TPP – Tecnología de la Programación Paralela

Máster en Computación Paralela y Distribuida

1. Paradigma de Memoria Compartida Multi-Hilo

José E. Román

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curso 2015/16

Contenido

Apartado 1

Paralelismo de Datos

- Conceptos Básicos
- Paralelización de Bucles
- Tipos de Variables
- Directiva `parallel`
- Reparto del Trabajo
- Paralelismo Anidado
- Sincronización

Modelo de Memoria Compartida con Hilos

Características:

- Un proceso puede tener múltiples hilos de ejecución
- Los hilos comparten recursos/memoria del proceso
- Cada hilo puede tener variables privadas
- El acceso a variables compartidas requiere sincronización
  - Semáforos, monitores, variables condición
- No hay que intercambiar datos explícitamente

Inconvenientes de `pthreads`:

- Portabilidad a algunos sistemas (Windows)
- No está orientado a computación (Fortran)
- Más orientado a paralelismo de tareas (no de datos)
- El API es de muy bajo nivel

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), 3.1 (2011), 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

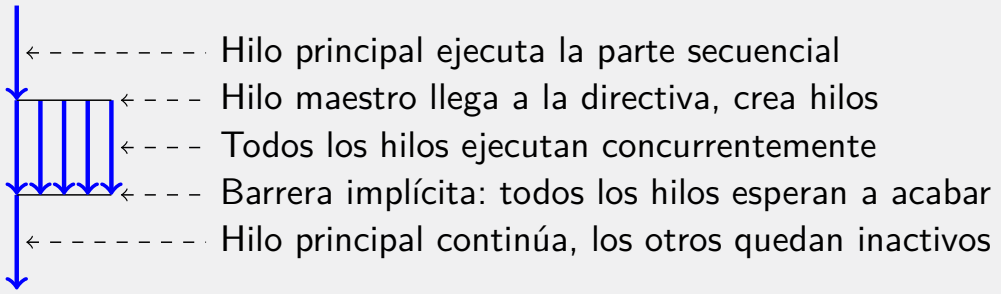
Además: funciones (ver `omp.h`) y variables de entorno

Modelo de Ejecución

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

Hay directivas para crear hilos y dividir el trabajo

Esquema



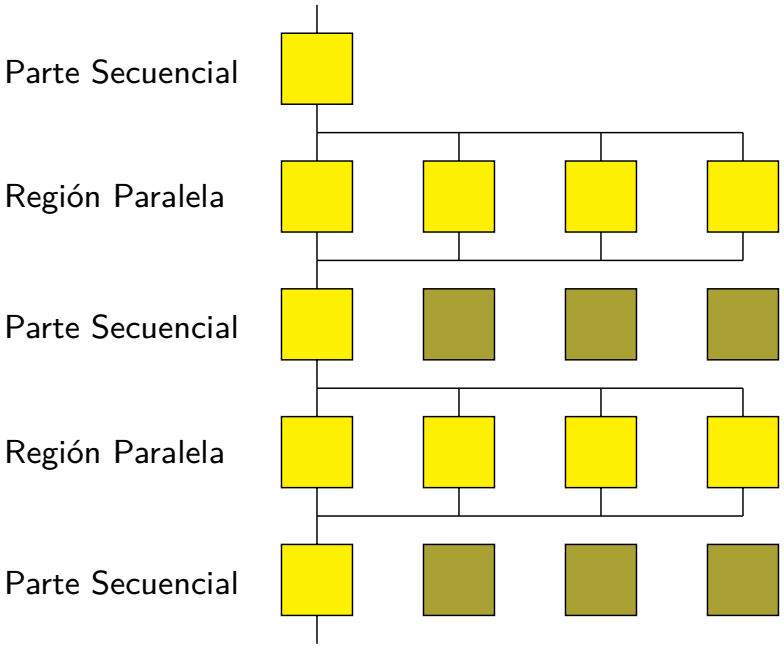
Las directivas definen regiones paralelas

Otras directivas/cláusulas:

