Programación en memoria compartida con OpenMP

Jesús Sánchez Cuadrado

Resumen. En este documento se describen algunas técnicas de programación en memoria compartida y cómo implementarlas con OpenMP. Los objetivos principales de este capítulo son:

- Compilar y ejecutar programas OpenMP
- Utilización de directivas OpenMP
- Paralelismo estructurado
- Paralelismo no estructurado con tareas

Índice

1. Introducción

En el modelo de programación paralela denominado *memoria compartida* los procesos comparten un espacio de direcciones global desde donde pueden leer y escribir datos de manera asíncrona como medio de comunicarse. Un aspecto fundamental es que son son necesarios mecanismos de sincronización para poder garantizar que se accede a la memoria de manera correcta y que los procesos actúan de manera coordinada.

Dependiendo del lenguaje y/o framework de paralelismo utilizando el programador puede especificar el paralelismo con un estilo explícito o implícito. En el estilo explicíto deben crearse y gestionarse "manualmente" los procesos o hilos de ejecución. Por ejemplo, en Java antes de las versiones 7 y 8 ese era el estilo habitual de programación. En el estilo implícito el compilador se encarga de la paralelización automáticamente, guiado a través de anotaciones o directivas proporcionadas por el programador.

En este capítulo se presenta la programación paralela en memoria compartida utilizando OpenMP.

1.1. ¿Qué es OpenMP

OpenMP es una API para implementar programas paralelos en sistemas de memoria compartida. Se trata de una especificación que es mantenida por organizaciones interesadas en el desarrollo de la computación paralela como Intel, IBM, AMD, etc. La especificación, así como documentación y recursos de aprendizaje, está disponible en `https://www.openmp.org/`. Al ser una especificación, las organizaciones ofrecen implementaciones para sus plataformas, algunas gratuitas y otras de pago. La especificación está orientada a los lenguajes C, C++ y Fortran, pero incluso hay adaptaciones para Java (ej., `https://github.com/omp4j/omp4j`).

El estilo de programación de OpenMP, al contrario que otras API como Pthreads, se basa en utilizar directivas del compilador junto con algunas funciones de librería. Mientras que en Pthreads el programador debe especificar explícitamente el comportamiento de cada hilo, en OpenMP el paralelismo "se declara" a través de las directivas y se deja al compilador que construya y organice los hilos necesarios para llevarlo a cabo. Así, Pthreads utiliza paralelismo explícito y OpenMP paralelismo implícito. En Pthreads y librerías de hilos similares (por ejemplo, hilos estándar de Java) el programador tiene un control de bajo nivel sobre el comportamiento de los hilos, pero ello implica la necesidad de tener en cuenta todos los detalles de la gestión del paralelismo. OpenMP, en cambio, al ser más declarativo facilita el desarrollo de muchos tipos de programas paralelos, pero es más difícil afinar cuestiones de más bajo nivel. OpenMP utiliza el modelo ejecución *fork-join* como paradigma básico. A partir de las version 3.0 también tiene soporte para el paradigma basado en tareas.

Otra diferencia fundamental es que Pthreads es una librería y puede usarse en cualquier arquitectura en que esté disponible. En cambio, OpenMP requiere soporte del compilador y por tanto sólo está disponible en aquellos entornos para los que exista un compilador que lo soporte. Afortunadamente, es en la mayoría.

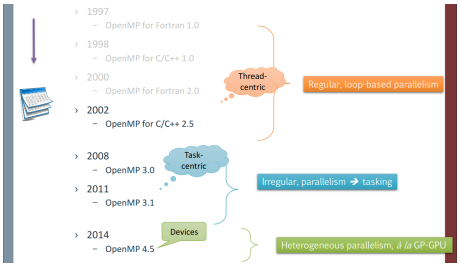

Figura 1: Historia de OpenMP (tomada de `http://algo.ing.unimo.it/people/andrea/Didattica/HPC/SlidesPDF/10.%20OMP%20tasks.pdf`)

2. Utilización básica de OpenMP

OpenMP está basado en el uso de directivas del compilador para indicar qué elementos del programa deben ejecutarse en paralelo, cómo realizar la sincronización, etc. Además de las directivas, las distribuciones de OpenMP tienen una librería estándar.

2.1. Ejemplo

La filosofía de OpenMP es realizar programas paralelos de la manera más declarativa posible. El siguiente código es un ejemplo sencillo. La librería estándar de OpenMP está definida en el fichero `omp.h` (ver sección 2.17). Es necesario incluirla sólo si se utilizan funciones de OpenMP (no confundir con las directivas). La directiva `parallel` permite indicar que el siguiente bloque de código debe ser ejecutado en paralelo por los hilos con los que esté configurado el sistema.

```c
#include <stdio.h>

int main(void) {
  printf("Hello\n");
  #pragma omp parallel
  {
    // El codigo del bloque se ejecuta en paralelo
    printf("world!\n");
  }
  printf("Bye!\n");
  return 0;
}
```

Al ejecutar el programa anterior con, por ejemplo, 2 hilos la salida será la siguiente.

```
Hello
world
world
Bye!
```

OpenMP se encarga de generar dos hilos, asignarles trabajo (imprimir `"world"`) y sincronizarlos al terminar para que el programa pueda continuar, en este caso imprimir `"Bye!"`.

La simplicidad de este código contrasta con lo complejo que puede llegar a ser programar con librerías de bajo nivel como pthreads (ver figura **??**).

Directivas y funciones

Las directivas tienen la forma: `#pragma omp <nombre> <args>` y las interpreta el compilador para transformar el código del programa según corresponda. Si no se activa el soporte para las directivas, se ignoran.

Las funciones tienen la forma `omp_nombre(args)` y están definidas en `omp.h`. Si se usan estas funciones hay que incluir la librería y enlazarlas.

omp/helloworld.c

2.2. Compilación

Los programas OpenMP requieren soporte especial del compilador para procesar sus directivas. Si se compila sin ese soporte las directivas no tienen efecto y el programa no trabaja en paralelo.

La opción de `gcc -fopenmp` se encarga de activar el módulo del compilador que realiza el procesamiento de las directivas y de enlazar la librería estandar al generar el ejecutable.

Por ejemplo:

```
gcc -fopenmp -o hello_world hello_word.c
```

Si no se indica el parámetro `-fopenmp` es posible que el programa compile si no se ha incluido la librería `omp.h` (ej., en programas muy básicos donde no haga falta). En este caso el compilador simplemente ignora las directivas y el programa se ejecutará sin paralelismo.

