Guía de Actividades Prácticas Nro 2: Estudio de la Programación Paralela con Memoria Compartida y OpenMP

Introducción

En esta guía se irán viendo otros constructores y cláusulas de OpenMP como también diversos conceptos y ejemplos que ayudarán a comprender el manejo de la memoria de los hilos durante una ejecución.

Constructor Critical

En una aplicación concurrente, una **región crítica** es una porción de código que contienen la actualización de una o más variables compartidas, por lo que puede ocurrir una condición de carrera. (**race conditions**)

La **exclusión mutua** consiste en que un solo proceso/hilo excluye temporalmente a todos los demás para usar un recurso compartido de forma que garantice la integridad del sistema.

En OpenMP existe una directiva que permite que un segmento de código que contiene una secuencia de instrucciones no sea interrumpido por otros hilos (realiza una exclusión mutua). Es decir, en el segmento de código delimitado por la directiva sólo puede entrar un hilo a la vez y así evitar una condición de carrera, el nombre de esta directiva es ***critical*** y la sintaxis de uso es:

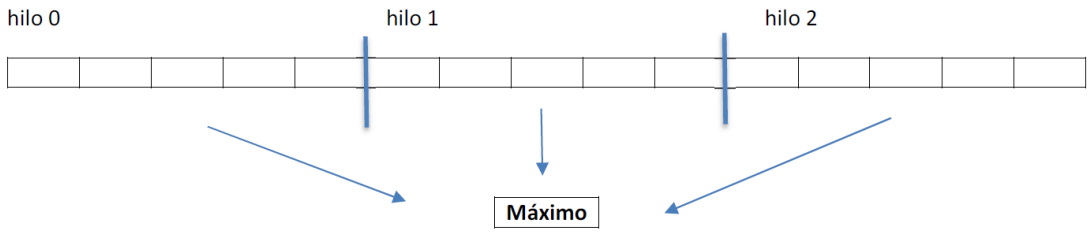
```
#pragma omp critical
{
}
```

Ejercicio Nro. 1

Dada una función que encuentra el máximo valor entero entre los elementos de un arreglo unidimensional de n elementos enteros, realizar una versión paralela.

```
Int buscaMaximo( int (*a, int desde, int hasta) {
    int max, I;
    max = a[ 0 ];
    for ( I = desde; I < hasta; I++) {
        if (a[ I ]) > max)
            max=a[ I ];
        }
    return max;
}
```

Analizar el problema. Ver que se puede realizar una descomposición de dominio o datos. En otras palabras, si se tienen varios hilos, cada uno puede buscar el máximo en un sub-arreglo del arreglo original, que le es asignado, y utilizar el mismo algoritmo de búsqueda sobre cada sub-arreglo.



Utilizando la función del ejemplo que realiza la búsqueda, para dividir el arreglo entre los hilos cada uno debe empezar y terminar su índice del arreglo en diferente valor. Para conseguir esto con OpenMP, como vimos en los ejemplos de la Guía de Actividades 1, se utiliza el constructor for que divide el computo entre los hilos.

pragma omp parallel

{

#pragma omp for

Se puede reunir en

pragma omp parallel for

Importante: Cuando cada hilo trabaje con una parte del arreglo puede ser que al revisar si a[i]>max y dos de ellos actualizan max pueda ocurrir que se dé una race condition. *Esto es posible, aunque las ejecuciones parezcan que están bien.*

Cláusula reduction

Para conocer cómo opera esta cláusula pensemos un ejemplo:

Realizar el producto escalar entre dos vectores de **n** elementos cada uno, el cual se grafica a continuación:

$$\mathbf{A} \cdot \mathbf{B} = \begin{bmatrix} A_1 & A_2 & \dots & A_n \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_n \end{bmatrix} = A_1 B_1 + A_2 B_2 + \dots + A_n B_n$$

Una solución serial sería:

```
double prodpunto(double *a, double *b, int n){
 double res=0;
 int i;

  for(i=0;i<n;i++){
    res+=a[i]*b[i];
   }
return res;
}
```

Para tener una solución paralela con hilos que cooperen en el cómputo se analizará dividir el cálculo entre los n hilos.

