Guía de Actividades Prácticas: Estudio de la Programación Paralela con Memoria Compartida y OpenMP

1. Conceptos básicos para el cómputo en paralelo

La unidad de software elemental activa y básica durante el cómputo, es el proceso. Los procesos se ejecutan en procesadores y podemos tener uno o más procesos en un procesador. La asociación de los procesos y los procesadores a los que están asignados, se llama mapeo.

Actualmente encontramos varios procesadores en una misma unidad de cómputo. Un mapeo adecuado permite agilizar las funcionalidades de dichos sistemas. Este mapeo lo puede hacer el sistema operativo o los desarrolladores de software en manera explícita al programar.

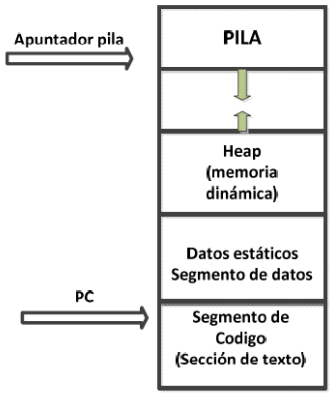
Es muy importante comprender qué es un proceso y sus variantes.

Procesos

Un proceso tiene varias definiciones, o modos de definirse, ya que no es solo un programa en ejecución. Una ejecución puede generar varios subprocesos (que son procesos) y no necesariamente estaban programados y varios procesos pueden estar ejecutando parte del código de un mismo programa.

Los procesos se componen de las siguientes partes básicas:

1) Segmento del código del programa
2) Instrucción actual señalada por el Program Counter (PC)
3) La pila con datos temporales (parámetros de funciones, variables locales, …) Y el puntero al inicio de la pila.
4) La sección de datos estáticos, variables globales.
5) La sección del montículo de memoria o heap, que se va asignando dinámicamente o en tiempo de ejecución.



Recordamos los estados de un proceso: *listo, en ejecución y bloqueado*. El planificador selecciona el nuevo de los estados *listo* para pasarlos a estar en *ejecución* como también los selecciona del estado de *ejecución* para pasarlos al estado *bloqueado* y a la espera que se liberen los recursos que necesitan.

Hilos

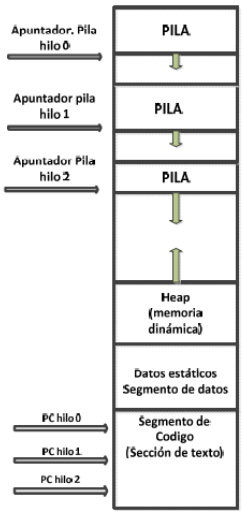
Cuando se ejecuta un programa, la CPU utiliza el valor del *contador de programa* del proceso para determinar cuál instrucción debe ejecutar a continuación. El flujo de instrucciones que se genera se denomina *hilo de ejecución del programa*, y es el flujo de control para el proceso representado por la secuencia de direcciones de instrucciones indicadas por el contador del programa.

Desde el punto de vista del programa el *hilo* es una secuencia ininterrumpida de direcciones, pero desde el punto de vista del procesador los hilos de ejecución se entrelazan y la ejecución cambia de un proceso a otro, lo que se llama *conmutación de contexto*.

Supongamos que el proceso 1 ejecuta sus direcciones 245, 246 y 247 en un ciclo. Su hilo de ejecución puede representarse así: 245_1, 246_1, 247_1, 245_1, 246_1, 247_1, 245_1, 246_1, 247_1, ... donde el subíndice 1 representa el único hilo.

Ahora consideremos también el proceso 2 que ejecuta las direcciones 11, 12, 13 en un ciclo. La CPU ejecuta en el orden 245_1, 246_1, 247_1, 245_1, 246_1, 11_2, 12_2, 13_2, 247_1, 245_1, 11_2, 12_2, 246_1, 247_1, 13_2, ...

Si el programa con un hilo de ejecución está ejecutando un procesador de textos, entonces el usuario no puede realizar más de una tarea por vez. Si quiere escribir y que el corrector ortográfico le corrija dentro del mismo proceso, entonces necesita dos hilos de ejecución y ejecutarían con una distribución similar al ejemplo anterior. Un proceso solo tiene un hilo de ejecución o proceso pesado. Cuando se descompone en varios hilos se les dice procesos livianos.