2.3. Ejecución

La ejecución de un programa en OpenMP se realiza como cualquier otro programa. La principal cuestión a tener en cuenta es cómo configurar el número de hilos que se asignan a la ejecución. Hay tres maneras de realizar esto:

- Utilizando una variable de entorno. Por ejemplo,

  ```
  $ OMP_NUM_THREADS=4 ./hello_world
  Hello
  ```

```
world
world
world
world
Bye!
```

- Utilizando la función `omp_set_num_threads`.
- Utilizando la directiva `num_threads(n)`.

Al ejecutar este programa con, por ejemplo, 4 hilos la salida será la siguiente.

```
$ OMP_NUM_THREADS=4 ./hello_world
Hello
world
world
world
```

Es posible anidar directivas `parallel` para conseguir paralelismo anidado, como se verá en la Sección **??**.

La directiva `parallel` no prescribe cómo debe distribuirse el trabajo entre los hilos, y por tanto todos los hilos ejecutan el mismo código. OpenMP proporciona directivas adicionales que permiten especificar cómo dividir el trabajo entre los hilos.

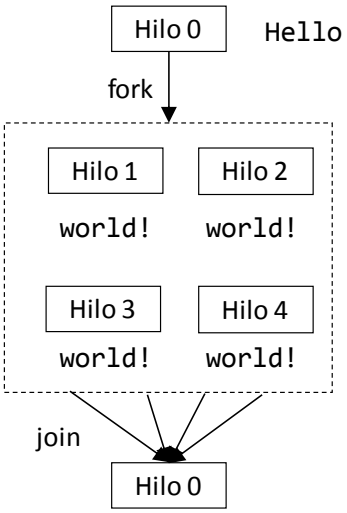

Figura 2: Funcionamiento de `parallel`

2.4. Manejo de variables

Por defecto las variables definidas fuera del bloque paralelo son compartidas por todos los hilos. En este caso se dice que es una variable de tipo `shared`. existen claúsulas para indicar explícitamente el ámbito y la forma de uso de las variables, son las siguientes.

`private(vars)`. Las variables privadas solo son accesibles por cada hilo, es decir, es una variable local al hilo. Se pueden leer y escribir sin que existan problemas de sincronización. Es muy importante tener en cuenta que **no se inicializan antes de comenzar** ni se guarda su valor al finalizar.

`firstprivate(vars)`. Son también variables privadas a los hilos, pero su valor se inicializa con el valor que tuviera la variable original de la que provienen. Por ejemplo, en el siguiente programa, si se ejecuta con 4 hilos, el programa imprimirá 4 veces el valor 10.

```c
int var = 10;
#pragma omp parallel firstprivate(var)
{
  printf("Valor = %d\n", var);
}

```

`lastprivate(vars)`. Son también variables privadas a los hilos, pero al finalizar la ejecución de su valor su valor se inicializa con el valor que tuviera la última iteración (cuando se ejecuta en el contexto de un bucle `for`) o la última sección.

`shared(vars)`. Se indica que la variable es compartida por todos los hilos. Es el valor por defecto, pero es conveniente indicarlo explícitamente para evitar confusiones. Las variables compartidas requieren especial cuidado, puesto que su acceso requiere de mecanismos de sincronización (p.ej., una sección crítica).

`default(shared | none)`. Indica cómo serán las variables por defecto. Puede ser buena idea indicar siempre `default(none)` para que el compilador fuerce a indicar las claúsulas private, firstprivate o shared.

Variables privadas

¿Qué valor se imprimirá?

```c
int var = 10;
#pragma omp parallel private(var)
{
  printf("%d\n", var);
}

```

El valor es indeterminado, no se puede saber a priori porque la variable no se inicializa.

2.5. Directiva for

La directiva `for` permite ejecutar bucles en paralelo. Para ello especifica que el trabajo de un bucle *for* (i.e., las iteraciones) deben distribuirse entre los hilos. El ejemplo siguiente ejecuta calcula una tabla con 360 valores de la función *seno* en paralelo.

omp/sintable.c

```c
#include <stdio.h>
#include <math.h>

#define SIZE 360

int main(void) {
  double sin_table[SIZE];

  #pragma omp parallel for
  for(int i = 0; i < SIZE; i++)
  {
    sin_table[i] = sin(2 * M_PI * i / SIZE);
  }

  printf("sin(%d) = %.2f\n", 0, sin_table[0]);
  printf("sin(%d) = %.2f\n", 45, sin_table[45]);
  printf("sin(%d) = %.2f\n", 90, sin_table[90]);
  printf("sin(%d) = %.2f\n", 180, sin_table[180]);

  return 0;
}
```

Es obligatorio que la directiva `for` esté dentro una región paralela o no se ejecutará en paralelo. Esto quiere decir que estas dos construcciones son equivalentes:

```c
#pragma omp parallel
#pragma omp for
for(...) { }

/* Esto es equivalente al anterior */
#pragma omp parallel for
for(...) { }
```

Internamente el compilador de OpenMP genera un código similar al siguiente, que es ejecutado por cada hilo de forma independiente.. En esencia, lo que hace el compilador es dividir el rango del bucle en tantas partes como hilos se van a utilizar. El "truco" para que este código funcione reside en que la función `omp_get_thread_num()` accede a una variable que es *thread local*, es decir, su valor es diferente en cada hilo.

```c
int this_thread = omp_get_thread_num();
int num_threads = omp_get_num_threads();
int my_start = (this_thread  ) * SIZE / num_threads;
int my_end   = (this_thread+1) * SIZE / num_threads;
for(int i=my_start; i<my_end; ++n)
  sin_table[i] = sin(2 * M_PI * i / SIZE);
```

La principal limitación de esta directiva es que solo puede paralelizar bucles de forma canónica, es decir, dado un búcle con la forma `for(index = start; index` cmp `end; index` op `incr)`, se deben cumplir las siguientes condiciones.

- cmp es una operación de comparación y op una operación aritmética.
- La variable índice (`index`) debe ser de tipo entero o puntero.
- Las expresiones `start`, `end`, e `incr` deben ser del tipo compatible (ej., `index` es un puntero, `incr` debe ser un entero).
- Los valores de `start`, `end` e `incr` no pueden cambiar durante la ejecución del bucle.

$$\texttt{for}\left(\texttt{index = start}\ ;\ \begin{array}{l}\texttt{index < end}\\ \texttt{index <= end}\\ \texttt{index >= end}\\ \texttt{index > end}\end{array}\ ;\ \begin{array}{l}\texttt{index++}\\ \texttt{++index}\\ \texttt{index--}\\ \texttt{--index}\\ \texttt{index += incr}\\ \texttt{index -= incr}\\ \texttt{index = index + incr}\\ \texttt{index = incr + index}\\ \texttt{index = index - incr}\end{array}\right)$$

Figura 3: Formas válidas de la directiva `for`

- La variable `index` sólo puede cambiarse a través de la operación de incremento de la sentencia for.

