Guía práctica de estudio 12

Algoritmos paralelos parte 1.

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaño

Facultad de Ingeniería

U.N.A.M.

Guía Práctica 12

Estructura de datos y Algoritmos II

Algoritmos paralelos.

Objetivo: El estudiante conocerá y aprenderá a utilizar algunas de las directivas de OpenMP para implementar algún algoritmo paralelo.

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

Antecedentes

- Guía de estudio práctica 10-
- Conocimientos sólidos de programación en Lenguaje C.

Introducción.

En esta guía se irán revisando otros constructores y cláusulas de OpenMP así como diversos conceptos y ejemplos que ayudarán a la comprensión de las mismas.

Constructor *critical*

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

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, que al segmento de código delimitado por la directiva solo pueda 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

{

}

La siguiente función permite encontrar el máximo valor entero de entre los elementos de un arreglo unidimensional de ***n*** elementos enteros. Y se requiere realizar su versión paralela.

```
int buscaMaximo(int *a, int n){
 int max,i;
 max=a[0];
  for(i=1;i<n;i++) {
     if(a[i]>max)
     max=a[i];
   }
   return max;
}
```

Analizando, 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 asignado del arreglo original y utilizar el mismo algoritmo de búsqueda sobre cada sub-arreglo. Figura 12.1.

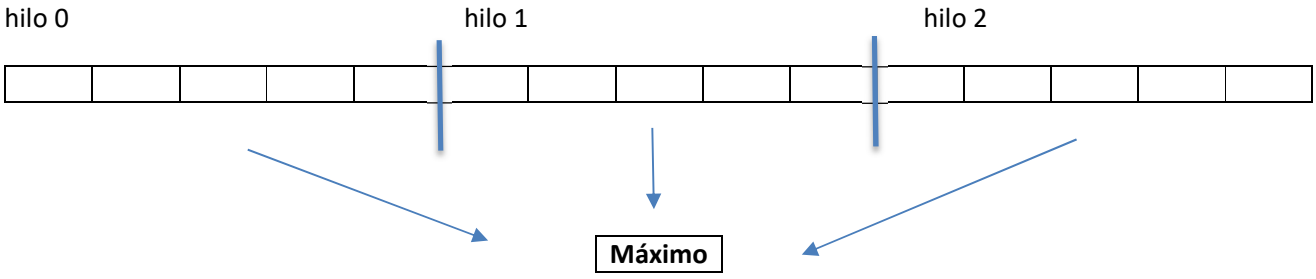

Figura 12.1

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 se vio en la práctica anterior) se puede utilizar el constructor **for** ya que este divide las iteraciones del ciclo entre los hilos. La función queda:

```
 int buscaMaximo(int *a, int n){
  int max,i;
  max=a[0];
#pragma omp parallel
{
   #pragma omp for
   for(i=1;i<n;i++){
     if(a[i]>max)
     max=a[i];
   }
}
 return max;
}
```

NOTA: Cuando lo que está dentro de la región paralela solo es una estructura for y esta es posible paralelizarla, se pueden anidar los constructores ***parallel*** y ***for*****,** entonces el código de la función anterior se reescribe como:

```
int buscaMaximo(int *a, int n){
 int max,i;
 max=a[0];
#pragma omp parallel for
   for(i=1;i<n;i++){
     if(a[i]>max)
     max=a[i];
   }
  return max;
}
```

Agregando el constructor ***for*** cada hilo trabaja con diferentes partes del arreglo, pero, cada uno revisa si **a[i]>max** y si por lo menos dos de ellos encuentran la proposición verdadera, actualizan la variable **max** donde se almacena el valor máximo encontrado, y entonces ocurrirá una condición de carrera. Una forma de arreglar este problema es que un hilo a la vez modifique la variable **max** y los demás esperen su turno, esto se consigue con el constructor ***critical*****.**

```
int buscaMaximo(int *a, int n){

 int max,i;
 max=a[0];
   #pragma omp parallel for
    for(i=1;i<n;i++){
        if(a[i]>max){
           #pragma omp critical
           {
            max=a[i];
           }
        }
    }
}
```

Hasta aquí todavía existe un problema ya que, aunque cada hilo espera su turno en actualizar **max**, el valor de **max** que utiliza uno de los hilos en espera puede que sea menor al valor de a[i] que analiza en ese momento debido a que fue actualizado por uno de los hilos que anteriormente entró a modificarlo. Entonces para que no exista ese problema, cuando cada hilo entra a actualizar a la variable **max** debe revisar nuevamente si a[i]>max.