Todos los hilos dentro de un proceso comparten la misma imagen de memoria como el segmento para código, el segmento de datos y no comparte el contador de programa (cada hilo está ejecutando en distintas direcciones) ni la pila de datos temporales.

Los sistemas operativos por lo general presentan un importante paralelismo el que mejoran notablemente utilizando varios hilos de ejecución simultáneos. Por lo cual, aprovechan el hardware paralelo.

Concurrencia vs. Paralelismo

La *concurrencia* es el hecho de compartir recursos en el mismo marco de tiempo. Esto significa que varios procesos comparten la misma CPU (o sea, son ejecutados concurrentemente), o la memoria o algún dispositivo de entrada/salida.

El manejo incorrecto de la concurrencia es la causa de programas que fallan sin motivo aparente cuando tienen la misma entrada y en otras oportunidades no tuvieron fallo.

Las computadoras personales con varios procesadores y los sistemas distribuidos son ejemplo de arquitecturas en las que el control de la concurrencia adquiere un significado de mayor importancia para los desarrolladores de sistemas.

Si tenemos un solo procesador la concurrencia no puede mejorarse con paralelismo.

El *paralelismo* permite realizar varias tareas simultáneamente, al mismo tiempo. **Para que dos procesos se ejecuten al mismo tiempo necesitan de hardware que se lo permita, o sea que tenga más de una unidad de procesamiento**.

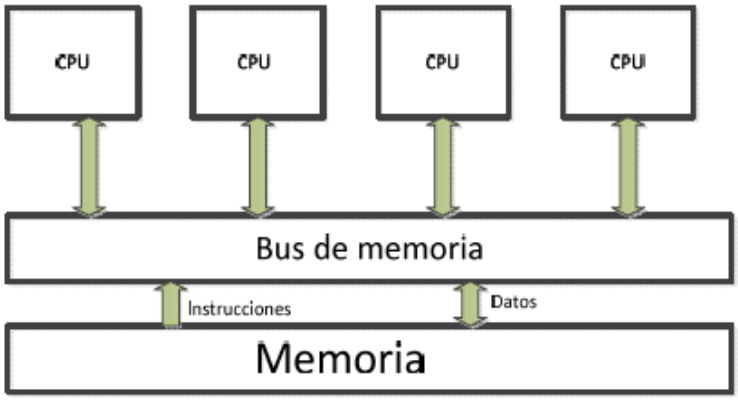
A modo de resumen:

A la ejecución de cada secuencia de instrucciones se le conoce como un hilo de procesamiento o thread. Cada thread posee sus propios registros de control y su propio stack de datos, mientas que comparte el uso de la memoria del montículo (heap) y data con los demás threads del programa. ***En las computadoras multi-core logra la mejor eficiencia cuando cada CPU ejecuta sólo un thread****.*

Cuando se tiene un solo hilo de ejecución en un solo procesador tenemos ***procesamiento secuencial****. Si tenemos varios hilos de ejecución y un solo procesador estamos ejecutando* ***procesamiento concurrente****, pero no en paralelo. Sólo cuando hay varios hilos de procesamiento en distintos procesadores decimos que tenemos* ***programación paralela****.*

2. Paradigma de programación paralela con Memoria Compartida

En este documento nos ocuparemos del procesamiento en paralelo que aprovecha el hardware que comparte una memoria principal en común entre varios núcleos de procesamiento. Las computadoras *multicore* responden a este paradigma ya que ofrecen varios núcleos (*cores*) que comparten una memoria común (espacio de direcciones de variables). Una representación de este modelo de cómputo se puede ver en el siguiente esquema:



La tecnología de software que permita aprovechar esta plataforma de hardware son los lenguajes o bibliotecas (libraries) que permiten la comunicación entre procesos/hilos, es decir intercambiar datos entre procesos o instrucciones que se ejecutan al mismo tiempo. **Un proceso/hilo creará un espacio en la memoria compartida (RAM) a la que otros procesos pueden tener acceso. Esto es muy beneficioso, pero siempre que la memoria esté correctamente administrada.**

