Instituto Tecnológico de Costa Rica

Área Académica de Ingeniería en Computadores

Curso: CE-430 Arquitectura de Computadores II

Taller 1: OpenMP

Realizado por:

Dennis Gerardo Porras Barrantes - 2015084004

Profesor:
Jefferson González Gómez

Grupo:
1

Fecha: Cartago, Agosto 24, 2018

Micro-Investigación

1- ¿Qué es OpenMP?

Básicamente, es un conjunto de directivas de compilador y bibliotecas con rutinas de tiempo de ejecución, para expresar paralelismo en memoria compartida. El lenguaje base queda sin especificación, y esto permite implementarlo en cualquier compilador.

2- ¿Cómo se define una región paralela en OpenMP utilizando pragmas?

Las regiones paralelas se definen con la siguiente sintaxis:

#pragma omp parallel [cláusulas]

Bloque

Esto crea un grupo de hilos, las clausulas son las formas en que se acceden a las variables.

3- ¿Cómo se define la cantidad de hilos a utilizar al paralelizar usando OpenMP?

Para configurar el número de hilos se utiliza la función:

omp_set_num_threads (int t)

Puede ser llamado en múltiples partes de programa y toma el argumento para crear esa cantidad de hilos en secciones paralelas

4- ¿Cómo se compila un código fuente c para utilizar OpenMP y qué encabezado debe incluirse?

Se debe agregar la bandera -fopenmp, como en el siguiente objeto:

gcc -o omp_helloc -fopenmp omp_hello.c

5- ¿Como maneja OpenMP la sincronización entre hilos y por qué esto es importante?

La sincronización en OpenMP, al igual que en con otras bibliotecas de hilos se pueden compartir variables. Sin embargo, también cuanta con ***pragmas*** de sincronización como ***critical*** o ***atomic*** que son mutuas. Y se pueden usar por eventos que obligan al hilo a esperar a que algo pase, como el uso de ***flush.***

Ejercicio Pi Serial

1- Analice la implementación del código y detecte que sección del código podrá paralelizarse por medio de la técnica de multihilo.

Analizando el código, se nota que la única parte que se puede paralelizar es el ciclo for donde se calcula el área bajo la curva. Esto se debe a que solamente se calcula un pequeño rectángulo del área y se suma a una variable con el área completa.

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

- **x:** debe ser privada, dado que es una variable que se calcula en un ciclo. Y no deben usar otros ciclos, entonces cada hilo calcula esta variable.
- **sum:** debe ser compartida, ya que cada calcula de áreas deben ser sumadas en una sola variable, que representa la integral. Entonces sum debe estar publica para que cada hilo pueda agregar su cálculo.

3- La ejecución del programa.

