Taller 1: OpenMP

Malcolm Davis, Miembro Estudiantil, IEEE
mdavis.cr@ieee.org

Resumen— En este documento se encuentran las respuestas a las preguntas propuestas en el taller 1 del curso Arquitectura de Computadores II relacionadas con los Hilos de un computador utilizando OpenMP.

Index Terms— TEC, Arquitectura De Computadores II, Threads, OpenMP, TLP.

1. Pregunta 1: Definición OpenMP

¿Qué es OpenMP?
OpenMP es un API para el procesamiento multi hilo con memoria compartida, la especificación incluye directivas de compilador, bibliotecas y variables de entorno. La información de las diferentes versiones de la especificación se puede encontrar en la página web http://www.openmp.org.

2. Pregunta 2: Regiones Paralelas en OpenMP

¿Cómo se define una región paralela en OpenMP utilizando pragmas?
Para entender la definición de una región paralela en OpenMp se necesita conocer el concepto de pragma; un **pragma** es una directiva de compilador que le especifica como interpretar el código.
Al definir una sección de código predecida por un pragma de openmp se puede definir diferentes formas de paralelizar el código. La más básica es la región paralela que se define como:

```
#pragma omp parallel [clause ...]
{
        //Do parallel stuff here
}
```

El código que se encuentre en una región paralela va a ser ejecutado por todos los hilos que se definan(a menos de que haya otra instrucción que especifique lo contrario.

3. Pregunta 3:

¿Cómo se define la cantidad de hilos a utilizar al paralelizar usando OpenMP?
Cómo se habló anteriormente, OpenMP incluye variables de entorno; por esta razón se puede utilizar la variable OMP_NUM_THREADS para definir el número **máximo** de hilos que el programa va a utilizar introduciendo antes de la ejecución del programa en la terminal:

```
$ export OMP_NUM_THREADS=8
```

O para tener más flexibilidad a la hora de definir los hilos que utiliza el programa(utilizar diferente cantidad de hilos en diferentes secciones paralelas) se puede utilizar la función omp_set_num_threads antes de definir la región paralela.

```
omp_set_num_threads(number_of_threads);
#pragma omp parallel [clause ...]
{
        //Do parallel stuff here
}
```

Otra forma de definirlo es utilizando las clausulas del pragma omp parallel con:

```
#pragma omp parallel num_threads(N)
{
        //Do parallel stuff here
}
```

4. Pregunta 4: Compilando OpenMP

¿Cómo se compila un código fuente c para utilizar OpenMP y qué encabezado debe incluirse?
Para compilar el código de C en GNU/Linux se debe incluir el header omp.h y utilizar el flag del compilador -fopenmp. El archivo y la compilación se vería de la siguiente forma:

```
// main.c
#include <omp.h>
#pragma omp parallel num_threads(N)
{
        //Do parallel stuff here
}
$ gcc -fopenmp main.c -o run
```

5. Pregunta 5:

¿Cómo maneja OpenMP la sincronización entre hilos y por qué esto es importante?
La sincronización entre hilos evita conflictos a la hora de utilizar los datos; si no se toma en cuenta el orden de las operaciones o el momento de las mismas el resultado del programa puede ser erróneo. OpenMP maneja la sincronización con diferentes directivas:

- **MASTER**: Se utiliza para definir que sólo el hilo maestro va a ejecutar una sección de código.
- **CRITICAL**: Se utiliza para definir secciones críticas de código que van a ser ejecutadas por 1 hilo a la vez.
- **BARRIER**: Se utiliza para hacer que todos los hilos lleguen a un punto antes de continuar.

- **TASKWAIT**: Se utiliza para definir que un hilo espere a que se complete una tarea.
- **ATOMIC**: Se utiliza para definir pequeñas regiones que definen partes de memoria que puede ser modificada atomicamente(similar a critical)
- **FLUSH**: Se utiliza para guardar el estado de las variables de los hilos a la memoria.
- **ORDERED**: Se utiliza para definir el modo de ejecución en los hilos como si fuera el orden secuencial.

