Guía práctica de estudio 11

Introducción a OpenMP.

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaño

Facultad de Ingeniería

U.N.A.M.

Guía Práctica 11

Estructura de datos y Algoritmos II

Introducción a OpenMP

Objetivo: El estudiante conocerá y aprenderá a utilizar algunas de las directivas de OpenMP utilizadas para realizar programas paralelos.

Al final de la práctica el estudiante tendrá las herramientas necesarias para trabajar con algunas directivas de OpenMP en el lenguaje C y podrá realizar su primer programa en paralelo.

Antecedentes

- Análisis previo del concepto de proceso e hilo visto en clase teórica.
- Conocimientos sólidos de programación en Lenguaje C.

Introducción

Conceptos básicos en el procesamiento paralelo.

La entidad software principal en un sistema de cómputo es **el proceso**, y su contraparte en hardware es **el procesador** o bien la unidad de procesamiento. En la actualidad tanto los sistemas de cómputo como los dispositivos de comunicación cuentan con varias unidades de procesamiento, lo que permite que los procesos se distribuyan en éstas para agilizar la funcionalidad de dichos sistemas/dispositivos. Esta distribución la puede hacer el sistema operativo o bien los desarrolladores de sistemas de software. Por lo tanto, es necesario entender lo que es un proceso, sus características y sus variantes.

El proceso.

Un proceso tiene varias definiciones, por mencionar algunas: a) programa en ejecución, b) procedimiento al que se le asigna el procesador y c) centro de control de un procedimiento. Por lo que no solamente un proceso se limita a ser un programa en ejecución. Por ejemplo, al ejecutar un programa éste puede generar varios procesos que no necesariamente están incluidos en un programa o bien partes del código de un mismo programa son ejecutados por diversos procesos.

Los procesos se conforman básicamente de las siguientes partes.

- El código del programa o segmento de código (sección de texto)
- La actividad actual representada por el contador de programa (PC) y por los contenidos de los registros del procesador.
- Una pila o stack que contiene datos temporales, como los parámetros de las funciones, las direcciones de retorno y variables locales. También el puntero de pila o stack pointer.
- Una sección de datos que contiene variables globales.

- Un heap o cúmulo de memoria, que es asignada dinámicamente al proceso en tiempo de ejecución.

Como se muestra en la figura 11.1

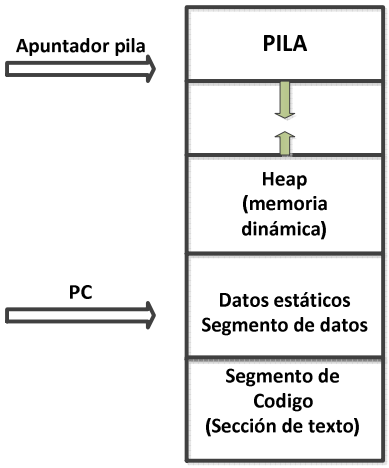

Figura 11.1

Respecto a los estados de un proceso, se puede encontrar en cualquiera de los siguientes: Listo, en ejecución y bloqueado. Los procesos en el estado listo son los que pueden pasar a estado de ejecución si el planificador los selecciona. Los procesos en el estado ejecución son los que se están ejecutando en el procesador en ese momento dado. Los procesos que se encuentran en estado bloqueado están esperando la respuesta de algún otro proceso para poder continuar con su ejecución (por ejemplo, realizar una operación de E/S).

Hilos

Un proceso puede ser un programa que tiene solo un hilo de ejecución. Por ejemplo, cuando un proceso está ejecutando un procesador de texto, se ejecuta solo un hilo de instrucciones. Este único hilo de control permite al proceso realizar una tarea a la vez, por ejemplo, el usuario no puede escribir simultáneamente caracteres y pasar al corrector ortográfico dentro del mismo proceso, para hacerlo se necesita trabajar con varios hilos en ejecución, para llevar a cabo más de una tarea de forma simultánea [1]. Entonces un proceso tradicional (o proceso pesado) tiene un solo hilo de control que realiza una sola tarea y un ´proceso con varios hilos de control, puede realizar más de una tarea a la vez. Así también se pueden tener varios procesos y cada uno con varios hilos. Figura11.2.

