Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación
Programación de Microprocesadores
Ing. Kimberly Barrera
Sección 10

Proyecto 03:
OpenMP

Martín Amado Girón 19020
Evelyn Andrea Amaya Malin 19357
Oliver Josué de León Milian 19270
Brandon Josué Hernández Marroquín 19376
Laura María Leví Tamath 19365

Guatemala, Ciudad de Guatemala 26 de octubre de 2020

Índice

OpenMp

Programación Paralela en ramas de la Física y Matemáticas

La programación paralela en algunas de las ramas de la matemáticas es utilizada para poder realizar operaciones repetitivas que al realizarlas muchas veces se vuelven exhaustivas. Además de ser un método confiable en el que el computador posee una menor probabilidad de tener un error matemático incluso luego de un millón de repeticiones, también son más exactos y precisos. Algunos ejemplos de operaciones que se trabajan en programación paralela son los siguientes:

- Multiplicación de matrices.
- Encontrar n números primos.
- Hacer la sumatoria de n números en una serie.

Y el ejemplo al que más ayuda es el algoritmo de encontrar la ruta más corta, en donde cada parte del programa calcula una posible ruta y el resto otras posibles. Al terminarlo ellos presentan la que menos tiempo en recorrer, un dato curioso es que para este algoritmo es necesario utilizar operaciones entre matrices.

(Aguilar, 2004)

Ahora hablemos de la rama de la física, la cual utiliza a gran medida en el uso de simulaciones que requieren una gran cantidad de cálculos. Por ejemplo, se tiene una simulación del pronóstico del tiempo, de cómo se comportan los huracanes o las corrientes de aire. En donde con tres celdas, en donde son necesarias 200 operaciones de punto flotante por cada una de las celdas en cada unidad de tiempo, haciendo alrededor de 10^11 operaciones de punto flotante por unidad de tiempo. Otro ejemplo que se puede dar es el predicciones de movimientos de cuerpos astronómicos. Este caso es el más extremo, porque se deben de modelar las fuerzas entre estrellas, planetas o agujeros negros. Es fácil calcular la trayectoria sumando todas las fuerzas ejercidas al cuerpo, pero si se cuenta con n cuerpos celestes (en donde n tiende al infinito) se poseen n - 1 fuerzas y por ende n^2 operaciones por cada unidad de tiempo. Al tener cada galaxia 10^11 estrellas se obtienen que son necesarios 10^9 años para una iteración. Que puede ser reducida considerablemente paralelizando este el cálculo de las fuerzas que ejercen entre ellos.

(Zamudio, 2010)

Aplicación de OpenMP para el cálculo de valores numéricos de precisión

OpenMP es una aplicación estandarizada basada en una interfaz programada utilizada para escribir códigos paralelos con múltiples hilos de ejecución. Emplea la programación paralela en todas las configuraciones de procesadores simétricos y ofrece un modelo de programación paralelo flexible para aplicaciones las cuales son ejecutadas bajo el modelo computacional de un único proceso con múltiple información; permitiendo que un problema traducido a un código computacional se divida en otros problemas más pequeños, con el fin de utilizar el poder de un súper computador para ejecutar tareas de manera simultánea.

El modelo que ofrece OpenMp para realizar aplicaciones se basa en tres aspectos:

- Especificaciones o directivas para ejecuciones concurrentes, es decir, la ejecución de hilos a través de diferentes procesos ocurre por medio de librerías, funciones y variables de entorno.
- La comunicación entre hilos de ejecución se especifica dentro de los mecanismos de control para asignar trabajo equivalente en cada proceso, es decir, evita que varios procesos finalicen antes que algún proceso continúe en ejecución.
- Posee una sincronización explícita la cual permite agrupar y asignar trabajo a diferentes hijos de ejecución con el fin de realizar el trabajo paralelo con el mejor uso de los recursos.