Además en la creación de una sección paralela se pueden definir variables privadas para cada thread con:

```
#pragma omp parallel private(PI)
{
        //Do parallel stuff here
}
```

6. Ejemplo 1: Calculo Secuencial y Paralelo de Pi

6.1. Pi Secuencial

6.1.1. Analice la implementación del código y detecte qué sección del código podría paralelizarse por medio de la técnica de multihilo.

Normalmente un buen punto para paralelizar un código es en dónde ocurre algún ciclo, la forma de hacerlo está limitado a si los elementos del ciclo tienen alguna dependencia con elementos anteriores o futuros. Para este caso no existe dependencia de datos así que un punto importante a paralelizar sería el ciclo:

```
for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
}
```

6.1.2. Con respecto a las variables de la aplicación (dentro del código paralelizable) ¿cuáles deberían ser privadas y cuáles deberían ser compartidas? ¿Por qué?

Se puede hacer un acercamiento ingenuo a la solución del problema simplemente definiendo x como una variable privada y la suma y los pasos como una variables globales, el problema es que con la forma en que se está haciendo la suma de elementos; al utilizar el valor actual de sum para hacer la suma los diferentes hilos van a obtener diferentes valores de sum y el último que escriba va a escribir un valor erróneo porque va a leer un valor erróneo.
Para resolver este problema hay varios acercamientos definir una seción critica(lo cual afectaría el desempeño al ser ejecutado uno a la vez), hacer cálculos parciales y usar al master para hacer las sumas... La solución más conocida es utilizar la clausula reduce con operación de suma sobre las sumas parciales. Por esta razón, las sumas parciales serían privadas(se hace automáticamente con el reduction, pero es importante recalcar que la suma va a tener sólo su valor), al igual que la x y sólo el step sería global ya que no se hacen modificaciones sobre el.

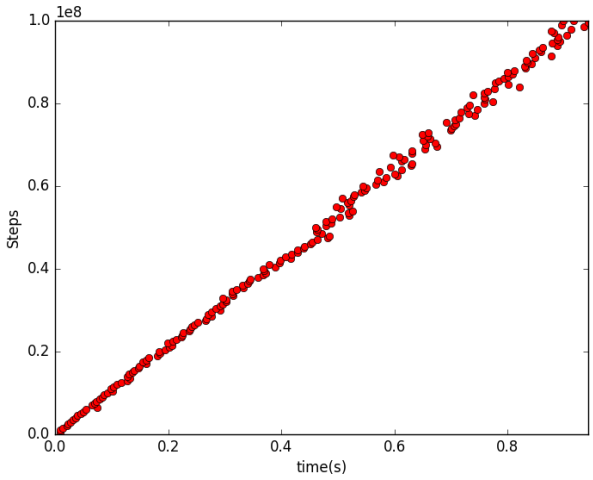

Figura 1. Pi_Ser: Cantidad de pasos Contra Tiempo

6.1.3. Ejecute la aplicación. Realice un gráfico de tiempo para al menos 4 números de pasos distintos (iteraciones para cálculo del valor de pi).

6.2. Pi Paralelo

6.2.1. Analice el código dado. ¿Cómo se define la cantidad de hilos a ejecutar? ¿Qué funcionalidad tiene el pragma omp single? ¿Qué función realiza la línea: #pragma omp for reduction(+:sum) private(x)

Para este código se deja abierta la **cantidad de hilos**, el compilador interpreta realizar 1 hilo por ciclo de la iteración(más el hilo master) cuando se hace el pragma omp for. Para delimitar la cantidad de hilos habría que desactivar la creación dinámica de hilos y establecer una cantidad fija con omp_set_num_threads(number_of_threads).
El **omp single** se utiliza para delimitar que sólo uno de los hilos ejecute una sección de código específica. La linea **#pragma omp for reduction(+:sum) private(x)** como se habló anteriormente, configura la variable x como privada para cada hilo y opera todos los resultados parciales de las sumas de los hilos.

6.2.2. Ejecute la aplicación. Realice un gráfico con tiempo de ejecución para las diferentes cantidades de hilos mostradas en la aplicación. Compare el mejor resultado con la cantidad de procesadores de su sistema. Aumente aún más la cantidad de hilos. Explique por qué, a partir de cierta cantidad de hilos, el tiempo aumenta

El tiempo aumenta según la cantidad de hilos en cierto punto porque el crear demasiados hilos genera un overhead mayor al beneficio de tener más hilos.