Un proceso es un hilo principal que puede crear hilos y cada hilo puede ejecutar concurrentemente varias secuencias de instrucciones asociadas a funciones dentro del mismo proceso principal. Si el hilo principal termina entonces los hilos creados por este salen también de ejecución.

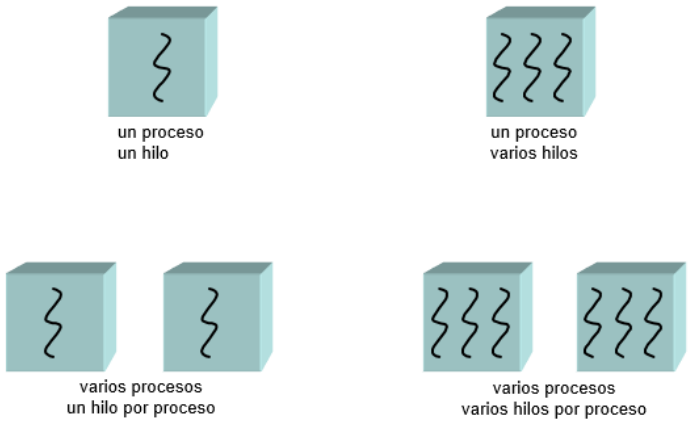

Figura 11.2

Todos los hilos dentro de un proceso comparten la misma imagen de memoria, como el segmento de código, el segmento de datos y recursos del sistema operativo. Pero no comparte el contador de programa (por lo que cada hilo podrá ejecutar una sección distinta de código), la pila en la que se crean las variables locales de las funciones llamadas por el hilo, así como su estado. Figura 11.3.

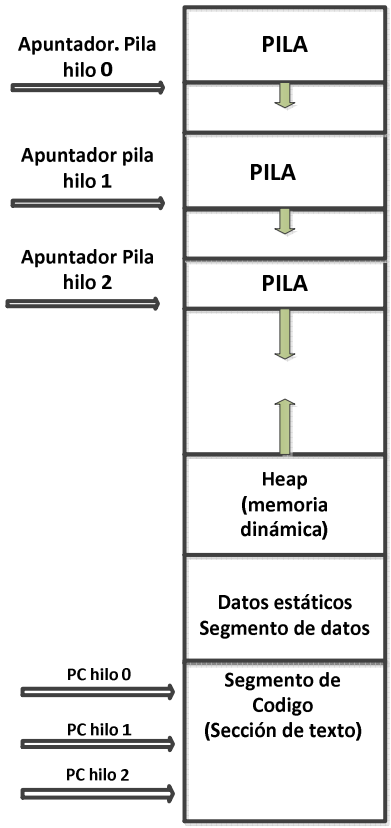

Figura 11.3

Concurrencia vs Paralelismo

Concurrencia: Es la existencia de varias actividades realizándose simultáneamente y requieren de una sincronización para trabajar en conjunto. Se dice que dos procesos o hilos son concurrentes cuando están en progreso simultáneamente pero no al mismo tiempo.

Figura 11.4

Paralelismo: Actividades que se pueden realizar al mismo tiempo. Dos procesos o hilos son paralelos cuando se están ejecutándose al mismo tiempo, para lo cual se requiere que existan dos unidades de procesamiento.

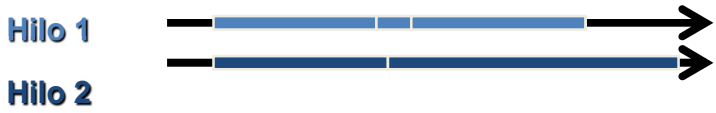

Figura 11.5

Programación secuencial, concurrente y paralela

Cuando se realiza un programa secuencial, las acciones o instrucciones se realizan una tras otra, en cambio en un programa concurrente se tendrán actividades que se pueden realizar de forma simultánea y se ejecutarán en un solo elemento de procesamiento.

