo

Posible enfoque para implementar el paralelismo de tareas:

- Los hilos realizan diferentes tareas según el id. de hilo
- Las dependencias se garantizan con barreras explícitas

Paralelismo de tareas con id. hilo

```
#pragma omp parallel private(myid)
{
  myid = omp_get_thread_num();
  if (myid==0) {
    /* Task 1 */
  } else if (myid==1) {
    /* Task 2 */
  }
  #pragma omp barrier
  if (myid==0) {
    /* Task 3 */
  }
}
```



El código debería comprobar que hay suficientes hilos

Utilidades en OpenMP para Paralismo de Tareas

La solución anterior es bastante primitiva

- El programador es responsable de asignar las tareas
- Código oscuro y complicado en programas grandes

OpenMP ofrece construcciones de reparto de trabajo:

- Construcción `for` para distribuir iteraciones de bucles
- Construcción `sections` para distribuir trozos de código
- Construcción `single` para código a ejecutar en un solo hilo

En todos los casos:

- Barrera implícita al final del bloque
- Todos los hilos deben entrar en la construcción

Una alternativa aún mejor: directiva `task` (no cubierta aquí)

Construcción `sections`

Para trozos de código independientes difíciles de paralelizar

- Individualmente suponen muy poco trabajo, o bien
- Cada fragmento es inherentemente secuencial

Puede también combinarse con `parallel`

Ejemplo de secciones

```
#pragma omp parallel sections
{
    #pragma omp section
      Xaxis();
    #pragma omp section
      Yaxis();
    #pragma omp section
      Zaxis();
}
```

Un hilo puede ejecutar más de una sección

Cláusulas: `private`, `first/lastprivate`, `reduction`, `nowait`

Construcción `single`

Fragmentos de código que deben ejecutarse por un solo hilo

Ejemplo `single`

```
#pragma omp parallel
{
    #pragma omp single nowait
      printf("Empieza work1\n");
    work1();

    #pragma omp single
      printf("Finalizando work1\n");

    #pragma omp single nowait
      printf("Terminado work1, empieza work2\n");
    work2();
}
```

Algunas cláusulas permitidas: `private`, `firstprivate`, `nowait`

Apartado 5

Sincronización

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

Condición de Carrera

El siguiente ejemplo ilustra una condición de carrera en OpenMP

Encontrar el valor máximo

```
cur_max = -100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] > cur_max) {
        cur_max = a[i];
    }
}
```

Secuencia con resultado incorrecto:

Hilo 0: lee `a[i]`=20, lee `cur_max`=15
Hilo 1: lee `a[i]`=16, lee `cur_max`=15
Hilo 0: comprueba `a[i]>cur_max`, escribe `cur_max`=20
Hilo 1: comprueba `a[i]>cur_max`, escribe `cur_max`=16

Exclusión Mutua

¿Cómo evitar 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)

Exclusión mutua mediante secciones críticas

- Fragmentos de código con más de una sentencia
- No permitir que haya más de un hilo ejecutándola
- Requiere mecanismos de sincronización: semáforos, etc.

Funcionalidad OpenMP

- Directivas `critical` y `atomic`
- Sincronización avanzada con funciones `*_lock`

Directiva `critical` (1)

En el ejemplo anterior, el acceso en exclusión mutua a la variable `cur_max` evita la condición de carrera

Búsqueda de máximo, sin condición de carrera

```
cur_max = -100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    #pragma omp critical
    if (a[i] > cur_max) {
        cur_max = a[i];
    }
}
```

Cuando un hilo llega al bloque `if` (la sección crítica), espera hasta que no hay otro hilo ejecutándolo al mismo tiempo

OpenMP garantiza progreso (al menos un hilo de los que espera entra en la sección crítica) pero no espera limitada

Directiva `critical` (2)

En la práctica, el ejemplo anterior resulta ser secuencial

Teniendo en cuenta que `cur_max` nunca se decrementa, se puede plantear la siguiente mejora

Búsqueda de máximo, mejorado

```
cur_max = -100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] > cur_max) {
        #pragma omp critical
        if (a[i] > cur_max)
            cur_max = a[i];
    }
}
```

El segundo `if` es necesario porque se ha leído `cur_max` fuera de la sección crítica

Esta solución entra en la sección crítica con menor frecuencia

Directiva `critical` con Nombre

Al añadir un nombre, se permite tener varias secciones críticas sin relación entre ellas

Búsqueda de máximo y mínimo

```
cur_max = -100000;
cur_min = 100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] > cur_max) {
        #pragma omp critical (maximo)
        if (a[i] > cur_max)
            cur_max = a[i];
    }
    if (a[i] < cur_min) {
        #pragma omp critical (minimo)
        if (a[i] < cur_min)
            cur_min = a[i];
    }
}
```

Directiva atomic

Operaciones atómicas de lectura-modificación-escritura

```
#pragma omp atomic
x <binop>= expr
```

```
#pragma omp atomic
x++, ++x, x--, --x
```

donde <binop> puede ser +, *, -, /, %, &, |, ^, <<, >>

Ejemplo atomic

```
#pragma omp parallel for shared(x, index, n)
for (i=0; i<n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
}
```

El código es mucho más eficiente que con critical y permite actualizar elementos de x en paralelo