UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE INGENIERÍA, DIE, COMPUTACIÓN
ESTRUCTURAS DE DATOS Y ALGORITMOS II
M. I. Fco. Javier Rodríguez García

Práctica 11
Introducción a OpenMP

M.I. FCO. JAVIER RODRÍGUEZ, EDA II

Version 0.1, 06/12/20

Tabla de contenido

1. OBJETIVOS

Al finalizar la práctica el alumno:

Habrá realizado programas simples en forma paralela utilizando la biblioteca-extensión **OpenMP**.

2. TERMINOLOGÍA

Programa

Conjunto de instrucciones en un medio no volátil.

Proceso

Conjunto de instrucciones en la memoria principal que se están ejecutando o están por ejecutarse.

Hilo

Subconjunto de instrucciones de un *proceso* en ejecución. Se pueden considerar como una especie de *procesos ligeros*.

Concurrencia

Es la situación donde uno o más procesos están compitiendo por la misma CPU (o núcleo).

Paralelismo

Es la situación donde cada *proceso* tiene su propio CPU (o núcleo).

Multitarea

Mecanismo del sistema operativo para la aparente ejecución simultánea de múltiples procesos.

Multithreading

Mecanismo para que los programadores dividan los procesos en hilos más o menos independientes, con la propiedad de que cuando un hilo se bloquea otro hilo comienza a ejecutarse.

Condiciones de carrera

Es la situación donde dos o más procesos, o dos o más hilos, quieren accesar al mismo recurso compartido, pero solamente uno puede hacerlo a la vez. Ejemplos de recursos compartidos: una área de memoria, una variable, un archivo, un periférico, etc.

La salida del sistema dependerá de cuál proceso llegó primero, pudiendo provocar inconsistencias de datos y comportamientos impredecibles y no compatibles con un sistema determinista.

Algoritmo paralelo

Es un algoritimo que puede ser ejecutado por partes en el mismo instante de tiempo por varias unidades de procesamiento, y al final une todas las partes para obtener el resultado

correcto.

Arquitectura Von Newman

(pend)

Arquitectura Harvard

(pend)

Memoria compartida

Es memoria que puede ser accesada por múltiples procesos o hilos, ya sea para pasarse información entre ellos o para evitar copias redundantes.

Memoria caché

(pend)

Estados de un proceso

Son los estados por los cuales pasa un proceso en su ciclo de vida. Varían de sistema operativo a sistema operativo.

Inconsistencia de datos

(pend)

3. PROGRAMACIÓN PARALELA CON OpenMP

3.1. OpenMP

Es una biblioteca (y en el caso del compilador GNU, una extensión) que proporciona -*directivas*, funciones y variables para crear y controlar la ejecución de programas paralelos. Se puede utilizar tanto para el lenguaje Fortran como para C/{c++}.

Para utilizarla en nuestros programas de C/C++ deberás incluir el encabezado `<omp.h>`, e indicarle al compilador que vas a usar la extensión, con `-fopen`:

```
$ gcc -Wall -std=c99 -osalida.out tu_prog.c -fopenmp
```

De la instrucción anterior podrás notar que `openmp` está indicada como bandera, `-f`, en lugar de biblioteca, `-l`. Como mencioné, OpenMP es una extensión, en los compiladores GNU, no una biblioteca. Si decides correr tu programa sin utilizar la paralelización, entonces basta con que elimines `-fopenmp` de la instrucción de la compilación y recompiles. Tu programa se

compilará como siempre para ejecutarse en un sólo *hilo*. Esta forma de desactivar a OpenMP ayuda a que no tengas que tocar tu código fuente. Más adelante podrás ver un programa tipo *Hola mundo* paralelo.

> Es recomendable que el estudiante se apoye en [CHAPMAN08, ch 4] y [PACHECO11, ch 5] para el desarrollo de las prácticas 11, 12 y 13.