La implementación de OpenMP puede alcanzar máxima aceleración al aprovechar las cualidades de los superordenadores, dado que estos se desempeñan con el más alto nivel de funcionamiento. Las ejecuciones de aplicaciones paralelas en supercomputadores son muy eficientes dado que entre más procesadores se utilicen, mayor es la aceleración de cualquier aplicación paralela. Esta no es lineal porque se presentan retardos por comunicación y sincronización entre procesos. Entre más procesadores se utilizan en una aplicación, más comunicación se presenta entre los hilos de ejecución. El número de hilos de ejecución se establece de manera acorde al tamaño del código, ya que cuando el código es pequeño las comunicaciones son redundantes entre procesos; es decir que, es posible que varios procesadores asignen la misma orden a cualquier proceso.
(Acosta, 2009)

Programa a: Número de Euler

1. $\sum_{n=0}^{\infty} \frac{1}{n!}$
2. $\lim_{n\to\infty} \left(1+\frac{1}{n}\right)^n$
3. $2+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{4+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{6+\cfrac{1}{1+\cdots}}}}}}}}}$

Para el cálculo de la constante de Euler ‘e’, existen tres métodos, representados arriba. el utilizado fue el método número uno.
Solución matemática: Se realizó una serie de sumatoria: 1/n!. El valor donde finaliza la serie es ingresado por el usuario, y mientras mayor sea el número, mayor exactitud tiene el valor de ‘e’.

Pseudocódigo:

1. Se importan las librerías omp, stdio y stdlib.

Función factorial

2. Recibe el valor ‘n’ al que se le calcula el factorial.
3. Si el valor es menor a 0, devuelve error ‘-1’
4. Si el valor es 0, devuelve ‘1’.
5. Si el valor es otro, devuelve n y realiza factorial de n-1.

Función main

6. Se define la variable entera ‘n’.
7. Se define la variable float ‘e’ con valor ‘0’.
8. Se le pide al usuario que ingrese ‘n’, número de términos para la sumatoria.
9. Ciclo for de 0 a n:
    a. Hilo 1, de 0 a n/4
        i. A la variable ‘e’ se le suma 1/i!
    b. Hilo 2, de n/4 a n/2
        i. A la variable ‘e’ se le suma 1/i!
    c. Hilo 3, de n/2 a 3n/4
        i. A la variable ‘e’ se le suma 1/i!
    d. Hilo 4, de 3n/4 a n
        i. A la variable ‘e’ se le suma 1/i!

10. Se muestra el valor de ‘e’ al usuario.

Programa solución:

```
/*
      Euler.cpp
      --------------------
      Calculo de la constante de Euler
      con implementacion de openmp
*/

#include <omp.h>                              // Inclusion de librerias
#include <stdio.h>
#include <stdlib.h>

float factorial(int n) {
      if (n < 0)
            return -1;
      else if (n == 0)
            return 1;
      else
            return n * factorial(n - 1);
}
int main() {
      int n;                                                  //Valor que se le
da a n para ciclo for
      float e = 0;
      printf("Ingrese el valor 'n'. A mayor valor de 'n' mayor exactitud de la
constante: ");
      scanf("%d", &n);
      #pragma omp parallel for
      for (int i = 0; i < n; i++)
            e += (1 / factorial(i));

      printf("El valor de la constante de Euler es: %f", e);
}
```

Programa b

$$\sum_{n=1}^{\infty} \frac{1}{n(n+1)}$$

Donde n es el número actual

Solución matemática: Para realizar esta serie, se calculará primero la multiplicación *n(n+1)* y luego, el recíproco del mismo. Luego, se sumarán todos los resultados obtenidos para obtener el valor de convergencia.