En un programa paralelo se tienen acciones que se pueden realizar también de forma simultánea, pero se pueden ejecutar en forma independiente por diferentes unidades de procesamiento.

Por lo anterior, para tener un programa paralelo se requiere de una computadora paralela, es decir una computadora que tenga dos o más unidades de procesamiento.

Memoria Compartida en una computadora

En el hardware de una computadora, la memoria compartida se refiere a un bloque de memoria de acceso aleatorio a la que se puede acceder por varias unidades de procesamiento diferentes. Tal es el caso de las computadoras *Multicore* donde se tienen varios núcleos que comparten la misma memoria principal. Figura 11.6.

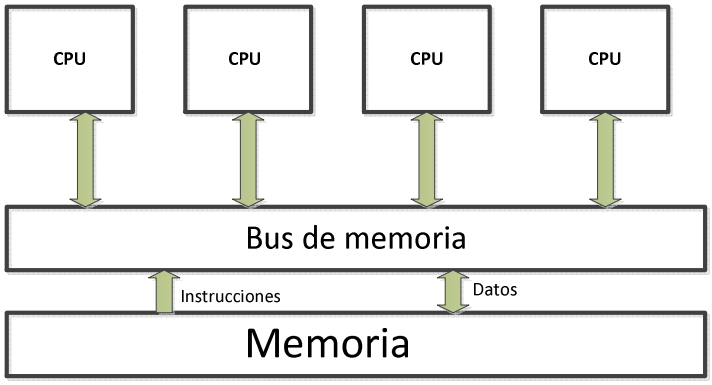

Figura 11.6

En software, la memoria compartida es un método de comunicación entre procesos/hilos, es decir, una manera de intercambiar datos entre programas o instrucciones que se ejecutan al mismo tiempo. Un proceso/hilo creará un espacio en la memoria RAM a la que otros procesos pueden tener acceso.

OpenMP (Open Multi-Processing)

Es un interfaz de programación de aplicaciones (API) multiproceso portable, para computadoras paralelas que tiene una arquitectura de memoria compartida.

OpenMp está formado

- Un conjunto de directivas del compilador, las cuales son ordenes abreviadas que instruyen al compilador para insertar ordenes en el código fuente y realizar una acción en particular.
- una biblioteca de funciones y
- variables de entorno.

que se utilizan para paralelizar programas escritos es lenguaje C, C++ y Fortran.

Entonces para utilizar OpenMP basta con contar con un compilador que incluya estas extensiones al lenguaje. En [2] se pueden encontrar la lista de compiladores que implementa este API.

Arquitectura de OpenMP

Para paralelizar un programa se tiene que hacer de forma explícita, es decir, el programador debe analizar e identificar qué partes del problema o programa se pueden realizar de forma concurrente y por tanto se pueda utilizar un conjunto de hilos que ayuden a resolver el problema.

OpenMp trabaja con la llamada arquitectura fork-join, donde a partir del proceso o hilo principal se genera un número de hilos que se utilizarán para la solución en paralelo llamada región paralela y después se unirán para

volver a tener solo el hilo o proceso principal. El programador especifica en qué partes del programa se requiere ese número de hilos. De aquí que se diga que OpenMP combina código serial y paralelo. Figura 11.7

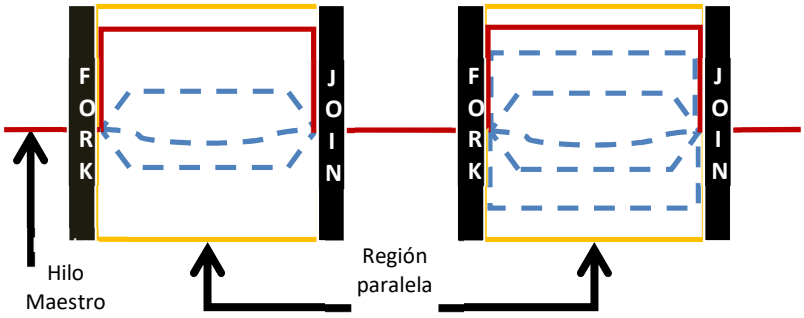

Figura 11.7

Directivas o pragmas

Agregar una directiva o pragma en el código es colocar una línea como la que sigue

#pragma omp **nombreDelC*onstructor*** ***<clausula o clausulas>***

Donde como se puede observar se tiene los llamados constructores y las cláusulas. Los constructores es el nombre de la directiva que se agrega y las clausulas son atributos dados a algunos constructores para un fin específico; una clausula nunca se coloca si no hay antes un constructor.