3. OpenMP (Open Multi-Processing)

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

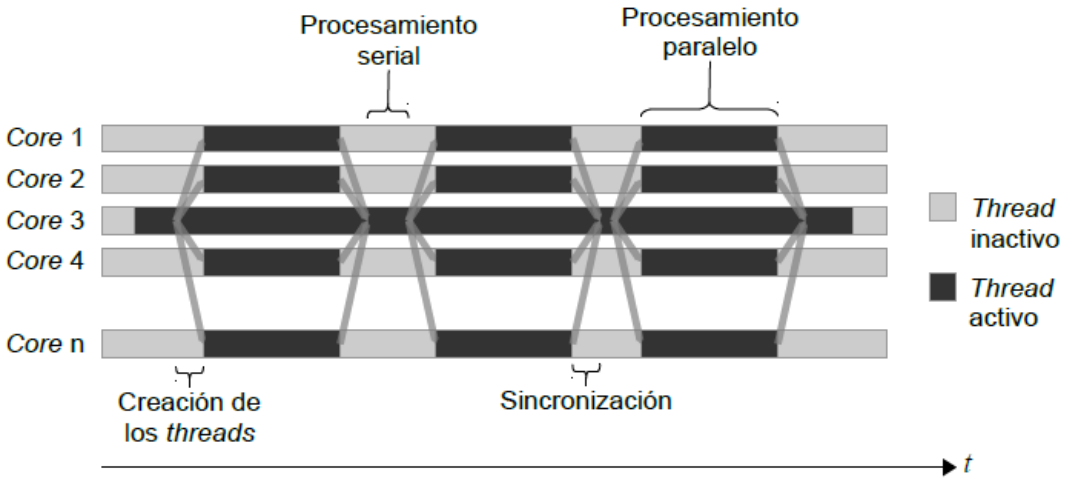
OpenMP está compuesto por:

a) Un conjunto de directivas del compilador, las cuales son órdenes abreviadas que instruyen al compilador para insertar órdenes en el código fuente y realizar una acción particular.
b) Una biblioteca de funciones
c) Un conjunto de variables de entorno.

Se utilizan en programas que pueden escribirse en lenguajes habituales de desarrollo como C, C++, Fortran.

Para paralelizar un programa se deben utilizar estas directivas de modo explícito en el código. 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 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 la ejecución del hilo o proceso principal, llamado **región secuencial**. El programador determina en qué parte del programa requiere un número de hilos en particular y los lanza. **OpenMP combina código serial con código en paralelo,** como se ve en el siguiente esquema:



Directivas o Pragmas

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

#pragma omp nombreDelConstructor <clausula/s>

La directiva se compone de un constructor, que es el nombre de la misma, y las cláusulas que son atributos dados a algunos constructores para un fin específico.

A continuación, iremos explicando constructores, cláusula y funciones de la biblioteca **omp.h** mediante actividades para ir comprendiendo cómo actúan.

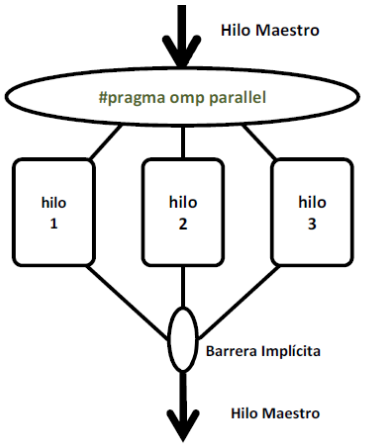
Será necesario contar con un compilador de C o C++ como gcc o g++, en Linux o con un entorno de desarrollo para Windows que haga referencia a las bibliotecas de openMP al momento de compilar. En el caso de usar línea de comando deberán utilizar un editor de textos, en lo posible que marque la sintaxis de C.

Parallel

La construcción ***parallel*** crea un equipo de hilos y comienza con la ejecución paralela.

A continuación, se esquematiza el efecto de desencadenar esta cláusula en el código:

Se puede ver cómo se generan los hilos que se pidan en las directivas o uno por cada core del procesador, en su defecto.



