TPP – Tecnología de la Programación Paralela

Master Universitario en Computación en la Nube y de Altas Prestaciones

Programación con OpenMP

Pedro Alonso

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Contenido

Apartado 1

Conceptos Básicos

- Modelo de Programación
- Ejemplo Simple

La Especificación OpenMP

Estándar de facto para programación en memoria compartida

`http://www.openmp.org`

Especificaciones:

- Fortran: 1.0 (1997), 2.0 (2000)
- C/C++: 1.0 (1998), 2.0 (2002)
- Fortran/C/C++: 2.5 (2005), 3.0 (2008), 4.0 (2013), **4.5 (2015)**

Antecedentes:

- Estándar ANSI X3H5 (1994)
- HPF, CMFortran

Modelo de Programación

La programación en OpenMP se basa principalmente en directivas del compilador

Ejemplo

```
void daxpy(int n, double a, double *x, double *y, double *z)
{
    int i;
    #pragma omp parallel for
    for (i=0; i<n; i++)
        z[i] = a*x[i] + y[i];
}
```

Ventajas

- Facilita la migración (el compilador ignora los `#pragma`)
- Permite la paralelización incremental
- Permite la optimización por parte del compilador

Además: funciones (ver `omp.h`) y variables de entorno

Modelo de Programación

La programación en OpenMP se basa principalmente en directivas del compilador

Ejemplo

```
void daxpy(int n, double a, double *x, double *y, double *z)
{
    int i;
    #pragma omp parallel for
    for (i=0; i<n; i++)
        z[i] = a*x[i] + y[i];
}
```

Ventajas

- Facilita la migración (el compilador ignora los `#pragma`)
- Permite la paralelización incremental
- Permite la optimización por parte del compilador

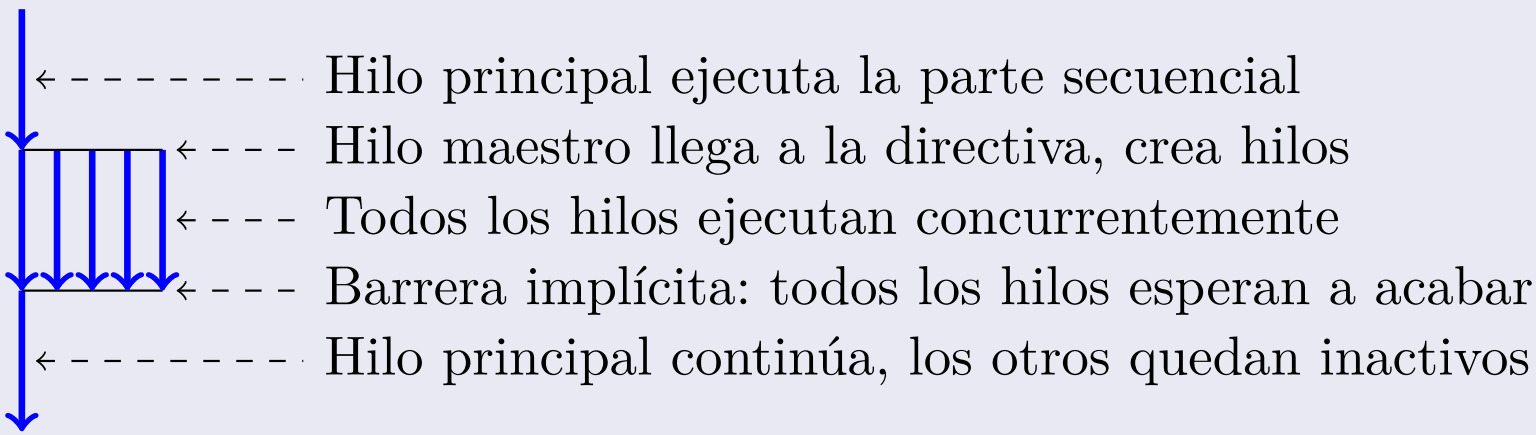
Además: funciones (ver `omp.h`) y variables de entorno

Modelo de Ejecución

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

Hay directivas para crear hilos y dividir el trabajo

Esquema



Las directivas definen regiones paralelas

Otras directivas/cláusulas:

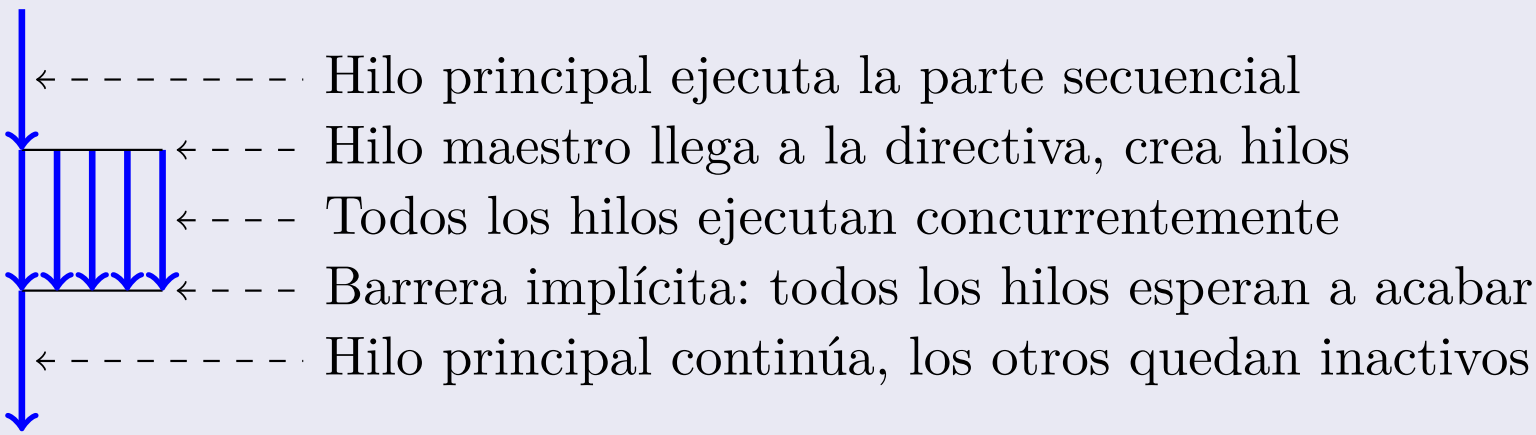
- Indicar tipo de variable: `private`, `shared`, `reduction`
- Sincronización: `critical`, `barrier`

