Capítulo 3

Programación multi-hilo usando OpenMP

Herramientas utilizadas: Lenguaje C, OpenMP, gcc (4.2 o posterior).

3.1. Introducción

Las herramientas que permiten programación multi-hilo actualmente son varias, tales como, Pthreads [14], TBB [15] u OpenMP[13]. En el presente capítulo utilizaremos OpenMP, debido a su sencillez a la hora de programar y depurar un código. Además, el directorio de OpenMP está formado por varias compañías, entre ellas Sun Mycrosystems, AMD, Intel, y otras. OpenMP ha sido desarrollado desde el año 1997, por lo tanto las versiones actuales cuenta con compiladores robustos.

OpenMP se organiza mediante *directivas* que comienzan con la sentencia `#pragma omp` y también mediante *funciones* de la librería `omp.h`.

3.2. Directivas y Funciones Básicas

Para comenzar a programar en OpenMP, debemos conocer algunas de sus funciones y directivas básicas. En los Anexos A y B se describen las funciones y directivas de OpenMP en extenso. Para realizar los primeros ejemplos tomemos en cuenta las siguientes funciones y directivas:

Funciones:

`omp_get_num_procs()`: Devuelve la cantidad de núcleos detectada por el sistema operativo.

`omp_set_num_threads(T)`: Establece el número de hilos de la región paralela siguiente (`T` hilos en este caso). Esto también se puede realizar asignando el número de hilos deseados a la variable de entorno `OMP_NUM_THREADS`.

`omp_get_thread_num()`: Devuelve el identificador del hilo. ejecución.

`omp_get_num_threads()`: Devuelve la cantidad de hilo de la región paralela actual.

Directivas:

`#pragma omp parallel`: Esta directiva cubre una región de código que será ejecutada por T hilos (T debe ser definido previamente). Todos los hilos ejecutarán exactamente el mismo código.

A lo largo del libro, llamaremos *región paralela* a la región de código ejecutada en paralelo. Estas regiones pueden ser creadas con la directiva `#pragma omp parallel` o con `#pragma omp for` (descrita en la Sección 3.3).

Debido a que todos los hilos ejecutan el mismo código, una manera de lograr que un hilo ejecute un código distinto es mediante su *identificador de hilo*, dado por la función `omp_get_thread_num()`.

Observemos el siguiente ejemplo en donde se emplean la directiva (en púrpura) y las funciones (en azúl) previamente señaladas:

Algoritmo 1 Algoritmo *Hola Mundo* en OpenMP [2] [Download]

```
1: #include <stdio.h>
2: #include <omp.h>
3: main()
4: {
5:     printf("Nuestro sistema tiene %d núcleos\n\n", omp_get_num_procs());
6:     omp_set_num_threads(8);
7:     #pragma omp parallel
8:     {
9:         int tid = omp_get_thread_num(), nprocs = omp_get_num_threads();
10:        printf("Hola Mundo. Soy el hilo %d, de un total de %d.\n", tid, nprocs);
11:    }
12: }
```

Compilación: gcc -fopenmp prog.c
Ejecución: ./a.out

Host: magistermco.ucm.cl
Puerto: 22
Ej.: ssh login@magistermco.ucm.cl

Login: (Se da en la clase)
Pass: 12345678

La compilación se puede realizar con `gcc`, usando cualquier versión desde la 4.2 en adelante, de la siguiente manera:

```
gcc -fopenmp mi_programa.c
```

Para el lector no familiarizado con programación multi-hilo quizás pueda sorprender la salida del programa, ya que cada vez que lo ejecute es probable que imprima una salida distinta. Supongamos que la salida de una ejecución fue la siguiente:

```
Nuestro sistema tiene 24 núcleos

Hola Mundo. Soy el hilo 1, de un total de 8.
Hola Mundo. Soy el hilo 7, de un total de 8.
Hola Mundo. Soy el hilo 0, de un total de 8.
Hola Mundo. Soy el hilo 5, de un total de 8.
Hola Mundo. Soy el hilo 2, de un total de 8.
Hola Mundo. Soy el hilo 3, de un total de 8.
Hola Mundo. Soy el hilo 4, de un total de 8.
Hola Mundo. Soy el hilo 6, de un total de 8.
```

Como observará, este programa fue ejecutado en una máquina con 24 núcleos, y además los hilos no se imprimieron en orden (del 0 al 7), sino que de manera desordenada. Esto es porque una vez creados los hilos, éstos se lanzan a ejecución, sin saber qué hilo podrá "adelantarse" a otro en la secuencia de instrucciones. Qué hilo es ejecutado en qué procesador es decidido por el sisema operativo. Una vez que la región paralela termina (línea 11 en Algoritmo 1), entonces se continúa ejecutando el programa de manera secuencial, pero esto ocurre sólo cuando todos los hilos llegan a este punto.