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

Modelo de Ejecución - Hilos

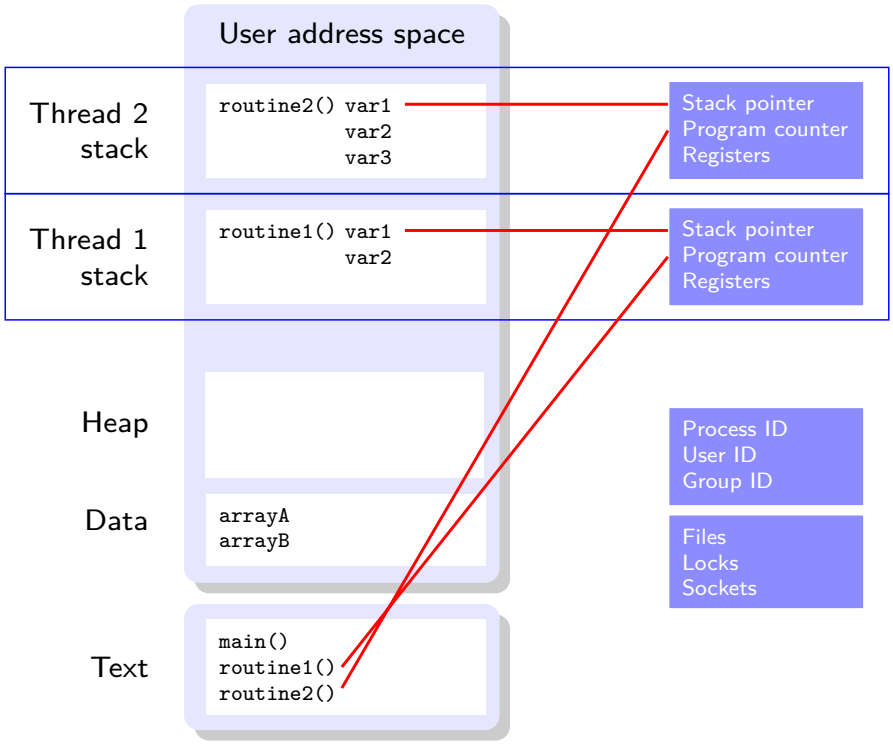
En OpenMP los hilos inactivos no se destruyen, quedan a la espera de la siguiente región paralela



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

C/C++

```
#pragma omp <directiva>
```

Fortran

```
!$omp <directiva>
```

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 -x03 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

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());
}
```

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
}
```

Fortran

```
!$omp parallel do [clausula [,] [...]]
    do index = first, last[, stride]
!       cuerpo del bucle
    enddo
[!$omp end parallel do]
```

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

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`
- `default`
- `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, 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

[Ojo: en Fortran las excepciones son diferentes]

En Fortran se puede cambiar para que por defecto sea `private`

```
#pragma omp parallel for default(private)
```

Se puede usar `default(none)` para depurar (da error)

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 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

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

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:

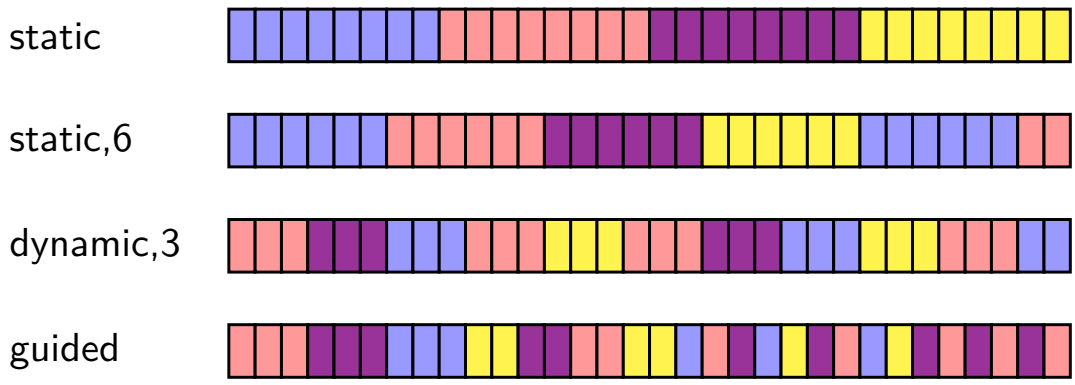
```
schedule(type[,chunk])
```

- `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
```



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 ejecuta de forma replicada el bloque que va a continuación

C/C++

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

Fortran

