

Sincronización de los threads y tareas

Sincronización de *threads*

- ▶ Cuando no pueden eliminarse las dependencias de datos entre los *threads*, es necesario sincronizar su ejecución.

OpenMP proporciona los mecanismos de sincronización más habituales: **exclusión mutua** y **sincronización por eventos**.

Exclusión mútua (SC)

1. Secciones Críticas

Define un trozo de código que no puede ser ejecutado por más de un *thread* a la vez.

OpenMP ofrece varias alternativas para la ejecución en exclusión mutua de secciones críticas. Las dos opciones principales son: **critical** y **atomic**.

Exclusión mútua

► Directiva `critical`

Define una única sección crítica para todo el programa, dado que no utiliza variables de *lock*.

```
#pragma omp parallel firstprivate(MAXL)
{
  ...
  #pragma omp for
  for (i=0; i<N; i++) {
    A[i] = B[i] / C[i];
    if (A[i]>MAXL) MAXL = A[i];
  }
  #pragma omp critical
  { if (MAXL>MAX) MAX = MAXL; }
  ...
}
```

OJO: la sección crítica debe ser lo "menor" posible!

Exclusión mútua

- Secciones críticas “específicas” (*named*)

También pueden definirse diferentes secciones críticas, controladas por la correspondiente variable cerrojo.

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
    A[i] = fun(i);

    if (A[i]>MAX)
        #pragma omp critical(M1)
        { if (A[i]>MAX) MAX = A[i]; }

    if (A[i]<MIN)
        #pragma omp critical(M2)
        { if (A[i]<MIN) MIN = A[i]; }
}
```

Exclusión mútua

► Directiva `atomic`

Es un caso particular de sección crítica, en el que se efectúa una operación RMW atómica sencilla (con limitaciones).

```
#pragma omp parallel ...
{
    ...
    #pragma omp atomic
    X = X + 1;
    ...
}
```

Para este tipo de operaciones, es más eficiente que definir una sección crítica.

Variables cerrojo (lock)

2. Funciones con cerrojos

Un conjunto de funciones de librería permite manejar variables cerrojo y definir así secciones críticas "ad hoc".

En C, las variables cerrojo deben ser del tipo predefinido: `omp_lock_t C;`

- `omp_init_lock(&C);`
reserva memoria e inicializa un cerrojo.
- `omp_destroy_lock(&C);`
libera memoria del cerrojo.

Variables cerrojo (lock)

- `omp_set_lock (&C) ;`
coge el cerrojo o espera a que esté libre.
- `omp_unset_lock (&C) ;`
libera el cerrojo.
- `omp_test_lock (&C) ;`
testea el valor del cerrojo; devuelve T/F.

Permiten gran flexibilidad en el acceso en exclusión mutua. La variable de *lock* puede pasarse como parámetro a una rutina.

Variables cerrojo (lock)

> Ejemplo

```
#pragma omp parallel private(nire_it)
{
    omp_set_lock(&C1);
    mi_it = i;
    i = i + 1;
    omp_unset_lock(&C1);

    while (mi_it < N)
    {
        A[mi_it] = A[mi_it] + 1;

        omp_set_lock(&C1);
        mi_it = i;
        i = i + 1;
        omp_unset_lock(&C1);
    }
}
```

Variables cerrojo (lock)

- ▶ Un grupo similar de funciones permite el anidamiento de las secciones críticas, sin que se produzca un bloqueo por recursividad.

La sintaxis es la misma que en el grupo anterior, añadiendo `_nest`. Por ejemplo:

- `omp_set_nest_lock(var);` entrada a la SC
- `omp_unset_nest_lock(var);` salida de la SC

Eventos

3. Eventos

La sincronización entre los *threads* puede hacerse esperando a que se produzca un determinado evento.

La sincronización puede ser:

- **global**: todos los *threads* se sincronizan en un punto determinado.
- **punto a punto**: los *threads* se sincronizan uno a uno a través de *flags*.

Eventos (barreras)

► Sincronización global: barreras

