Tecnológico de Costa Rica

CE4303: Arquitectura de Computadores

Taller 1

Profesor:

Ronald García

Estudiante:

Brandon Gómez Gómez – 2019173506

Fecha de entrega: 9 de octubre del 2023

Semestre II, 2023

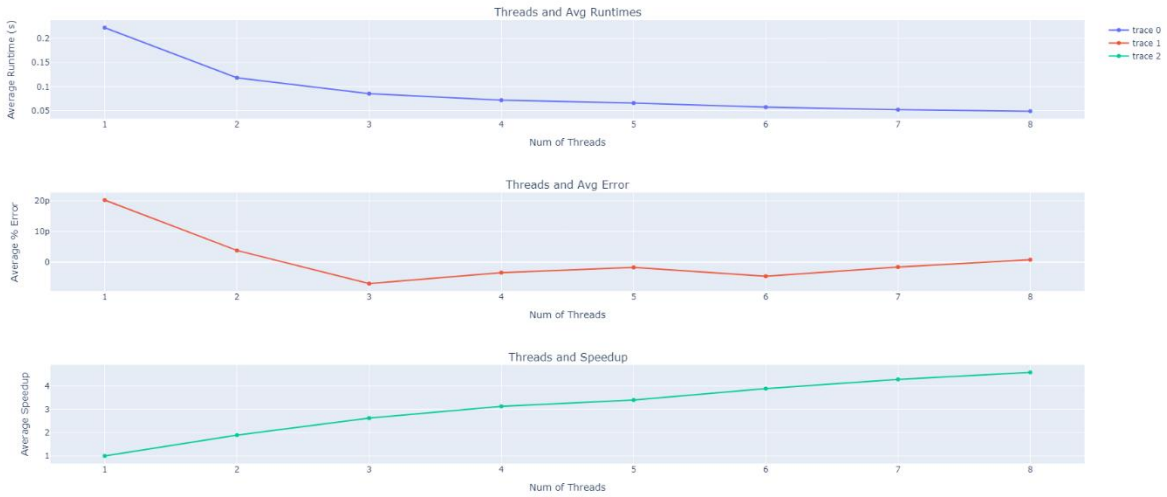
Ejecutable 1 pi.c sin modificaciones

```
PS C:\Users\Brandon\Downloads\taller01_openMP-20231008T192028Z-001\taller01_openMP\openmp> ./executable1

 pi with 100000000 steps is 3.141593 in 0.224000 seconds

PS C:\Users\Brandon\Downloads\taller01_openMP-20231008T192028Z-001\taller01_openMP\openmp> ./executable2
num_threads = 1 computed pi = 3.141592653590426 in 0.221999883651733 seconds threads =  1 % error = 0.000000000020257
num_threads = 2 computed pi = 3.141592653589910 in 0.140000104904175 seconds threads =  2 % error = 0.000000000003817
num_threads = 3 computed pi = 3.141592653589570 in 0.089999914169312 seconds threads =  3 % error = -0.000000000006997
num_threads = 4 computed pi = 3.141592653589683 in 0.073000192642212 seconds threads =  4 % error = -0.000000000003421
PS C:\Users\Brandon\Downloads\taller01_openMP-20231008T192028Z-001\taller01_openMP\openmp>
```

Ejecutable 1 pi_loop.c con modificaciones

```
PS C:\Users\Brandon\Downloads\taller01_openMP-20231008T192028Z-001\taller01_openMP\openmp> ./executable2
num_threads = 8
num_threads = 1 computed pi = 3.141592653590426 in 0.222999811172485 seconds threads =  1 % error = 0.000000000020257
num_threads = 2 computed pi = 3.141592653589910 in 0.113999843597412 seconds threads =  2 % error = 0.000000000003817
num_threads = 3 computed pi = 3.141592653589570 in 0.085999965667725 seconds threads =  3 % error = -0.000000000006997
num_threads = 4 computed pi = 3.141592653589683 in 0.072000026702881 seconds threads =  4 % error = -0.000000000003421
num_threads = 5 computed pi = 3.141592653589737 in 0.065999984741211 seconds threads =  5 % error = -0.000000000001696
num_threads = 6 computed pi = 3.141592653589646 in 0.062999963760376 seconds threads =  6 % error = -0.000000000004594
num_threads = 7 computed pi = 3.141592653589740 in 0.052999973297119 seconds threads =  7 % error = -0.000000000001583
num_threads = 8 computed pi = 3.141592653589816 in 0.049000024795532 seconds threads =  8 % error = 0.000000000000820
```

Calculo de PI graficado



Análisis punto #3

De las imágenes anteriores empezando por los datos obtenidos de consola, se puede observar inicialmente como el calculo de pi con ambos ejecutables sin modificaciones, presentan mayor duración en el cálculo del valor. Esto se debe a que los datos y cálculos se realizan en una sola secuencia y la tarea no es fragmentada en partes como si sucede cuando se usan múltiples hilos.

En la segunda imagen, la cual, si utiliza multithreading y la totalidad de los hilos del sistema, se puede observar como el tiempo de ejecución disminuye conforme a mas hilos se utilizan, comprobando la teoría donde la distribución de tareas entre los núcleos permite una ejecución simultanea de valores lo cual permite obtener valores rápidamente. Se observa la diferencia entre 1 y 8 hilos donde es bastante grande siendo aproximadamente 5 veces menor.

De la grafica se puede observar el comportamiento anteriormente mencionado. Donde conforme incrementan los hilos se tiene un tiempo de ejecución menor y un speed up mayor, este fenómeno es conocido como paralelización efectiva y se relaciona con la ley de Amdahl y Gustafson.