```
!$omp parallel [clausula [,] [...]]
!       bloque
!$omp end parallel
```

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

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 la instrucción `data` en Fortran 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 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

Hay una barrera implícita al final del bloque

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

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`, `copyprivate`

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);
}
```


Construcciones Huérfanas

Las construcciones de reparto de trabajo pueden ser huérfanas

Ejemplo

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

En el ejemplo, la función process_array puede ser invocada sin problemas desde fuera de la región paralela

Regiones Paralelas Anidadas (1)

Se permite un esquema similar pero con directivas `parallel`

Ejemplo paralelismo anidado

```
int myindex;
#pragma omp parallel private(myindex)
{
    myindex = get_next_task();
    while (myindex>-1) {
        process_task(myindex);
        myindex = get_next_task();
    }
}
...
void process_task(myindex) {
    int i;
    if(myindex>0) {
        #pragma omp parallel for
        for (i=1; i<n; i++)
            result[i] = ...
    }
}
```


Regiones Paralelas Anidadas (2)

Paralelismo anidado en el mismo bloque

```
int i, j;
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<n; i++) {
        #pragma omp parallel shared(i, n)
        {
            #pragma omp for
            for (j=0; j<n; j++)
                dowork(i, j);
        }
    }
}
```

No todos los hilos han de encontrarse con la región paralela interna (en el caso de construcciones de reparto de trabajo sí)

Regiones Paralelas Anidadas (3)

El paralelismo anidado puede ser contraproducente, suele estar desactivado por defecto

- Algunas implementaciones de OpenMP serializan las regiones internas
- El programa se ejecuta correctamente pero la región interna no proporciona mayor grado de paralelismo

---

El comportamiento por defecto se puede cambiar con

```
omp_set_nested(1)
```

o con la variable de entorno

```
OMP_NESTED=TRUE
```

$\longrightarrow$ ICV: *Internal Control Variable*

La función `omp_in_parallel()` se puede usar para evitar el paralelismo anidado (con la directiva `if`)

Hilos Dinámicos

Para usar tantos hilos como núcleos tiene el sistema, se podría hacer

```
omp_set_num_threads(omp_get_num_procs())
```

Los hilos dinámicos son el mecanismo de OpenMP para hacer un buen uso de los recursos del sistema

El número de hilos se reajusta al entrar en cada región paralela

```
OMP_DYNAMIC=TRUE
```

Se recomienda llamar a `omp_get_num_threads()` cada vez

Fusión de Bucles (OpenMP 3.0)

Paralelismo anidado

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

Bucles fundidos

```
#pragma omp parallel for
            collapse(2)
for (i=0; i<n; i++) {
    for (j=0; j<m; j++) {
        // cuerpo del bucle
    }
}
```

Se reparten las iteraciones globalmente

- Más eficiente que el paralelismo anidado
- Efectivo cuando `n` y `m` son pequeños

Restricciones:

- Han de ser "perfectamente" anidados
- Espacio de iteraciones rectangular

Mejora de Paralelismo Anidado (OpenMP 3.0)

Algunas variables ICV se definen por tarea: dyn-var, nest-var, nthreads-var, run-sched-var

Regiones paralelas de tamaño variable

```
omp_set_num_threads(3);
#pragma omp parallel
{
   omp_set_num_threads(omp_get_thread_num()+2);
   #pragma omp parallel
      do_work();
}
```

Nuevas funciones:

- Cuántos niveles: `omp_get_level`, `omp_get_active_level`
- Info de niveles anteriores: `omp_get_team_size(level)`, `omp_get_ancestor_thread_num(level)`,