Y finalmente el código queda:

```
int buscaMaximo(int *a, int n){
 int max,i;
 max=a[0];
  #pragma omp parallel for
   for(i=1;i<n;i++){
      if(a[i]>max) {
        #pragma omp critical
        {
           if(a[i]>max)
           max=a[i];
        }
      }
   }
}
```

. **Clausula *reduction***

Para explicar esta cláusula, se partirá del ejemplo de realizar el producto punto entre dos vectores de ***n*** elementos. El cual se realiza como se muestra en la figura12.2.

$$\mathbf{A} \cdot \mathbf{B} = \begin{bmatrix} A_1 & A_2 & \dots & A_n \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_n \end{bmatrix} = A_1B_1 + A_2B_2 + \dots + A_nB_n$$

Figura 12.2

Una función para la solución es la siguiente:

```
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;
}
```

Si se quiere una solución concurrente/paralela utilizando OpenMP, es decir que n hilos cooperen en la solución, se analiza primero como dividir las tareas entre los *n* hilos. La solución es similar al ejemplo anterior, que cada hilo trabaje con diferentes elementos de los vectores A y B y cada uno obtenga resultados parciales. Por ejemplo, si los vectores son de 15 elementos y se tienen 3 hilos, el producto punto se puede dividir como:

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

Y cada hilo hace los siguientes cálculos:

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

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

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

Se observa que cada hilo realiza los mismos cálculos, pero sobre diferentes elementos del vector. Así que para asignar diferentes elementos del vector a cada hilo se puede utilizar el constructor ***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;
}
```

Pero todavía no se tiene la solución, porque, todos acumulan los resultados de sus sumas en una variable **res** que es compartida por lo que se sobre escriben los resultados parciales. Para una posible solución lo que se hará es utilizar un arreglo resp**[ ]** del tamaño del número de hilos que se quiere tener en la región paralela y en cada elemento guardar las soluciones parciales de cada hilo. Después un solo hilo suma esos resultados parciales y obtener la solución final. La solución queda:

```
double prodpunto(double *a,double *b, int n){

 double res=0,resp[n_hilos];
 int i, tid, nth;

  #pragma omp parallel  private (tid) nthreads(n_hilos)
   {
     tid = omp_get_thread_num();
      resp[tid]=0;
     #pragma omp for
      for(i=0;i<n;i++){
          resp[tid]+=a[i]*b[i];
      }

     if(tid==0){
       nth = omp_get_num_threads();
       for(i=0;i<nth;i++){
         res+= resp[i];
       }
  }
return res;
}
```

Ahora, en lugar de utilizar un arreglo para almacenar resultados parciales se puede utilizar la cláusula ***reduction***

Lo que hace la cláusula ***reduction*** es tomar el valor de una variable aportada por cada hilo y aplicar la operación indicada sobre esos datos para obtener un resultado . Así en el ejemplo del producto punto cada hilo aportaría su cálculo parcial **res** y después se sumarian todos los **res** y el resultado quedaría sobre la misma variable.

Sintaxis

```
#pragma omp parallel reduction (variable:operador)
{

}
```

La solución del producto punto con la cláusula ***reduction*** queda:

```
double prodpunto(double *a,double *b, int n){

  double res=0;
  int i;

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

Algunos de los operadores utilizados en la cláusula ***reduction*** son los siguientes:

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

Es importante mencionar que a las actividades realizadas por la cláusula ***reduction*** se le conoce como operaciones de reducción y son muy utilizadas en la programación paralela.

Constructor *sections*

Este constructor permite usar paralelismo funcional (*descomposición funcional*) debido a que permite asignar secciones de código independiente a *hilos* diferentes para que trabajen de forma concurrente/paralela. Figura 12.3.

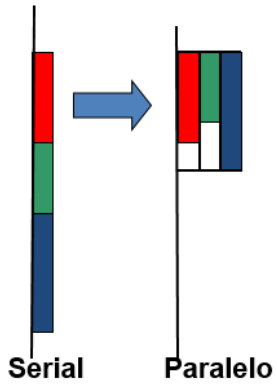

Figura 12.3

Cada sección paralela es ejecutada por un sólo *hilo*, y cada *hilo* ejecuta ninguna o alguna sección. Este constructor tiene una barrera implícita que sincroniza el final de las secciones.