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

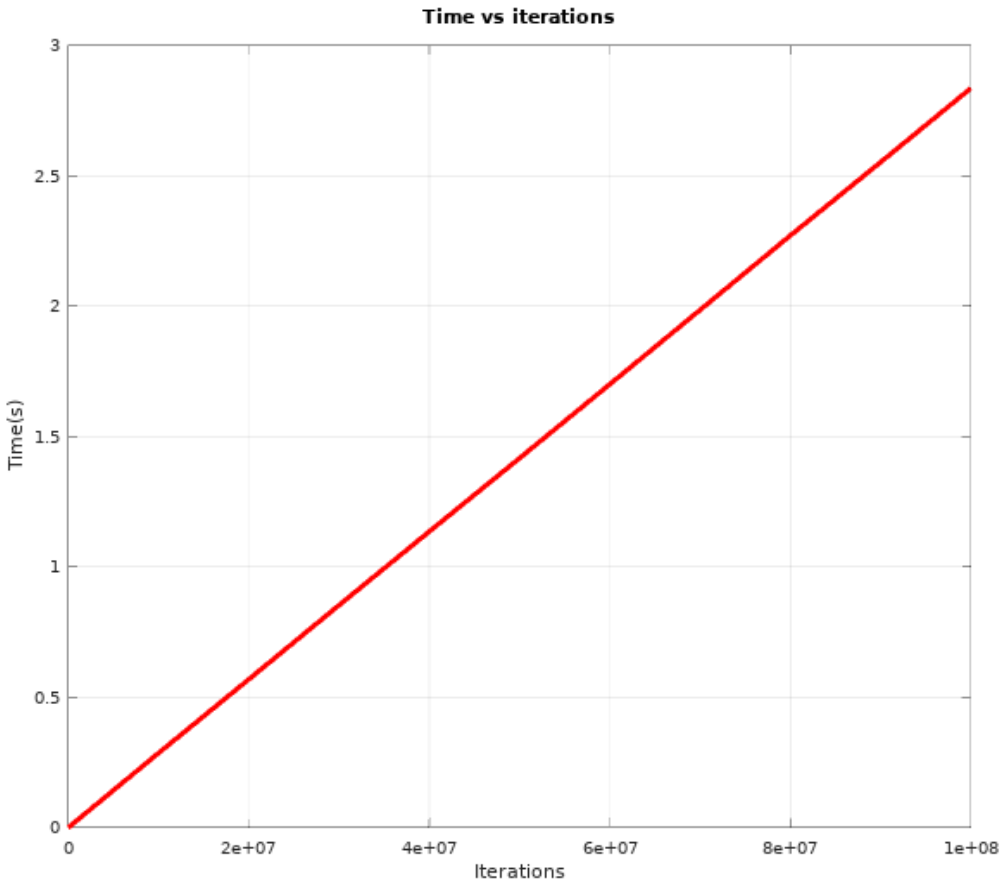

FIGURA 1. TIEMPO VS ITERACIONES EN CÁLCULO DE PI SERIAL.

Ejercicio Pi Paralelo:

1- Analice el codigo dado. ¿Como se define la cantidad de hilos a ejecutar? ¿Que funcionalidad tiene el pragma omp single? ¿Qué funcion realiza la línea: #pragma omp for reduction(+:sum) private(x)?

Primero se obtiene el número de procesadores que tiene la máquina, y se multiplica por dos y luego con la función omp_set_num_threads() se van cambiando la cantidad de hilos hasta que llegue a la variable calculada.

La función de omp single es que ese bloque de código se ejecute solo una vez por un hilo.

A la hora de paralelizar el for se tiene la línea: ***#pragma omp for reduction(+:sum) private(x)***

Esto primero para hacer ese for paralelo, luego con ***reduction*** permite decir que la variable ***sum*** es compartida y que al final se va a hacer un suma con esa variable. Private es para que cada hilo tenga un ***x*** privado, y solo ellos tengan ese valor.

2- La ejecución del programa
3- Ejecute la aplicación. Realice un gráfico con tiempo de ejecucion para las diferentes cantidades de hilos mostradas en la aplicación. Compare el mejor resultado con la cantidad de procesadores de su sistema. Aumente aun más la cantidad de hilos. Explique por qué, a partir de cierta cantidad de hilos, el tiempo aumenta.

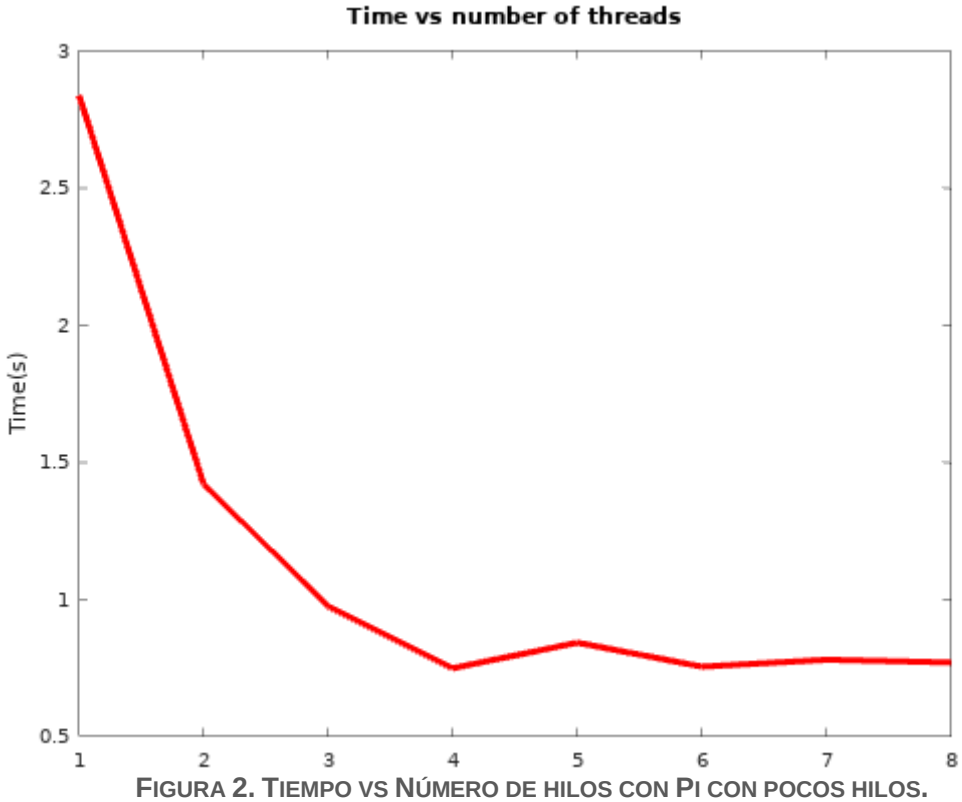

FIGURA 2. TIEMPO VS NÚMERO DE HILOS CON PI CON POCOS HILOS.