Modelo de Ejecución

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

Hay directivas para crear hilos y dividir el trabajo

Esquema



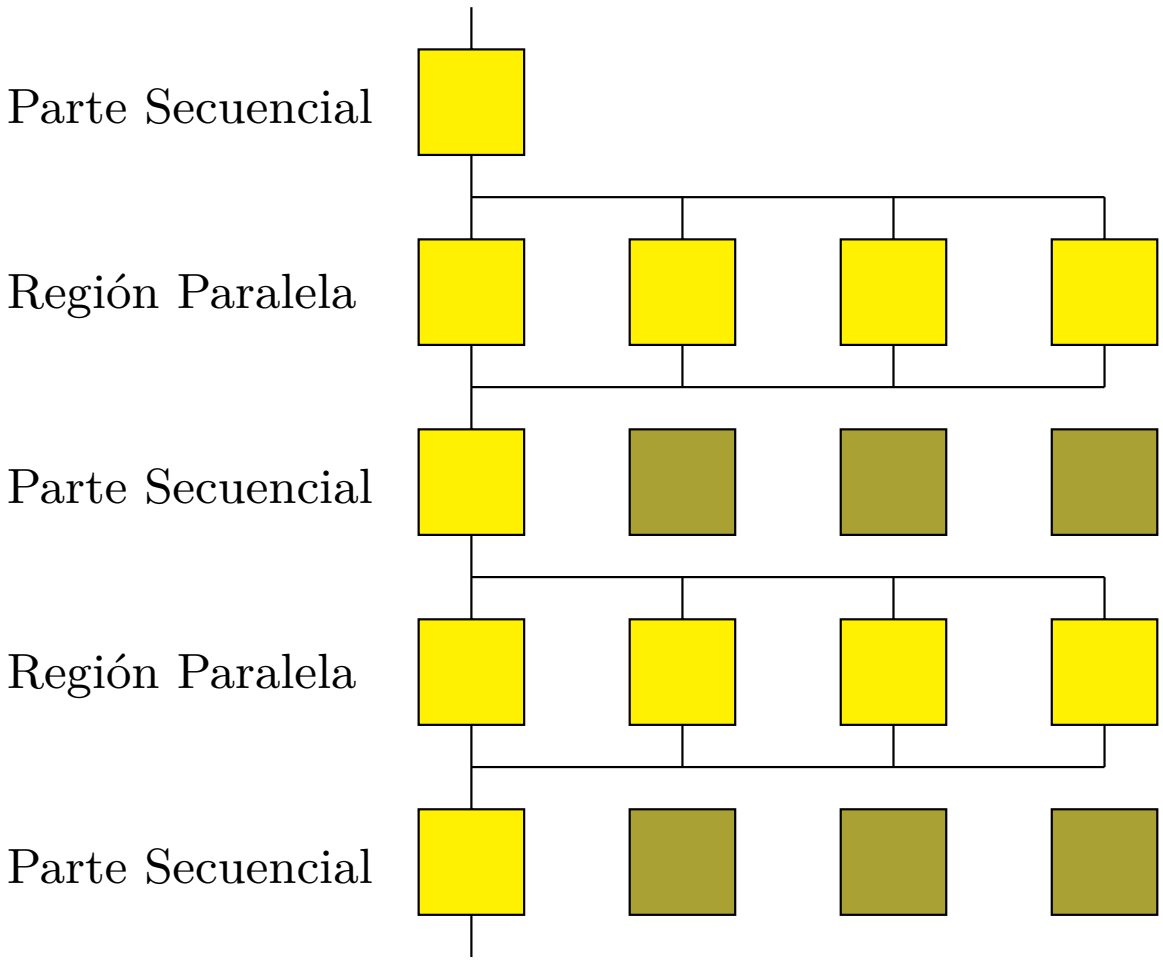
Las directivas definen regiones paralelas

Otras directivas/cláusulas:

- Indicar tipo de variable: `private`, `shared`, `reduction`
- Sincronización: `critical`, `barrier`

Modelo de Ejecución - Hilos

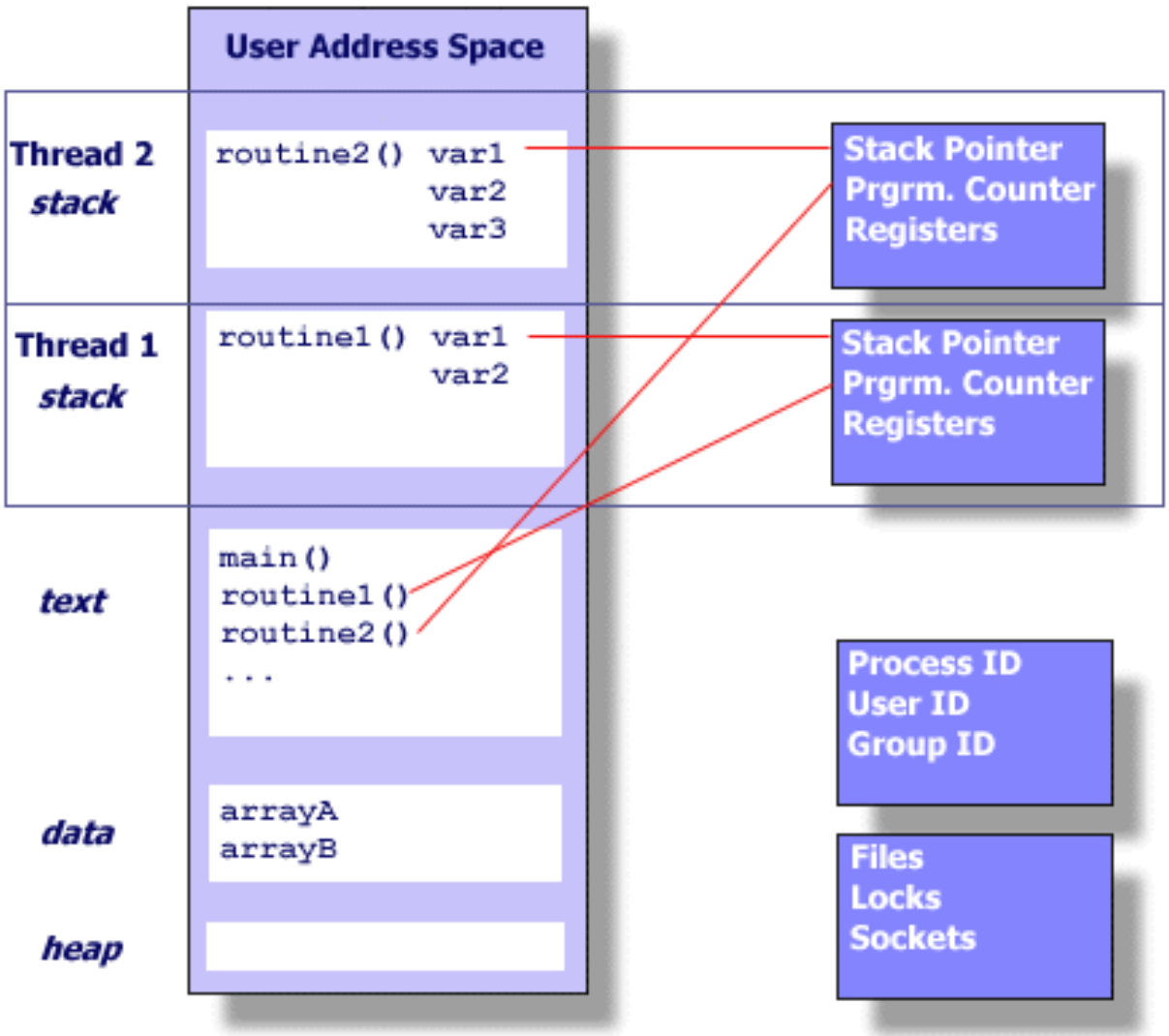
En OpenMP los hilos inactivos no se destruyen, quedan a la espera de la siguiente región paralela