Se utiliza mediante la siguiente instrucción:

```
#pragma omp parallel
{
        <BLOQUE DE CÓDIGO>
}
```

4. Actividades Prácticas de Programación

Actividad 1:

En un editor ingresar el siguiente código y guardarlo como *hola.c*

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

Desde la línea de comandos ejecutar

gcc (o g++) –o hola hola.c

Para ejecutarlo ingresar en la línea de comandos

./hola

Ahora agregar el siguiente constructor paralelo:

```
#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.

Si están en Linux se deberá escribir de esta manera la compilación:

gcc (o g++) –o holaPar holaPar.c -fopenmp

- ¿Qué diferencia hay en la salida?
- Explique en función del hardware sobre el que está ejecutando el programa.

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 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 (debe estar incluida).
3. Agregar la clúsula num_threads(n) seguida después del constructor parallel, esto es:
   #pragma omp parallel num_threads(n)

Actividad 3:

Crear un programa en OpenMP que informe

- la cantidad de hilos que se pidió al ejecutar el programa (lo solicitamos desde la línea de comandos),

- la cantidad de procesadores disponibles y que muestre el modelo fork/join mostrando en pantalla que está ejecutando un solo hilo,
- a continuación, que se ingresa a la región paralela mostrando que se abre la ejecución de n hilos
- y por último se cierra la región paralela y sigue con 1 hilo de ejecución para finalizar.

Escribir en un archivo de texto el programa, cuáles son las directivas openMP que se utiliza en cada paso con sus parámetros o cláusulas. Subirlo al Campus con su apellido-Tarea1. Utilizar las funciones que se muestran a continuación.

Rutinas a tiempo de ejecución

Rutinas de entorno de ejecución

Las rutinas de entorno de ejecución afectan y monitorean hilos, procesos y el entorno paralelo en general.

- *void omp_set_num_threads (int num_threads);*

Afecta el número de hilos usados por las regiones paralelas subsecuentes que no especifican una clausula num_threads.

- *int omp get_num_threads(void);*

Regresa el número de hilos de equipo actual.

- *int omp get_max_threads(void);*

Regresa el número máximo de hilos que pueden ser usados por un nuevo equipo que use la construcción parallel sin la cláusula num_threads.

- *int omp get_thread_num(void);*

Regresa la identidad del hilo en que se encuentra donde la identidad tiene un rango de 0 al tamaño del equipo de hilos menos 1.

- *int omp get_num_procs(void);*

Regresa el número de procesadores disponibles para el programa

---

Actividad 4:

En esta programación puede presentarse la llamada condición de carrera o *race condition* que ocurre cuando carios 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.

Continuar con el código de la actividad 2. 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? ¿Por qué sucede esto? De una explicación.

Actividad 5

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 comas 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. Si tenemos p hilos entonces 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 4, agregar la cláusula *private( )* después del constructor *parallel* y colocar la variable *i*:

```
#pragma omp parallel private(i)
{
}
```

¿Qué sucedió?

Actividad 6

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 que se pueden seleccionar de la lista anterior (elegir la correcta).

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


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

Actividad 7:

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. Ver la figura siguiente:

Hilo 0					Hilo 1				
A									
1	2	3	4	5	6	7	8	9	10
B		+					+		
11	12	13	14	15	16	17	18	19	20
C		=					=		
12	14	16	18	20	22	24	26	28	30

La versión serial suma A y B de la siguiente manera:

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

a. Realizar el programa en su versión serial. Su inicio debería ser:

```
#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;

. . .
```

b. En 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.
Inicio y *fin* pueden calcularse de la siguiente manera, sabiendo que *tid* es el identificador de cada hilo:

```
Inicio = tid * 5;
Fin = (tid + 1) * 5 -1;
```

La *función suma* queda así:

```
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

c. Implementar el código anterior para un arreglo de cantidad variable de elementos, aunque siempre A y B tienen la misma cantidad. Este valor debe ser un parámetro de entrada del programa. Buscar funciones de OpenMP que permitan medir los tiempos de ejecución de toda la ejecución. Comparar con los tiempos de la versión serial para valores muy grandes de elementos.
d. Subir al Campus el código de la parte c y un gráfico que compare los tiempos seriales y paralelo para valores de n lo suficientemente grandes.

Actividad 8

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 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;
}
```