Práctica 5

Arquitectura de Computadoras
Programación Multihilos con OpenMP

Fernanda Mora & Luis Román, ITAM 12/05/2016

Objetivos

El objetivo de esta práctica es:

- Implementar los conceptos teóricos de paralelismo de tareas.
- Inicializarnos en la programación en OpenMP que permite explotar el paralelismo de tareas.

1. Introducción

En esta práctica se probó OpenMP, un programa que permite implementar modelos de paralelismo de tareas. OpenMP es soportado por la mayoría de compiladores y procesadores. Desde el punto de vista del programador, OpenMP es una librería que tiene directivas para el compilador que son mandadas llamar en las líneas de código que se deseen paralelizar.

2. Desarrollo

2.1. Introducción: Hello Worlds

- Compile el programa HelloWorlds.c y ejecútelo. Despliegue el resultado.
- Edite el código fuente y agregue una directiva omp parallel que incluya las cuatro primeras líneas, pero deje fuera la última línea: printf(“GoodBye....

```
1     #pragma omp parallel
2  omp_set_num_threads(threads);
3
4    // Ejecutar for en parallelo.
5    #pragma omp parallel for
6    for(i=0; i<6; i++){
7      //Obtener id del thread que ejecuta.
8      id = omp_get_thread_num();
9      printf("Iter:%d, Thread: %d \n", i, id);
10   }
```

¿Necesita incluir la cabecera omp.h? ¿Por qué? Porque utiliza directivas propias de la biblioteca de openmp, que se encuentran en omp.

- Compile el programa. Si el compilador reporta errores, corríjalos. Modifique la variable de ambiente OMP_NUM_THREADS para definir seis hilos. Compile el programa y ejecútelo varias veces. Observe si el resultado es el mismo en todos los casos.

  ¿Qué puede deducir de los resultados observados? Debido a que las operaciones no se realizan secuencialmente, los resultados no son deterministas.

2.2. Cálculo de PI por integración numérica

- Abra el archivo pi_serial.c y explique brevemente qué hace.
  Hace el calculo de pi mediante una aproximación numérica.
- Compile y ejecute el programa. ¿Cuál fue el tiempo de ejecución?
  12,456.276 milisegundos
- Identifique la sección del código que puede ser paralelizable y agregue la directiva correspondiente.

```
1  for (i=0; i<num_steps; i++)
2     {
3         x = (i + .5)*step;
4         sum = sum + 4.0/(1.+ x*x);
5     }
```

Identifique el ciclo a paralelizar y agregue la directiva correspondiente.

```
1  #pragma omp parallel for reduce(:= sum)
2   for(i = 0; i < num_steps; i ++){
3       sum = sum + 4.0/(1. + (((i + .5)*step)*((i + .5)*step)));
4   }
```

Examine todas las variables y determine cuáles deben ser declaradas de forma especial. Considere el siguiente pseudo-código como guía:

```
1  #pragma omp parallel private(varname,varname)\ reduction(+:varname,↩
      varname) \
2
3  shared(varname,varname)
4  {
5  ...[C digo paralelo]...
6
7  }
```

Las variables que solo debería modificar un thread a la vez son declaradas como privadas. Si no se usa la alternativa reduce, tanto los indices como la suma debería ser declarada como privada.

Dependiendo de su implementación, quizás también necesite definir secciones críticas para actualizar variables compartidas, etc. Recuerde que la directiva para ello es la siguiente:

```
#pragma omp critical

{
...[Codigo seccion critica]...
}
```

- Muestre el código resultante.

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

long long num_steps = 2000000000;
double step;

int main(int argc, char* argv[])
{
  clock_t start, stop;
  double pi, sum, x;
  int i, id;
  sum  = 0.0;
  step = 1./(double)num_steps;
  // N mero de threads
  omp_set_num_threads(4);
  start = clock();

  #pragma omp parallel for reduction(+: sum)
  for(i = 0; i < num_steps; i ++){
      sum = sum + 4.0/(1. + (((i + .5)*step)*((i + .5)*step)));
  }
  pi   = sum*step;
  stop = clock();

  printf("El valor calculado de Pi es: %15.12f\n",pi);
  printf("El tiempo requerido para calcular Pi fue %f milisegundos\n"↩
     ,
         ((double)(stop - start)/1000.0));
```