3.2. Directivas

Las *directivas* de OpenMP están formadas de la siguiente manera:

```
#pragma omp construcción [cláusula [,] cláusula ...]
```

La directiva más importante, que de hecho es la que lleva a cabo la paralelización, es la que incluye a la *construcción* `parallel`:

```
#pragma omp parallel
```

Las construcciones pueden estar acompañadas de *cláusulas*. Cada construcción acepta un conjunto de cláusulas. Una cláusula muy utilizada con `parallel` es `num_threads()`, la cual le indica al compilador el número de hilos que se deben usar:

```
#pragma omp parallel num_threads( 4 )
```

> Aunque `num_threads()` parece una función, **no** lo es. Y así como ésta, hay más cláusulas que parecen funciones pero no lo son. En [CHAPMAN08, pp 56] podrá ver la lista de cláusulas para `parallel`.

El cuerpo de cada directiva **debe** ser un *bloque estructurado*. En C, un bloque estructurado es cualquier instrucción simple, o conjunto de instrucciones dentro de llaves, `{` y `}`.

```
// instrucción simple (cualquier llamada a función entra en esta categoría
#pragma omp parallel num_threads( 4 )
    printf( "hola mundo" );

// bloque
#pragma omp parallel num_threads( 4 )
{ // abre la región paralela

   int i;
   i = 5;
   printf( "hola mundo" );

} // cierra la región paralela
```

3.3. Hola mundo paralelo

Nuestro primer programa será el clásico *Hola mundo*, pero paralelizado. Este pequeño programa nos mostrará varias cosas:

1. El entorno de trabajo está instalado.
2. La construcción `parallel`.
3. La directiva `num_threads()`.
4. Las funciones de biblioteca `omp_get_num_threads()` y `omp_get_thread_num()`.

Para diferenciar entre *directivas* y funciones de la biblioteca/extensión deberás recordar que si el elemento del cual dudas es parte de una *construcción*, entonces se trata de una *directiva*; si el elemento es parte del código fuente, entonces es una función.

Listado 1. hello.c

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

#include <omp.h>
// para las funciones de biblioteca de OpenMP

void hello();

int main( int argc, char* argv[] )
{
   if( argc < 2 ){
      printf( "Usage:\n"
            "%s num_threads\n"
            "num_threads is the number of threads\n", argv[0] );
      return 0;
   }

   int thread_count = strtol( argv[1], NULL, 10 );

#pragma omp parallel num_threads( thread_count )
{ // abre la región paralela

   hello();

} // cierra la región paralela

} // cierra main()

void hello()
{
   int my_rank = omp_get_thread_num();

   int thread_count = omp_get_num_threads();

   printf( "Hello from thread %d of %d\n", my_rank, thread_count );
}
```

```
$ gcc -Wall -std=c99 -osalida.out hello.c -fopenmp
```

Para que un programa sea realmente paralelo, el valor de `num_threads` debería coincidir con el número de núcleos en la computadora en la que se ejecutará (en general, $2 \Leftarrow$ `num_threads` $\Leftarrow$ `N`, donde `N` es el número de núcleos físicos). Sin embargo, nada nos impide indicar un valor mayor o menor para dicha variable. En caso de escoger un número mayor el programa dejará de ser *paralelo* para convertirse en *concurrente*. Como experimento ejecuta el programa con valores para `num_threads` mayores al número de núcleos físicos y observa la salida.

3.4. Regiones paralelas

El siguiente programa muestra que dentro de una misma región paralela, uno o más hilos pueden ejecutar códigos distintos. Por simplicidad estaremos utilizando 4 hilos.

Listado 2. reg_para.c

```c
/*
 * Adaptación de [CHAPMAN08, pp55]
 */

#include <stdio.h>
#include <stdlib.h>

#include <omp.h>
// para las funciones de biblioteca de OpenMP

int main()
{

   // aquí iría código secuencial

#pragma omp parallel num_threads( 4 )
{ // abre la región paralela

  printf( "La región paralela es ejecutada por el hilo %d\n", omp_get_thread_num() );

  if( omp_get_thread_num() == 2 ){
     printf( "   El hilo %d hace las cosas diferentes\n", omp_get_thread_num() );
  }

} // cierra la región paralela


   // aquí iría más código secuencial

}
```

3.5. Ejecución condicional de las regiones paralelas

Quizás sea necesario evitar que nuestro programa se paralelice si el número de núcleos es menor a una cierta cantidad. La cláusula `if` de la construcción `parallel` nos permite paralelizar en forma condicional:

Listado 3. reg_para_cond.c