```
#pragma omp barrier
```

Típica barrera de sincronización global para todos los *threads* de una región paralela.

Recordad que muchos constructores paralelos llevan implícita una barrera final.

Eventos (barreras)

> Ejemplo

```
#pragma omp parallel private(tid)
{
    tid = omp_get_thread_num();
    A[tid] = fun(tid);
    #pragma omp barrier

    #pragma omp for
    for (i=0; i<N; i++) B[i] = fun(A,i);
    #pragma omp for nowait
    for (i=0; i<N; i++) C[i] = fun(A,B,i);
    D[tid] = fun(tid);
}
```

Eventos (*flags*)

► Sincronización punto a punto

La sincronización entre procesos puede hacerse mediante *flags* (memoria común), siguiendo un modelo de tipo **productor / consumidor**.

```
/* productor */  
...  
dat = ...;  
flag = 1;  
...
```

```
/* consumidor */  
...  
while (flag==0) { };  
... = dat;  
...
```



Eventos (*flags*)

Sin embargo, sabemos que el código anterior puede no funcionar correctamente en un sistema paralelo, dependiendo del **modelo de consistencia** de la máquina.

Tal vez sea necesario **desactivar las optimizaciones** del compilador antes del acceso a las variables de sincronización.

Eventos (*flags*)

Para asegurar que el modelo de consistencia aplicado es el secuencial, OpenMP ofrece como alternativa la directiva:

```
#pragma omp flush(X)
```

que marca puntos de consistencia en la visión de la memoria (*fence*).

```
/* productor */  
...  
dat = ...;  
#pragma omp flush(dat)  
flag = 1;  
#pragma omp flush(flag)  
...
```

```
/* consumidor */  
...  
while (flag==0)  
{ #pragma omp flush(flag) };  
#pragma omp flush(dat)  
... = dat;  
...
```

Eventos (*flags*)

El modelo de consistencia de OpenMP implica tener que realizar una operación de *flush* tras escribir y antes de leer cualquier variable compartida.

En C se puede conseguir esto declarando las variables de tipo `volatile`.

```
volatile int dat, flag;  
...  
  
/* productor */  
...  
dat = ...;  
flag = 1;  
...
```

```
volatile int dat, flag;  
...  
  
/* consumidor */  
...  
while (flag==0) {};  
... = dat;  
...
```

Eventos (ordered)

4. Secciones "ordenadas"

```
#pragma omp ordered
```

Junto con la cláusula `ordered`, impone el orden secuencial original en la ejecución de una parte de código de un `for` paralelo.

```
#pragma omp parallel for ordered
for (i=0; i<N; i++)
{
    A[i] = ... (cálculo);
    #pragma omp ordered
    printf("A(%d)= %d \n", i, A[i]);
}
```

Equivale a un `wait - post` construido mediante contadores.

Sincronización

5. Directiva **master**

```
#pragma omp master
```

Marca un bloque de código para que sea ejecutado solamente por el *thread* máster, el 0.

Es similar a la directiva **single**, pero sin incluir la barrera al final y sabiendo qué *thread* va a ejecutar el código.

Tareas

► Directiva **task**

Sirve para definir tareas explícitas, y permite gestionar eficientemente procedimientos recursivos y bucles en los que el número de iteraciones es indeterminado.

```
#pragma omp task [cláusulas]
{
}
```

cláusulas **if** (expresión)
untied
default (shared | none)
private (var), **firstprivate** (var), **shared** (var)

Tareas

► Directiva **task**

Una directiva que permite esperar a todas las tareas hijas generadas desde la tarea actual.