En la parte del desarrollo de esta guía se irán explicando constructores, cláusulas y funciones de la biblioteca omp.h. mediante actividades que facilitan la comprensión

Desarrollo

Para aprender más sobre openMP en lo siguiente se desarrollan actividades que facilitaran la comprensión.

Para el desarrollo de la parte práctica, en esta guía se utilizará el compilador gcc de Linux. Por lo que para realizar las siguientes actividades es necesario estar en un ambiente Linux contar con un editor (vi, emacs, gedit, nano, etc..) y tener el compilador gcc (versión mayor o igual a la 4.2).

El primer constructor a revisar es el más importante, ***parallel,*** el cual permite crear regiones paralelas, es decir generar un número de hilos que ejecutarán ciertas instrucciones y cuando terminen su actividad se tendrá solo al maestro. Figura 11.8

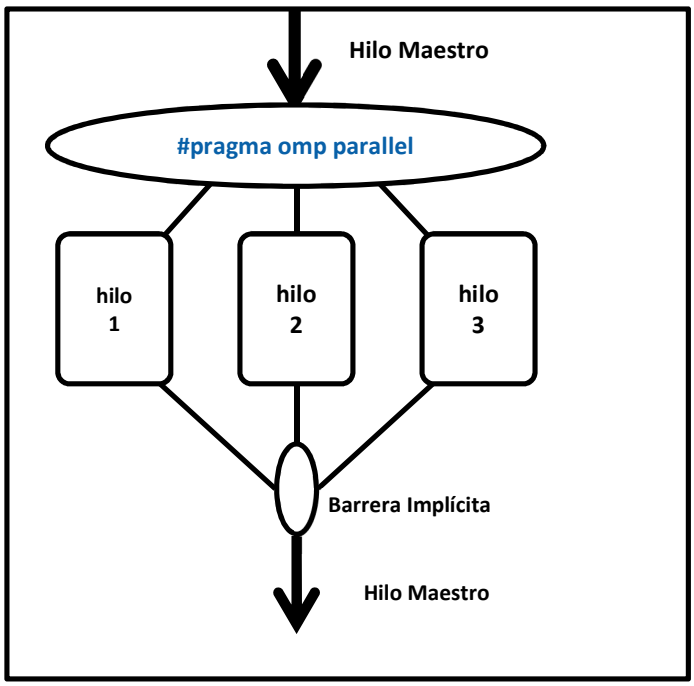

Figura 11.8

La sintaxis es como sigue:

```
#pragma omp parallel
{
//Bloque de código
}
```

Nota: La llave de inicio de la región debe ir en el renglón siguiente de la directiva, si no se coloca así ocurrirá error.

Actividad 1

En un editor de texto teclear el siguiente código y guardarlo con extensión ".c" , por ejemplo *hola.c* .

```
#include <stdio.h>

int main() {
        int i;
        printf("Hola Mundo\n");
        for(i=0;i<10;i++)
                printf("Iteración:%d\n",i);
        printf("Adiós \n");
return 0;
}
```

Después desde la consola o línea de comandos de Linux, situarse en el directorio donde se encuentra el archivo fuente y compilarlo de la siguiente manera.

gcc hola.c -o hola

Para ejecutarlo, igual en la línea de comandos escribir:

./hola

Una vez que ya se tiene el código en su versión serial, se formará una región paralela. Entonces, agregar al código anterior el constructor **parallel** desde la declaración de la variable y hasta antes de la última impresión a pantalla.

```
#include <stdio.h>

int main() {
#pragma omp parallel
{
        int i;
        printf("Hola Mundo\n");
        for(i=0;i<10;i++)
                printf("Iteración:%d\n",i);
}
        printf("Adiós \n");
return 0;
}
```

Guardar el archivo, compilarlo y ejecutarlo.