```
#include <stdio.h>
#include <stdlib.h>

#include <omp.h>
// para las funciones de biblioteca de OpenMP

int main()
{
   int n = 2; ①

#pragma omp parallel if( n >= 4 ) num_threads( 4 )
{ // abre la región paralela

  printf( "La región paralela es ejecutada por el hilo %d\n", omp_get_thread_num() );

  if( omp_get_thread_num() == 2 ){
     printf( "   El hilo %d hace las cosas diferentes\n", omp_get_thread_num() );
  }

} // cierra la región paralela

}
```

① Cambie este valor entre 2 y 4 y observe los resultados.

3.6. Variables privadas y compartidas

Todas las variables declaradas fuera de una directiva se consideran compartidas (*shared*); mientras que todas las variables declaradas dentro de una directiva se consideran privadas (*privadas*). Las cláusulas `shared( lista )` y `private( lista )` permiten cambiar dicho comportamiento; es decir, podemos convertir a las variables privadas en compartidas y viceversa.

Condiciones de carrera

Lo anterior es importante porque en la programación paralela pueden aparecer *condiciones de carrera*. Una condición de carrera (o *race condition*) es cuando un recurso (variable, archivo, etc) es accesado sin control por dos o más hilos. El resultado es la corrupción del recurso.

El siguiente ejemplo muestra una variable compartida, `i`, que puede ser accesada por todos los hilos.

Listado 4. hello_no_private.c

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

#include <omp.h>
// para las funciones de biblioteca de OpenMP

int main()
{
   int i;
   // variable compartida en una región paralela

#pragma omp parallel num_threads( 4 )
{ // abre la región paralela

  printf( "Hola mundo\n" );

  for( i = 0; i < 10; ++i ){
     printf( "Contador: %d\n", i );
  }

} // cierra la región paralela

}
```

Este programa se debe ejecutar varias veces para apreciar la condición de carrera. Y en caso de que no llegara a aparecer, el resumen es el siguiente: la salida es un caos.

Un primer intento para evitar las condiciones de carrera es hacer que cada hilo tenga su propia copia de `i`. Para ello tenemos dos caminos:

- Declarar a `i` dentro de la directiva, o
- Marcar a `i` como privada.

El siguiente programa muestra el segundo caso:

Listado 5. hello_private.c

```
#include <stdio.h>
#include <stdlib.h>

#include <omp.h>
// para las funciones de biblioteca de OpenMP

int main()
{
   int i;
   // variable compartida en una región paralela

#pragma omp parallel num_threads( 4 ) private( i ) ①
{ // abre la región paralela

  printf( "Hola mundo\n" );

  for( i = 0; i < 10; ++i ){
     printf( "Contador: %d\n", i );
  }

} // cierra la región paralela

}
```

① Le indicamos a OpenMP que `i` es privada. Esto es, cada hilo tendrá su propia `i`.

Es muy probable que la variable contadora `i` la utilicemos precisamente como eso, una variable contadora dentro de un ciclo `for` o `while`. Y nosotros, a lo largo del curso, hemos estado declarando a dicha variable dentro del propio `for`, por lo que sin duda alguna ésta será casi siempre privada. Sin embargo, habrá situaciones donde esto no sea la norma.

Variables compartidas

El caso contrario es cuando deseamos que una variable privada (declarada dentro de una directiva) sea compartida. Para ello se utiliza la cláusula `shared()`. Sin embargo, las variables compartidas son tan malas como las variables globales y deben evitarse a toda costa, excepto cuando:

- Sean de sólo lectura, por ejemplo, que representen algún tipo de límite.
- Es la única forma que se encontró en el algoritmo.

3.7. Trabajando con colecciones

La tarea más extendida de la programación paralela es el procesamiento de colecciones, elemento por elemento.

El ejemplo a continuación suma un arreglo a otro arreglo, y el resultado se almacena en un tercer arreglo. Para hacer efectiva la programación paralela, nosotros como programadores debemos indicarle al algoritmo el rango de datos que cada hilo va a procesar.