```
#pragma omp taskwait
```

La directiva **task** permite introducir paralelismo de una forma sencilla en ciertas situaciones en las que era difícil hacerlo. Veamos un par de ejemplos.

Tareas

> Ejemplo 1: lista ligada

```
...  
while (puntero)  
{  
    (void) ejecutar_tarea(puntero);  
    puntero = puntero->sig;  
}  
...
```

Sin la directiva **task**, habría que contar el número de iteraciones previamente para transformar el **while** en un **for**.

Tareas

> Ejemplo 1: lista ligada - openmp

```
puntero = cabecera;
#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(puntero) {
            #pragma omp task firstprivate(puntero)
            {
                (void) ejecutar_tarea(puntero);
            }
            puntero = puntero->sig ;
        }
    }
}
```

Tareas

> Ejemplo 2: fibonacci

```
long fibonacci(int n)
{ // f(0)=f(1)=1, f(n) = f(n-1) + f(n-2)

    long f1, f2, fn;

    if ( n == 0 || n == 1 ) return(n);

    f1 = fibonacci(n-1);
    f2 = fibonacci(n-2);

    fn = f1 + f2;

    return(fn);
}
```

Tareas

> Ejemplo 2: fibonacci - openmp

```
long fibonacci(int n)
{
    long f1, f2, fn;
    if ( n == 0 || n == 1 ) return(n);

    #pragma omp task shared(f1)
        {f1 = fibonacci(n-1);}

    #pragma omp task shared(f2)
        {f2 = fibonacci(n-2);}

    #pragma omp taskwait

    fn = f1 + f2;
    return(fn);
}
```

Tareas

> Ejemplo 2: fibonacci - openmp

```
#pragma omp parallel shared(nth)
{
    #pragma omp single nowait
    {
        result = fibonacci(n);
    }
}
```

Posibilidad de aplicar recursividad paralela a partir de un tamaño mínimo de cálculo?

Otras cuestiones

► Un par de funciones para “medir tiempos”

- `omp_get_wtime()` ;

```
t1 = omp_get_wtime();
```

```
...
```

```
t2 = omp_get_wtime();
```

```
tiempo = t2 - t1;
```

- `omp_get_wtick()` ; precisión del reloj

Otras cuestiones

- ▶ Programar aplicaciones SMP resulta “más sencillo” que repartir datos por diferentes procesadores y comunicarse por paso de mensajes.

Pero el uso de variables compartidas por varios *threads* puede llevar a **errores** no previstos si no se analiza detenidamente su comportamiento.

Algunos errores típicos pueden producir **carreras** (*races*) en los resultados o dejar bloqueada la ejecución (*deadlock*).

Otras cuestiones: carreras

► Carreras

Definimos una carrera (*race*) como la consecución de resultados inesperados e irreproducibles debido a problemas en el acceso y sincronización de variables compartidas.

```
#pragma omp parallel sections
{
    #pragma omp section
        A = B + C;

    #pragma omp section
        B = A + C;

    #pragma omp section
        C = B + A;
}
```

!?

Otras cuestiones: carreras

```
CONT = 0;
#pragma omp parallel sections
{ #pragma omp section
    A = B + C;
    #pragma omp flush (A)
    CONT = 1;
    #pragma omp flush (CONT)
#pragma omp section
{ while (CONT<1)
    { #pragma omp flush (CONT) }
    B = A + C;
    #pragma omp flush (B)
    CONT = 2;
    #pragma omp flush (CONT)
}
#pragma omp section
{ while (CONT<2)
    { #pragma omp flush (CONT) }
    C = B + A;
}
}
```

el contador permite la sincronización entre las secciones (eventos)

las operaciones de *flush* aseguran la consistencia de la memoria.

Otras cuestiones: carreras

```
#pragma omp parallel private(tid, X)
{
    tid = omp_get_thread_num();

    #pragma omp for reduction(+:total) nowait
    for (i=0; i<N; i++)
    {
        x = fun(i);
        total = total + x;
    }
    Y[tid] = fun2(tid, total);
}
```

!?

Otras cuestiones: deadlock

► ***Deadlock***

Si no se usan correctamente las operaciones de sincronización, es posible que el programa se bloquee (*deadlock*).

Por ejemplo, he aquí un par de errores típicos usando secciones críticas construidas mediante cerrojos:

Otras cuestiones: deadlock

(región paralela)

```
...
omp_set_lock(&C1);
A = A + func1();
if (A>0) omp_unset_lock(&C1);
else
{
    ...
}
...
```

(región paralela con secciones)