Para compilarlo hay que indicar que se agregaran las directivas de openMP, lo que se realiza agregando la bandera en la compilación -fopenmp. Ejemplo.

gcc -fopenmp hola.c -o hola.c

Para ejecutarlo, se realiza de la misma manera que el secuencial.

¿Qué diferencia hay en la salida del programa con respecto a la secuencial?

_____

¿Por qué se obtiene esa salida?

_____

Actividad 2

En cada región paralela hay un número de hilos generados por defecto y ese número es igual al de unidades de procesamiento que se tengan en la computadora paralela que se esté utilizando, en este caso el número de núcleos que tenga el procesador.

En el mismo código, cambiar el número de hilos que habrá en la región paralela a un número diferente n (entero), probar cada una de las formas indicadas a continuación. Primero modificar, después compilar y ejecutar nuevamente el programa en cada cambio.

1-Modificar la variable de ambiente *OMP_NUM_THREADS desde* la consola, de la siguiente forma:

export OMP_NUM_THREADS=4

2- Cambiar el número de hilos a n (un entero llamando a la función **omp_set_num_threads(n)** que se encuentra en la biblioteca omp.h (hay que incluirla).

3-Agregar la cláusula **num_threads(n)** seguida después del constructor **parallel,** esto es**:**

#pragma omp parallel num_threads

¿Qué sucedió en la ejecución con respecto al de la actividad 1?

Actividad 3

En la programación paralela en computadoras que tienen memoria compartida puede presentarse la llamada condición de carrera (race condition) que ocurre cuando varios hilos tienen acceso a recursos compartidos **sin control.** El caso más común se da cuando en un programa varios hilos tienen acceso concurrente a una misma dirección de memoria (variable) y todos o algunos en algún momento intentan escribir en la misma localidad al mismo tiempo. Esto es un conflicto que genera salidas incorrectas o impredecibles del programa.

En OpenMP al trabajar con hilos se sabe que hay partes de la memoria que comparten entre ellos y otras no. Por lo que habrá variables que serán compartidas entre los hilos, (a las cuales todos los hilos tienen acceso y las pueden modificar) y habrá otras que serán propias o privadas de cada uno.

Dentro del código se dirá que cualquier variable que esté declarada fuera de la región paralela será compartida y cualquier variable declarada dentro de la región paralela será privada.

Del código que se está trabajando, sacar de la región paralela la declaración de la variable entera i , compilar y ejecutar el programa varias veces.

```
#include <stdio.h>

int main() {
int i;
#pragma omp parallel
{
                printf("Hola Mundo\n");
                for(i=0;i<10;i++)
                        printf("Iteración:%d\n",i);
}
                printf("Adiós \n");
return 0;
}
```

¿Qué sucedió? Y ¿Por qué? _____

Actividad 4

Existen dos cláusulas que pueden forzar a que una variable privada sea compartida y una compartida sea privada y son las siguientes:

shared(): Las variables colocadas separadas por coma dentro del paréntesis serán compartidas entre todos los hilos de la región paralela. Sólo existe una copia, y todos los *hilos* acceden y modifican dicha copia.

private(): Las variables colocadas separadas por coma dentro del paréntesis serán privadas. Se crean p copias, una por hilo, las cuales no se inicializan y no tienen un valor definido al final de la región paralela ya que se destruyen al finalizar la ejecución de los hilos.

Al código resultante de la actividad 3, agregar la cláusula private( ) después del constructor parallel y colocar la variable i:

```
#pragma omp parallel private(i)

{

}
```

¿Que sucedió?_____

Actividad 5

Dentro de una región paralela hay un número de hilos generados y cada uno tiene asignado un identificador. Estos datos se pueden conocer durante la ejecución con la llamada a las funciones de la biblioteca omp_get_num_threads() y omp_get_thread_num() respectivamente.