2.6. Loop scheduling

Para organizar el trabajo de los hilos al paralelizar un bucle `for` OpenMP debe decidir qué iteraciones debe asignar a cada hilo. Se pueden aplicar varios políticas, y cada política puede ser más adecuada en un caso u otro.

La claúsula `schedule` permite configurar el tipo de *scheduling* que se usará. Admite cinco valores: `static`, `dynamic`, `guided`, `auto`, `runtime`.

`schedule(static, tamaño)`. Las iteraciones se dividen según un tamaño fijo y la asignación del trabajo se hace estáticamente. Es decir, al comenzar el bucle se asigna a cada hilo cuantas iteraciones y de qué tamaño hará. Si no se indica el tamaño, se calcula en función del tamaño total del bucle.

La figura 4 muestra un ejemplo de un bucle con 24 iteraciones cuyo trabajo se reparte entre 3 hilos. Como se ha indicado tamaño 4, cada hilo realiza 4 iteraciones y siempre realizará esas mismas iteraciones independientemente de cuándo las termine. Eso significa que aunque un hilo termine todo su trabajo mucho antes que los demás, no recibirá más trabajo porque no se le ha sido asignado de antemano.

```
#pragma omp parallel for schedule(static, 4)
for (…) { }
```

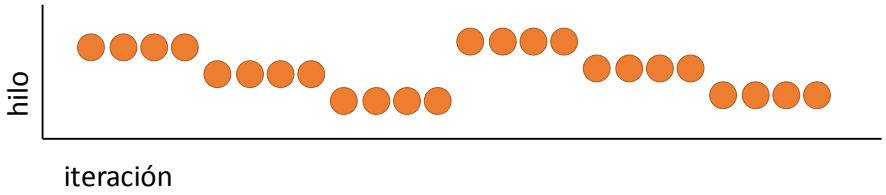

Figura 4: Funcionamiento de la claúsula `schedule(static)`.

`schedule(dynamic, tamaño)`. Con esta política cada hilo va pidiendo trabajo cuando está ocioso. El tamaño del trabajo sigue siendo fijo, pero si un hilo termina antes que los demás puede continuar realizando trabajo útil. Esta política es adecuada cuando cada iteración puede tener un coste de ejecución diferente. Su principal inconveniente es que hay una sobrecarga asociada a la asignación dinámica de trabajo.

La figura 5 muestra un ejemplo de un bucle con 24 iteraciones cuyo trabajo se reparte entre 3 hilos. El tamaño de cada trabajo es de 2 iteraciones. Todos los hilos comienzan procesando 2 iteraciones cada vez, pero (en el ejemplo) el hilo 1 termina el procesamiento de la iteración 10 antes de que el resto de hilos hayan terminado su trabajo. Por tanto, se le asignan las iteraciones 11 y 12. Al contrario que con la claúsula *static*, qué iteraciones ejecutan qué hilos se va calculando dinámicamente a medida que se ejecuta el búcle.

En ocasiones puede suceder que el coste de las iteraciones sea mayor a medida que el bucle progresa. En esos casos puede ser conveniente utilizar `guided` en lugar de `dynamic`.

`schedule(guided)`. La política `guided` es también una política dinámica, pero en este caso la cantidad de trabajo es asignada de manera dinámica en función del trabajo que queda por realizar. En particular, cada vez que un hilo toma nuevo trabajo, el tamaño de éste es proporcional al número de iteraciones que quedan sin asignar dividido por el número de hilos, esto es: $iteracionesSinAsignar/numHilos$. La figura 6 muestra un ejemplo con 24 iteraciones y 3 hilos.

Ejemplo de uso de guided

La claúsula `guided` es recomendable cuando el coste de la iteración es mayor al final de la ejecución del bucle.

```
1 #pragma omp parallel for
2 for(int i = 0;i <= n;i++) {
3   factorial(i);
4 }
5
```

```
#pragma omp parallel for schedule(dynamic, 2)
for (…) { }
```

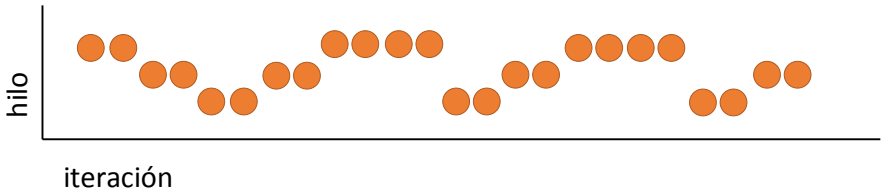

Figura 5: Funcionamiento de la claúsula `schedule(dynamic)`.

```
#pragma omp parallel for schedule(guided)
for (…) { }
```

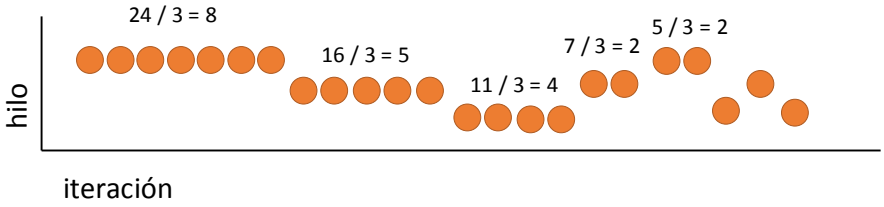

Figura 6: Funcionamiento de la claúsula `schedule(guided)`.

Por otra parte, las diferentes cláusulas, `static`, `dynamic` y `guided`, pueden tener un efecto importante dependiendo de la arquitectura destino[1], por lo que es conveniente realizar pruebas de rendimiento para entender cuál puede convenir mejor en cada caso.

2.7. Sections

La directiva `sections` es útil cuando se desea ejecutar dos o más bloques de código diferente en paralelo.

Si hay más hilos que secciones, entonces algunos quedan ociosos (idle). Si hay menos hilos que secciones, entonces algunas secciones deben esperar para ser ejecutadas cuando acaben una de las primeras.

2.8. Secciones críticas

El acceso a variables compartidas debe realizarse en exclusión mutua, es decir, debe garantizarse que solo uno de los hilos lea o escriba una variable compartida.