Los hilos creados por una directiva se llaman equipo (*team*)

Modelo de Ejecución - Memoria

Cada hilo tiene su propio contexto de ejecución (incluyendo la pila)



Sintaxis

Directivas:

```
#pragma omp <directiva> [clausula [...]]
```

Uso de funciones:

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

Compilación condicional: la macro `_OPENMP` contiene la fecha de la versión de OpenMP soportada, p.e. 201107

Compilación:

```
gcc-4.2> gcc -fopenmp prg-omp.c
sun> cc -xopenmp -xO3 prg-omp.c
intel> icc -openmp prg-omp.c
```

Ejemplo Simple

Ejemplo

```
void daxpy(int n, double a, double *x, double *y, double *z)
{
    int i;
    #pragma omp parallel for
    for (i=0; i<n; i++)
        z[i] = a*x[i] + y[i];
}
```

- Al llegar a la directiva `parallel` se crean los hilos (si no se han creado antes)
- Las iteraciones del bucle se reparten entre los hilos
- Por defecto, todas las variables son compartidas, excepto la variable del bucle (`i`) que es privada
- Al finalizar se sincronizan todos los hilos

Ejemplo Simple

Ejemplo

```
void daxpy(int n, double a, double *x, double *y, double *z)
{
    int i;
    #pragma omp parallel for
    for (i=0; i<n; i++)
        z[i] = a*x[i] + y[i];
}
```

- Al llegar a la directiva `parallel` se crean los hilos (si no se han creado antes)
- Las iteraciones del bucle se reparten entre los hilos
- Por defecto, todas las variables son compartidas, excepto la variable del bucle (`i`) que es privada
- Al finalizar se sincronizan todos los hilos

Número e Identificador de Hilo

El número de hilos se puede especificar:

- Con la cláusula `num_threads`
- Con la función `omp_set_num_threads()` *antes* de la región paralela
- Al ejecutar, con `OMP_NUM_THREADS`

Funciones útiles:

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

```
omp_set_num_threads(3);
printf("hilos antes = %d\n",omp_get_num_threads());
#pragma omp parallel for
for (i=0; i<n; i++) {
  printf("hilos = %d\n",omp_get_num_threads());
  printf("yo soy %d\n",omp_get_thread_num());
}
```

Número e Identificador de Hilo

El número de hilos se puede especificar:

- Con la cláusula `num_threads`
- Con la función `omp_set_num_threads()` *antes* de la región paralela
- Al ejecutar, con `OMP_NUM_THREADS`

Funciones útiles:

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

```
omp_set_num_threads(3);
printf("hilos antes = %d\n",omp_get_num_threads());
#pragma omp parallel for
for (i=0; i<n; i++) {
  printf("hilos = %d\n",omp_get_num_threads());
  printf("yo soy %d\n",omp_get_thread_num());
}
```

Apartado 2

Paralelización de Bucles

- Directiva `parallel for`
- Tipos de Variables
- Mejora de Prestaciones

Directiva `parallel for`

Se paraleliza el bucle que va a continuación

C/C++

```
#pragma omp parallel for [clausula [...]]
for (index=first; test_expr; increment_expr) {
    // cuerpo del bucle
}
```

OpenMP impone restricciones al tipo de bucle

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 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

Tipos de Variables

Se clasifican las variables según su alcance (*scope*)

- Privadas: cada hilo tiene una réplica distinta
- Compartidas: todos los hilos pueden leer y escribir

Fuente común de errores: no elegir correctamente el alcance

El alcance se puede modificar con cláusulas añadidas a las directivas:

- `private`, `shared`
- `reduction`
- `firstprivate`, `lastprivate`

Tipos de Variables

Se clasifican las variables según su alcance (*scope*)

- Privadas: cada hilo tiene una réplica distinta
- Compartidas: todos los hilos pueden leer y escribir

Fuente común de errores: no elegir correctamente el alcance

El alcance se puede modificar con cláusulas añadidas a las directivas:

- `private`, `shared`
- `reduction`
- `firstprivate`, `lastprivate`

private, shared

private

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

Incorrecto: tras el bucle sólo existe la suma del hilo principal (con valor 0) - además, las copias de cada hilo no se inicializan

shared

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

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

private, shared

private

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

Incorrecto: tras el bucle sólo existe la suma del hilo principal (con valor 0) - además, las copias de cada hilo no se inicializan

shared

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

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

`private`, `shared`, `default`

Si no se especifica el alcance de una variable, por defecto es `shared`

Excepciones (`private`):

- Índice del bucle que se paraleliza
- En subrutinas invocadas, las variables locales (excepto si se declaran `static`)
- Variables automáticas declaradas dentro del bucle

Cláusula `default`

- `default(none)` obliga a especificar el alcance de todas
- En Fortran, `default(private)` cambia el alcance por defecto

reduction

Para realizar reducciones con operadores conmutativos y asociativos (+, *, &, ...)

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 se inicializa correctamente

`firstprivate`, `lastprivate`

Las variables privadas se crean sin un valor inicial y tras el bloque `parallel` quedan indefinidas

- `firstprivate`: inicializa al valor del hilo principal
- `lastprivate`: se queda con el valor de la "última" iteración

Ejemplo

```
alpha = 5.0;
#pragma omp parallel for firstprivate(alpha) lastprivate(i)
for (i=0; i<n; i++) {
    z[i] = alpha*x[i];
}
k = i;    /* i tiene el valor n */
```

El comportamiento por defecto intenta evitar copias innecesarias

`firstprivate`, `lastprivate`

Las variables privadas se crean sin un valor inicial y tras el bloque `parallel` quedan indefinidas

- `firstprivate`: inicializa al valor del hilo principal
- `lastprivate`: se queda con el valor de la "última" iteración

Ejemplo

```
alpha = 5.0;
#pragma omp parallel for firstprivate(alpha) lastprivate(i)
for (i=0; i<n; i++) {
    z[i] = alpha*x[i];
}
k = i;    /* i tiene el valor n */
```

El comportamiento por defecto intenta evitar copias innecesarias

Garantizar Suficiente Trabajo (1)

La paralelización de bucles supone un *overhead*: activación y desactivación de hilos, sincronización