3.3. Bucle `for` Paralelo

Dentro de una directiva `#pragma omp parallel` (en adelante directiva `parallel`) es posible utilizar una directiva `#pragma omp for` (en adelante directiva `for`), en donde las iteraciones del bucle `for` inmediatamente después de esta directiva son distribuidas entre los hilos. Su sintaxis es como sigue:

```
#pragma omp for [opciones]
for(inicializar var; var expr limite; incremento/decremento de var)
```

La expresión *`inicializar var`* debe inicializar una variable de tipo `int`, que en este caso la denominamos *`var`*. En la expresión *`var expr limite`*, la variable limite debe ser de tipo `int`, y *`expr`* debe ser una de las siguientes expresiones: `<`, `<=`, `>`, `>=`. La expresión *`incremento/decremento de var`* debe incrementar o decrementar la variable *`var`* en un número entero. El Algoritmo 2 muestra un ejemplo de la directiva `for`.

Algoritmo 2 Algoritmo *Hola Mundo* en OpenMP [12] [Download]

```
1: #include <stdio.h>
2: #include <omp.h>
3: #define N 10
4: main()
5: {
6:     omp_set_num_threads(4);
7:     #pragma omp parallel
8:     {
9:         int i;
10:        #pragma omp for
11:        for(i=0; i<N; i++)
12:            printf("Hilo %d - Ejecutando la iteración: %d\n", omp_get_thread_num(), i);
13: }
```

La salida del Algoritmo 2 (utilizando 4 hilos y un valor de $N = 10$) es una impresión en pantalla desordenada, pero para ilustrar de mejor manera la distribución de las iteraciones entre los hilos, supondremos la siguiente impresión por pantalla:

```
Hilo 0 - Ejecutando la iteración: 0
Hilo 0 - Ejecutando la iteración: 1
Hilo 0 - Ejecutando la iteración: 2
Hilo 1 - Ejecutando la iteración: 3
Hilo 1 - Ejecutando la iteración: 4
Hilo 1 - Ejecutando la iteración: 5
Hilo 2 - Ejecutando la iteración: 6
Hilo 2 - Ejecutando la iteración: 7
Hilo 3 - Ejecutando la iteración: 8
Hilo 3 - Ejecutando la iteración: 9
```

Se puede observar claramente la manera de distribuir las iteraciones entre los hilos en este ejemplo. Si no se declara explícitamente la manera de distribuir las iteraciones entre los hilos, entonces el compilador toma la decisión, la que puede variar entre distintos bucles paralelos en el mismo programa. La manera de asignar una distribución a un bucle `for` paralelo está descrito en la Sección 3.3.1.

Cabe destacar que luego de una directiva `for` es agregada una barrera de sincronización (Sección 3.5) automáticamente.

También, es posible fusionar las directivas `paralell` y `for` en una sola denominada `#pragma omp parallel for`, la que debe continuarse con un bucle `for` siguiendo las misma reglas vistas previamente para la directiva `for`.

3.3.1. Opción `schedule`

La directiva `for` tiene una opción denominada `schedule(`*`tipo`*`[,`*`tamaño_lote`*`])`, que permite establecer la forma de asignación de los bucles de una directiva `for` a hilos. El parámetro *`tamaño_lote`* es opcional. Los distintos valores que puede tener el parámetro *`tipo`* son definidos a continuación:

static: Las iteraciones son divididas cada *`tamaño_lote`* iteraciones. Si la opción *`tamaño_lote`* no es especificada, entonces el número de iteraciones es divido en lotes de (aproximadamente) igual tamaño.

dynamic: Las iteraciones son asignadas dinámicamente según se vayan requiriendo por los hilos. Primeramente se asignan *`tamaño_lote`* iteraciones a cada hilo, y luego el siguiente lote de iteraciones es asignado al hilo que lo solicite. Un hilo solicita un nuevo lote de iteraciones cuando ya procesó el anterior y por lo tanto está disponible para procesar el siguiente. Si el parámetro *`tamaño_lote`* no es especificado, entonces su valor será igual a 1.

guided: Es similar a `dynamic`, pero el tamaño de lote de las iteraciones asignadas a los hilos va decreciendo hasta llegar al valor *`tamaño_lote`*. Por defecto el valor de *`tamaño_lote`* es aproximadamente el número de iteraciones dividido por el número de hilos.

runtime: Si este valor es utilizado, entonces la desición de qué distribución usar se hace en tiempo de ejecución con los valores establecidos por la variable de ambiente OMP_SCHEDULE. Esta variable tiene el formato OMP_SCHEDULE="kind[,chunk_size]".

En la Tabla **??**

3.4. Variables Compartidas y Privadas

Las variables que son declaradas antes de una región paralela pueden ser usadas como variables compartidas o privadas. Para ello existen las siguientes opciones para la directiva `#pragma omp parallel`:

`shared(x)`: Establece la variable `x` como compartida, lo que significa que todos los hilos tendrán acceso a ella.

