Instituto Tecnológico de Costa Rica
Área Académica Ingeniería en Computadores
Arquitectura de Computadores II
II Semestre 2023

Taller 1: OpenMP

Estudiante:

Oscar Jesús Méndez Granados

Carné:

2019150432

Profesor:

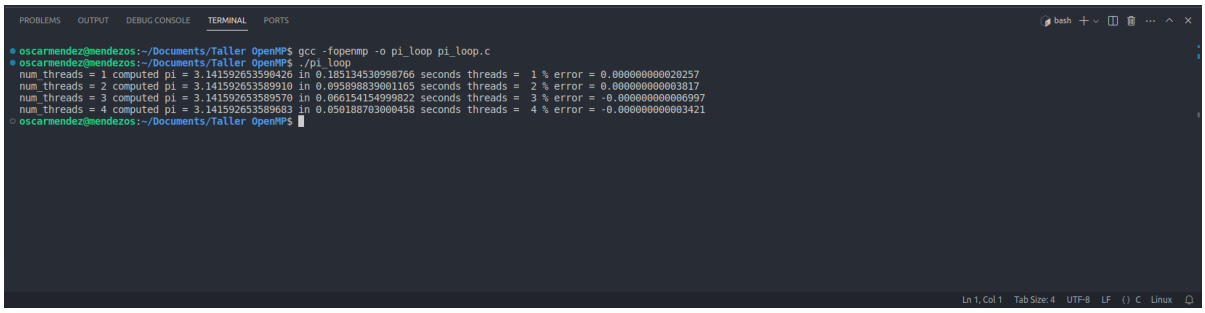
Ronald Eduardo García Fernández

Punto 3

a- Compile ambos códigos y compare sus tiempos de ejecución

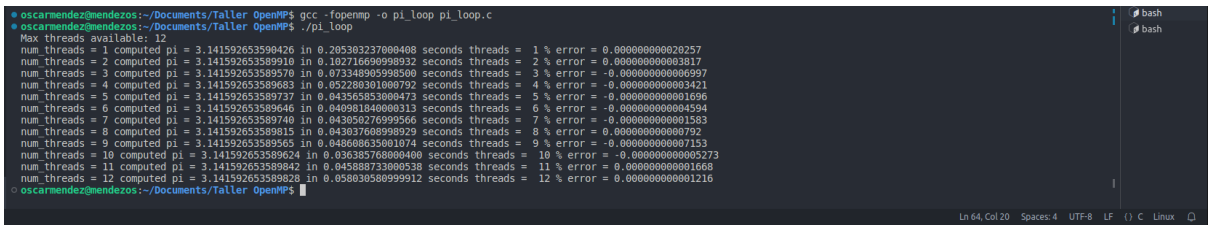
Pi.c

Pi_loop.c



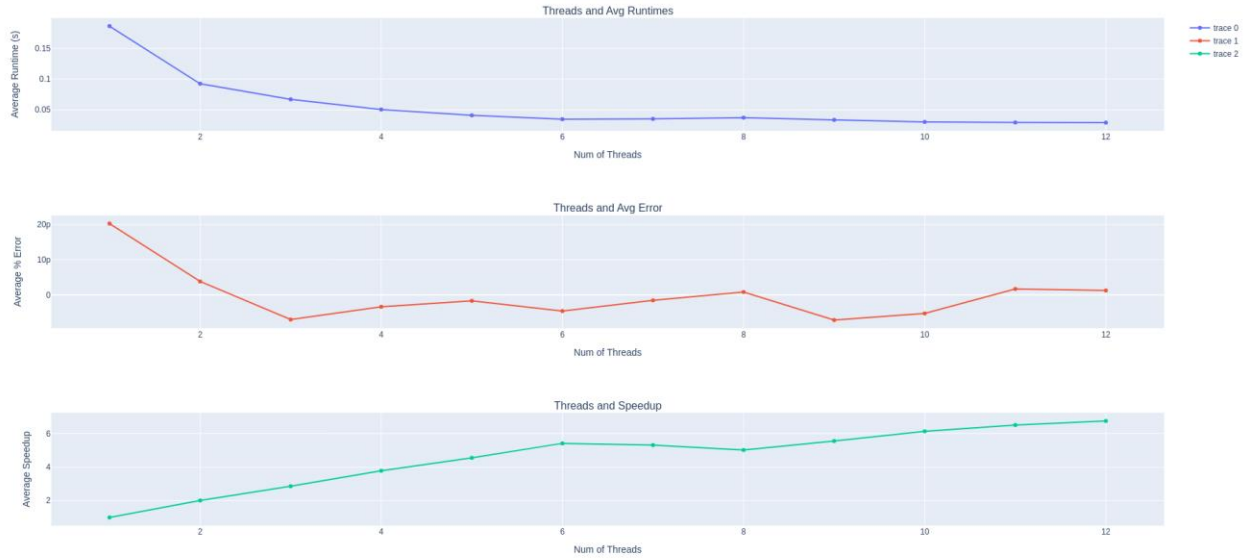
Realizando una comparación del tiempo de ejecución se puede observar que los tiempos de ejecución en Pi_loop son menores y esto es se debe a que OpenMP permite dividir la tarea en fragmentos más pequeños que pueden ser ejecutados en paralelo por diferentes hilos y en este caso el código es lo suficientemente paralelizable como para aprovechar esos beneficios.

b- Modifique pi_loop.c de forma que puede ejecutar el número de threads disponible en su sistema



c- Ejecute pi_loop.c múltiples veces (~100) y almacene los resultados en un directorio results por ejemplo pi_loop_01.txt, ..., pi_loop_100.txt

d- Utilice algún script o herramienta para graficar: tiempo de ejecución vs número de threads, porcentaje de error vs número de threads, y mejora (speedup) vs número de threads, considere el speedup como el tiempo de ejecución con un solo thread entre el tiempo de ejecución con múltiples threads puede referirse al ejemplo proporcionado plotter.py