En bucles muy sencillos, el overhead puede ser mayor que el tiempo de cálculo

Cláusula `if`

```
#pragma omp parallel for if(n>800)
for (i=0; i<n; i++)
    z[i] = a*x[i] + y[i];
```

Si la expresión es falsa, el bucle se ejecuta secuencialmente

Esta cláusula se podría usar también para evitar dependencias de datos detectadas en tiempo de ejecución

Garantizar Suficiente Trabajo (1)

La paralelización de bucles supone un *overhead*: activación y desactivación de hilos, sincronización

En bucles muy sencillos, el overhead puede ser mayor que el tiempo de cálculo

Cláusula `if`

```
#pragma omp parallel for if(n>800)
for (i=0; i<n; i++)
    z[i] = a*x[i] + y[i];
```

Si la expresión es falsa, el bucle se ejecuta secuencialmente

Esta cláusula se podría usar también para evitar dependencias de datos detectadas en tiempo de ejecución

Garantizar Suficiente Trabajo (2)

En bucles anidados se recomienda paralelizar el más externo

- En casos en que las dependencias de datos impiden esto, se puede intentar intercambiar los bucles

Código secuencial

```
for (j=1; j<n; j++)
    for (i=0; i<n; i++)
        a[i][j] = a[i][j] + a[i][j-1];
```

Código paralelo con bucles intercambiados

```
#pragma omp parallel for private(j)
for (i=0; i<n; i++)
    for (j=1; j<n; j++)
        a[i][j] = a[i][j] + a[i][j-1];
```

Estas modificaciones pueden tener impacto en el uso de cache

Garantizar Suficiente Trabajo (2)

En bucles anidados se recomienda paralelizar el más externo

- En casos en que las dependencias de datos impiden esto, se puede intentar intercambiar los bucles

Código secuencial

```
for (j=1; j<n; j++)
    for (i=0; i<n; i++)
        a[i][j] = a[i][j] + a[i][j-1];
```

Código paralelo con bucles intercambiados

```
#pragma omp parallel for private(j)
for (i=0; i<n; i++)
    for (j=1; j<n; j++)
        a[i][j] = a[i][j] + a[i][j-1];
```

Estas modificaciones pueden tener impacto en el uso de cache

Planificación (1)

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 (1)

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 (2)

Sintaxis de la cláusula de planificación:

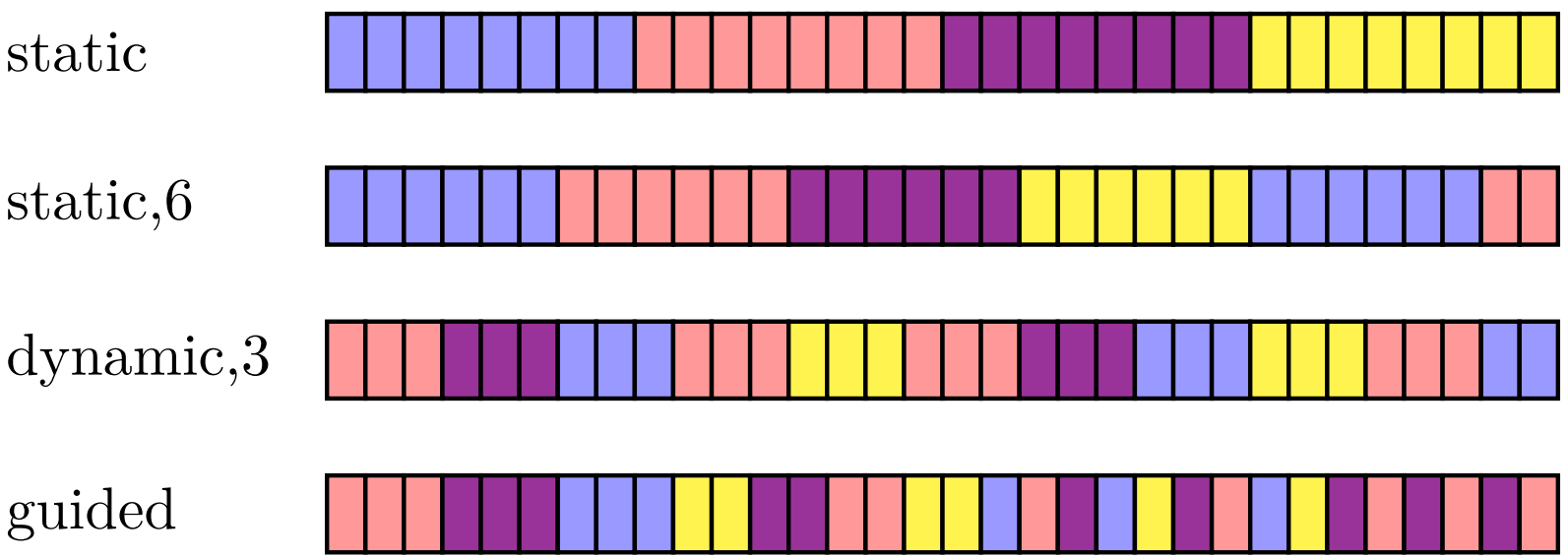
```
schedule(type[,chunk])
```

- `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 (3)

Ejemplo: bucle de 32 iteraciones ejecutado con 4 hilos

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



Apartado 3

Regiones Paralelas

- Directiva `parallel`
- Reparto del Trabajo

Regiones Paralelas

La paralelización a nivel de bucles tiene inconvenientes:

- No permite paralelizar porciones grandes de código
- No permite bucles *while* o paralelismo no iterativo
- Puede dar *speedup* bajo

Regiones Paralelas

- No están restringidas a un bucle
- Se ajustan al modelo SPMD
- Admiten construcciones para repartir el trabajo

Regiones Paralelas

La paralelización a nivel de bucles tiene inconvenientes:

- No permite paralelizar porciones grandes de código
- No permite bucles *while* o paralelismo no iterativo
- Puede dar *speedup* bajo

Regiones Paralelas

- No están restringidas a un bucle
- Se ajustan al modelo SPMD
- Admiten construcciones para repartir el trabajo

Directiva `parallel`

Se paraleliza el bloque que va a continuación

C/C++

```
#pragma omp parallel [clause [clause ...]]
{
    // bloque
}
```

Las cláusulas permitidas son: `private`, `shared`, `default`, `reduction`, `if`, `copyin`

OpenMP impone restricciones al bloque (p.e. sin saltos)

Funcionamiento de la Directiva `parallel`

Se activan los hilos y se replica el código del bloque

Ejecución replicada

```
#pragma omp parallel private(i)
for (i=0; i<10; i++) {
    printf("Iteración %d\n", i);
}
```

Con 2 hilos, se imprimirían 20 líneas

Reparto de iteraciones

```
#pragma omp parallel for
for (i=0; i<10; i++) {
    printf("Iteración %d\n", i);
}
```