7. Ejercicio 1: SAPYX Serial - Paralelo

Realice un programa que aplique la operación SAXPY tanto serial (normal) como paralelo (OpenMP), para al menos tres tamaños diferentes de vectores. Mida y compare el tiempo de ejecución entre ambos. Una operación SAXPYX(**S**ingle-precision **A**lpha***X** **P**lus) **Y** es una operación de precisión simple con la forma $Y = aX + Y$. Para solucionar este problema se propone el siguiente código:

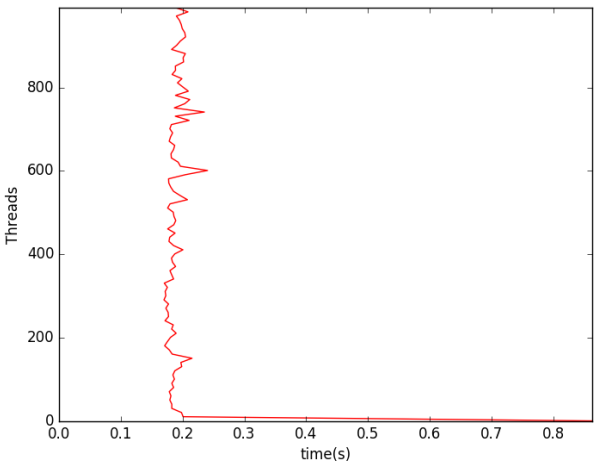

Figura 2. Pi_Par: Cantidad de hilos Contra Tiempo

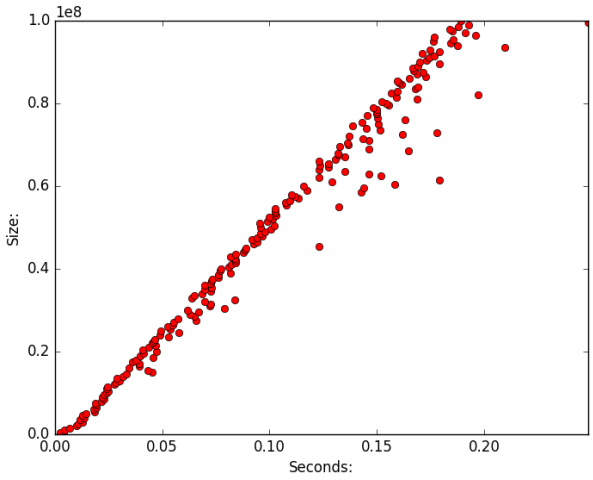

Figura 4. SAXPY_Par: Cantidad de hilos Contra Tiempo

```
void SAXPY(long size, float alpha, float* X,
        float* Y){
        long i;
        #ifdef PARALLEL
        #pragma omp parallel for  private(i)\
      shared(size, alpha, X, Y)
        #endif
        for(i = 0; i<size; i++){
                Y[i]=alpha*X[i] + Y[i];
        }
}
```

Así se puede obtener con el make tanto la versión serial como la versión paralela al utilizar la bandera de compilación $-DPARALLEL$. Como se puede observar en las figuras 3 y 4 la mejora de serial a paralelo es notable; ya que para calcular el SAXPY de vectores de 10e8 elementos se obtiene alrededor de 0.7s y 0.2 segundos respectivamente. Aunque se puede observar un comportamiento lineal en ambos(entre más grande la entrada más tiempo se dura calculando) el tiempo es menor con el uso de las directivas de OpenMP.