El mejor resultdo se da cuando tiene 4 hilos corriendo, tomando en cuanta que mi computadora tiene 4 núcleos, es esperado que se tengo el mejor ahí. Aunque, en otro corridas de programa se tenian bueno resultados con 8 procesodores debido talvez a que cada nucleo se puede ver como 2 nucleos.

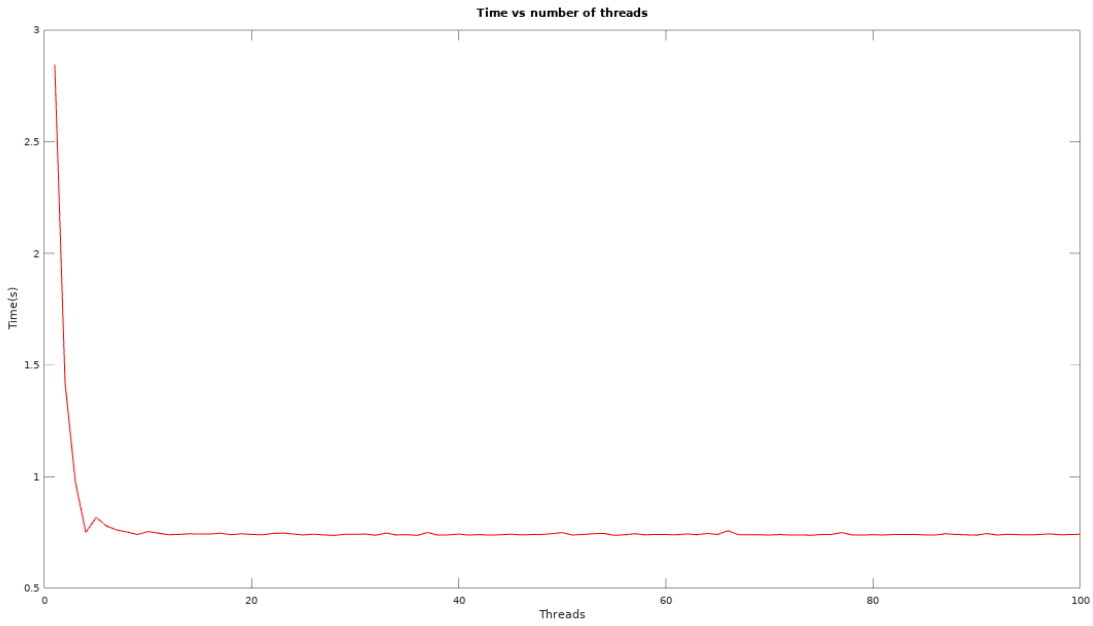

FIGURA 3. TIEMPO VS NÚMERO DE HILOS CON MUCHOS HILOS.

Como se observa cuando se usan muchos hilos, ya no hay mejor. Influye mucho el hecho de que solo hayan 4 núcleos y todos los hilos no pueden acceder a memoria simultaneamente si no que siempre habrá un delay. Además, la ley de Ahmdal demuestra esto, siempre habrá un tope debido a la parte no paralelizable.

Ejercicios prácticos

1- Operación SAXPY.

Resultados en serial

TABLA 1. RESULTADOS DE SAXPY SERIAL.

Tamaño de vector	Tiempo (s)
10000000	0.050662
20000000	0.102456
30000000	0.255523

Resultados en paralelo

TABLA 2. RESULTADOS DE SAXPY PARALELO.

Tamaño de vector	Tiempo (s)
10000000	0.024238
20000000	0.049101
30000000	0.114356

Como se observa en ambas tablas, el tiempo en paralelo es aproximandamente dos veces más rápido. Tal vez se debería esperar que sea cuatro veces por ser un procesador con cuatro núcleos, sin embargo por muchas razones de accesos a memoria esto no podría ser así y se dé el caso que se observa.

El código se encuentra en el siguiente enlace de GitHub, como la solución al segundo problema:

https://github.com/denporras/CE-4302-TallerOpeMP

Bibliografía

(s.f.).

College, D. (23 de Marzo de 2007). *Dartmouth College*. Obtenido de How to Compile and Run An OpenMP Program: https://www.dartmouth.edu/~rc/classes/intro_openmp/compile_run.html

Javier, C. (s.f.). *Departamento de Ingeniería y Tecnología de Computadores*. Obtenido de Programación en el Supercomputador Ben Arabi Programación con OpenMP: http://www.ditec.um.es/~javiercm/curso_psba/sesion_03_openmp/PSBA_OpenMP.pdf

Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering, 5(1), 46-55.

Quinn, M. J. (2003). Parallel Programming. TMH CSE, 526.