`private(x)`: Establece la variable `x` como privada, por lo tanto, se creará una variable `x` por cada hilo, y cada uno sólo tendrá acceso a dicha variable local. El valor de la variable original no es copiado a las que se crean para cada hilo.

`firstprivate(x)`: Es igual que `private(x)`, pero mantiene el valor de la variable original en las variables privadas creadas para cada hilo.

Comprobemos lo anterior con el siguiente ejemplo, en donde además de imprimir las variables compartidas y privadas, también se imprimen sus direcciones de memoria.

Algoritmo 3 Ejemplo de variables compartidas y privadas en OpenMP [3] [Download]

```
1: #include <stdio.h>
2: #include <omp.h>
3: main()
4: {
5:     int var_sh=1, var_priv=2, var_fpriv=3;
6:     omp_set_num_threads(8);
7:     #pragma omp parallel shared(var_sh) private(var_priv) firstprivate(var_fpriv)
8:     {
9:         int tid = omp_get_thread_num();
10:        var_priv = 5;
11:        printf("Hilo %d: var_sh = %d (%p), var_priv = %d (%p), var_fpriv = %d (%p)\n", tid, var_sh,
   &var_sh, var_priv, &var_priv, var_fpriv, &var_fpriv);
12:    }
13: }
```

El resultado sería algo como lo siguiente:

```
Hilo 2: var_sh = 1 (0x7ffedae64a6c), var_priv = 5 (0x7faa52823e88), var_fpriv = 3 (0x7faa52823e84)
Hilo 7: var_sh = 1 (0x7ffedae64a6c), var_priv = 5 (0x7faa5001ee88), var_fpriv = 3 (0x7faa5001ee84)
Hilo 0: var_sh = 1 (0x7ffedae64a6c), var_priv = 5 (0x7ffedae64a48), var_fpriv = 3 (0x7ffedae64a44)
Hilo 5: var_sh = 1 (0x7ffedae64a6c), var_priv = 5 (0x7faa51020e88), var_fpriv = 3 (0x7faa51020e84)
Hilo 1: var_sh = 1 (0x7ffedae64a6c), var_priv = 5 (0x7faa53024e88), var_fpriv = 3 (0x7faa53024e84)
Hilo 4: var_sh = 1 (0x7ffedae64a6c), var_priv = 5 (0x7faa51821e88), var_fpriv = 3 (0x7faa51821e84)
Hilo 6: var_sh = 1 (0x7ffedae64a6c), var_priv = 5 (0x7faa5081fe88), var_fpriv = 3 (0x7faa5081fe84)
Hilo 3: var_sh = 1 (0x7ffedae64a6c), var_priv = 5 (0x7faa52022e88), var_fpriv = 3 (0x7faa52022e84)
```

Donde se observa que todos los hilos están accediendo a la misma variable `var_sh` en memoria, y cada uno de ellos tiene las variables privadas (`var_priv` y `var_fpriv`) almacenadas en una dirección de memoria distinta a la original. Por lo anterior, al modificar

el valor una variable compartida, esta modificación se mantiene luego de terminar la región paralela.

3.5. Regiones Críticas y Barreras de Sincronización

A continuación se definen tres directivas, que son las regiones críticas, barreras de sincronización, y ejecución exclusiva del hilo maestro.

`#pragma omp critical [nombre]:` Esta directiva cubre una región de código que será ejecutada por un hilo a la vez. T definidos previamente. Es decir, si un hilo entra en esta región de código, entonces la bloquea, y los demás hilos tendrán que esperar hasta que el hilo que está actualmente en esta región termine. También es posible asignarle un nombre a una región de esta directiva, de tal manera que cuando un hilo bloquea una región, tmbién bloqueará todas las demás regiones que tengan el mismo nombre. Si no se asigna ningún nobre, por defecto todas las regiones críticas tienen el mismo nombre. Es una buena práctica siempre colocarle nombre a las regiones críticas. La región de código encerrada en las llaves de esta directiva la llamaremos *región crítica* a lo largo del libro.

`#pragma omp barrier:` Esta directiva no encierra una región de código. Cuando un hilo llega a esta directiva, entonces se detiene hasta que todos los demás hilos también hayan llegado hasta este punto en el código. Una vez que todo los hilos alcanzan este punto, entonces continúan.

`#pragma omp master:` La región de código encerrada por esta directiva es ejecutada solamente por el hilo con identificador igual a 0o.

Para ilustrar las directivas anteriores veamos el siguiente ejemplo, en donde cada hilo sumará el valor de su identificador a una variable compartida, para luego imprimir el resultado final.

