Guía práctica de estudio 13

Algoritmos paralelos parte 2.

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaño

Facultad de Ingeniería

U.N.A.M.

Guía Práctica 13

Estructura de datos y Algoritmos II

Algoritmos paralelos.

Objetivo: El estudiante utilizará algunas de las directivas de OpenMP para paralelizar algunos problemas y con ello adquiera experiencia en el desarrollo de programas multihilo en sistemas multiprocesador de memoria compartida.

Al final de la práctica el estudiante reforzará el uso de directivas de OpenMP en el lenguaje C y podrá realizar programas en paralelo.

Antecedentes

- Guía de estudio práctica 11 y 12.
- Conocimientos sólidos de programación en Lenguaje C.

Introducción.

En el proceso de análisis de problemas que se deseen paralelizar, surgen diversas opciones o formas de resolverlos. Ello dependerá, en gran medida, de las diferentes técnicas y herramientas que nos pueda proporcionar el software que empleemos en esta tarea. En esta guía revisaremos algunos problemas ejemplo que emplean estas técnicas con las directivas de OpenMP.

Ejemplos de problemas y algoritmos paralelos

Ejemplo 1-Multiplicación de matrices

Los algoritmos basados en arreglos unidimensionales y bidimensionales a menudo son paralelizables de forma simple debido a que es posible tener acceso simultáneamente a todas las partes de la estructura, no se tienen que seguir ligas como en las listas ligadas o árboles.

Un ejemplo común es el producto de dos matrices A y B de orden nxm y mxr . Recordando la fórmula y la forma de obtener un elemento de C. Figura 13.1.

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \qquad \text{para } 0 \leq i, j < n.$$

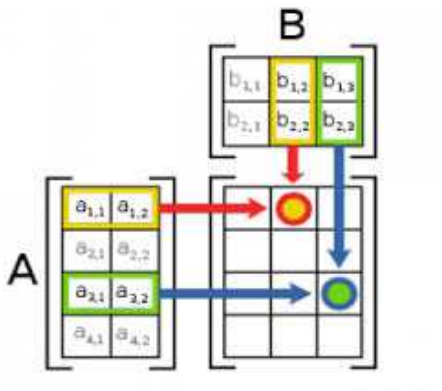

Figura 13.1

Para lograr dividir este problema se realiza un paralelismo de datos, donde cada hilo trabaje con datos distintos, pero con el mismo algoritmo.

Un planteamiento de solución paralela es considerar que, en un ambiente con varios hilos, cada uno puede trabajar con diferentes datos, de acuerdo a las siguientes opciones:

- Calcular un elemento de la matriz resultante, se requiere un renglón de A y una columna de B,
- Calcular todo un renglón de C, se requiere un renglón de A y toda la matriz B
- Calcular i renglones de C, se requieren i renglones de A y toda B

Dependiendo de la granularidad escogida se necesitan un número de hilos en la región paralela y como lo ideal es tener un número de hilos igual al número de unidades de procesamiento, que no puede ser muy grande se toma la tercera opción.

Partiendo del algoritmo secuencial, el segmento de código para el cálculo de C es:

```
for (i=0; i<NRA; i++)
    {
        for(j=0; j<NCB; j++)
                for (k=0; k<NCA; k++)
                        c[i][j] += a[i][k] * b[k][j];
    }
```

Donde NRA=número de renglones de A, NCB= número de columnas de B y NCA= número de columnas de A.

Como cada hilo tomará n diferentes renglones de la matriz A y calculará n diferentes renglones de la matriz C , se pueden dividir las iteraciones del primer ciclo for que hace referencia al renglón i de A utilizado en el cálculo actual de c[i][j], con esto cada hilo usará distintos valores del índice i para la lectura de A y para el cálculo de C.

```
#pragma omp parallel for
for (i=0; i<NRA; i++)
    {
        for(j=0; j<NCB; j++)
                for (k=0; k<NCA; k++)
                        c[i][j] += a[i][k] * b[k][j];
    }
```

Hasta aquí hace falta considerar si todas las variables involucradas deben ser compartidas o algunas deben ser privadas. A, B y C deben ser compartidas ya que en ningún momento los hilos tratan de escribir en un elemento compartido, pero j y k deben ser privadas porque cada hilo las modifica al realizar los ciclos internos.