Listado 6. suma_parallel.c

```
#include <stdio.h>

#include <stdlib.h>
// para la función rand()

#include <time.h>
// para inicializar la semilla de aleatoriedad

#include <omp.h>
// para las funciones de biblioteca de OpenMP

// No vamos a usar esta función, pero está aquí para que se vea la
// diferencia con su contraparte paralela.
void suma_sec( int arr1[], int arr2[], int res[], int tam )
{
   for( int i = 0; i < tam; ++i ){
      res[ i ] = arr1[ i ] + arr2[ i ];
   }
}

void suma_par( int arr1[], int arr2[], int res[], int tam )
{
   int mitad = tam / 2;
   // para que este ejemplo funcione correctamente, 'tam' DEBE ser múltiplo
   // de 2

#pragma omp parallel num_threads( 2 )
{ // abre la región paralela

   int id = omp_get_thread_num();
   // obtenemos el id de este hilo

   int inicio = id * mitad;
   // calculamos el inicio de la parte de datos que le corresponde procesar a
   // este hilo

   int fin = inicio + mitad;
   // calculamos el final de la parte de datos que le corresponde procesar a
   // este hilo

   for( int i = inicio; i < fin; ++i ){
      printf( "El hilo %d está calculando res[ %d ]\n", id, i );

      res[ i ] = arr1[ i ] + arr2[ i ];
   }

} // cierra la región paralela

}

```

```
51 #define TAM 10
52
53 int main()
54 {
55    int a[ TAM ];
56    int b[ TAM ];
57    int c[ TAM ];
58
59    srand( time( NULL ) );
60
61    for( int i = 0; i < TAM; ++i ){
62       a[ i ] = rand() % 100;
63       b[ i ] = rand() % 100;
64    }
65
66    suma_par( a, b, c, TAM );
67
68    for( int i = 0; i < TAM; ++i ){
69       printf( "%2d) %d + %d = %d\n", i, a[ i ], b[ i ], c[ i ] );
70    }
71 }
```

Ya que el procesamiento de colecciones es muy importante en la programación en paralelo, OpenMP incluye la construcción `for` que se encarga de distribuir los datos entre los hilos. Ésta se explica a continuación.

3.8. Construcción `for`

Esta directiva existe para procesar específicamente ciclos realizando, entre otras cosas, la distribución de datos entre los hilos. En la sección anterior lo hicimos a mano, pero además de ser tedioso, es propenso a errores. (Pregúntele a su profesor por uno de estos errores.)

Así, para procesar ciclos utilizaremos a la directiva `for`. Y aunque ésta es muy conveniente, tiene algunas restricciones:

Restricciones para la construcción `for`

1. Sólo están permitidos los ciclos `for`; esto es, no se puede utilizar con ciclos `while` ni `do-while`.
2. Sólo puede haber un punto de entrada y uno de salida; esto es, no está permitido utilizar `break`s ni `return`s dentro del cuerpo del ciclo. Como excepción, las llamadas a `exit()` sí están permitidas.
3. El `for` de C puede tomar muchas formas; sólo está permitida la forma *canónica*. Por ejemplo, las siguientes formas del `for(;;)` no están permitidas:

```
for(;;){...} // for infinito
for(int i = n; i > 0; i /= 2 ){...} // /= y *= no están permitidas
for(; i > 0; i += 2 ){...} // está incompleto
```

> Consulte [PACHECO11, pp 226] para una explicación más amplia del `for` *canónico*.

4. El número de iteraciones debe poder determinarse de antemano o con facilidad. Estas restricciones existen por ello.
5. La variable contadora (o de control) debe ser entera (`int` o apuntadora).

El siguiente programa muestra el uso básico de la construcción `for`. Ponga atención al hecho de que esta construcción **no** requiere llaves; esto es así porque el propio ciclo `for` con o sin llaves representa un bloque.