Probar el siguiente ejemplo, y notar que para su buen funcionamiento se debe indicar que la variable *tid* sea privada dentro de la región paralela, ya que de no ser así todos los hilos escribirán en la dirección de memoria asignada a dicha variable sin un control (race condition), es decir "competirán " para ver quién llega antes y el resultado visualizado puede ser inconsistente e impredecible.

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

int main(){
    int tid,nth;

    #pragma omp parallel private(tid)
    {

        tid = omp_get_thread_num();
        nth = omp_get_num_threads();
        printf("Hola Mundo desde  el hilo %d de un total de %d\n",tid,nth);


    }
        printf("Adios");
return 0;
}
```

Probar el ejemplo quitando del código la cláusula *private* para visualizar el comportamiento del programa.

¿Qué sucedió? _____

Actividad 6

Se requiere realizar la suma de dos arreglos unidimensionales de 10 elementos de forma paralela utilizando solo dos hilos. Para ello se utilizará un paralelismo de datos o descomposición de dominio, es decir, cada hilo trabajará con diferentes elementos de los arreglos a sumar A y B, pero ambos utilizarán el mismo algoritmo para realizar la suma. Figura.11.9.

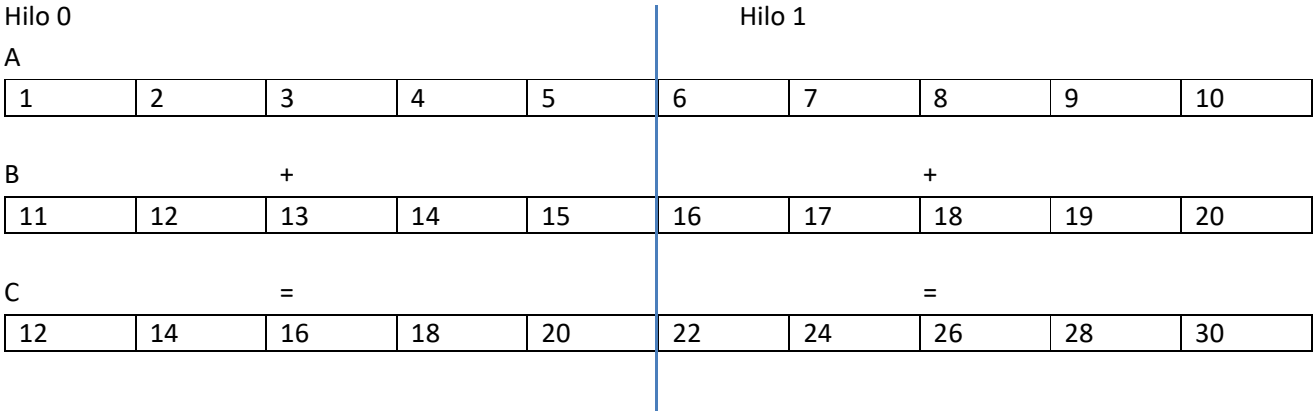

Figura 11.9

Se parte de la versión serial, donde la suma se realiza de la siguiente manera (se asume que A y B ya tienen valores para ser sumados):

```
for(i=0; i<10; i++)
        C[i]=A[i]+B[i]
```

Actividad 6.1

Realizar programa en su versión serial.

```
#include<stdio.h>
#include<stdlib.h>
#include <math.h>
#define n 10

void llenaArreglo(int *a);
void suma(int *a,int *b,int *c);

main(){

int max,*a,*b,*c;

a=(int *)malloc(sizeof(int)*n);
b=(int *)malloc(sizeof(int)*n);
```

```
c=(int *)malloc(sizeof(int)*n);

llenaArreglo(a);
llenaArreglo(b);

suma(a,b,c);
}

void llenaArreglo(int *a){
 int i;
 for(i=0;i<n;i++)
   {
    a[i]=rand()%n;
   printf("%d\t", a[i]);
    }
   printf("\n");
}

void suma(int *A, int *B, int *C){
int i;
  for(i=0;i<n;i++){
       C[i]=A[i]+B[i];
        printf("%d\t", C[i]);
   }
}
```

Actividad 6.2

Para la versión paralela, el hilo 0 sumará la primera mitad de A con la primera de B y el hilo 1 sumará la segunda mitad de A con la segunda de B. Para conseguir esto cada hilo realizará las mismas instrucciones, pero utilizará índices diferentes para referirse a diferentes elementos de los arreglos, entonces cada uno iniciará y terminará el índice i en valores diferentes