Por ejemplo, supongamos que tenemos un arreglo con 15 elementos y 3 cores, y queremos dividir el cálculo en 3 hilos. Sería algo así:

$$A \cdot B = A_0 \cdot B_0 + A_1 \cdot B_1 + A_2 \cdot B_3$$

Y cada hilo hace el siguiente trabajo:

$$A_1 \cdot B_1 = a_0 \cdot b_{0,} + a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3 + a_4 \cdot b_4$$

$$A_2 \cdot B_2 = a_5 \cdot b_5 + a_6 \cdot b_6 + a_7 \cdot b_7 + a_8 \cdot b_8 + a_9 \cdot b_9$$

$$A_3 \cdot B_3 = a_{10} \cdot b_{10} + a_{11} \cdot b_{11} + a_{12} \cdot b_{12} + a_{13} \cdot b_{13} + a_{14} \cdot b_{14}$$

Otra vez hay que repartir el arreglo en sub-arreglos y encontrar los índices en función del Id del hilo.

La siguiente es una primera manera de paralelizarlo usando la cláusula **for**:

```
double prodpunto(double *a,double *b, int n){
 double res=0;
 int i;
  #pragma omp parallel for
  for(i=0;i<n;i++){
    res+=a[i]*b[i];
   }
return res;
}
```

- Implementar este código.
- ¿Los resultados obtenidos son correctos?
- ¿Cuál es la función de la variable RES?
- Explique el comportamiento del algoritmo

Ejercicio 2:

---

a) *Escribir una solución posible para el problema anterior usando un arreglo de resultados RES.*
b) *Reescribir la solución anterior usando la cláusula **reduction***

La ***cláusula reduction*** toma el valor de una variable aportada por cada hilo y aplica la operación indicada sobre esos datos para obtener un resultado correcto libre ***de race conditions.***

Esta cláusula suma todos los **res** parciales de cada hilo y deja la suma acumulada de todos en la misma variable **res**.

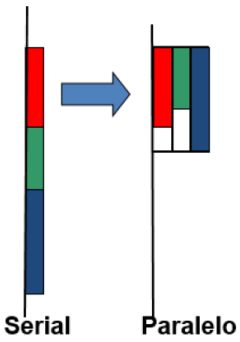
Algunos operadores válidos para utilizar en la cláusula ***reduction***:

Operador	Valor inicial
+	0
*	1
-	0
^	0
&	0
|	0
&&	1
||	0
min y max	

Las operaciones de reducción son útiles y necesarias en la programación paralela en general.

Constructor Sections

Sections es un constructor que permite la programación paralela funcional (descomposición funcional) debido a que permite construir secciones de código independiente a hilos diferentes que trabajarán en modo concurrente o paralelo. Como se ve en el siguiente esquema:



Por ejemplo, para el siguiente código:

V = alfa();

W = beta();

X = gama(v,w);

Y = delta();

printf (“%6.2f\n”, épsilon(x,y));

Se supone que pueden ejecutarse en paralelo V, W y Y y se pueden separar así:

```
#pragma omp parallel sections
 {
  #pragma omp section
     v = alfa();
  #pragma omp section
     w = beta();
  #pragma omp section
    y = delta();
 }
 x = gama(v, w);
 printf ("%6.2f\n", epsilon(x,y));
```

Otra posibilidad es que se ejecuten así:

```
#pragma omp parallel
 {
   #pragma omp sections
    {
      #pragma omp section
       v = alfa();
      #pragma omp section
       w = beta();
    }
  #pragma omp sections
    {
      #pragma omp section
        x = gama(v, w);
     #pragma omp section
       y = delta();
    }
 }
 printf ("%6.2f\n", epsilon(x,y));
```