Se imprimen 10 líneas, repartiéndose entre los hilos

Funcionamiento de la Directiva `parallel`

Se activan los hilos y se replica el código del bloque

Ejecución replicada

```
#pragma omp parallel private(i)
for (i=0; i<10; i++) {
    printf("Iteración %d\n", i);
}
```

Con 2 hilos, se imprimirían 20 líneas

Reparto de iteraciones

```
#pragma omp parallel for
for (i=0; i<10; i++) {
    printf("Iteración %d\n", i);
}
```

Se imprimen 10 líneas, repartiéndose entre los hilos

Tipos de Variables / Alcance

Cláusulas permitidas: `private`, `shared`, `default`, `reduction`

- Se aplican al *alcance estático* (interior de la directiva), pero no al *alcance dinámico* (funciones invocadas)

Problema con variables globales

```
int initialized = 0;  /* variable global */
...
#pragma omp parallel
{
    procesa_array();
}
...
void procesa_array() {
    if (!initialized) { initialized=1; /* inicializa */ }
    /* procesamiento normal */
}
```

No es correcto, ni tampoco con `private(initialized)`

Tipos de Variables / Alcance

Cláusulas permitidas: `private`, `shared`, `default`, `reduction`

- Se aplican al *alcance estático* (interior de la directiva), pero no al *alcance dinámico* (funciones invocadas)

Problema con variables globales

```
int initialized = 0;  /* variable global */
...
#pragma omp parallel
{
    procesa_array();
}
...
void procesa_array() {
    if (!initialized) { initialized=1; /* inicializa */ }
    /* procesamiento normal */
}
```

No es correcto, ni tampoco con `private(initialized)`

Variables `threadprivate`

Declara variables globales privadas a nivel de hilo

Problema resuelto

```
int initialized = 0;  /* variable global */
#pragma omp threadprivate(initialized)
...
#pragma omp parallel
{
    procesa_array();
}
...
void procesa_array() {
    if (!initialized) { initialized=1; /* inicializa */ }
    /* procesamiento normal */
}
```

Estas variables se mantienen de una región paralela a otra - los hilos no se destruyen (a no ser que cambie el número de hilos)

Cláusula `copyin`

En el ejemplo anterior, las copias privadas de la variable `initialized` se inicializan según la declaración (o el constructor en C++)

La cláusula `copyin` permite pasar el valor que tiene el hilo principal a todas las copias privadas

Ejemplo con `copyin`

```
int n;   /* variable global */
#pragma omp threadprivate(n)
...
fread(&n, ...);   /* el hilo principal lee de fichero */
...
#pragma omp parallel copyin(n)
{
    procesa_array(n,...);
}
```

Cláusula `copyin`

En el ejemplo anterior, las copias privadas de la variable `initialized` se inicializan según la declaración (o el constructor en C++)

La cláusula `copyin` permite pasar el valor que tiene el hilo principal a todas las copias privadas

Ejemplo con `copyin`

```
int n;   /* variable global */
#pragma omp threadprivate(n)
...
fread(&n, ...);   /* el hilo principal lee de fichero */
...
#pragma omp parallel copyin(n)
{
    procesa_array(n,...);
}
```

Reparto del Trabajo

Además de la ejecución replicada, suele ser necesario repartir el trabajo entre los hilos

- Cada hilo opera sobre una parte de una estructura de datos, o bien
- Cada hilo realiza una operación distinta

Posibles formas de realizar el reparto:

- Cola de tareas paralelas
- Según el identificador de hilo
- Mediante construcciones OpenMP específicas

Reparto del Trabajo

Además de la ejecución replicada, suele ser necesario repartir el trabajo entre los hilos

- Cada hilo opera sobre una parte de una estructura de datos, o bien
- Cada hilo realiza una operación distinta

Posibles formas de realizar el reparto:

- Cola de tareas paralelas
- Según el identificador de hilo
- Mediante construcciones OpenMP específicas

Reparto mediante Cola de Tareas Paralelas

Una cola de tareas paralelas es una estructura de datos compartida que contiene una lista de "tareas" a realizar

- Las tareas se pueden procesar concurrentemente
- Cualquier tarea puede realizarse por cualquier hilo

```
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();
        }
    }
```

Reparto mediante Cola de Tareas Paralelas

Una cola de tareas paralelas es una estructura de datos compartida que contiene una lista de "tareas" a realizar

- Las tareas se pueden procesar concurrentemente
- Cualquier tarea puede realizarse por cualquier hilo

```
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();
        }
    }
```

Reparto según Identificador de Hilo

Se utilizan las funciones ya comentadas

- `omp_get_num_threads()`: devuelve el número de hilos
- `omp_get_thread_num()`: devuelve el identificador de hilo

para determinar qué parte del trabajo realiza cada hilo

Ejemplo - identificadores de hilo

```
#pragma omp parallel private(myid)
{
    nthreads = omp_get_num_threads();
    myid = omp_get_thread_num();
    dowork(myid, nthreads);
}
```

Construcciones de Reparto del Trabajo

Las soluciones anteriores son bastante primitivas

- El programador se encarga de dividir el trabajo
- Código oscuro y complicado en programas largos

OpenMP dispone de construcciones específicas (*work-sharing constructs*)

Hay de tres tipos:

- Construcción de bucle (`for`/`do`) para repartir iteraciones
- Secciones para distinguir porciones del código
- Código a ejecutar por un solo hilo

Construcciones de Reparto del Trabajo

Las soluciones anteriores son bastante primitivas

- El programador se encarga de dividir el trabajo
- Código oscuro y complicado en programas largos

OpenMP dispone de construcciones específicas (*work-sharing constructs*)

Hay de tres tipos:

- Construcción de bucle (`for`/`do`) para repartir iteraciones
- Secciones para distinguir porciones del código
- Código a ejecutar por un solo hilo

Construcción de Bucle (`for/do`)

Reparte de forma automática las iteraciones del bucle

Ejemplo de bucle compartido

```
#pragma omp parallel
{
    ...
    #pragma omp for
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

El bucle se *comparte* entre los hilos, en vez de hacerlo replicado

Las directivas `parallel` y `for` se pueden combinar en una

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]);
    }
}
```

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

Hay una barrera implícita al final del bloque

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();
}
```

Cláusulas permitidas: `private`, `firstprivate`, `nowait`

Construcción `single` y `copyprivate`

Para inicializar variables privadas a partir de valores de línea de comandos, por ejemplo

Similar al ejemplo de `copyin`

Ejemplo `single` con `copyprivate`

```
int Nsize, choice;
#pragma omp parallel private(Nsize,choice)
{
    #pragma omp single copyprivate(Nsize,choice)
    input_parameters(&Nsize,&choice);

    do_work(Nsize,choice);
}
```

Apartado 4

Sincronización

- Exclusión Mutua
- Sincronización por Eventos

Condición de Carrera (1)

El siguiente ejemplo ilustra una condición de carrera

Ejemplo de condición de carrera