Pseudocódigo:

1. Se importan las librerías omp y iostream
2. Se define la variable infinito=100

Función double serie(int actual)

1. Recibe el número actual a calcular
2. Se suma 1 al número actual
3. Se multiplica el número actual por el resultado del paso 2

4. Se calcula el recíproco del resultado del paso 3
5. Se retorna el valor del paso 4

Función int main()

1. Se definen las variables int num=0 y sumar=0
2. Se define la variable float total=0
3. Se coloca el número de threads a utilizar en 100
4. Ciclo *for* de 0 a 10 con la variable actual
    a. Se calcula la sumatoria a realizar 100*actual
    b. Se define #pragma omp parallel private(num) para hacer uso de paralelismo
    c. Se define #pragma omp for ordered para hacer uso de un ciclo
        i. Ciclo *for* de 0 a infinito
        ii. Se define #pragma omp ordered para realizar el ciclo en orden
        iii. Se obtiene el número de thread, se le suma 1 y se multiplica por la variable actual
        iv. Se calcula la serie y se suma a total
5. Se muestra el resultado de convergencia de la serie de 1 a 1,000

Programa solución:

```
#include <omp.h>  // OpenMP include file
#include <iostream> //Cout
using namespace std;

#define infinite 100 //Define hasta donde hacer la sumatoria *10
                     //Con 100 -> se realiza la sumatoria hasta 1000

double serie(int actual){ //Calcula el valor del thread actual
      double resultado = actual*(actual+1.0);
      resultado = 1/resultado;
      return resultado;
}

int main()
{
      int num=0, sumar=0;
      double total=0;
      omp_set_num_threads(100); //Se definen 100 threads

      //Secuencial
      for(int actual=0;actual<10;actual++){
            //Se revisa por cual ciclo vamos
            sumar = 100*actual;
            //Paralelismo
            #pragma omp parallel private(num)
            {
                  //Se hace de forma ordenada
                  #pragma omp for ordered
                  for(int i=0;i<infinite;i++) //Hasta llegar a infinite
                  {
                        #pragma omp ordered
                        {
                        //Sumo +1 al thread (comienza en 1 la sumatoria)
                              //Se suma el ciclo actual
                              num = (omp_get_thread_num()+1)+sumar;
```

```
                        //Se suma el resultado al total
                        total+=serie(num);
                    } //ordered
                }//for
            } //pragma
        } //for
        cout << "La sumatoria es: " << total << endl; //Se muestra
        return 0;
}//main
```

Programa C

$$\sum_{n=1}^{\infty}\left(1-\frac{1}{\sqrt{n}}\right)^{n}$$

Donde n es el número actual

Solución matemática**:** En este caso, primero se realiza la división dentro de 1 y la raíz cuadrada de n, este resultado siempre es menor cuando n es mayor a uno. Por ende, cuando da uno esta división todo el factor da cero, pero si n es mayor a uno entonces esta resta es un número menor a uno. Luego se eleva las veces que sean necesarias, cabe recalcar que cuando n tiende al infinito la resta cada vez se acercará más a uno, pero al ser elevado a una potencia sumamente grande este valor tenderá a ser un valor muy pequeño que cada vez aporta menos a la sumatoria. Entonces se puede concluir que la serie convergerá en un valor menor a uno.

Pseudocódigo**:**

1. Se importan las bibliotecas necesarias (omp, math y iostream)
2. Se le indica al programa que estamos utilizando std.
   Función Main
3. Se declara la variable ‘limit’ de tipo entero (hasta donde llega la sumatoria).
4. Se declaran dos variables una ‘resultado’ inicializada en 0 y la otra ‘preResultado’ ambos de tipo float o double.
5. Se le indica al programa que se utilizaran 10 hilos en las partes paralelas.
6. Se le pide al usuario cuantos números quiere sumar y se guarda en la variable ‘limit’.
7. Se le indica al programa que el proximo for debe de ser distribuido dentro de todos los hilos y que cada uno debe de tener su propia copia de ‘preResultado’.
   a) Para cada hilo empezando en 1, mientras sea menor o igual al ‘limit’ y se le suma uno por cada iteración.
   b) Se le indica al programa que en la siguiente línea se realizará una operación atómica del tipo expression.
   c) Se suma el valor de ‘preResult’ con el de ‘result’ y se guarda en ‘result’.
8. Fuera de la parte paralela se imprime el valor de resultado.
9. Se retorna el valor cero.