```
...
#pragma omp section
{ omp_set_lock(&C1);
  A = A + func1();
  omp_set_lock(&C2);
  B = B * A;
  omp_unset_lock(&C2);
  omp_unset_lock(&C1);
}

#pragma omp section
{ omp_set_lock(&C2);
  B = B + func2();
  omp_set_lock(&C1);
  A = A * B;
  omp_unset_lock(&C1);
  omp_unset_lock(&C2);
}
...
```

Otras cuestiones: deadlock

► Recomendaciones:

- prestar atención al ámbito de las variables: `shared`, `private`, etc.
- utilizar con cuidado las funciones de sincronización.
- disponer de una versión equivalente secuencial para comparar resultados (serán siempre iguales?).

Otras cuestiones

- ▶ Llamadas en paralelo a funciones de librería
¿habrá problemas con la activación simultánea de más de una instancia de dichas funciones?

Una librería es *thread-safe* (re-entrante) si lo anterior no es un problema. Si no es así, habría que utilizar una secuencia tipo:

LOCK / CALL / UNLOCK

Otras cuestiones: *speed-up*

► *Speed-up*

El objetivo de programar una aplicación en un sistema paralelo es:

- ejecutar el problema **más rápido**.
- ejecutar un problema de **mayor tamaño**.

En ambos casos hay que tener en cuenta el ***overhead*** añadido al paralelizar el código.

Limitaciones al rendimiento

- ▶ Escribir programas paralelos OpenMP es fácil... y también lo es escribir programas de bajo rendimiento.
- ▶ Principales fuentes de pérdida de eficiencia
 - el algoritmo en ejecución: **Amdahl** // **desequilibrio de carga**
 - sincronización: **grano** muy fino
 - comunicación: acceso a variables **shared**, **cache** (fallos, falsa compartición...)
 - implementación de OpenMP / S.O.

Limitaciones al rendimiento

► Ejemplos de mejora de la eficiencia:

```
#pragma omp parallel for
for (i=0;i<N;i++) { ... }

XMED = xnorm / sum;

#pragma omp parallel for
for (i=0;i<M;i++) { ... }
```

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i<N;i++) { ... }

    #pragma omp single
        XMED = xnorm / sum;

    #pragma omp for
    for (i=0;i<M;i++) { ... }
}
```

```
#pragma omp parallel
    private (XMED)
{
    #pragma omp for nowait
    for (i=0;i<N;i++) { ... }

    XMED = xnorm / sum;

    #pragma omp for
    for (i=0;i<M;i++) { ... }
}
```

una sola sección paralela

cálculo repetido

Limitaciones al rendimiento

► Ejemplos de mejora de la eficiencia

Utilizar el mismo reparto (**¡los mismos datos!**) para todos los bucles (hacer a “mano” el *work sharing*).

```
...
#pragma omp parallel ...
{
    ...
    ini = pid*N/npr;
    fin = (pid+1)*N/npr;

    for (i=ini; i<fin; i++) { ... }
    ...
    for (i=ini; i<fin; i++) { ... }
    ...
}
```

Otras cuestiones: *speed-up*

- ▶ Por ello, interesa minimizar costes de creación y terminación de *threads*, sincronización, etc.

Hay que considerar también los problemas de **localidad de los datos**.

Diferentes aspectos:

Otras cuestiones: *speed-up*

- **La máquina: SMP o DSM**

La localidad de los datos es un factor determinante en una máquina DSM.

También es importante en el caso SMP, para mejorar la tasa de acierto de la cache.

- **La compartición falsa de datos**

Aquí puede resultar importante el tipo de *scheduling* que se efectúe, para evitar continuas anulaciones de bloques de cache.

Otras cuestiones: *speed-up*

- ▶ En resumen, es necesario un cuidadoso análisis del reparto y uso de los datos, el tamaño de grano, la sincronización de los *threads*, el uso de memoria, etc., para minimizar *overheads* (p.e., el uso de barreras implícitas) y conseguir el máximo rendimiento de un sistema paralelo.

Existen herramientas que ayudan en estas tareas:
KAI, PORTLAND...