Nuevas variables: `OMP_MAX_NESTED_LEVEL`, `OMP_THREAD_LIMIT`

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

En OpenMP 3.1 se resuelve con `reduction(max:cur_max)`

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`

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

Para aprovechar instrucciones especiales del procesador para actualizar una posición de memoria (ej. CMPXCHG en Intel)
En C/C++ la sintaxis es:

```
#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)

Sincronización a Medida

Hay casos en que es necesaria la sincronización aunque no sea obvio

Ejemplo productor/consumidor

Hilo productor

```
data = ...
flag = 1
```

Hilo consumidor

```
while(flag==0) {}
process(data);
```

El resultado puede ser distinto del esperado:

- La modificación de `flag` no se actualiza en memoria principal inmediatamente
- El compilador puede reordenar las operaciones (p.e., asignar `data` después de `flag`)

Directiva `flush`

Define un punto en que un hilo tiene garantizada una visión consistente de memoria respecto de una variable (o varias)

- Todas las operaciones de lectura/escritura anteriores al `flush` se completan antes de que se ejecute el `flush`
- Todas las operaciones de lectura/escritura posteriores al `flush` no se ejecutan hasta después del `flush`

De esta forma, `flush` fuerza la actualización de memoria para que otros hilos vean el valor más reciente

Ejemplo productor/consumidor

Hilo productor

```
data = ...
#pragma omp flush(data)
flag = 1
#pragma omp flush(flag)
```

Hilo consumidor

```
while(flag==0) {
  #pragma omp flush(flag)
}
#pragma omp flush(data)
process(data);
```

Apartado 2

Paralelismo de Tareas

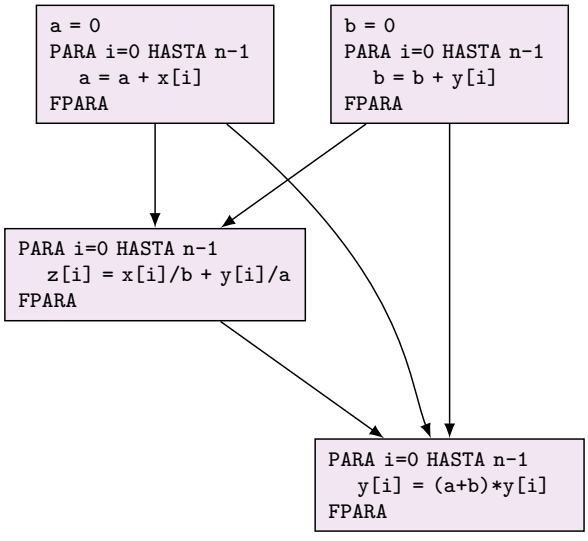
- Dependencias de Datos
- Paralelismo de Tareas
- Tareas OpenMP 4.0
- Otros Sistemas

Paralelización de Algoritmos

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

Las dependencias entre tareas imponen un orden parcial

```
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
```



Granularidad fina o gruesa, según el tamaño de las tareas

Diseño de Algoritmos Paralelos

Básicamente dos fases:

1. Descomposición en tareas
   - Requiere un análisis detallado del problema
     $\longrightarrow$ Grafo de Dependencia de Tareas
2. Asignación de tareas (estática o dinámica)
   - Qué hilo/proceso ejecuta cada tarea
   - Normalmente implica agrupar varias tareas

Para una asignación dada, las dependencias implican:

- Sincronización en memoria compartida
- Comunicación en paso de mensajes

Hay que maximizar el grado de concurrencia (muchas tareas similares y pocas dependencias)

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 las tareas (y su coste asociado)
- Las aristas representan las dependencias entre tareas

Definiciones:

- Longitud del camino crítico (L): suma de los costes c_i de los nodos que componen el camino más largo entre un nodo inicial y un nodo final
- Grado medio de concurrencia: $M = \sum_{i=1}^{n} \frac{c_i}{L}$
  (se puede ver como una cota superior del speed-up)

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. $O_i \cap O_j = \varnothing$

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

Tipos de dependencias (T_i precede a T_j en secuencial):

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

Ejemplo Dependencia RaW en Bucles (1)

Estas dependencias se pueden eliminar a veces con `reduction`

En aquellos casos en que `reduction` no se puede usar: *Eliminación de variables conflictivas por inducción*

Código secuencial