```
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

Condición de Carrera (2)

Hay casos en que la condición de carrera no da problemas

Ejemplo de condición de carrera aceptable

```
encontrado = 0;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] == valor) {
        encontrado = 1;
    }
}
```

Aunque varios hilos escriban a la vez, el resultado es correcto

---

En general, se necesitan mecanismos de sincronización:

- Exclusión mutua
- Sincronización por eventos

Condición de Carrera (2)

Hay casos en que la condición de carrera no da problemas

Ejemplo de condición de carrera aceptable

```
encontrado = 0;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] == valor) {
        encontrado = 1;
    }
}
```

Aunque varios hilos escriban a la vez, el resultado es correcto

---

En general, se necesitan mecanismos de sincronización:

- Exclusión mutua
- Sincronización por eventos

Exclusión Mutua

La *exclusión mutua* en el acceso a las variables compartidas evita cualquier condición de carrera

OpenMP proporciona tres construcciones diferentes:

- Secciones críticas: directiva `critical`
- Operaciones atómicas: directiva `atomic`
- Cerrojos: rutinas `*_lock`

En esta lista, las construcciones están ordenadas por complejidad / flexibilidad creciente

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 y modificación

```
#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

Directiva `atomic`

Operaciones atómicas de lectura y modificación

```
#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

Cerrojos: rutinas `*_lock`

Funcionamiento muy similar a los *mutex* de POSIX

```
omp_lock_t lck;
int id;
omp_init_lock(&lck);
#pragma omp parallel shared(lck) private(id)
{
    id = omp_get_thread_num();
    omp_set_lock(&lck);
    /* solo un hilo a la vez puede ejecutar este printf */
    printf("Mi id de hilo es %d.\n", id);
    omp_unset_lock(&lck);
    while (! omp_test_lock(&lck)) {
        skip(id); /* aun no tenemos el cerrojo, por lo
                     que tenemos que hacer otra cosa */
    }
    work(id); /* ahora tenemos el cerrojo
                 y podemos hacer el trabajo */
    omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Sincronización por Eventos

Las construcciones de exclusión mutua proporcionan acceso exclusivo pero no imponen ningún orden en la ejecución de las secciones críticas

La sincronización por eventos permite ordenar la ejecución de los hilos

- Barreras: directiva `barrier`
- Secciones ordenadas: directiva `ordered`
- La directiva `master`

Directiva `barrier`

Al llegar a una barrera, los hilos esperan a que lleguen todos

```
#pragma omp parallel private(index)
{
    index = generate_next_index();
    while (index>0) {
        add_index(index);
        index = generate_next_index();
    }
    #pragma omp barrier
    index = get_next_index();
    while (index>0) {
        process_index(index);
        index = get_next_index();
    }
}
```

Se suele usar para asegurar que una fase la han terminado todos antes de pasar a la siguiente fase

Directiva `ordered`

Para permitir que una porción del código de las iteraciones se ejecute en el orden secuencial original

Ejemplo `ordered`

```
#pragma omp parallel for ordered
for (i=0; i<n; i++) {
    a[i] = ...   /* cálculo complejo */
    #pragma omp ordered
    fprintf(fd, "%d %g\n", i, a[i]);
}
```

Restricciones:

- Si un bucle paralelo contiene una directiva `ordered`, hay que añadir la cláusula `ordered` también al bucle
- Sólo se permite una única sección `ordered` por iteración

Directiva `master`

Identifica un bloque de código dentro de una región paralela que se ha de ejecutar solo por el hilo principal (*master*)

Ejemplo `master`

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    calc1();     /* realizar cálculos */
    #pragma omp master
        printf("Resultados intermedios: ...\n", ...);
    calc2();     /* realizar más cálculos */
}
```

Diferencias con `single`:

- No se requiere que todos los hilos alcancen esta construcción
- No hay barrera implícita (los otros hilos simplemente se saltan ese código)

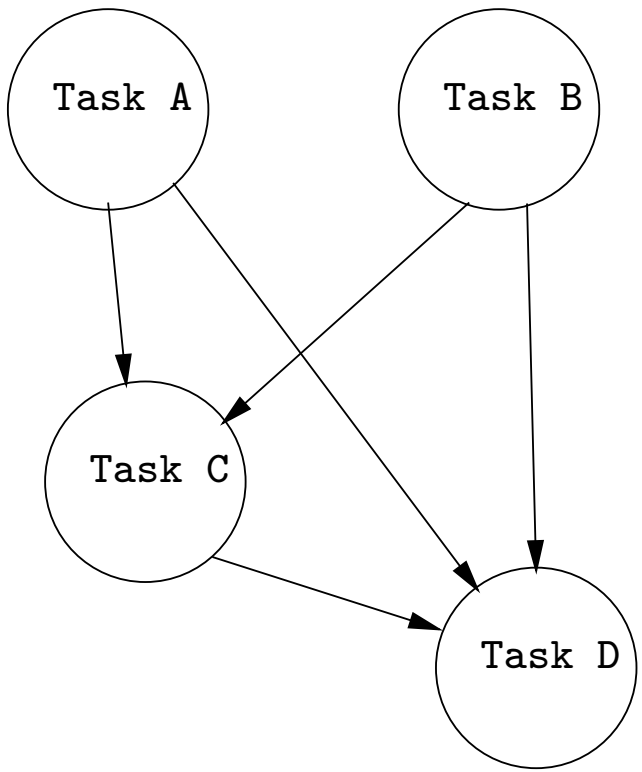
Apartado 5

Paralelismo de Tareas

Paralelización de Algoritmos

La paralelización de un algoritmo consiste en identificar tareas y las relaciones que existen entre sí (dependencias) para poder ejecutarlas concurrentemente siempre que sea posible.

```
/* Task A */
a=0;
for (int i=0; i<n-1; i++ ) {
  a = a + x[i];
}
/* Task B */
b=0;
for (int i=0; i<n-1; i++ ) {
  b = b + y[i];
}
/* Task C */
for (int i=0; i<n-1; i++ ) {
  z[i] = x[i]/b + y[i]/a;
}
/* Task D */
for (int i=0; i<n-1; i++ ) {
  y[i] = (a+b)*y[i];
}
```



Diseño de Algoritmos Paralelos basado en Tareas

El diseño basado en tareas implica dos aspectos básicos:

1. Descomponer el problema en tareas.
   1. Consiste en realizar un análisis detallado del problema para obtener un Grafo de Dependencia de Tareas.
2. *Mapear* tareas a recursos (procesadores o cores).
   1. Consiste en asignar las tareas a hilos. Esta asignación puede ser estática o dinámica.
   2. Las dependencias inducen una sincronización entre tareas.

Como estrategia general se debe maximizar el grado de concurrencia y minimizar las dependencias con objeto de obtener la máxima eficiencia.