Algoritmo 4 Ejemplo donde cada hilo suma su identificador a una variable compartida [6] [Download]

```
1: #include <stdio.h>
2: #include <omp.h>
3: main()
4: {
5:     int var_sh=0;
6:     omp_set_num_threads(8);
7:     #pragma omp parallel shared(var_sh)
8:     {
9:         int tid = omp_get_thread_num();
10:        #pragma omp critical (zona_1)
11:        {
12:            var_sh = var_sh + tid;
13:        }
14:        #pragma omp barrier
15:        #pragma omp master
16:        {
17:            printf("Hilo %d: var_sh = %d\n", tid, var_sh);
18:        }
19:    }
20: }
```

La región crítica que denominamos *zona_1* asegura que la suma a la variable compartida (línea 13, Algoritmo 4) es realizada por un hilo a la vez, evitando una escritura simultánea de más de un hilo. Luego, para imprimir el resultado correcto es necesario agregar una barrera de sincronización. Si no se hubiese colocado dicha barrera, entonces al momento de imprimir el resultado (línea 18) algún hilo podría no haber realizado la suma aún, o incluso peor, intentar modificar el valor de `var_sh` justo en el momento en que el hilo con identificador 0 imprime dicha variable.

A continuación resolveremos otro problema, que nos ilustrará la importancia de abordar la programación multi-hilo de una manera eficiente. Un programa paralelo mal diseñado podría ser incluso más lento que su versión secuencial. El ejercicio se trata de lo siguiente: Generar un arreglo con valores enteros aleatorios, y luego utilizando un conjunto de hilos encontrar cuál es el elemento del arreglo de mayor valor.

Antes de atender la solución en sí, asumiremos a lo largo del libro que tenemos una función para generar números aleatorios, como la que se muestra en el Algoritmo 5.

Algoritmo 5 Función para generar un arreglo de números aleatorios de 0 a 99.

```
1: int *genera_num_aleatorios(int cantidad_elementos)
2: {
3:     int i, *arr;
4:     srand(time(NULL));
5:     arr = (int *)malloc(sizeof(int)*cantidad_elementos);
6:     for (i=0; i<cantidad_elementos; i++)
7:         arr[i] = rand() %100; //Se genera un número aleatorio entre 0 y 99
8:     return arr;
9: }
```

Una primera solución a este problema podría ser el tener una variable compartida que almacene el mayor valor encontrado hasta el momento por todos los hilos. De esta manera, luego que todos los hilos recorrieron el arreglo, esta variable compartida tendrá el mayor de los valores, tal como se muestra el Algoritmo 6.

Algoritmo 6 Algoritmo para encontrar el mayor elemento de un arreglo utilizando una variable compartida [4] [Download]

```
1: #include <stdio.h>
2: #include <omp.h>
3: #define N 1000000000
4: #define T 8
5: main()
6: {
7:     int mayor = 0, *arr;
8:     arr = genera_num_aleatorios(N);
9:     omp_set_num_threads(T);
10:    #pragma omp parallel shared(mayor, arr)
11:    {
12:        int i, tid = omp_get_thread_num(), nprocs = omp_get_num_threads();
13:        for (i=tid; i<N; i=i+nprocs)
14:        {
15:            #pragma omp critical (zona_1)
16:            {
17:                if (arr[i] >mayor)
18:                    mayor = arr[i];
19:            }
20:        }
21:        #pragma omp barrier
22:        #pragma omp master
23:        {
24:            printf("Elemento mayor = %d\n", mayor);
25:        }
26:    }
27: }
```

Observemos que la distribución de los elementos entre los hilos fue circular, es decir, el hilo con identificador 0 accede a los elementos 0, 8, 16, 32, etc. mientras que el hilo con identificador 1 accede a los elementos 1, 9, 17, 33, etc. y así sucesivamente. Esta distribución se realiza debido al `for` de la línea 13 (Algoritmo 6).

Si se toma en cuenta el tiempo de ejecución del programa (Algoritmo 6) y se compara con una versión secuencial del mismo algoritmo ([1] [Download]), se dará cuenta que la versión secuencial es más eficiente al agrandar el valor de N implicando menos tiempo de ejecución. La razón de esto es que cada vez que un hilo accede a un elemento del arreglo `arr` también accede a una región crítica, es decir, no hay paralelismo involucrado, ya que el código en una región crítica es siempre ejecutado por un hilo a la vez. La versión secuencial es más eficiente porque si bien también accede de manera secuencial al arreglo, no tiene que lidiar con bloqueos de regiones críticas. Como recomendación se debe siempre intentar evitar las regiones críticas dado el decremento en el rendimiento que provocan.

Tomemos en cuenta una segunda solución, que utilice menos accesos a regiones críticas. En el Algoritmo 7, cada hilo luego de recorrer el arreglo `arr` (siguiendo una distribución circular), obtiene un *máximo local*, que representa el valor máximo encontrado entre los elementos que visitó. Posteriormente, se debe encontrar el valor máximo entre los máximos locales, que será la respuesta final.