```
i_sum = 0;
for (i=0; i<n/2; i++) {
    b[i] = i_sum;
    i_sum = i_sum + (i+1);
}
```

Código paralelo sin dependencias

```
#pragma omp parallel for shared(b)
for (i=0; i<n/2; i++)
    b[i] = (i+1)*(i+2)/2;
```

Ejemplo Dependencia RaW en Bucles (2)

Eliminación de la dependencia mediante *sesgado del bucle*

Código secuencial

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

Código paralelo sin dependencias

```
b[1] = b[1] + a[0];
#pragma omp parallel for shared(a,b,c)
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];
```


Ejemplo Dependencia WaR en Bucles

Siempre se pueden eliminar (aunque puede resultar ineficiente)

Código secuencial

```
for (i=0; i<n-1; i++) {
    x = (b[i] + c[i])/2;
    a[i] = a[i+1] + x;
}
```

Código paralelo sin dependencias

```
#pragma omp parallel for shared(a, a2)
for (i=0; i<n-1; i++)
    a2[i] = a[i+1];
#pragma omp parallel for shared(a, a2) private(x)
for (i=0; i<n-1; i++) {
    x = (b[i] + c[i])/2;
    a[i] = a2[i] + x;
}
```

Ejemplo Dependencia WaW en Bucles

Código secuencial

```
for (i=0; i<n; i++) {
    x = (b[i] + c[i])/2;
    a[i] = a[i] + x;
    d[1] = 2*x;
}
y = x + d[1] + d[2];
```

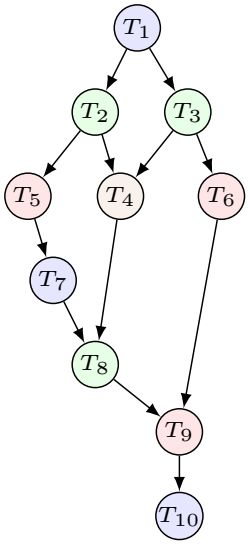
Código paralelo sin dependencias

```
#pragma omp parallel for shared(a) lastprivate(x,d1)
for (i=0; i<n; i++) {
    x = (b[i] + c[i])/2;
    a[i] = a[i] + x;
    d1 = 2*x;
}
d[1] = d1;
y = x + d[1] + d[2];
```


Ejemplo: Cholesky por Bloques

$$\begin{bmatrix} A_{11} & & \text{sim} \\ A_{21} & A_{22} & \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{bmatrix}$$

$$\begin{array}{lllll} k=1 & A_{11} \leftarrow \text{chol}(A_{11}) & & & \\ & j=2 & A_{21} \leftarrow A_{21}A_{11}^{-T} & & \\ & j=3 & A_{31} \leftarrow A_{31}A_{11}^{-T} & & \\ & j=2 & i=3 & A_{32} \leftarrow A_{32} - A_{31}A_{21}^T & \\ & & & A_{22} \leftarrow A_{22} - A_{21}A_{21}^T & \\ & j=3 & & A_{33} \leftarrow A_{33} - A_{31}A_{31}^T & \\ k=2 & A_{22} \leftarrow \text{chol}(A_{22}) & & & \\ & j=3 & A_{32} \leftarrow A_{32}A_{22}^{-T} & & \\ & j=3 & & A_{33} \leftarrow A_{33} - A_{32}A_{32}^T & \\ k=3 & A_{33} \leftarrow \text{chol}(A_{33}) & & & \end{array}$$



Paralelismo de Tareas en OpenMP

La construcción `sections` permite implementar el paralelismo de tareas (en contraposición a paralelismo de datos) 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 varios `sections` y sincronización entre ellos

Posteriormente se añadió soporte para tareas dinámicas

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

Implementación del Modelo de Tareas

Tanto en OpenMP como en otros sistemas, 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
- Para mejorar el balanceo de la carga, se prevé la posibilidad de *work-stealing* de tareas pertenecientes a otros hilos

E. Ayguadé et al. The design of OpenMP tasks.
IEEE Trans. Parallel Distrib. Systems, 20(3):404-418, 2009.

Paralelismo Irregular

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);
}
```