El segmento paralelizado queda.

```
#pragma omp parallel for private (j,k)
for (i=0; i<NRA; i++)
    {
        for(j=0; j<NCB; j++)
               for (k=0; k<NCA; k++)
                  c[i][j] += a[i][k] * b[k][j];
```

Ejemplo 2- Cálculo del histograma de una imagen en tono de grises.

El histograma es el número de pixeles de cada nivel de gris encontrado en la imagen. En el programa la imagen es representada por una matriz de nxn cuyos elementos pueden ser mayores o iguales a 0 y menores a NG (tonos de gris) y se cuenta el número de veces que aparece un número(pixel) en la matriz y éste es almacenado en un arreglo unidimensional .

Por ejemplo, para la matriz (imagen) de 5x5 de la figura 13.2 que almacena valores de entre 0 y 5, en un arreglo unidimensional llamado histograma se almacena en el elemento con índice 0 el número de ceros encontrados, en el elemento con índice 1 el número de unos, en el elemento con índice 2 el número de dos etc..

Imagen

1	3	5	0	1
0	3	4	2	3
1	0	5	3	3
2	0	4	0	1
2	0	5	3	3

histograma

0	1	2	3	4
6	4	3	7	2

Figura 13.2

El segmento de código correspondiente al cálculo del vector histograma es:

```
for(i=0; i<NG; i++)
    histo[i] = 0;

for (i=0; i<N; i++)
       for (j=0; j<N; j++)
            histograma[IMAGEN[i][j]]++;
```

Para paralelizarlo, se realiza una descomposición de datos, es decir, se dividirá la matriz de forma que cada hilo trabaje con un número de renglones diferentes y almacene la frecuencia de aparición de cada valor entre 0 y NG de la sub-matriz en un arreglo privado o propio de cada hilo llamado histop[]. Después se deben sumar todos los arreglos histop[] de cada hilo para obtener el resultado final en el arreglo histo[ ] . Figura 13.3.

1	3	5	0	1
0	3	4	2	3
1	0	5	3	3
2	0	4	0	1
2	0	5	3	3

Hilo 0

Hilo 1

histop de hilo 0

0	1	2	3	4
2	2	1	3	1

histop de hilo 1

0	1	2	3	4
4	2	2	4	1

histo[ ]= histop de hilo 0 + histop de hilo1

0	1	2	3	4
6	4	3	7	2

Figura 13.3

El segmento de código que realiza lo planteado es:

```
for(i=0; i<NG; i++)
       histo[i] = 0;

/*Calculo del histograma de IMAGEN*/
#pragma omp parallel  private(histop) num_threads(2)
{
      for(i=0; i< NG; i++)
         histop[i]=0;

      #pragma omp for  private(j)
        for (i=0; i<N; i++)
                  for (j=0; j<N; j++)
                       histop[IMA[i][j]] ++;
```

```
      #pragma omp critical
        {
        for( i=0 ; i < NG ; i++)
         histo[i]+=histop[i];

      }
}
```

Ejemplo 3. Cálculo de los números de la sucesión de Fibonacci

Para este ejemplo se requiere conocer de otros dos constructores que son útiles para paralelizar otro tipo de problemas como los que requieren manejo de recursividad o los que contienen ciclos con un número indeterminado de iteraciones. Estos son **task** y **taskwait.**

El constructor **task** sirve para generar tareas de forma explícita y esas tareas generadas pueden ser asignadas a otros hilos que se encuentran en la región paralela. Figura 13.4.

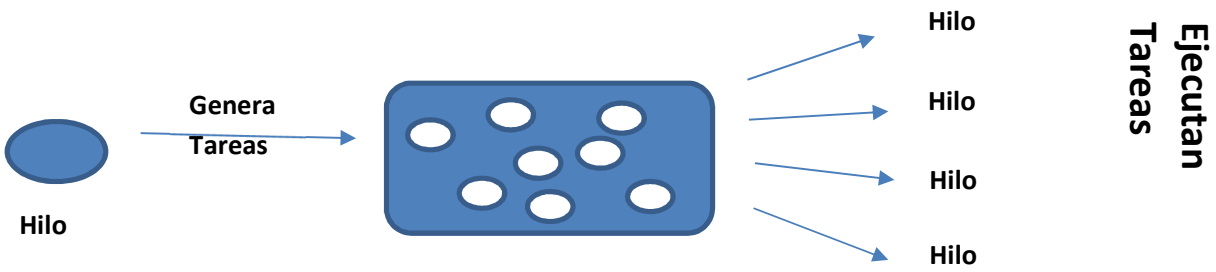

Figura 13.4

Cuando un hilo encuentra el constructor **task**, se genera una nueva tarea, así que, si un hilo encuentra n veces al constructor **task**, genera n tareas.