Algoritmo 7 Algoritmo para encontrar el mayor elemento de un arreglo utilizando mínimos locales [5] [Download]

```
1: #include <stdio.h>
2: #include <omp.h>
3: #define N 1000000000
4: #define T 8
5: main()
6: {
7:     int i, mayor_global = 0, *arr, mayores_locales[T];
8:     arr = genera_num_aleatorios(N);
9:     for (i=0; i<T; i++)
10:        mayores_locales[i] = 0;
11:    omp_set_num_threads(T);
12:    #pragma omp parallel shared(mayores_locales, arr)
13:    {
14:        int i, mayor_global, tid = omp_get_thread_num(), nprocs = omp_get_num_threads();
15:        for (i=tid; i<N; i=i+nprocs)
16:        {
17:            if (arr[i] >mayores_locales[tid])
18:                mayores_locales[tid] = arr[i];
19:        }
20:        #pragma omp barrier
21:        #pragma omp master
22:        {
23:            mayor_global = 0;
24:            for (i=0; i<T; i++)
25:                if (mayor_global <mayores_locales[i])
26:                    mayor_global = mayores_locales[i];
27:            printf("Elemento mayor =%d\n", mayor);
28:        }
29:    }
30: }
```

3.6. Directiva `atomic`

La directiva `atomic` es similar a una directiva `critical`, pero sólo aplicado a una sentencia, es decir, esta directiva obliga a que dicha sentencia sea ejecutada por un hilo a la vez. Esta sentencia debe ser una asignación, incremento o decremento, donde los operadores aceptados son: `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, `>>`. Tome en cuenta las siguientes sentencias válidas:

1.
```
#pragma omp atomic
ic = ic + 1;
```

2. ```
   #pragma omp atomic
   ic = ic + var;
   ```

3. ```
   #pragma omp atomic
   ic = ic + fun();
   ```

La sentencia 2 no impide el acceso concurrente a la variable `var`. La sentencia 3 tampoco impide el acceso concurrrente a la función `fun()`, pero ambas sentencias (2 y 3) protegen (bloquean) a la variable del lado izquierdo `ic` para obligar al acceso a ella por parte de un hilo a la vez.

3.7. Locks

Además de las directivas bloqueantes `critical` y `atomic`, también existe un grupo de funciones de OpenMP que permiten bloquear una región de código. La gran ventaja que tienen estas funciones es que permiten bloquear una región de código para un grupo de hilos y no obligatoriamente para todos ellos. Es decir, su comportamiento es similar a una directiva `critical` pero aplicado solamente a un grupo de hilos que uno puede seleccionar. La sintaxis de estas funciones se muestran a continuación.

1. `void omp_init_lock (omp_lock_t *cerrojo)`: Permite el inicializar la variable `cerrojo`, que es un proceso obligatorio antes de poder utilizar dicha variable en el bloqueo de una regón de código.

2. `void omp_set_lock (omp_lock_t *cerrojo)`: Permite bloquear una región de código identificada con la variable `cerrojo`. Si un hilo ejecuta esta función, pueden suceder dos cosas: (1) el hilo inicia un bloqueo sobre una región de código identificada con la variable `cerrojo`, (2) si la región de código está bloqueada (un hilo ha ejecutado previamente esta función con la misma variable `cerrojo`), entonces el hilo se detiene a esperar a que la variable `cerrojo` sea liberada.

3. `void omp_test_lock (omp_lock_t *cerrojo)`: Esta función es similar a la función anterior `omp_set_lock`, pero si otro hilo tiene establecido un bloqueo con la varibale `cerrojo`, entonces el hilo continúa sin esperar.

4. `void omp_unset_lock (omp_lock_t *cerrojo)`: Libera la variable `cerrojo`, desbloqueando la región de código correspondiente.

5. `void omp_destroy_lock (omp_lock_t *cerrojo)`: Elimina la inicialización a la variable `cerrojo` establecida previamente por la función `omp_init_lock`.

Tome en cuenta el ejemplo del Algoritmo 8, que utiliza las funciones previamente definidas. Este ejemplo es igual al Algoritmo 4, en donde cada hilo suma el valor de su identificador a una variable compartida.

Algoritmo 8 Algoritmo que utiliza las funciones de bloqueo definidas en la Sección 3.7 [7]

[Download]

```
#include <stdio.h>
#include <omp.h>
main()
{
    int var_sh=0;
    omp_lock_t cerrojo;
    omp_init_lock(&cerrojo);
    omp_set_num_threads(8);
    #pragma omp parallel shared(var_sh, cerrojo)
    {
        int tid = omp_get_thread_num();
        omp_set_lock(&cerrojo);
            var_sh = var_sh + tid;
        omp_unset_lock(&cerrojo);
        #pragma omp barrier
        #pragma omp master
        {
            printf("Hilo %d: var_sh = %d\n", tid, var_sh);
        }
    }
    omp_destroy_lock(&cerrojo);
}
```

El Algoritmo 9 muestra como implementar un conjunto de 4 regiones críticas, una por grupo, donde cada grupo está formado por 4 hilos. Cada grupo de hilos suma su identificador a una celda del arreglo compartido `var_sh`, y finalmente un hilo por grupo hace una impresión por pantalla.