Listado 7. for_1.c

```
#include <stdio.h>
#include <stdlib.h>

#include <omp.h>
// para las funciones de biblioteca de OpenMP

int main()
{
#pragma omp parallel num_threads( 4 )
{ // abre la región paralela

   int id = omp_get_thread_num();

#pragma omp for
  for( int i = 0; i < 20; ++i ){
     printf( "Iteración: %d, Hilo %d\n", i, id );
  }

} // cierra la región paralela

}
```

Una corrida de este programa arrojó la siguiente salida:

```
$ ./for_1.c.out
Iteración: 5, Hilo 1
Iteración: 6, Hilo 1
Iteración: 7, Hilo 1
Iteración: 8, Hilo 1
Iteración: 9, Hilo 1
Iteración: 10, Hilo 2
Iteración: 11, Hilo 2
Iteración: 12, Hilo 2
Iteración: 13, Hilo 2
Iteración: 14, Hilo 2
Iteración: 0, Hilo 0
Iteración: 1, Hilo 0
Iteración: 2, Hilo 0
Iteración: 3, Hilo 0
Iteración: 4, Hilo 0
Iteración: 15, Hilo 3
Iteración: 16, Hilo 3
Iteración: 17, Hilo 3
Iteración: 18, Hilo 3
Iteración: 19, Hilo 3
$
```

Combinación de construcciones

OpenMP permite combinar algunas construcciones. Por ejemplo, podemos combinar las construcciones `parallel` y `for` en una sola:

```
#pragma omp parallel for
    for( ... ; ... ; ... ){
      // cuerpo
   }
```

Si combinamos ambas construcciones el programa anterior nos queda de la siguiente manera:

Listado 8. for_2.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #include <omp.h>
 5 // para las funciones de biblioteca de OpenMP
 6
 7 int main()
 8 {
 9 #pragma omp parallel for num_threads( 4 )
10    for( int i = 0; i < 20; ++i ){
11       printf( "Iteración: %d, Hilo %d\n", i, omp_get_thread_num() );
12    }
13 }
```

Las llaves de apertura y cierre de las regiones paralelas **no** está permitido en esta combinación; esto es, la única instrucción que siga a la directiva **debe** ser un ciclo `for` (con o sin sus llaves, dependiendo de su naturaleza).

¿Esta construcción/combinación funciona con el `for` corto de C++?

Hacer el experimento.

4. PRE-LABORATORIO

El entorno de programación con OpenMP ya está instalado y listo para usarse en nuestro laboratorio. Las siguientes instrucciones son para que Ud instale, o verifique si ya lo tiene instalado, dicho entorno en su casa.

4.1. Linux

El entorno se instala automáticamente junto con las versiones más recientes de **gcc**. Verifique que la biblioteca está instalada ejecutando el siguiente comando en una consola (la instrucción es válida para Linux Mint, Ubuntu y derivados):

```
$ sudo apt install libomp-dev <ENTER>
[sudo] contraseña para eda1:
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
libomp-dev ya está en su versión más reciente (5.0.1-1).
0 actualizados, 0 nuevos se instalarán, 0 para eliminar y 6 no actualizados.
$
```

Si no está instalada, se instalará. Si está instalada el sistema se lo hará saber.

4.2. Windows

Siga estas instrucciones:

https://youtu.be/oiARwi0fJOU

4.3. Mac

Echen un vistazo a este thread y avísenme si lo logran.

https://stackoverflow.com/questions/35134681/installing-openmp-on-mac-os-x-10-11

5. IN-LABORATORIO

5.1. Hola mundo paralelo

Capture, compile y ejecute el programa hello.c. Utilice los siguientes comandos:

```
$ gcc -Wall -std=c99 -fopenmp -o hello.out hello.c <ENTER>
$ ./hello.out 4 <ENTER>
```

5.2. Regiones paralelas

Capture, compile y ejecute el programa reg_para.c. Utilice los siguientes comandos:

```
$ gcc -Wall -std=c99 -fopenmp -o reg_para.out reg_para.c <ENTER>
$ ./reg_para.out <ENTER>
```

5.3. Condiciones de carrera

Capture, compile y ejecute los programas hello_no_private.c y hello_private.c. Use un número muy grande de iteraciones, por ejemplo, 1000.

5.4. Suma paralela 1

Modifique el programa suma_parallel.c:

1. Para que TAM sea 1000 o mayor. Compile, ejecute, y comente los resultados.
2. Escriba una función que imprima en forma paralela el contenido de un arreglo. La firma de la función es:

   ```
   void Imprime( int arr[], int tam );
   ```

5.5. Suma paralela 2

Modifique el programa suma_parallel.c para que utilice la directiva `for` combinada:

```
#pragma omp parallel for num_threads( 2 )
```

Repita el paso 1 del ejercicio anterior. Nombre a este archivo como *suma_parallel_2.c*.

5.6. Ejemplo práctico

(pend)

6. POST-LABORATORIO

Observe las diferencias (a nivel ensamblador) de un programa secuencial y uno paralelo. Compile el siguiente programa utilizando la instrucción a continuación:

```
$gcc -std=c99 -S -fverbose-asm hello.c
```

Guarde el archivo de salida *hello.s* como *hello_no_openmp.s*.

Vuelva a compilar con esta instrucción:

```
$gcc -std=c99 -S -fverbose-asm hello.c -fopenmp
```

Guarde el archivo de salida *hello.s* como *hello_with_openmp.s*.

Abra ambos archivos y busque las diferencias.

7. ENTREGABLES

1. La función `Imprime()` de la sección 4.4.
2. El programa de la sección 4.5. Nómbrelo suma_parallel_2.c.

8. BIBLIOGRAFÍA

8.1. Específica para las prácticas 11, 12 y 13

Estos libros se encuentran en la Biblioteca del Ed. Principal de la F.I.

[CHAPMAN08]

Chapman, Barbara; Jost, Gabrielle; Pas, Ruud van der. *Using OpenMP: portable shared memory parallel programming*. MIT. USA: 2008.

[PACHECO11]

Pacheco, Peter S. *An introduction to parallel programming*. Elsevier Inc. USA: 2011.

En este *padlet* Ud encontrará enlaces útiles sobre OpenMP:

https://padlet.com/eda1_fiunam/5o4qqxj2ov6h

8.2. General para Estructuras de Datos y Algoritmos

[DALE01]

Dale, Nell B; Teague David. *C++ plus Data Structures*. 2nd. ed. Jones and Bartlett Publishers. USA: 2001.

[GONNET99]

Gonnet, G.H.; Baeza-Yates, R. *Handbook of Algorithms and Data Structures In Pascal and C*. 2nd ed. Addison-Wesley Publishers. GREAT BRITAIN: 1991.

[GOODRICH11]

Goodrich, Michael T; Tamassia, Roberto; Mount, David M. *Data Structures and Algorithms in C++*. 2nd. ed. John Wiley & Sons. USA: 2011.

[GREGOIRE11]

Gregoire, Marc; Solter, Nicholas A.; Kleper, Scott J. *Professional C++*. 2nd ed. John Wiley & Sons. USA: 2011.

[JOYANES05]

Joyanes A., Luis; Zahonero M., Ignacio. *Programación en C, Metodología, algoritmos y estructura de datos*. 2da ed. Mc Graw Hill. SPAIN: 2005.

[KOFFMAN08]

Koffman, Elliot B., Wolfgang, Paul A. T. *Estructura de datos con C++, Objetos, abstracciones y diseño*. McGraw Hill. MÉXICO: 2008.

[KRUSE97]

Kruse, Robert L; Tondo, Clovis L; Leung, Bruce P. *Data structures and program design in C*. Prentice Hall. USA: 1997.

[NYHOFF92]

Nyhoff, Larry R. *Data structures and program design in Pascal*. 2nd ed. Macmillan Publishing. USA:1992.

[SHAFFER11]

Shaffer, Clifford A. *Data structures and algorithm analysis in C++*. 3rd ed. Dover Publications Inc. USA: 2011.

[THOMAS13]

Thomas, Dave; Fowler, Chad; Hunt, Andy. *Programming Ruby 1.9 & 2.0*. The Pragmatic Programmers. USA: 2013.

[WEISS13]

Weiss, Mark A. *Estructuras de datos en Java*. 4ta ed. Pearson Educación, S.A. SPAIN: 2013.

Referencia del lenguage C

http://www.cplusplus.com/reference/