Por ejemplo, para el segmento de código:

```
v = alfa ();
w = beta ();
x = gama (v, w);
y = delta ();
printf ("%6.2f\n", épsilon(x,y));
```

Se observa que se pueden ejecutar en paralelo alfa(), beta() y delta() y por tanto separarlas como sigue:

```
#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 primero se ejecuten alfa () y beta () y una vez que terminen gama () y delta() :

```
#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));
```

Otros constructores y cláusulas

Constructor *barrier*

¿Cómo obtener la secuencia apropiada cuando hay dependencias presentes? Por ejemplo, cuando un hilo 0 produzca información en alguna variable y otro hilo 1 quiere imprimir esa variable, el hilo 1 debe esperar a que el hilo 0 termine. Para hacerlo se necesita alguna forma de sincronización por ejemplo las barreras.

En OpenMp se tiene el constructor ***barrier*,** que coloca una barrera explícita para que cada hilo espere hasta que todos lleguen a la barrera. Es una forma de sincronización.

Ejemplo:

```
#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");
}
```

Constructor single

Este constructor permite definir un bloque básico de código dentro de una región paralela, que debe ser ejecutado por un único hilo. Aquí todos los hilos esperan

Por ejemplo, una operación de entrada/salida solo debe realizarse por un solo hilo mientras todos los demás esperan. En el constructor no se especifica qué hilo ejecutará la tarea, puede ser cualquiera de los que están en la región paralela.

Ejemplo:

```
#pragma omp parallel
{
        todosRalizanUnasActividades();
        #pragma omp single
        {
                ActividadE/S();
        }       // Hilos esperan
        RealizanMasActividades();
}
```

Constructor master

Este constructor es similar a single con la diferencia de que las actividades realizadas son hechas por el hilo maestro y no se tiene una barrera implícita, es decir, los hilos restantes no esperan a que el hilo maestro termine la actividad asignada.

Sintaxis

```
#pragma omp master
{

}
```

Constructor *barrier*

Permite colocar una barrera de forma explícita. Se coloca cuando se requiere que todos los hilos esperen en un punto del programa.

Sintaxis

```
#pragma omp barrier
```

Ejemplo: En el siguiente código primero 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 de 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;
  }
}
```

Cláusula *nowait*

Esta cláusula permite quitar las barreras implícitas que tienen los constructores, como lo son ***for, single, sections,*** etc.. Quitar las barreras que no son necesarias ayudan a mejorar el desempeño del programa.

Desarrollo

Después de haber leído y analizado el contenido de la guía, realizar con ayuda del profesor las siguientes actividades:

Actividad 1

Completar la versión serie y paralela del ejemplo explicado de búsqueda del valor mayor de los elementos de un arreglo unidimensional de enteros.

Actividad 2

Completar la versión serie y sus dos versiones paralelas del ejemplo explicado del producto punto de dos vectores de n elementos enteros.

Actividad 3

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

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

A continuación, se proporciona el código en su versión serial y se requiere se obtenga su versión paralela.

```
#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;
}
```

Actividad 4

Para el siguiente programa obtener dos versiones paralelas, una utilizando el constructor ***section*** y otra con el constructor **for**. ¿Cuál de las dos versiones tarda más tiempo?.

```
#include<stdio.h>
#include<omp.h>
#define N 100000

int main (int argc, char *argv[]) {

        double empezar,terminar;
        int  i,j;
        float a[N], b[N], c[N],d[N], e[N],f[N];

        for (i=0; i < N; i++)
                a[i] = b[i] = i * 1.0;

        empezar=omp_get_wtime( );

         for(i=0;i<N;i++)
            c[i]=a[i]+b[i];

        for(j=0;j<N;j++)
             d[j]=e[j]+f[j];

        terminar=omp_get_wtime();

        printf("TIEMPO=%lf\n",empezar-terminar);
}
```

Actividad 5

Probar el siguiente ejemplo visto en la guía y responder a las siguientes preguntas:

¿Qué sucede si se quita la barrera? _____

Si en lugar de utilizar el constructor ***master*** se utilizara ***single***, ¿Qué otros cambios se tienen que hacer en el código? _____

Realizar los cambios.

```
#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;
   }
}
```

Actividad 6

Ejercicios sugeridos por el profesor

Referencias

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

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

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

[4] R. Chandra et al,Parallel Programming in OpenMP,Morgan Kaufmann, 2001.