Cada hilo recorre la lista completa y determina para cada elemento si ya ha sido procesado por otro hilo

El Modelo de Tareas en OpenMP

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

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

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)`
- Por defecto `firstprivate`, pero `shared` se hereda
- Las barreras de los hilos afectan a las tareas

Directiva sincronización:

```
#pragma omp taskwait
```

- Bloquea una tarea (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 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(float *data, int n)
{
    int p;
    if (n < N_MIN) {
        insertion_sort(data, n);
    } else {
        p = partition(data, n);
        #pragma omp parallel sections firstprivate(data,p,n)
        {
            #pragma omp section
            quick_sort(data, p);
            #pragma omp section
            quick_sort(data+p+1, n-p-1);
        }
    }
}
```

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 explícita de tareas es necesaria

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

Fibonacci con 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;
}
```

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

Tareas - Mejoras en OpenMP 4.0

Cláusula `final`

- En problemas recursivos que realizan descomposición de tareas, para la creación de tareas a cierta profundidad
- Todas las tareas hijas son también finales

```
#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 que está incluida
- En tareas que se ejecutan sin retraso

```
#pragma omp task mergeable
```

Diferencia entre `if` y `final`

```
int i;
#pragma omp task if(0)     // This task is undeferred
{
  #pragma omp task         // This task is a regular task
  for (i = 0; i < 3; i++) {
    #pragma omp task       // This task is a regular task
    bar();
  }
}
#pragma omp task final(1)  // This task is a regular task
{
  #pragma omp task         // This task is included
  for (i = 0; i < 3; i++) {
    #pragma omp tas        // This task is also included
    bar();
  }
}
```


Ejemplo `final` y `mergeable`

```
#define LIMIT 3   /* arbitrary limit on recursion depth */
void bin_search (int pos, int n, char *state) {
  if ( pos == n ) { check_solution(state); return; }
  #pragma omp task final(pos > LIMIT) mergeable
  {
    char new_state[n];
    if (!omp_in_final()) {
      memcpy(new_state, state, pos);
      state = new_state;
    }
    state[pos] = 0;
    bin_search(pos+1, n, state);
  }
  #pragma omp task final(pos > LIMIT) mergeable
  {
    char new_state[n];
    if (!omp_in_final()) {
      memcpy(new_state, state, pos);
      state = new_state;
    }
    state[pos] = 1;
    bin_search(pos+1, n, state);
  }
  #pragma omp taskwait
}
```

Prevención de Interbloqueos

La directive `taskyield` especifica que la tarea actual puede ser suspendida en favor de la ejecución de una tarea distinta

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);
    }
}
```


Sincronización por Grupos y Cancelación

```
#pragma omp taskgroup
```

Especifica una espera sobre la finalización de las tareas hijas y sus tareas descendientes

- Sincronización más profunda que `taskwait`, pero
- con la opción de restringir a un subconjunto de tareas (al contrario que `barrier`)

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

Ejemplo taskgroup y cancel taskgroup

```
binary_tree_t* search_tree_parallel(binary_tree_t* tree, int value) {
  binary_tree_t* found = NULL;
  #pragma omp parallel shared(found,tree,value)
  {
    #pragma omp master
    {
      #pragma omp taskgroup
      {
      found = search_tree(tree, value);
      }
    }
  }
  return found;
}
binary_tree_t* search_tree( binary_tree_t* tree, int value) {
  binary_tree_t* found = NULL;
  if (!tree) return NULL;
  if (tree->value == value) found = tree;
  else {
    #pragma omp task shared(found)
    {
      binary_tree_t* found_left = search_tree(tree->left, value);
      if (found_left) {
        #pragma omp atomic write
        found = found_left;
        #pragma omp cancel taskgroup
      }
    }
    // lo mismo para found_right
  }
  return found;
}
```