Una **condición de carrera** ocurre cuando hay dos o más hilos que pueden acceder a un dato compartido y cambiarlo al mismo tiempo, y este comportamiento puede dar lugar a resultados incorrectos. El sistema operativo puede ejecutar los hilos en cualquier orden por lo que no es posible saber en qué orden se producirán estos accesos y el resultado del programa puede ser diferente en diferentes ejecuciones. Por eso se dice que los hilos están compitiendo (en una carrera) para acceder o cambiar el dato.

Por ejemplo, el siguiente código accede a una variable compartida de manera **incorrecta**. El problema radica en la actualización de la variable `v`. Esta actualización implica leer `v`, hacer una suma y escribirla. Puesto que los hilos se están ejecutando de manera paralela, es posible que varios hilos estén leyendo el mismo valor de v, que estará desactualizado. Esto se ilustra en la figura 7.

[1] https://stackoverflow.com/questions/10850155/whats-the-difference-between-static-and-dynamic-schedule-in-openmp

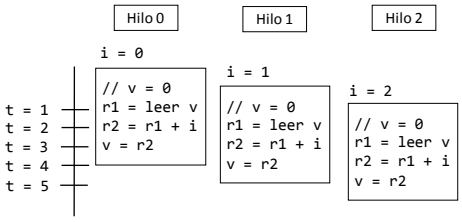

Figura 7: Intercalación de hilos que provoca un comportamiento incorrecto.

```
void main(void) {
  int v = 0;
  const int n = 100;

  #pragma omp parallel for
  for(int i = 0;i <= n;i++) {
    if (i % 2 == 0) {
      v += i;
    }
  }

  printf("v = %d\n", v);
}
```

Para comprobar este comportamiento podemos ejecutar el código anterior con diferente cantidad de hilos. A continuación se muestran los resultados de algunas de estas ejecuciones. Al ejecutar con un hilo el resultado es el esperado.

```
$ OMP_NUM_THREADS=1 ./main  => v = 2550  ✔
$ OMP_NUM_THREADS=1 ./main  => v = 2550  ✔
$ OMP_NUM_THREADS=2 ./main  => v = 2550  ✔
$ OMP_NUM_THREADS=2 ./main  => v = 2406  ✖
$ OMP_NUM_THREADS=3 ./main  => v = 2550  ✔
$ OMP_NUM_THREADS=3 ./main  => v = 850   ✖
$ OMP_NUM_THREADS=3 ./main  => v = 850   ✖
$ OMP_NUM_THREADS=4 ./main  => v = 2056  ✖
$ OMP_NUM_THREADS=4 ./main  => v = 2414  ✖
```

Para solucionar este problema hay dos estrategias básicas: a) que los hilos accedan en exclusión mutua a los datos compartidos, o b) reescribir los algoritmos para evitar el problema de sincronización.

Una sección crítica permite abordar la primera estrategia. Para ello la claúsula `critical` define una sección crítica, es decir, un fragmento de código en el que es seguro puede acceder una variable compartida. El código anterior puede reescribirse de la siguiente manera:

```
void main(void) {
  int v = 0;
  const int n = 100;

  #pragma omp parallel for
  for(int i = 0;i <= n;i++) {
    if (i % 2 == 0) {
      #pragma omp critical
      v += i;
    }
  }

  printf("v = %d\n", v);
}
```

De esta manera en el mismo instante de tiempo sólo un hilo ejecutará la sentencia `v += i`. Esto significa que el resto de hilos estará esperando para poder entrar a la sección crítica. Por tanto, una región crítica en efecto "serializa" la ejecución en ese punto, convirtiéndolo en secuencial, lo que provocará un decremento en el rendimiento. Es por ello que es importante que las secciones críticas sean lo más pequeña posibles.

Una alternativa al ejemplo anterior es la siguiente:

```
int v = 0;
const int n = 100;
#pragma omp parallel shared(n,v) private(sumLocal)
{
  int TID = omp_get_thread_num();
  int v_local = 0;
  #pragma omp for
  for (i=0; i<=n; i++)
    if (i % 2 == 0)
```

Errores de sincronización

Detectar errores a la hora de sincronizar el acceso a variables compartidas puede ser complicado. Diferentes ejecuciones del mismo código pueden dar resultados diferentes. Es más, en ocasiones el resultado del algoritmo puede ser correcto y otros no.
Es muy importante prestar atención a este aspecto para escribir algoritmos paralelos correctos.

Formato `critical`

`#pragma omp critical \`
`[(nombre)]`
El nombre es opcional, si se especifica permite tener más de una región crítica en el programa. Es decir, dos regiones críticas con nombres diferentes se podrían ejecutar a la vez.

```
    v_local += i;

  /* Cada hilo actualiza la variable en exclusion mutua */
  #pragma omp critical (update_sum)
  {
    v += v_local;
  }
}

printf("v = %d\n", v);
```

2.9. Directiva `atomic`

La actualización de variables a través de una operación aritmética no es atómica, es decir, está compuesta de al menos dos operaciones a nivel hardware. Por ejemplo, la operación `x += 2` a nivel hardware típicamente se descompone en: `r1 = load x; r2 = r1 + 2; store r2, x`. Esto puede provocar problemas de sincronización cómo los mencionados en la sección anterior.

Sin embargo, los procesadores actuales ofrecen instrucciones especiales que permiten realizar algunas de estas operaciones simples de manera atómica. La directiva `atomic` permite indica que se usen este tipo de instrucciones, evitando el uso de una sección crítica. Por ejemplo:

```
void main(void) {
  int v = 0;
  const int n = 100;

  #pragma omp parallel for
  for(int i = 0;i <= n;i++) {
    if (i % 2 == 0) {
      #pragma omp atomic
      v += i;
    }
  }

  printf("v = %d\n", v);
}
```

La ventaja de las instrucciones atómicas es que son más rápidas que una sección crítica, pero están limitadas a unos pocos casos, descritos en la tabla 1.

2.10. Locks

Un lock (o mutex o candado) es un mecanismo de sincronización para limitar el acceso a un recurso compartido. Utilizando un lock es posible definir una sección crítica. Aunque la directiva `critical` permite realizar la misma tarea, el uso de objetos lock explícitos puede proporcionar más flexilibidad en algunos casos.

En OpenMP un lock es un objeto de tipo `omp_lock` y sus funciones asociadas son las siguientes:

- `void omp_init_lock(omp_lock_t *lock)`. Para inicializar un objeto de tipo `omp_lock`. Un lock se inicializa como no bloqueado.
- `void omp_init_destroy(omp_lock_t *lock)`. Para destruir lock cuando no es necesario más.
- `void omp_set_lock(omp_lock_t *lock)`. Se utiliza para adquirir un lock de manera bloqueante. Es decir, si ya hay un hilo que ha adquirido el lock previamente y todavía no lo ha liberado, el hilo que está intentando adquirirlo se bloquea y se pone en una cola a esperar su turnu para acceder a la sección crítica.