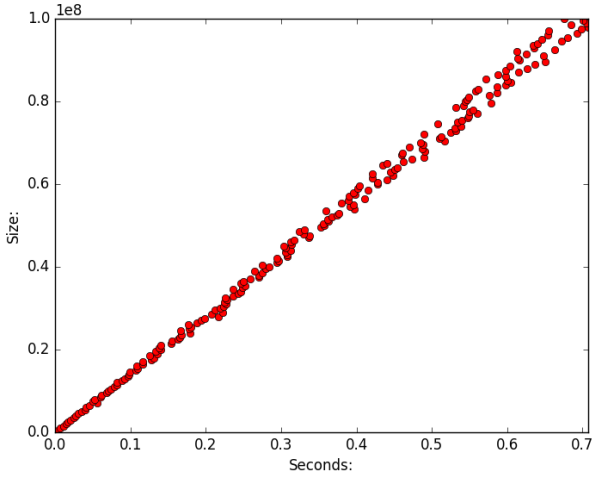

Figura 3. SAXPY_Ser: Cantidad de hilos Contra Tiempo

8. Ejercicio 2: Multiplicación de Matrices

Realice una aplicación paralelizable que requiera gran cantidad de procesamiento. Su aplicación podría ser una aproximación a una integral, una serie, un conjunto de operaciones, etc. Por su naturaleza de crecer rápido, el algoritmo de multiplicación de matrices es un buen ejemplo para parelizar(además porque numerosos problemas se resuelven con la multiplicación de matrices). Para este taller se propone el acercamiento más sencillo a esta solución:

```
void matMult(struct doubleMatrix* A,
             struct doubleMatrix* B,
             struct doubleMatrix* C){
        long i, j, k;
        double sum = 0;
        #ifdef PARALLEL
        #pragma omp parallel for \
    private(i,j,k,sum) shared(A, B, C)
        #endif
        for (i = 0; i < A->nrows; i++) {
          for (j = 0; j < B->ncols; j++) {
            for (k = 0; k < B->nrows; k++) {
              sum += A->data[i*A->nrows+k] *
              B->data[k*B->nrows+j];
            }
            C->data[i*C->nrows+j] = sum;
            sum = 0;
          }
        }
}
```

Al igual que con el ejercicio de SAXPY se puede concluir observando las figuras 5 y 6 que la solución paralela aprovecha más los recursos y logra resolver el mismo problema para la misma cantidad de entradas en menor tiempo que la solución serial.

Referencias

[1] B. Blaise.*OpenMP*". Computing.llnl.gov, 2017.Available: https://computing.llnl.gov/tutorials/openMP/. [Accessed: 23- Feb- 2018]

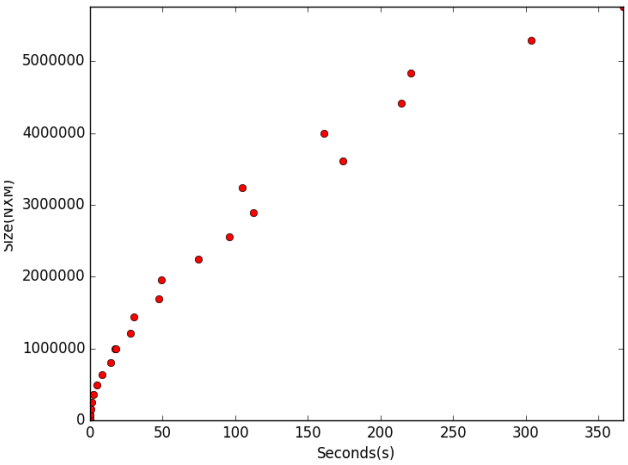

Figura 5. matMul_Ser: Cantidad de hilos Contra Tiempo

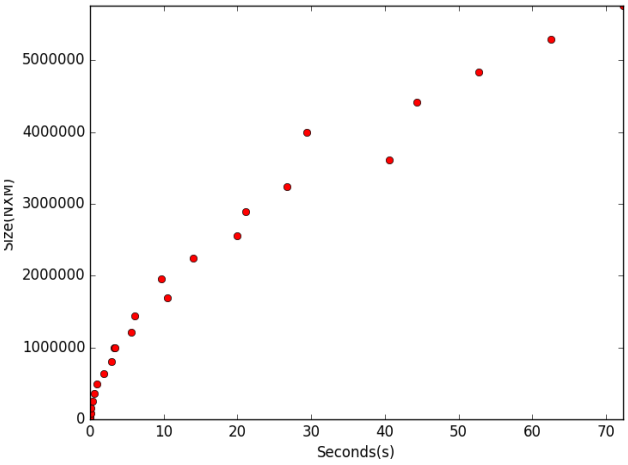

Figura 6. matMul_Par: Cantidad de hilos Contra Tiempo