Constructor Barrier

Barrier permite obtener la secuencia apropiada cuando hay dependencias presentes. Por ejemplo, cuando el hilo 0 produce información en alguna variable y otro hilo 1 quiere imprimir esa variable, entonces el hilo 1 debe esperar que el hilo 0 termine. ***Barrier*** ofrece la sincronización correcta.

a. **Por ejemplo, analizar el siguiente código**

```
#pragma omp parallel shared (A, B, C)
{
realizaUnTrabajo(A,B);
        printf("Procesado A y  B\n");
#pragma omp barrier  // esperan
        realizaUnTrabajo (B,C);
        printf("Procesando B y C\n");
}
```

b. Veamos otro ejemplo con Barrier.

Se genera una región paralela y dentro de esta con el ***constructor for*** cada hilo asigna valores a diferentes elementos del arreglo a, después con el ***constructor master*** se indica que solo el hilo maestro imprima el contenido del arreglo. Como el ***constructor master*** no tiene barrera implícita se usa el ***constructor barrier*** para lograr que todos los hilos esperen a que se imprima el arreglo a antes de modificarlo.

```
#include <stdio.h>
int main( )
{
    int a[5], i;
  #pragma omp parallel
   {
       #pragma omp for
         for (i = 0; i < 5; i++)
                 a[i] = i * i;

       #pragma omp master
          for (i = 0; i < 5; i++)
             printf("a[%d] = %d\n", i, a[i]);

       #pragma omp barrier

       #pragma omp for
        for (i = 0; i < 5; i++)
                 a[i] += i;
   }
}
```

Ejercicio 3

Probar el ejemplo b. del constructor ***Barrier*** visto anteriormente y responder las siguientes preguntas

a) ¿Qué sucede si se quita la barrera?
b) Si en lugar del constructor ***Master*** se utiliza ***Single***, ¿qué otros cambios se tienen que hacer en el código?

c) Realice los cambios.

Ejercicio 4 – Cálculo de PI

Una forma de obtener la aproximación del número irracional PI es utilizar la regla del trapecio para dar solución aproximada a la integral definida

$$\pi = \int_0^1 \frac{4}{(1-x^2)}\ dx.$$

Se da a continuación el código serial para que se comprenda el método numérico que se utiliza y se pide escribir una versión para OpenMP.

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

long long num_steps = 100000000;
double step;
double empezar,terminar;

int main(int argc, char* argv[])
{
      double x, pi, sum=0.0;
      int i;
      step = 1.0/(double)num_steps;
      empezar=omp_get_wtime( );

      for (i=0; i<num_steps; i++)
      {
            x = (i + .5)*step;
            sum = sum + 4.0/(1.+ x*x);
      }

      pi = sum*step;
      terminar=omp_get_wtime();

      printf("El  valor de PI es %15.12f\n",pi);
      printf("El tiempo de calculo del numero pi es: %lf segundos ",terminar-empezar);
      return 0;
}
```

Constructor Single

Veamos este constructor con un ejemplo. Solo un hilo imprime el mensaje del progreso del trabajo. Todos los hilos saltan la región ***single*** y paran en el ***barrier*** al final del constructor ***single*** hasta que todos los hilos hayan alcanzado la barrera. Si otros hilos pudieran continuar

sin esperar por la ejecución de la región ***single***, entonces una cláusula ***nowait*** puede especificarse como se muestra en el tercer constructor ***single*** en este ejemplo. No se puede hacer ninguna suposición previa sobre qué hilo ejecutará la región ***single***

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

void work1() {}
void work2() {}

void single_example(){
    #pragma omp parallel
    {
        #pragma omp single
            printf("Comienza work1. \n");

        work1();

        #pragma omp single
            printf("Finaliza work1. \n");

        #pragma omp single nowait
            printf("Finaliza work1 y comienza work2. \n");

        work2();
    }
}
```