```
30    return 0;
31  }
```

El tiempo de ejecución fue de 25,459.423 milisegundos.

- Compile y ejecute el programa con diferentes valores para la variable de ambiente OMP_NUM_THREADS.

  Muestre una tabla con el número de hilos y el tiempo de ejecución obtenido. ¿Qué puede deducir de los resultados esperados?

NUMERO DE HILOS	TIEMPO EJECUCIÓN (msg)
1	24,740.331
2	25,089.014
3	25,380.076
4	25,459.305
5	25,508.1060
6	25,522.427
7	25,539.149
8	25,557.285
9	25,520.893
10	25534.776

2.3. Cálculo de números primos

Ejecución secuencial

En esta sección se buscarán números primos dentro de un rango de enteros con un algoritmo de "fuerza bruta" que se encuentra en el archivo *primos_serial.c*. Se divide el candidato potencial entre sus posibles factores. Si existe el factor, entonces el número no es primo. Los primos encontrados se almacenan en un arreglo.

El programa despliega un indicador de avance que se actualiza con cada incremento de 10 % de los números procesados. Finalmente, el programa también despliega el tiempo de ejecución.

- Compile y ejecute el programa. ¿Cuántos primos encontró en el rango [1..10,000,000]? 664,579 primos
- ¿Cuánto tiempo tardó el programa? 4.29 secs

Ejecución paralela

Modifique el programa para paralelizar el ciclo dentro de la rutina FindPrimes. Compile y ejecute el programa varias veces.

- En promedio, ¿Cuántos primos encontró en el rango [1..10,000,000]? 664,563 primos

- En promedio, ¿Cuánto tiempo tardó el programa? 519.73 secs

En general, deberá haber observado diferencias en el número de primos encontrados. Esto se debe a que hay variables compartidas por los hilos en el código, lo que ocasiona condiciones de carrera.

- Identifique las regiones de código en las que estas condiciones de carrera se dan, y protéjalas insertando directivas pragma omp critical (Ayuda: típicamente hay dos secciones, pero solo una de ellas afecta el resultado)

```
1  int TestForPrime(int val)
2  {
3      int limit, factor = 3;
4
5      limit = (long)(sqrtf((float)val)+0.5f);
6       #pragma omp parallel shared(factor)
7      while( (factor <= limit) && (val % factor))
8          #pragma omp critical
9          factor ++;
10
11     return (factor > limit);
12 }
13 void FindPrimes(int start, int end)
14 {
15     // start siempre es non
16     int i,range = end - start + 1;
17      #pragma omp parallel for private(i) shared(gPrimesFound)
18     for( i = start; i <= end; i += 2 )
19     {
20         if( TestForPrime(i) )
21         #pragma omp critical
22             globalPrimes[gPrimesFound++] = i;
23
24         ShowProgress(i, range);
25     }
26 }
```

- Compile y ejecute el programa varias veces para comprobar que el resultado no cambia y es igual a la versión serie del código.

Y entonces ya encontramos 664,579 primos en 541.34.

Para una mayor precisión al computar el tiempo de ejecución en OpenMP, modifique el código para cambiar los llamados a la función clock() por omp_get_wtime(). Observe que esta última devuelve un double y que ya solo es necesario calcular la diferencia de valores sin dividir por CLOCKS_PER_SEC.

- Compile y ejecute el programa para un rango de valores de [1..10,000,000], variando el número de hilos de 1 a 8. Muestre una tabla con los resultados obtenidos y coméntelos.

NUM THREADS	TIEMPO
1	25.16 secs
2	126.81 secs
3	303.99 secs
4	571.26 secs
5	827.09 secs
6	1,036.79 secs
7	1,280.24 secs
8	1,302.08 secs

Conclusiones

El paralelismo a nivel de tareas permite eficientar la ejecución de un programa que tenga secciones que puedan ser paralelizadas. Para poder elegir la cantidad adecuada de hilos a ejecutar es necesario conocer el problema a paralelizar. También hay que considerar el hardware. Dado que OpenMP está implementado para la mayoría de compiladores y procesadores entonces resulta una buena elección cuando queramos paralelizar tareas. Adicionalmente, es posible customizar los programas en OpenMP, escogiendo, por ejemplo, el stride de vectores para procesar mayor cantidad de datos en un thread, o utilizar la directiva GOMP_CPU_AFFINITY para especificar la manera en cómo se podrían ejecutar los threads.

Referencias

[1] Stallings, William. Computer organization and architecture: designing for performance. Pearson Education India, 2000.

[2] Hennessy, John L., and David A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2011.

[3] Sanders, Jason, and Edward Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional, 2010.