```
for(i=inicio; i<fin; i++)
        C[i]=A[i]+B[i]
```

Inicio y fin se pueden calcular de la siguiente manera, siendo tid el identificador de cada hilo:

Inicio = tid* 5

fin = (tid+1)*5-1

Entonces la función donde se realiza la suma queda:

```
void suma(int *A, int *B, int *C){
int i,tid,inicio,fin;

  omp_set_num_threads(2);
  #pragma omp parallel private(inicio,fin,tid,i)

  {
     tid = omp_get_thread_num();

     inicio = tid* 5;
     fin = (tid+1)*5-1;

     for(i=inicio;i<fin;i++){
          C[i]=A[i]+B[i];
          printf("hilo %d calculo C[%d]= %d\n",tid,i, C[i]);
      }

  }
}
```

Implementar el código completo.

Actividad 7

Otro constructor es el **for,** el cual divide las iteraciones de una estructura de repetición *for*. Para utilizarlo se debe estar dentro de una región paralela.

Su sintaxis es:

```
#pragma omp parallel
{
…..
#pragma omp for
          for(i=0;i<12;i++) {
             Realizar Trabajo();
          }
.....
}
```

La variable índice de control i se hará privada de forma automática. Esto para que cada hilo trabaje con su propia variable i.

Este constructor se utiliza comúnmente en el llamado paralelismo de datos o descomposición de dominio, lo que significa que, cuando en el análisis del algoritmo se detecta que varios hilos pueden trabajar con el mismo algoritmo o instrucciones que se repetirán, pero sobre diferentes datos y no hay dependencias con iteraciones anteriores.

Por lo anterior, **no siempre** es conveniente dividir las iteraciones de un ciclo for.

Modificar el código de la actividad 1, de manera que se dividan las iteraciones de la estructura de repetición for.

```
#include <stdio.h>

int main() {
#pragma omp parallel
{

                printf("Hola Mundo\n");
               #progma omp for
                for(i=0;i<10;i++)
                        printf("Iteración:%d\n",i);
}
                printf("Adiós \n");
return 0;
}
```

Actividad 8

Realizar la suma de dos arreglos unidimensionales de n elementos de forma paralela, utilizando los hilos por defecto que se generen en la región paralela y el constructor for.

Como se explicó en la actividad 6 los hilos realizarán las mismas operaciones, pero sobre diferentes elementos del arreglo y eso se consigue cuando cada hilo inicia y termina sus iteraciones en valores diferentes para referirse a diferentes elementos de los arreglos A y B. Esto lo hace el constructor for, ya que al dividir las iteraciones cada hilo trabaja con diferentes valores del índice de control.

Entonces la solución queda:

```
void suma(int *A, int *B, int *C){
int i,tid;
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        #pragma omp for
        for(i=0;i<n;i++){
            C[i]=A[i]+B[i];
            printf("hilo %d calculo C[%d]= %d\n",tid,i, C[i]);
        }
    }
}
```

Actividad 9

Ejercicios sugeridos por el profesor

En las siguientes guías se revisarán otros constructores y cláusulas.

Referencias

[1] Abraham Silberschatz, Peter Baer Galvin & Greg Gagne
Fundamentos de Sistemas Operativos
Séptima Edición
MacGrall Hill

[2] http://openmp.org/wp/

[3] Introduction to parallel programming, Intel Software College, junio 2007

[4] B. Chapman ,Using OpenMP, The MIT Press, 2008