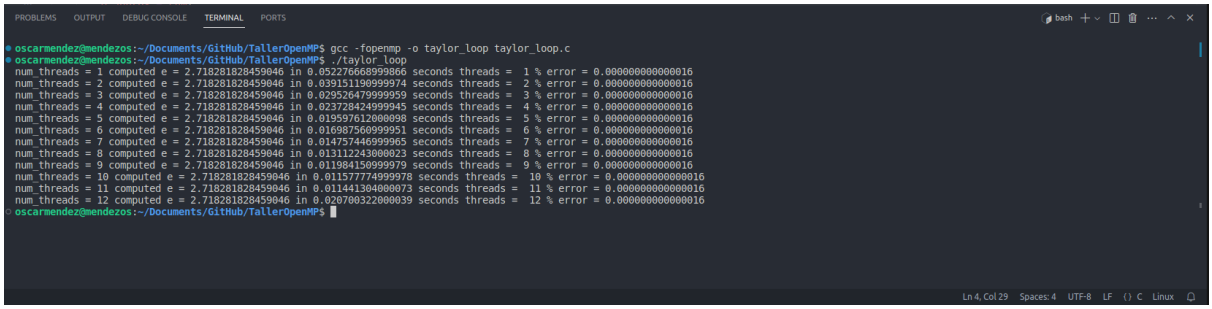


Punto 4

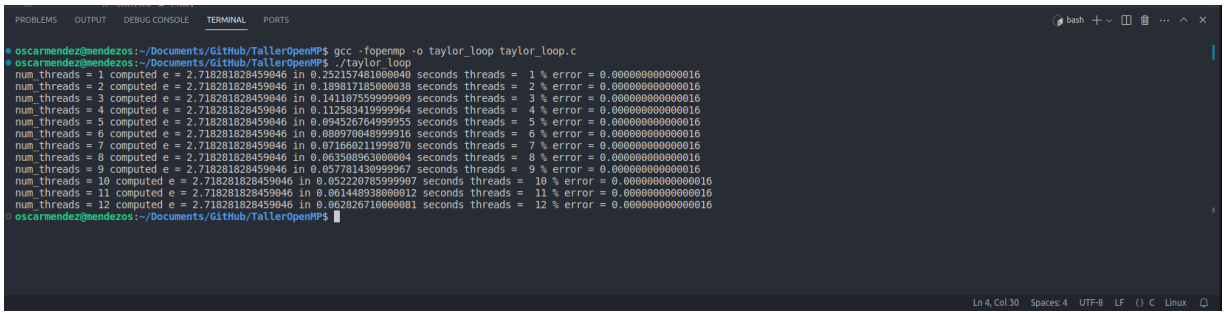
a- Un programa que permita emplear múltiples threads para el cálculo de la constante e mediante el uso de series de Taylor con diferentes términos n (justifique cuantos términos va a emplear).

Inicialmente se hicieron varias pruebas con distintos valores de N, como se muestra a continuación:

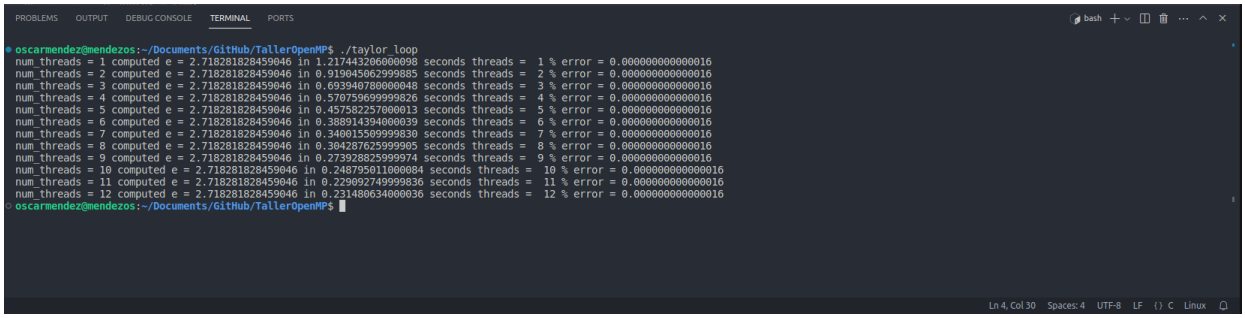
Con N = 4800



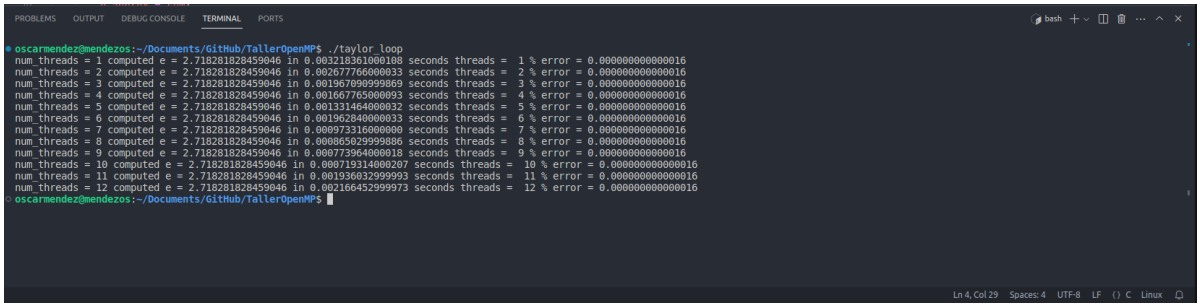
Con N = 10800



Con N = 24000



N = 1200



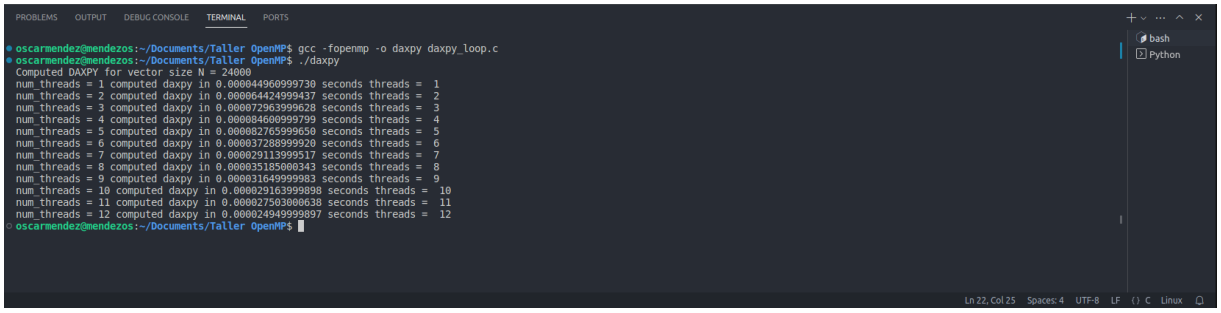
Para todas estas ejecuciones una de las que mejor comportamiento tuvo fue la de ejecución de N igual a 1200, además como se puede observar, todas las ejecuciones se realizaron con un N múltiplo de 12 y esto se debe a que el número de threads de mi equipo es 12 y, por lo tanto, un buen número N debía de ser uno divisible por 12 a fin de que se pueda distribuir de manera uniforme el trabajo entre los hilos, evitando desequilibrios en las cargas de trabajo. Por su parte, si se observan los valores de N, en la mayoría de ellos el error es bastante similar, y si varía este lo hace en el orden de los +15 decimales, es por esto que se considera que el valor de 1200 es un buen valor de N ya que no pone en riesgo la exactitud del cálculo y a su vez el valor es uno de los que mejor se comportó en una relación (exactitud/velocidad del cálculo).

b- Un programa que implemente la operación DAXPY de forma single thread y multithread (openMP), dicho programa deberá generar vectores aleatorios de tamaño definido N, considere las capacidades de su sistema para definir los valores de prueba (al menos 4 diferentes valores)

De igual forma en la que se realizó en el caso anterior con el cálculo de Euler por series de Taylor, a continuación, se presentan los resultados de DAXPY para diferentes valores de N:

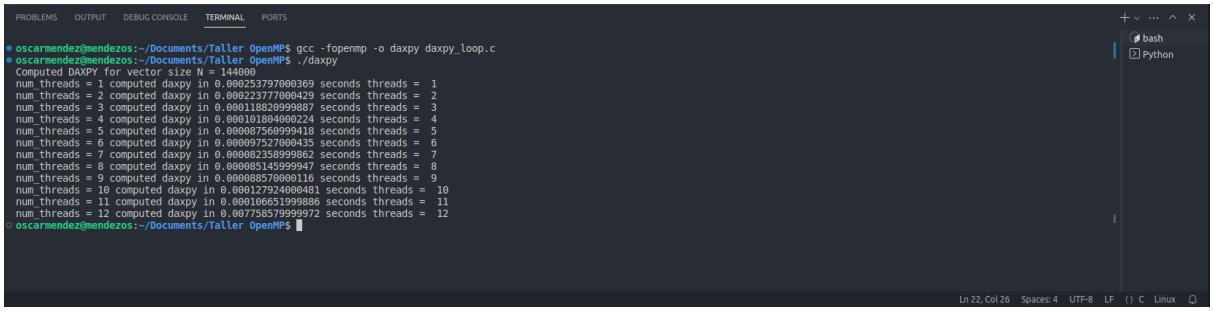
N = 1200

N = 24000

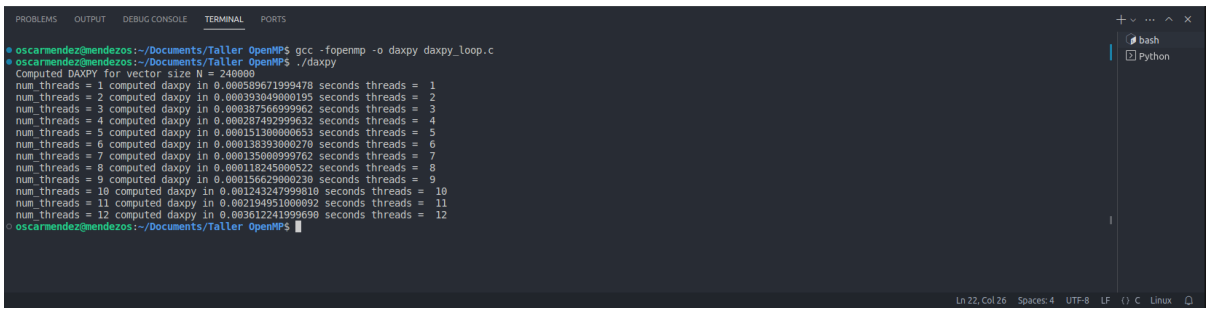


N = 72000

N = 144000



N = 240000

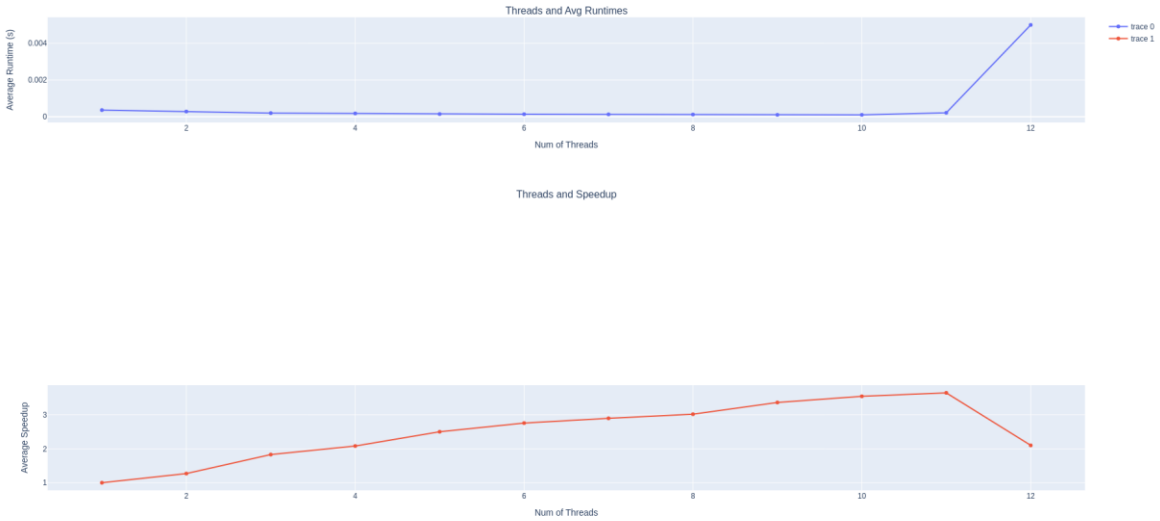


Como se puede observar en las capturas de pantalla anteriores, el tiempo de ejecución en la mayoría de los casos va en descenso una vez se llega a los 10 threads, sin embargo, luego de este valor, este tiende a crecer, esto puede deberse a que no todo el programa es compatible para una paralelización con esta cantidad

de hilos y, además, que esto puede deberse a la sobrecarga de la concurrencia y a la competencia por recursos que los hilos agregan.