Ejercicio

Implementa una solución similar a la anterior pero utilizando un array para almacenar los valores temporales y así evitar el uso de la sección crítica.
Compara los tiempos de ejecución, modificando ligeramente los ejemplos para que hagan trabajo útil y más costoso.

`x <op> = expresion` donde `<op>` = +, *, -, /, &, ^, |, <<, >>
x++
x−−
++x
−−x

Cuadro 1: Operadores soportados por `atomic`.

- `void omp_unset_lock(omp_lock_t *lock)`. Libera el lock para que otro hilo que pueda estar esperando entre en la sección crítica.
- `int omp_test_lock(omp_lock_t *lock)`. Intenta adquirir el lock pero no se bloquea. Si no lo consigue devuelve 0, y el hilo puede continuar haciendo otra tarea en lugar de quedar bloqueado.

El siguiente listado usa algunas de estas funciones:

```
int main()
{
  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);
    // Trabajo que debe realizarse en exclusion mutua
    printf("Hilo %d ejecutando en exclusion mutua\n", id);
    omp_unset_lock(&lck);

    while (! omp_test_lock(&lck))
    {
      // Realizar trabjo util hasta adquirir el lock
    }

    // Realizar trabajo en exclusion mutua

    omp_unset_lock(&lck);

  }

  omp_destroy_lock(&lck);
}
```

2.11. Barreras

Un barrera establece un punto en la ejecución del programa en el que los hilos deben detenerse hasta que el resto de hilos lleguen a ese punto. Es un mecanismo básico para asegurar que todos los hilos han terminado sus cálculos antes de proceder realizando nuevo trabajo que dependa de esos cálculos.

Por ejemplo, supongamos que tenemos un array de 4 elementos y queremos ejecutar un cálculo de manera iterativa que depende de los valores calculados en otras celdas del array. Tras cada cálculo, es necesario esperar a que el resto de hilos terminen para obtener su valor calculado.

```
int v[4] = ...

#pragma omp parallel num_threads(4)
{
  int tid = omp_get_thread_num();

  // Todos los hilos ejecutaran las 20
  // iteraciones de este bucle
  for(int i = 0; i < 20; i++) {
    int valor = calculo(x[mytid);

    #pragma omp barrier
    /* Esperar a que todos los hilos hagan el calculo */

    /* Tomar el valor del vecino */
    x[tid] = x[(tid+1) % 4] + 1;
```

```
  }
}
```

2.12. Directiva single

Dentro de un bloque `parallel` sirve para asegurar que el bloque de código solo sea ejecutado por un único hilo (puede ser cualquiera de los del equipo).

Un ejemplo típico de su uso es asegurar que sólo un hilo es el encargado de escribir el resultado en pantalla o en otro recurso externo.

```
int v = ...;
#pragma omp parallel
{
    v = calculo_paralelo();
    #pragma omp single
    {
      printf(" %d\n", v);
    }
}
```

Con la directiva `single` hay un barrera implícita al finalizar, lo que hace que todos los hilos esperen a que el hilo que está trabajando solo termine para poder continuar. Si se desea evitar esta barrera es necesario usar la opción `single nowait`.

2.13. Directiva master

El bloque de código solo será ejecutado por el hilo maestro. A diferencia de la directiva `single`, no hay barrera implícita al terminar. Eso significa que el resto de hilos "se saltan" el bloque de código y continúan su trabajo sin esperar al hilo maestro.

2.14. Opción no nowait

Al final de las directivas de reparto del trabajo (work sharing), como `for` o `sections` existe una barrera implícita, esto es, los hilos que van terminando quedan esperando al resto de hilos antes de continuar. Esta barrera puede eliminarse utilizando la opción `nowait`. De esta manera los hilos que han terminado pueden continuar cuanto antes ejecutando el resto de del código de la sección paralela.

Por ejemplo,

```
#pragma omp parallel
{
  x = local_computation()
#pragma omp for schedule(static) nowait
  for (i=0; i<N; i++) {
    x[i] = ...
  }
#pragma omp for schedule(static)
  for (i=0; i<N; i++) {
    y[i] = ... x[i] ...
  }
}
```

2.15. Directiva `reduction`

En ocasiones, el cálculo que debe realizarse en un bucle implica agregar los valores de una operación (ej., suma, resta, multiplicación) sobre una variable, como en el ejemplo anterior.

Para esos casos es posible utilizar la cláusula `reduction`, indicando la variable de agregación y la operación, de manera que OpenMP pueda generar código eficiente.

Op.	Descripción	Valor inicial
+	suma	0
*	producto	1
−	resta	0
&	bit-and	1111111...
|	bit-or	0
^	bit-xor	0
&&	and	1
||	or	0

Cuadro 2: Operadores soportados por `reduction`

El ejemplo anterior puede resolverse fácilmente utilizando esta cláusula:

```
void main(void) {
  int v = 0;
  const int n = 100;

  #pragma omp parallel for shared(n) reduction(+:v)
  for(int i = 0;i <= n;i++) {
    if (i % 2 == 0) {
      v += i;
    }
  }

  printf("v = %d\n", v);}
```

La claúsula reduction por tanto realiza el siguiente trabajo:

- Crea una copia local de la variable reducida en cada hilo, inicializada adecuadamente según el operador elegido (ver tabla 2).
- Cada hilo ejecuta las itearciones del búcle que le correspondan y van calculando su propio valor local.
- Al finalizar la ejecución de todos los bucles por parte de los hilos los valores se combinan según el operador.

2.16. Directiva sections

La directiva sections permite configurar un número fijo de tareas y asignarlas a hilos diferentes. El siguiente es un ejemplo de uso. La figura 8 muestra gráficamente su funcionamiento.

```
int main()
{
   #pragma omp parallel sections
   {
      #pragma omp section
      function_1();

      #pragma omp section
      function_2();
   }

   return 0;
}
```

La directiva `sections` se encarga de distribuir cada bloque entre los hilos del equipo, y cada bloque se ejecuta una sola vez y hay un barrera implícita al finalizar la ejecución. Si hay más tareas que hilos, algunos hilos ejecutarán varias tareas y si hay más hilos que tareas algunos hilos permenecerán ociosos.