Código solución**:**

```
#include <iostream>
#include <math.h>
#include <omp.h>
```

```
using namespace std;

int main(){

	// Declaración de variables a utilizar
	int limit;
	double result = 0, preResult;
	omp_set_num_threads(10);		//Se definen 10 threads

	cout << "Ingrese cuantos numeros de la sumatoria quiere operar: ";
	cin >> limit;

	#pragma omp parallel for private(preResult)
	for(int i = 1; i <= limit; i++){
		preResult = pow((1 - (1/(sqrt(i)))), i);

		#pragma omp atomic
		result += preResult;
	}	// Pragma For

	cout << "El resultado es: " << result << endl;

	return 0;
} // Main
```

Programa d

$$\sum_{n=1}^{\infty} e - \left(1 + \frac{1}{n^2}\right)^n$$

Donde n es el número actual

Solución matemática**:** En primera instancia se calcula el valor de $1/n^2$ y posteriormente se suma el número 1, elevando este resultado al valor n actual. Finalmente este valor es restado al valor e. Este proceso es realizado hasta que converja.

Pseudocódigo:

1. Se importan las librerías omp, iostream y cmath
2. Se indica el programa que se está utilizando

Función int main()

3. Se declara la variable límite de tipo int.
4. Se declara la variable final de tipo double.
5. Se declara la variable resultado de tipo double
6. Se define el valor de e de tipo double
7. Se define el valor de potencia de tipo int
8. Se declaran los hilos a utilizar

9. Se pide al usuario que indique cuántos números desea sumar
10. Se guarda en la variable límite
11. Se indica que el for que se utilizará deberá de ser distribuidos en los hilos
12. Se indica que cada hilo debe de tener una copia del resultado
    1. Cada hilo comienza con el valor 1
    2. Si es menor o igual al número de limite, entonces
    3. Se le suma uno por cada iteración
    4. Se declara cua de tipo double, el cual eleva al cuadrado el i
    5. Se declara resultado 1 de tipo dobule , que suma 1 al recíproco del paso 4
    6. Se declara res1 de tipo double, este eleva el resultado del paso 5 a i
    7. Se resta e con el resultado del paso 6 y se guarda en resultado
    8. Se indica que habrá una operación atómica
    9. Se suma el valor del resultado y se guarda en final
13. Se imprime el resultado
14. Se retorna el valor cero

Programa solución:

```
#include <omp.h>      // Biblioteca de OpenMP
#include <iostream> //Biblioteca Cout
#include <cmath> // Biblioteca de funciones matematicas

using namespace std;

int main ()
{
        // Atributos a utilizar
        int limite;
        double final = 0, resultado;
        double e = 2.178282;
        double potencia = 2;
        omp_set_num_threads(10);  //numero de hilos

        cout << "Ingrese hasta donde quiere que la sumatoria opere: ";
        cin >> limite;

        #pragma omp parallel for private(resultado)
        for(int i = 1; i <= limite; i++){
                double cua = pow(i, potencia); //eleva n²
                double resultado1 = (1+1/cua); //se realiza 1 + 1/n²
                double res1 = pow(resultado1, i); // (1 + 1/n²)^n
                resultado = e - res1; // e - (1 + 1/n²)^n

                #pragma omp atomic
                final += resultado;

        }       // Pragma For

        cout << "El resultado es: " << final << endl;
        return 0;
} //main
```

Literatura Citada

- Acosta, C. (2009). Implementación de computación de alto rendimiento y programación paralela en códigos computacionales. Universidad EAFIT: https://core.ac.uk/download/pdf/47244964.pdf
- Aguilar, J. (2004). *introducción a la Computación Paralela*. Universidad de los Andes: http://so-unlam.com.ar/material-clase/HPC/Introducci%C3%B3n%20a%20la%20Computaci%C3%B3n%20Paralela.pdf
- Zamudio, A. (2010). *¿Por qué estudiar el paralelismo?* Extraído de: https://compinformatidf.wordpress.com/2010/12/27/programacion-paralela/