Grafo de Tareas

Es una abstracción que expresa las dependencias que existen entre las tareas y, por tanto, el orden en que se ejecutarán:

- Es un grafo acíclico (*Direct Acyclic Graph o DAG*).
- Los nodos representan las tareas.
- Las aristas o arcos representan las dependencias.

Para un grafo dado tenemos:

- Si c_i representa al coste de un arco, entonces la longitud del camino crítico de un grafo es la suma de los costes (c_i) de los arcos del camino más largo desde un nodo inicial hasta el nodo final.
- El grado medio de concurrencia: $M = \sum_{i=1}^{n} \frac{c_i}{L}$.

  M es un límite superior para el speedup.

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 son independientes si

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

donde I_i y O_i representan el conjunto de variables leídas y escritas por T_i, respectivamente.
Tipos de dependencias (T_i precede a T_j en secuencial):

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

Paralelismo de Tareas en OpenMP

Se puede intentar una aproximación al paralelismo de tareas mediante la construcción `sections` de forma muy básica.

- Esquema fork-join con un número fijo de tareas concurrentes (en tiempo de compilación).
- Un grafo de dependencias se implementaría con varias sections y sincronización entre ellas.

La utilidad para crear y manejar tareas reales y dinámicas en OpenMP apareción con los estándares:

- OpenMP 3.0: construcción `task` con sincronización explícita (`taskwait`).
- OpenMP 4.0: descripción del grafo a través de las dependencias con la cláusula `depend`.

Implementación del Paralelismo de Tareas en OpenMP

El modelo de tareas se fundamenta en:

- La creación de tareas es una operación no-bloqueante, el hilo crea la tarea y continúa ejecutando el programa.
- El runtime se encarga de la planificación de las tareas pendientes que son ejecutadas por el resto de hilos.
- La implementación es dependiente del sistema. Algunos runtimes, para mejorar el balanceo de la carga, utilizan work-stealing.

Paralelismo de Tareas

En problemas irregulares, el paralelismo de tareas es mejor solución que otras construcciones

- Bucles sin límites establecidos
- Algoritmos recursivos
- Esquemas productor-consumidor

Ejemplo de paralelización enrevesada e ineficiente

```
void traverse_list(List l)
{
  Element e;
  #pragma omp parallel private(e)
  for (e = l->first; e; e = e->next)
    #pragma omp single nowait
    process(e);
}
```

El Modelo de Tareas en OpenMP

Tareas: unidades de trabajo cuya ejecución se *puede* posponer

- También se pueden ejecutar inmediatamente
- Se componen de código y datos

```
#pragma omp task [clauses]
```

Directiva creación:

Cada hilo crea una tarea (empaquetando código y datos)

- Alcance de variables: `shared`, `private`, `firstprivate` (se inicializa en la creación), `default(shared|none)`
- En una construcción `task`, si no está presente la cláusula `default`,
  - una variable que está cualificada como `shared` en todas las construcciones que incluyen a la construcción actual será `shared`.
  - una variable que no está cualificada según la regla anterior, entonces, es `firsprivate`.

  Esto implica que:
  - si una construcción `task` está incluida en una región `parallel`, las variables que son `shared` en dicha región lo seuirán siendo en la `task`.
  - si una construcción `task` es *huérfana*, las variables son implícitamente `firstprivate`.
- Las barreras de los hilos afectan a las tareas.

```
#pragma omp taskwait
```

Directiva sincronización:

- Bloquea un hilo hasta que acaban sus tareas hijas (directas)

El Modelo de Tareas en OpenMP

Tareas: unidades de trabajo cuya ejecución se *puede* posponer

- También se pueden ejecutar inmediatamente
- Se componen de código y datos

Directiva creación:

```
#pragma omp task [clauses]
```

Cada hilo crea una tarea (empaquetando código y datos)

- Alcance de variables: `shared`, `private`, `firstprivate` (se inicializa en la creación), `default(shared|none)`
- En una construcción `task`, si no está presente la cláusula `default`,
  - una variable que está cualificada como `shared` en todas las construcciones que incluyen a la construcción actual será `shared`.
  - una variable que no está cualificada según la regla anterior, entonces, es `firsprivate`.

  Esto implica que:
  - si una construcción `task` está incluida en una región `parallel`, las variables que son `shared` en dicha región lo seuirán siendo en la `task`.
  - si una construcción `task` es *huérfana*, las variables son implícitamente `firstprivate`.
- Las barreras de los hilos afectan a las tareas.

Directiva sincronización:

```
#pragma omp taskwait
```

- Bloquea un hilo hasta que acaban sus tareas hijas (directas)

El Modelo de Tareas en OpenMP

Tareas: unidades de trabajo cuya ejecución se *puede* posponer

- También se pueden ejecutar inmediatamente
- Se componen de código y datos

Directiva creación:

```
#pragma omp task [clauses]
```

Cada hilo crea una tarea (empaquetando código y datos)

- Alcance de variables: `shared`, `private`, `firstprivate` (se inicializa en la creación), `default(shared|none)`
- En una construcción `task`, si no está presente la cláusula `default`,
  - una variable que está cualificada como `shared` en todas las construcciones que incluyen a la construcción actual será `shared`.
  - una variable que no está cualificada según la regla anterior, entonces, es `firsprivate`.

  Esto implica que:
  - si una construcción `task` está incluida en una región `parallel`, las variables que son `shared` en dicha región lo seuirán siendo en la `task`.
  - si una construcción `task` es *huérfana*, las variables son implícitamente `firstprivate`.
- Las barreras de los hilos afectan a las tareas.

Directiva sincronización:

```
#pragma omp taskwait
```

- Bloquea un hilo hasta que acaban sus tareas hijas (directas)

Modelo de Ejecución de Tareas

Las tareas se ejecutan por algún hilo del equipo que la generó

Recorrido de listas con `task`

```
void traverse_list(List l)
{
  Element e;
  for (e = l->first; e; e = e->next)
    #pragma omp task
    process(e);      /* la variable e es firstprivate */
  #pragma omp taskwait
}
...
#pragma omp parallel    /* Un hilo crea las tareas y */
#pragma omp single      /* todos cooperan en su ejecución */
traverse_list(l);
```

- `if` permite controlar la creación de tareas
- `untied` permite migrar tareas entre hilos

Modelo de Ejecución de Tareas

Las tareas se ejecutan por algún hilo del equipo que la generó

Recorrido de listas con `task`

```
void traverse_list(List l)
{
  Element e;
  for (e = l->first; e; e = e->next)
    #pragma omp task
    process(e);      /* la variable e es firstprivate */
  #pragma omp taskwait
}
...
#pragma omp parallel    /* Un hilo crea las tareas y */
#pragma omp single      /* todos cooperan en su ejecución */
traverse_list(l);
```