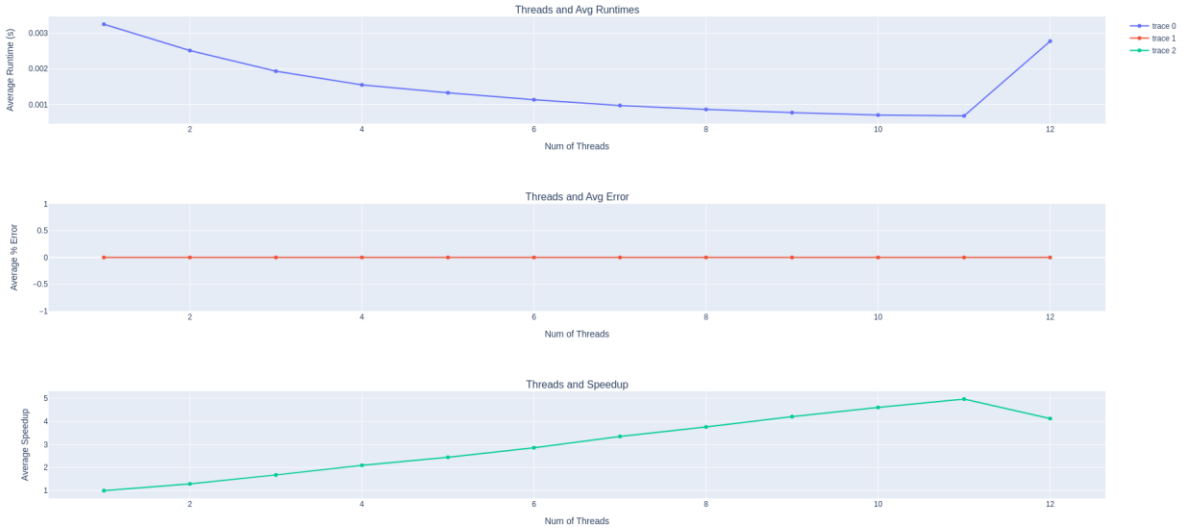
c- Un programa/script que permita graficar los resultados después de ejecutar el programa implementado en el punto a y b, de forma que se puede visualizar tiempo de ejecución, speedup, y porcentaje error (para el caso a).

Daxpy_loop.c



Inicialmente, en cuanto a la ejecución del DAXPY 100 veces, se puede observar que el comportamiento de este en cuanto a tiempo de ejecución es bastante constante, pero luego de llegar a un umbral este tiempo se dispara, esto se puede deber a como bien se mencionó en el punto anterior que existe mucha competencia por los recursos y que además la administración de hilos, la sincronización y la comunicación entre ellos, puede verse afectada a medida en que se agregan más hilos (dependiendo del programa), haciendo que la ejecución sea más lenta. Por su parte, en cuanto al speedup se puede observar cómo el mismo va en crecimiento hasta que llega a un umbral y esto puede ser fundamentado debido a la Ley de Amdahl donde el speedup máximo se puede lograr al paralelizar un programa, pero este está limitado por la parte del programa que no se puede ejecutar en paralelo. A medida que se agregan más hilos para paralelizar una task, la fracción no paralelizable se convierte en un cuello de botella, limitando el aumento del speedup.

Taylor_loop.c (Euler)



Por su parte, en la ejecución del código del cálculo de Euler se puede observar cómo al igual que con el DAXPY, el tiempo de ejecución en principio viene decreciendo y una vez llegado a un umbral, este tiempo se dispara, esto se puede deber a como bien se mencionó, a las dificultades de sincronización de los hilos y de además de las competencias que tienen estos por los recursos. Por su parte, se puede observar como el speedup tiene un comportamiento creciente, pero una vez llegado a un umbral este ya no crece más, esto puede ser producto de que el código no es completamente paralelizable para correr en un entorno con 12 hilos. Por su parte, con respecto al error obtenido, si se observa el mismo es muy cercano a cero, lo que puede ser considerado no tan crítico dependiendo la aplicación que se le dé al código, en este caso el error empieza a tener variaciones en el orden de los +15 decimales, siendo este 0.000000000000016 por lo cual, puede ser no considerado un valor muy grande.