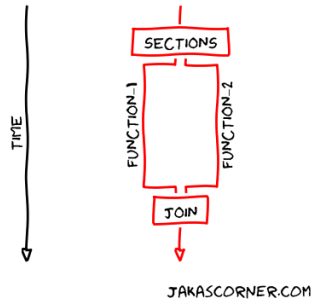

Figura 8: Funcionamiento de `sections`. Fuente: `jakascorner.com`.

2.17. Otras funciones

- `void omp_set_num_threads(int num_threads)`. Establece el número de hilos a usar en la siguiente región paralela.
- `int omp_get_num_threads(void)`. Obtiene el número de hilos que se están usando en una región paralela.
- `int omp_get_max_threads(void)`. Obtiene la máxima cantidad posible de hilos.
- `int omp_get_thread_num(void)`. Devuelve el identificador del hilo que se está ejecutando.
- `int omp_get_num_procs(void)`. Devuelve el máximo número de procesadores que tiene el sistema.

- `int omp_in_parallel(void)`. Devuelve valor distinto de cero si se está ejecutando dentro de una región paralela.
- `int omp_set_dynamic(void)`. Para permitir o no permitir que el número de threads se pueda ajustar dinámicamente en las regiones paralelas.
- `int omp_get_dynamic(void)`. Devuelve un valor distinto de cero si está permitido el ajuste dinámico del número de threads.
- `int omp_set_nested(int)`. Para permitir o no el paralelismo anidado
- `int omp_get_nested(void)`. Devuelve un valor distinto de cero si está permitido el paralelismo anidado

3. Consistencia de memoria

En los procesadores actuales las intrucciones no siempre se ejecutan en el orden en que el programador las ha escrito. La razón es que tanto los compiladores y los compiladores tienen libertad para reorganizar las instrucciones con el objetivo de obtener el máximo rendimiento posible. La única restricción es que el resultado de la ejecución debe ser el mismo que con la ejecución secuencial sin reordenación. Además, es posible que los valores de las variables estén temporalmente en registros o en una cola para ser escritas en cache, por lo que tales valores no necesariamente serían visibles en otros cores.

Supongamos el siguiente trozo de código:

```
1 a = b = c = d = 0;
2 #pragma omp sections
3 {
4 #pragma omp section
5   { a = 1; c = b; }
6 #pragma omp section
7   { b = 1; d = a; }
8 }
```

Si ejecutamos este código "en paralelo" intercalando de todas las formas posibles las ejecuciones secuenciales del mismo, las variables podrían tener los siguientes valores: `(c = 1, d = 1)`, `(c = 0, d = 1)` y `(c = 1, d = 0)`. Esta es la noción de consistencia secuencial.

Sin embargo, hay dos escenarios que pueden suceder en la práctica en los que el valor final de las variables podrían ser `(c = 0, d = 0)`. Esto puede ser debido a:

- Las variables `a` y `b` podrían no ser escritas immediatamente en memoria, por lo que los valores de `c` y `d` se actualizarían con un valor antiguo (e incorrecto).
- El compilador podría reordenar las asignaciones.

Por esta razón es importante disponer de ciertas nociones acerca del modelo de memoria utilizado por el lenguaje y/o librería de paralelismo.

3.1. El modelo de memoria de OpenMP

OpenMP tiene un modelo de memoria compartida "relajado" (*relaxed memory model*[2] [2]. Todos los hilos pueden escribir y leer variables en la memoria compartida, pero además cada hilo tiene una visión temporal de la memoria. Esta visión temporal es lo que convierte al modelo en relajado, y es necesaria para evitar que cada escritura y lectura de una variable implique ir a buscarla a memoria principal.

[2] `https://www.openmp.org/spec-html/5.0/openmpsu9.html`

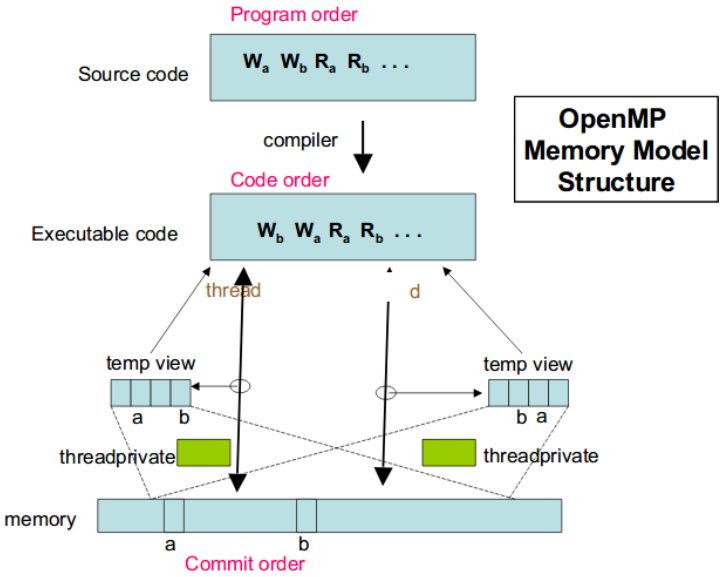

Figura 9: Esquema del modelo de memoria de OpenMP. Tomado de `http://www.nic.uoregon.edu/iwomp2005/Talks/hoeflinger.pdf`

La figura 9 muestra cómo se organiza el modelo de memoria. Un programa está compuesto de instrucciones que el compilador o el propio procesador tiene la libertad de reordenar. Cada hilo tiene una vista temporal de las variables compartidas, de manera que pueden no estar actualizados. El programador es responsable de insertar las directivas adecuadas para garantizar la sincronización de los valores de las variables cuando sea necesario.

En particular, OpenMP define la operación *flush* como la única manera en la que se garanza que un valor se moverá (se hará visible) de un hilo a otro. Para que esto suceda OpenMP requiere que se realicen estas cuatro acciones en este orden:

1. El primer hilo escribe el valor en una variable compartida.
2. El primer hilo hace un flush de la variable.
3. El segundo hilo hace un flush de la variable.
4. El segundo hilo lee la variable.

OpenMP proporciona dos maneras de realizar la operación *flush*.

- Explícitamente a través de la directiva `#pragma omp flush`.
- Implícitamente en otras directivas como son: barreras, al entrar y salir de una región paralela, de un región crítica, de `ordered`, de parallel `for` y de `section` (a no ser que se use `nowait`, antes y después de `atomic` y también al realizar un lock explícito usando las funciones de la API.