El constructor **taskwait** permite esperar a que todas las tareas hijas generadas desde la tarea actual terminen su actividad.

Usar la directiva task en un programa recursivo para calcular el número f(n) de la sucesión Fibonacci.

$f(n) = f(n-1) + f(n-2)$ con $f(0) = 1\, y\, f(1) = 1$

Una solución secuencial es:

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

```
long fibonacci(int n);

 main (){
 int nthr=0;
 int n;
 long resul;

  printf("\n Número a calcular? ");
  scanf("%d", &n);

  resul = fibonacci(n);
  printf ("\nEl numero Fibonacci de %5d es %d", n, resul);
}


long fibonacci(int n) {

long fn1, fn2, fn;
int id;

if ( n == 0 || n == 1 )
 return(n);

 if ( n < 30 )
  {
        fn1 = fibonacci(n-1);
        fn2 = fibonacci(n-2);
        fn = fn1 + fn2;
   }
         return(fn);
}
```

Ahora para la versión paralela se puede que un hilo empiece con la llamada a la función fibonacci() y después genere dos tareas una para el cálculo de $f1$ y otra para $f2$ que puedan ser realizadas por otros hilos de la región paralela y que estos a su vez al llamar a la función fibonacci() generaran otras nuevas tareas hasta que se llegue al caso base. Finalmente, el hilo que generó las primeras tareas realizará el cálculo final de fn, solo que con el constructor **taskwait** esperará a que terminen todas las tareas generadas anteriormente.

El código en su versión paralela queda:

```
#include <stdio.h>

long fibonacci(int n);

main () {
 int nthr=0;
 int n;
 long resul;
```

```
printf("\n Numero a calcular? ");
 scanf("%d", &n);

 #pragma omp parallel
 {
         #pragma omp single
        {
                resul = fibonacci(n);
         }
 }
 printf ("\nEl numero Fibonacci de %5d es %d", n, resul);
 }


long fibonacci(int n)
{
long fn1, fn2, fn;

if ( n == 0 || n == 1 )
 return(n);

 if ( n < 30 ){
        #pragma omp task shared(fn1)
        {

                fn1 = fibonacci(n-1);
        }
         #pragma omp task shared(fn2)
         {
                fn2 = fibonacci(n-2);
          }
         #pragma omp taskwait
        {
                fn = fn1 + fn2;
        }
         return(fn);
    }
}
```

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 1 explicado en la guía y medir el tiempo de ejecución de ambas versiones utilizando matrices de orden 500x500.

Actividad 2

Completar la versión serie y paralela del ejemplo 2 explicado en la guía y medir el tiempo de ejecución de ambas versiones utilizando N = 1000 y NG=256. Tomar tres lecturas y sacar el tiempo promedio para cada caso.

¿Cuánto tiempo tardaron ambas versiones?

¿Por qué en la versión paralela el cálculo de histo[] está delimitado con el constructor **critical**?

Actividad 3

Probar las versiones serie y paralela del ejemplo 3 para verificar que se obtienen los mismos resultados. Además, agregar a la versión paralela la impresión del identificador del hilo en la generación de cada tarea para visualizar los hilos que participan en los cálculos.

¿Qué pasa si $f1$ y $f2$ no se colocan como compartidas? Probar

¿Por qué sucede lo observado?

Actividad 4

Ejercicios sugeridos por el profesor

Nota: Existen otros constructores y cláusulas y se espera que con lo visto en la guía 11,12 y 13 el estudiante pueda comprenderlas con mayor facilidad.

Referencias

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

[2] B. Chapman ,Using OpenMP, The MIT Press, 2002