Algoritmo 9 Algoritmo que utiliza un arreglo de cerrojos para simular 4 regiones críticas, una por grupo, y cada grupo de 4 hilos [8] [Download]

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <omp.h>
4: main()
5: {
6:     int var_sh[4], i;
7:     omp_lock_t *arr_cerrojos;
8:     arr_cerrojos = (omp_lock_t *)malloc(sizeof(omp_lock_t)*4);
9:     for (i=0; i<4; i++)
10:    {
11:        omp_init_lock(&(arr_cerrojos[i]));
12:        var_sh[i] = 0;
13:    }
14:    omp_set_num_threads(16);
15:    #pragma omp parallel shared(var_sh)
16:    {
17:        int tid = omp_get_thread_num(), id_grupo;
18:        id_grupo = (int)(tid/4);
19:        omp_set_lock(&(arr_cerrojos[id_grupo]));
20:            var_sh[id_grupo] = var_sh[id_grupo] + tid;
21:        omp_unset_lock(&(arr_cerrojos[id_grupo]));
22:        #pragma omp barrier
23:        if (tid % 4 == 0)
24:            printf("Grupo %d: var_sh = %d\n", id_grupo, var_sh[id_grupo]);
25:    }
26:    for (i=0; i<4; i++)
27:        omp_destroy_lock(&(arr_cerrojos[i]));
28: }
```

Este grupo de funciones debe ser implementado cuidadosamente, ya que es posible que existan *deadlocks* si se utilizan de manera equivocada.

3.8. Directiva `single`

Esta directiva encierra una región de código que será ejecutada una sola vez, por solo un hilo. El primer hilo en alcanzar esta directiva es el encargado de ejecutar sus sentencias. Un punto a tener en cuenta es que esta directiva agrega automáticamente una barrera al final de ella. Es usualmente utilizada cuando es necesario que algún hilo (sin importar cuál) actualice una variable compartida.

El Algoritmo 10 muestra un ejemplo del uso de la directiva `single`. En este ejemplo

solo un hilo actualiza la varibale compartida `var_sh`. Todos los hilos imprimen el valor 10 de la variable `var_sh`, lo que es posible porque una barrera es agregada automáticamente luego de la directiva `single`.

Algoritmo 10 Ejemplo de la directiva `single` en OpenMP [10] [Download]

```
1: #include <stdio.h>
2: #include <omp.h>
3: main()
4: {
5:     int var_sh=0;
6:     omp_set_num_threads(16);
7:     #pragma omp parallel shared(var_sh)
8:     {
9:         int tid = omp_get_thread_num();
10:        #pragma omp single(var_sh)
11:        {
12:            printf("Hilo %d :: Actualizando var_sh a 10\n", tid);
13:            var_priv = 10;
14:        }
15:        /* En este punto una barrera es agregada automáticamente */
16:        printf("Hilo %d: var_sh = %d\n", tid, var_sh);
17:    }
18: }
```

3.9. Paralelismo Condicional: Opción `if`

La opción `if` puede ser utilizada solamente en la directiva `#pragma omp parallel`. Esta opción permite ejecutar una región de código de manera paralela o secuencial dependiendo si el valor de la opción `if` es verdadera o falsa respectivamente.

Algoritmo 11 Algoritmo en OpenMP utilizando la opción `if` [11] [Download]

```
1: #include <stdio.h>
2: #include <omp.h>
3: main()
4: {
5:     int n = 2, i;
6:     omp_set_num_threads(8);
7:     for (i=0; i <n; i++)
8:     {
9:         printf("\n Iteracion %d:\n", i);
10:        #pragma omp parallel if (i == 1)
11:        {
12:            int tid = omp_get_thread_num();
13:            printf("Hilo %d, i = %d\n", tid, i);
14:        }
15:    }
16: }
```

Observe el Algoritmo 11, en donde se realizan 2 iteraciones. En la primera de ellas se ejecuta la región paralela de manera secuencial, es decir, ejecutándose con solo 1 hilo. En la segunda iteración la ejecución si es en paralelo con 8 hilos, debido al valor de la opción `if`. La salida del algoritmo es como se muestra a continuación:

```
Iteracion 0:
Hilo 0, i = 0

Iteracion 1:
Hilo 1, i = 1
Hilo 3, i = 1
Hilo 4, i = 1
Hilo 2, i = 1
Hilo 5, i = 1
Hilo 6, i = 1
Hilo 0, i = 1
Hilo 7, i = 1
```

3.10. Paralelismo Anidado

OpenMP permite el paralelismo anidado, es decir, un hilo en una región paralela puede crear otra sub-región paralela dentro de ella, con un nuevo grupo de hilos.

La propiedad de paralelismo anidado puede ser habilitada o deshabilitada con la función `void omp_set_nested(int val)`. Para consultar si esta propiedad está habilitada se puede utilizar la función `int omp_get_thread_num()`. Cabe destacar que los

identificadores de hilo en una sub-región paralela comienzan también en 0. El Algoritmo 12 muestra un ejemplo de aplicación de la propiedad paralelismo anidado. Tenga en cuenta que esta propiedad puede ser utilizada por ambas directivas `#pragma omp parallel` y `#pragma omp parallel for`.

Algoritmo 12 Algoritmo en OpenMP aplicando Paralelismo Anidado [9] [Download]

```
1: #include <stdio.h>
2: #include <omp.h>
3: main()
4: {
5:     if (!omp_get_nested()) // Si no está habilitado el paralelismo anidado
6:         omp_set_nested(1); //Habilitando paralelismo anidado. Con omp_set_nested(0) se puede deshabilitar
7:     omp_set_num_threads(4);
8:     #pragma omp parallel
9:     {
10:        int tid = omp_get_thread_num();
11:        printf("Hilo %d, Región Externa\n", tid);
12:        #pragma omp parallel num_threads(2) firstprivate(tid)
13:        {
14:            printf("Hilo %d, Región Interna. Invocado por Hilo %d\n", omp_get_thread_num(), tid);
15:        }
16:    }
17: }
```

En al Algoritmo 12, cada uno de los 4 hilos de la región paralela (de la línea 8) crea una nueva sub-región paralela (línea 12) de 2 hilos. Para indicar la cantidad de hilos en la sub-región paralela se utilizó la opción `num_threads` de la directiva `parallel`. En este algoritmo se utilizó el identificador del hilo dentro de la sub-región paralela declarándola como una variable privada. La salida de este algoritmo (suponiendo una impresión ordenada) se muestra a continuación:

```
Hilo 0, Región Externa
Hilo 3, Región Externa
Hilo 1, Región Externa
Hilo 2, Región Externa
    Hilo 0, Región Interna. Invocado por Hilo 0
    Hilo 1, Región Interna. Invocado por Hilo 0
    Hilo 0, Región Interna. Invocado por Hilo 1
    Hilo 1, Región Interna. Invocado por Hilo 1
    Hilo 0, Región Interna. Invocado por Hilo 2
    Hilo 1, Región Interna. Invocado por Hilo 2
    Hilo 1, Región Interna. Invocado por Hilo 3
    Hilo 0, Región Interna. Invocado por Hilo 3
```

3.11. Planificación de Ejecución de Hilos en Núcleos

3.12. Ejercicios Propuestos

3.12.1. Índice Invertido

Este ejercicio consiste en realizar una implementación a escala muy pequeña de un motor de búsqueda Web mediante un *índice invertido*. La creación del índice debe ser implementada en paralelo usando OpenMP. Posteriormente, para comprobar el correcto funcionamiento del índice, usted deberá procesar consultas de manera secuencial.

Un índice invertido consiste en un conjunto de términos, y por cada término se tiene asociada una lista de documentos, que son los documentos donde dicho término aparece. Por ejemplo, considere que se tienen los siguientes archivos html con los siguientes términos:

primero.html:
```
hola mundo perro
```

segundo.html:
```
hola perro gato auto
```

tercero.html:
```
perro auto casa
```

El procesar dichos archivos debe generar un índice invertido como indica la Figura 3.1.

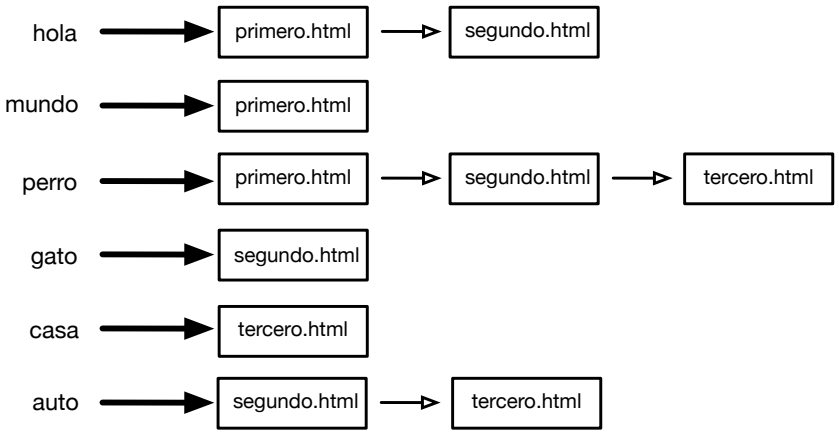

Figura 3.1: Ejemplo de Índice Invertido.

En esta figura cada término tiene asociado una lista enlazada de documentos, que contiene los nombres de los archivos donde aparece dicho término.

Con lo anterior, si un usuario desea realizar la consulta "auto", su sistema debería entregar como respuesta los documentos "segundo.html" y "tercero.html", porque en esos documentos es donde el término "auto" está presente.

Se utilizará un archivo de entrada para ingresar los documentos, y también las consultas, redirigiendo la entrada de los datos con "`<`" (Ej.: `./a.out <in.txt`). Para el ejemplo anterior, el archivo de entrada sería del siguiente tipo:

```
3
primero.html
3
hola mundo perro
segundo.html
4
hola perro gato auto
tercero.html
3
perro auto casa
```

```
5
gato perro mundo edificio hotel
```

En donde la primera línea indica el número de documentos, la segunda línea el título del primer documento, la tercera línea el número de términos del primer documento, la cuarta línea indica los términos del primer documento, y se repite lo mismo para los siguientes 2 archivos. Las últimas 2 líneas indican el número de consultas y las consultas mismas, que están compuestas por sólo 1 término cada una. En este ejemplo, las consultas son 5: "gato", "perro", "mundo", "edificio" y "hotel". Fíjese que es posible que una consulta tenga ningún archivo como resultado.

Todos los hilos deben involucrarse en la creación del índice invertido. Usted decida como distribuye el trabajo. Una vez que el índice invertido está creado, debe implementar el resolver consultas. Pero, la búsqueda debe ser implementada de manera secuencial, utilizando sólo un hilo.

Al momento de resolver consultas, su programa debe imprimir por pantalla, de manera muy clara, cuales son los nombres de archivo resultado de cada consulta. En caso que no haya ningún resultado, debe indicarlo claramente por pantalla. Por ejemplo, de la siguiente manera:

```
Resultados para "gato": segundo.html
Resultados para "perro": primero.html segundo.html tercero.html
Resultados para "mundo": primero.html
Resultados para "edificio": No hay resultados.
Resultados para "hotel": No hay resultados.
```

3.12.2. Multiplicación de Matrices

Este ejercicio consiste en realizar una multiplicación de matrices utilizando un conjunto de hilos en paralelo.

Usted deberá pedir por teclado los elementos para rellenar 2 matrices `A` y `B` de tamaño `N` $\times$ `N` ambas, para posteriormente multiplicarlas y almacenar el resultado en una matriz `C`. Debe crear `T` hilos (donde `T` es el número de núcleos de la máquina). Cada hilo debe realizar las operaciones necesarias para calcular una fila de `C`. Finalmente su programa debe imprimir el valor del coeficiente más alto de C.

Suponga el ejemplo de la Figura 3.2.

$$A = \begin{bmatrix} 4 & 2 & 4 \\ 3 & 1 & 5 \\ 2 & 5 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 1 & 3 \\ 5 & 1 & 2 \\ 5 & 5 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 38 & 26 & 20 \\ 36 & 29 & 16 \\ 34 & 12 & 17 \end{bmatrix}$$

Figura 3.2:

El cálculo de la primera fila de `C` (`[38 26 20]`) debe ser realizado por el hilo con identificador igual a 0. Es decir, este hilo 0 debe acceder a la primera fila de `A` y a todas las columnas de `B`. El cálculo de la segunda fila de `C` debe ser realizado por el hilo 1 y el de la tercera fila por el hilo 2.

Cuando el número de hilos sea menor al número de filas entonces la distribución es circular. Por ejemplo, si el número de filas de las matrices es 6 y sólo hay 2 hilos, entonces el hilo 0 calcularía la fila 0, 2 y 4 de `C`, y el hilo 1 calcularía las filas 1, 3 y 5 de `C`. En nuestro ejemplo en particular, si hubieran sólo 2 hilos (`T=2`), entonces el cálculo de la tercera fila de `C` sería realizado por el hilo 0.

Finalmente, su programa debe imprimir por pantalla el mayor de los coeficiente de `C`:

```
Coeficiente mayor = 38
```

Es importante que **todos** los hilos colaboren de alguna manera para encontrar el elemento de mayor valor de `C`.

Recuerde, los elementos de `A` y `B` debe pedirlos por teclado. Se sugiere utilizar archivos de entrada para poder utilizar matrices de gran tamaño. Es decir, para el ejemplo anterior, redireccionando la entrada con "`<`", el archivo de entrada hubiera sido:

```
4 2 4
3 1 5
2 5 1
2 1 3
5 1 2
5 5 1
```

Su programa debe pedir solamente los elementos de las matrices `A` y `B` y ningún otro dato. El `N` que indica el tamaño de la matriz, debe ser declarado con `#define` para facilitar su cambio a través del programa.