Operación flush

La operación *flush* se realiza explícitamente a través de la directiva `flush` o implícitamente en otras directivas.
La existencia de una vista temporal de variables permite a las implementaciones decidir cuándo sincronizar las variables con la memoria principal (a través de una barrera de memoria; *memory fence*). En el siguiente código el valor `a = 1` podría hacerse disponible a los otros hilos tan pronto como la línea 1 o tan tarde como la línea 3. Esta flexilibidad permite amortizar la latencia del acceso a memoria.

```
1 a = 1
2 ... otras operaciones ...
3 flush(a)
```

3.2. Directiva flush

OpenMP proporciona la directiva `flush` para forzar la sincronización de las variables usadas por los hilos.

Supongamos que queremos implementar un algoritmo al estilo productor-consumidor, en el que un hilo produce datos que son consumidos por otro hilo. Se podría implementar un algoritmo como el siguiente (ver [3]).

```
int N = 10000;
double *A = (double *) malloc(N*sizeof(double));
int sum;

#pragma omp parallel sections
{
  // productor
  #pragma omp section
  {
    fill_rand(A, N);
    flag = 1;
  }
  // consumidor
  #pragma omp section
  {
    while (flag==0) { }
    sum = sum_array(A, N);
  }
}
```

Este programa puede no funcionar porque el hilo consumidor nunca "vea" la actualización de la variable `flag`. Es más, el compilador podría reordenar las instrucciones y hacer que `flag = 1` antes de que `fill_rand` terminara. La solución es indicar explícitamente que el valor de la variable debe leerse de memoria y cuándo.

```
int N = 10000;
double *A = (double *) malloc(N*sizeof(double));
int sum;

#pragma omp parallel sections
{
  // productor
  #pragma omp section
  {
    fill_rand(A, N);
    #pragma omp flush // Asegura que flag sera escrito despues de fill_rand
    flag = 1;
    #pragma omp flush (flag) // Asegura que flag es escrito en memoria
  }
  // consumidor
  #pragma omp section
  {
    // Asegura que el flag es leido de memoria
    #pragma omp flush (flag)
    while (flag==0) {
      #pragma omp flush (flag)
    }
    #pragma omp flush // Evita reordenar la suma con el bucle
    sum = sum_array(A, N);
  }
}
```

3.3. Variables `volatile`

El lenguaje C/C++ (y también Java) incluye el modificador `volatile` que permite indicar que una variable debe mantenerse siempre consistente con la memoria principal.

Cuando hay una operación de lectura sobre una variable `volatile`, el programa se comporta como si existiera una operación *flush* justo antes de esa lectura. Cuando hay una operación de escritura sobre una variable `volatile`, el programa se comporta como si existiera una operación *flush* justo despues de esa lectura.

4. Tareas

Las directivas vistas hasta el momento no son muy adecuadas para expresar paralelismo irregular y no estructurado. El estilo tradicional de OpenMP ha estado tradicionalmente basado en el modelo *fork-join* o *maestro-esclavo*: con la directiva `parallel` se crea un un equipo de hilos y se debe establecer una estrategia para dividir el trabajo entre esos hilos. Directivas como `for` y `sections` permiten dividir el trabajo de manera sencilla entre hilos, pero la cantidad de trabajo tiene que ser conocida a priori.

Por ejemplo, si deseamos recorrer una lista ejecutando acciones en paralelo para cada elemento de la lista no podemos utilizar la directiva `for`.

```
list *p = list_head(l);
while (! list_end(p)) {
  procesar(p);  // se desea ejecutar en paralelo
  p = list_next(p);
}
```

Para abordar este tipo de situaciones, a partir de OpenMP 3.0 se introdujo el concepto de tareas (`task`) con el objetivo de dar soporte a un paralelismo más dinámico.

El ejemplo anterior se paralelizaría con tareas de la siguiente manera:

```
#pragma omp parallel
{
  #pragma omp single
  {
    list *p = list_head(l);
    while (! list_end(p)) {
      #pragma omp task
      procesar(p);

      p = list_next(p);
    }

    #pragma omp taskwait
  }
}
```

La directiva `task` permite indicar un bloque de código cuyo contenido deberá ser ejecutado por un hilo "en algún momento". Las tareas OpenMP están basadas en el estilo productor-consumidor *producer-consumer* a través de un *pool* de tareas. Las unidades de trabajo paralelo del programa se denominan tareas (*task*). El programa es el productor ya que genera nuevas tareas que se introducen en el *pool*. OpenMP actúa como consumidor y va ejecutando estas tareas en los hilos disponibles.

4.1. Directiva task

La directiva tiene el siguiente formato `#pragma omp task [claúsula [,claúsula]*]` donde las cláusulas pueden ser algunas de las siguientes:

- `if (expresión)`. Condición según la cual se generará la tarea o no. Si el resultado es falso, entonces simplemente se ejecuta el trozo de código como si no fuera parte de una nueva tarea. Esta directiva resulta útil para comprobar condiciones según las cuales no deseamos crear una nueva tarea porque su trabajo será pequeño (esto es, ya se ha dividido suficientemente el trabajo).
- `untied`. Especifica que la tarea no está asociada al hilo que comenzó su ejecución. Esto significa que cualquier hilo del equipo puede continuar la ejecución de la tarea después de que sea suspendida. Esto puede ser útil para balancear mejor la carga entre los hilos.

- `priority (num)`. Es un valor entero no negativo para indicar la prioridad de la tarea. El *scheduler* puede tener este valor en cuenta para favorecer la ejecución de la tareas con mayor prioridad.
- Modificadores del ámbito de las variables. Similar a otras directivas como `for`. Puede ser `private (vars)`, `firstprivate(vars)`, `lastprivate(vars)`, `shared (vars)`.

Cuando un hilo ejecuta una directiva `task`, el *runtime* puede elegir entre ejecutar la tarea immediatamente o diferir su ejecución. Si la tarea es diferida (*deferred*) se pone en una cola de tareas asociada con la región paralela actual. Los hilos del equipo irán tomando tareas de esta cola hasta vaciarla. El hilo que genera la tarea no tiene que ser necesariamente el mismo que empiece a ejecutarla. Además, si la tarea es `untied` es posible que varios hilos puedan contribuir a que termine su ejecución.

4.2. Ejemplo

El siguiente código implementa la secuencia de Fibonacci utilizando tareas.

```
int fib(int n) {
  int l, r;
  if (n<2) return n;
  #pragma omp task shared(l) firstprivate(n) if(n > 100)
  l = fib(n - 1);
  #pragma omptask shared(r) firstprivate(n) if(n > 100)
  r = fib(n - 2);

  #pragma omp taskwait
  return l + r;
}
```

4.3. Tareas o secciones

La diferencia entre las tareas y las secciones radica en el momento en el que el código se ejecuta en paralelo[3]..