- `if` permite controlar la creación de tareas
- `untied` permite migrar tareas entre hilos

Recursión con Regiones Paralelas Anidadas

Quicksort con regiones paralelas anidadas

```
void quick_sort (int p, int r, float *data) {
    if (p < r) {
        int q = partition (p, r, data);
        #pragma omp parallel sections firstprivate(data,p,q,r)
        {
            #pragma omp section
            quick_sort (p, q-1, data);
            #pragma omp section
            quick_sort (q+1, r, data);
        }
    }
}
void par_quick_sort (int n, float *data) {
    quick_sort (0, n-1, data);
}
```

Demasiadas regiones paralelas: mucho overhead, muchas sincronizaciones, no siempre bien soportado

Recursión con Directiva `task`

Quicksort con `task`

```
void quick_sort(float *data, int n) {
    int p;
    if (n < N_MIN) {
        insertion_sort(data, n);
    } else {
        p = partition(data, n);
        #pragma omp task
        quick_sort(data, p);
        #pragma omp task
        quick_sort(data+p+1, n-p-1);
    }
}
...
#pragma omp parallel
#pragma omp single
quick_sort(data, n);
```

Uso de `taskwait`

La sincronización de tareas es necesaria:

- cuando una operación require un resultado de la tarea, o
- para evitar que una variable `shared` pueda desaparecer antes de que la tarea finalice.

Fibonacci `task`

```
int fib(int n) {
    int x,y;
    if (n < 2) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

El Modelo de Tareas de OpenMP

- Cuando un hilo encuentra una construcción de tarea `task`, puede optar por ejecutar la tarea inmediatamente o posponer su ejecución hasta un momento posterior.
- Si se difiere la ejecución de la tarea, entonces la tarea se coloca en un repositorio conceptual de tareas que está asociado con la región paralela actual.
- Los hilos del equipo actual extraen las tareas del repositorio y las ejecutarán hasta que el repositorio quede vacío.
- El hilo que ejecuta una tarea puede ser diferente del hilo que la comenzó a ejecutar originalmente.

El Modelo de Tareas de OpenMP

- El código asociado a una tarea se ejecuta solo una vez.
- Una tarea se dice que está vinculada o "atada" (`tied`) si el código es ejecutado por el mismo hilo de principio a fin.
- La tarea es `untied` si el código puede ser ejecutado por más de un hilo, de modo que diferentes hilos ejecutan diferentes partes del código.
- Por defecto, las tareas son `tied`.

El Modelo de Tareas de OpenMP

- Los hilos pueden suspender la ejecución de una región de tarea en un punto de interrupción para ejecutar una tarea diferente.
- Si la tarea suspendida es `tied`, el mismo hilo más tarde reanuda la ejecución de la tarea suspendida.
- Si la tarea suspendida es `untied`, cualquier hilo en el equipo actual puede reanudar la ejecución de la tarea.

La especificación OpenMP define los siguientes puntos de interrupcción:

- una construcción `task`,
- una construcción `taskwait`,
- una construcción `taskyield`,
- una barrera implícita o explícita, y
- la terminación de la tarea.

Cláusula `untied`

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
...
#pragma omp parallel
{
  #pragma omp single
  {
    int i;
    #pragma omp task untied
    {
      for (i=0;i<LARGE_NUMBER;i++)
        #pragma omp task
        process(item[i]);
    }
  }
}
```

El primer `task` permite que el hilo que genera tareas pueda ponerse a ejecutar una y otro hilo continúe generando

Cláusulas `final` y `mergeable`

Cláusula `final (expresion)`

- Si la expresión se evalua a cierto la tarea no tendrá descendientes.
- En problemas recursivos que realizan descomposición de tareas, para la creación de tareas hasta cierta profundidad.

```
#pragma omp task final(expr)
```

Cláusula `mergeable`

- Permite que la implementación mezcle el entorno de datos de la tarea con el de la región en la que está incluida.
- La implementación puede optar por crear una tarea normal o ejecutarla en el mismo instante de la creación.

```
#pragma omp task mergeable
```

Directiva `taskyield`

- La directiva `taskyield` especifica que la tarea actual puede ser suspendida en favor de la ejecución de una tarea distinta.
- Sirve para evitar interbloqueos.

Ejemplo de `taskyield`

```
void foo( omp_lock_t *lock, int n ) {
int i;
for ( i = 0; i < n; i++ )
  #pragma omp task
  {
    something_useful();
    while ( !omp_test_lock(lock) ) {
      #pragma omp taskyield
    }
    something_critical();
    omp_unset_lock(lock);
  }
}
```

Directiva `taskgroup`

```
#pragma omp taskgroup
```

- Especifica una espera sobre la finalización de las tareas hijas de la tarea actual y sobre la finalización de sus tareas descendientes.
- Se trata de una sincronización parecida a **`taskwait`** pero con la opción de restringir a un subconjunto de tareas sobre el que esperar.
- Cuando un hilo encuentra una construcción **`taskgroup`**, comienza a ejecutar la región.
- Todas las tareas hijas generadas en la región **`taskgroup`** y todos sus descendientes son parte del grupo de tareas asociado con el **`taskgroup`**.
- Hay un punto de sincronización de tareas implícito al final de la región del **`taskgroup`**.
- La tarea actual es suspendida en el punto de sincronización de la tarea hasta que todas las tareas en el grupo hayan completado su ejecución.

Se puede cancelar un grupo de tareas con **`cancel taskgroup`**.

- Las tareas que encuentran la directiva saltan al final de la tarea.
- Las tareas no iniciadas se descartan.

OpenMP 4.0: La cláusula depend

Cláusula `depend`

```
#pragma omp task depend(dependency-type: list)
... structured block ...
```

- Una dependencia de tarea se produce cuando la tarea predecesora se ha completado.
- La especificación dice lo siguiente:
  - `in` dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause.
  - `out` and `inout` dependency-type: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` clause.
  - The list items in a depend clause may include array sections.

La cláusula depend: ejemplo

```
void process_in_parallel() {
 #pragma omp parallel
 #pragma omp single
 {
   int x = 1;
   ...
   for( int i = 0; i < T; ++i ) {
   #pragma omp task shared(x) depend(out:x)
   tarea1(...);
   #pragma omp task shared(x) depend(in:x)
   tarea2(...);
   #pragma omp task shared(x) depend(in:x)
   tarea3(...);
   }
 } // end omp single, omp parallel
```

La cláusula depend: ejemplo

```
void process_in_parallel() {
 #pragma omp parallel
 #pragma omp single
 {
   int x = 1;
   ...
   for( int i = 0; i < T; ++i ) {
   #pragma omp task shared(x) depend(out:x)
   tarea1(...);
   #pragma omp task shared(x) depend(in:x)
   tarea2(...);
   #pragma omp task shared(x) depend(in:x)
   tarea3(...);
   }
 } // end omp single, omp parallel
```