Dependencias Explícitas

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

La dependencia se satisface cuando las tareas predecesoras se han completado

- Dependencias de tipo `in`: la tarea generada será dependiente de todas las tareas hijas previamente generadas que referencian al menos un item de la lista en una cláusula `out` o `inout`
- Dependencias de tipo `out` o `inout`: la tarea generada será dependiente de todas las tareas hijas previamente generadas que referencian al menos un item de la lista en una cláusula `in`, `out` o `inout`

Ejemplo de depend

```
#pragma omp parallel
#pragma omp single
{
  int x = 1;
  ...
  for (int i = 0; i < T; ++i) {
    #pragma omp task shared(x, ...) depend(out: x)
      preprocess_some_data(...);
    #pragma omp task shared(x, ...) depend(in: x)
      do_something_with_data(...);
    #pragma omp task shared(x, ...) depend(in: x)
      do_something_independent_with_data(...);
  }
}
```

- La primera tarea ha de completarse antes de que las otras dos puedan ejecutarse
- Las variables en la cláusula depend no han de indicar necesariamente el flujo de datos

Producto de Matrices

En las dependencias se pueden especificar secciones de arrays

Producto de Matrices

```
/* suponemos N divisible por BS */
void matmul_depend(int N, int BS, float A[N][N],
                   float B[N][N], float C[N][N] )
{
  int i, j, k, ii, jj, kk;
  for (i = 0; i < N; i+=BS) {
    for (j = 0; j < N; j+=BS) {
      for (k = 0; k < N; k+=BS) {
        #pragma omp task depend (in: A[i:BS][k:BS], B[k:BS][j:BS]) \
                        depend (inout: C[i:BS][j:BS])
        for (ii = i; ii < i+BS; ii++ )
          for (jj = j; jj < j+BS; jj++ )
            for (kk = k; kk < k+BS; kk++ )
              C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
```

Quark

La librería Quark forma parte del proyecto PLASMA (U. Tennessee)

- Basada en paralelismo de tareas, con un planificador dinámico capaz de ejecutar las tareas en orden diferente al que se crearon, respetando las dependencias entre ellas
- Las tareas se insertan con `QUARK_Insert_Task` y el planificador deduce las dependencias
- Los hilos ejecutan las tareas usando una combinación de asignación de tareas por defecto y *work-stealing*
- El DAG nunca se crea de forma explicita en el planificador
- Permite mostrar el DAG de forma gráfica

PLASMA implementa subrutinas de álgebra lineal basándose en un formato especial de representación de matrices a bloques (*tile*)

Quark: Ejemplo

```
/* función que ejecuta las tareas */
void matmul_quark_task( Quark *quark )
{
  double *A, *B, *C;
  int NB;
  quark_unpack_args_4( quark, A, B, C, NB );
  GEMM( A, B, C, NB );
}

/* función que inserta las tareas */
void matmul_quark_call( Quark *quark, double *A, double *B, double *C, int NB )
{
  QUARK_Insert_Task( quark, matmul_quark_task, NULL,
      sizeof(double)*NB*NB, A, INPUT,
      sizeof(double)*NB*NB, B, INPUT,
      sizeof(double)*NB*NB, C, INOUT,
      sizeof(int), &NB, VALUE,
      0 );
}

void main(){
  int THREADS=4;
  Quark *quark = QUARK_New(THREADS);
  for(i=0;i<BB;i++)
    for(j=0;j<BB;j++)
      for (k=0;k<BB;k++)
        matmul_quark_call( quark, &Ablk[NB*NB*i+NB*NB*BB*k],
                  &Bblk[NB*NB*k+NB*NB*BB*j], &Cblk[NB*NB*i+NB*NB*BB*j], NB);
  QUARK_Delete(quark)
}
```