El siguiente listado muestra dos trozos de código que son equivalente, uno realizado con secciones y otro con tareas. En la práctica, si el número de tareas es conocido al escribir el programa, preferiremos usar secciones.

```
// sections
...
#pragma omp sections
{
  #pragma omp section
  foo();
  #pragma omp section
  bar();
}
...

// tasks
...
#pragma omp single nowait
{
  #pragma omp task
  foo();
  #pragma omp task
  bar();
}
#pragma omp taskwait
...
```

[3]Ver `https://stackoverflow.com/questions/13788638/difference-between-section-and-task-openmp`

Ejecución de tareas

Supongamos el siguiente código cuya intención es crear dos tareas asociadas a dos hilos.

```
/* Crear dos hilos */
#pragma omp parallel num_threads(2)
      {
  /* Poner la tarea t0 a la cola */
  #pragma omp task
  {
    t0();
  }
  /* Poner la tarea t1 a la cola */
  #pragma omp task
  {
    t1();
  }
}
```

¿Cuántas tareas se están creando? La región paralela ejecuta todo su código en cada hilo. Por tanto, cada hilo genera dos tareas y en total hay 4 tareas. Es por ello que habitualmente se sigue la estrategia de utilizar la directiva `single` para asegurar que solo un hilo genere las tareas iniciales.

5. Buenas prácticas y errores comunes

En esta sección se discutirán algunas prácticas que es recomendable seguir a la hora de escribir programas con OpenMP para obtener un mejor rendimiento. También se presentan algunos errores que suelen cometerse y que pueden dar lugar programas incorrectos que son difíciles de depurar.

5.1. Programas portables

Si se desea que el programa sea portable y pueda compilarse incluso sin OpenMP no está disponible se puede hacer uso directivas condicionales y de la macro `_OPENMP` que estará activa cuando OpenMP está activo.

```
#include <stdio.h>
#if defined (_OPENMP)
  #include <omp.h>
#endif

int main()
{
  int t = 0;
  int np = 1;
  #pragma omp parallel private(t, np)
  {
     #if defined (_OPENMP)
     np = omp_get_num_threads();
      t = omp_get_thread_num();
     #endif

     printf("Soy el thread %d\n",t, np);
  }
}
```

5.2. Condiciones de carrera

Una condición de carrera aparece cuando dos o más hilos acceden concurrentemente a la misma zona de memoria sin que exista sincronización entre ellos y al menos una de las operaciones es una escritura.

El siguiente código tiene una condición carrera debida a la dependencia que tienen las iteraciones del bucle. Al ejecutarlo en paralelo puede suceder que, por ejemplo, para `i = 2` se calcule el valor de `a[2]` antes de que se ejecute la iteración `i = 1`, por lo que el resultado no sería el mismo que la ejecución secuencial.

```
#pragma omp parallel for
for(int i = 0; i < n - 1; i++) {
  a[i] = a[i+1] + b[i];
}
```

La dificultad para depurar las condiciones de carrera es que el programa se comporta habitualmente de manera no determinista, de manera dos ejecuciones producirán resultados diferentes. Es más, a veces el problema solo será observable para determinadas entradas, configuraciones de hilos, hardware concreto, etc.

Para detectar este tipo de problemas se pueden utilizar herramientas especializadas, pero siempre es buena idea comparar el resultado de la ejecución en paralelo con el mismo algoritmo en secuencial (p.ej., estableciendo el número de hilos a 1).

5.3. Visibilidad de variables

La directiva `parallel` incluye varias opciones para controlar la visibilidad y la inicialización de las variables. La visibilidad por defecto es `shared` por lo que es muy fácil introducir condiciones de carrera inadvertidamente. Por ejemplo, el

siguiente código es incorrecto porque `x` es una variable compartida y por tanto puede suceder que un hilo escriba en ella (línea 9) justo antes de que otro hilo la lea (línea 10).

```
1  void pi(int n) {
2    double h, x, sum;
3
4    h = 1.0 / (double) n;
5    sum = 0.0;
6
7    #pragma omp parallel for reduction(+:sum)
8    for(int i = 0; i < n; i++) {
9      x = h * ((double) i - 0.5);
10     sum += 1.0 / (1.0 + x * x);
11   }
12 }
```

Para evitar este tipo de problema suele ser buena idea añadir el modificador `default(none)` a `parallel` para que el compilador obligue a especificar la visibilidad de las variables explícitamente.

5.4. Inicialización de variables privadas

Las variables privadas no se inicializan por defecto, incluso si la variable padre de la que provienen ha sido efectivamente inicializada. Además, el resultado de la variable padre tampoco está definido una vez que acabe la sección paralela.

El siguiente código es incorrecto porque no hay garantías de que la variable `a` tenga el valor 10.

```
1  int a = 10;
2  #pragma omp parallel for private(a)
3  for(int i = 0; i < 100; i++)
4  {
5    if (i % a == 0) {
6      printf("%d es multiplo de 10\n", i);
7    }
8  }
```

Para solucionarlo podemos usar `firstprivate` o mantener la variable como compartida.

Ejercicio

¿Qué posibles soluciones hay para solucionar este problema?
¿Cómo podrías comprobar que es correcto?

6. Ejercicios

- Utiliza tareas para paralelizar mergesort
- Construye un histograma en paralelo.

7. Bibliografía comentada

- *Using OpenMP: portable shared memory parallel programming* [1]. Los primeros capítulos ofrecen una introducción detallada a OpenMP y a la programación paralela en memoria compartida. Los capítulos 5 y 6 presentan trucos y consejos para obtener buen rendimiento y el capítulo 7 discute las causas frecuentes de "bugs" en programas OpenMP. Este último capítulo es bastante recomendable para disponer una lista de chequeo propia y evitar errores.
- *A Hands-on Introduction To OpenMP* de Tim Mattson [3], disponible en: `https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf`. Un tutorial detallado sobre OpenMP.
- *An introduction to parallel programming* de Peter Pacheco [4]. El capítulo 5 presenta OpenMP e incluye varios ejemplos.

- Ejemplos oficiales de OpenMP. `https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.Examples.pdf`

Referencias

[1] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[2] J. P. Hoeflinger and B. R. De Supinski. The openmp memory model. In *International Workshop on OpenMP*, pages 167–177. Springer, 2005.

[3] T. Mattson and L. Meadows. A "hands-on" introduction to openmp. *Intel Corporation*, 2014.

[4] P. Pacheco. *An introduction to parallel programming*. Elsevier, 2011.