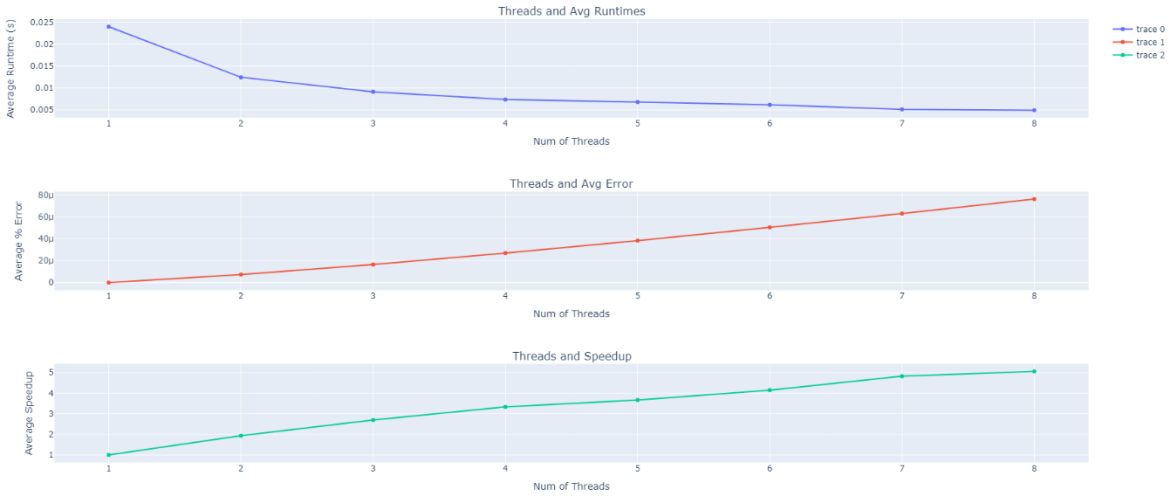
La ley de Amdahl nos menciona que el speedup máximo obtenible, esta limitado por la fracción secuencial del programa, relacionando la fracción secuencial del programa con el numero de hilos o unidades de procesamiento del sistema. Con lo cual de la misma se obtiene la conclusión de que incluso con hilos infinitos, el speedup que podemos obtener esta limitado por la parte secuencial de nuestro programa. Y esto se puede observar en la grafica anterior, donde se puede ver que conforme aumentan los hilos el speedup y el tiempo de ejecución se van estabilizando a un valor, si aumentáramos los hilos a infinito esta tendencia se estabilizaría ya que por cada hilo el speedup que obtendríamos seria mínimo o incluso nulo.

En cuanto a la ley de Gustafson, este nos habla sobre la fracción paralelizable de nuestro programa y que a mayor sea esta, mayor será el speedup del sistema. De la grafica se puede observar que el speedup/threads es aproximadamente ½ por cada hilo agregado, lo cual esta largo del comportamiento lineal ideal que se desearía y por lo tanto nos da una pista de que el programa se podría paralelizar aun más.

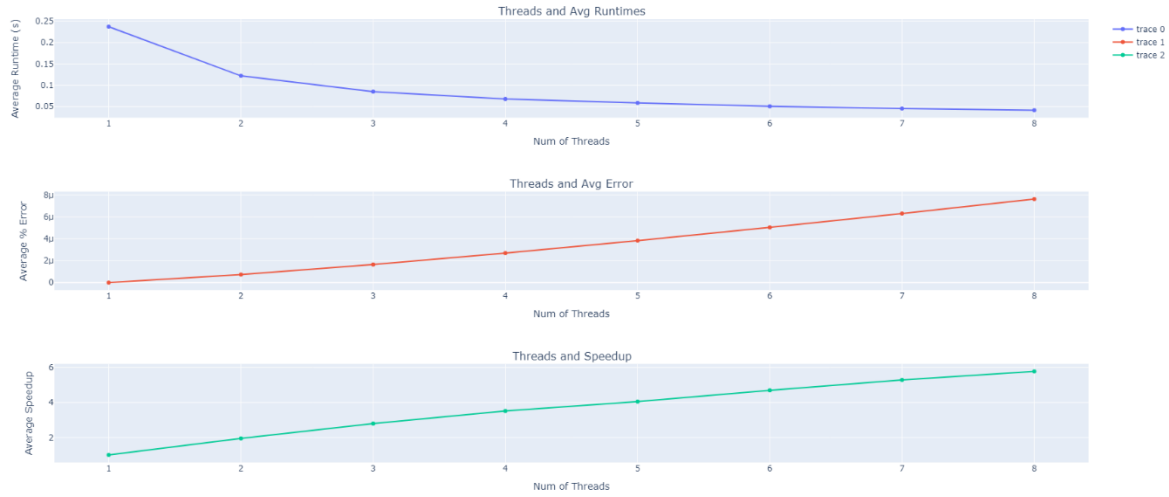
En cuanto al porcentaje de error obtenido de la gráfica, se observa inicialmente una decaída, para posteriormente tender al alza, lo cual puede deberse a múltiples factores. Donde la carga, el tipo de operación, la cantidad de recursos y la fragmentación de las tareas influye en el valor entregado por los hilos. Puede que al darle una mayor carga a un hilo se obtenga un error acumulativo mayor que cuando se le dan cargas menores y viceversa, sin embargo, este error, depende de varios factores donde el mismo código influye en el resultado esperado.

Análisis punto #4

Taylor 10 millones



Taylor 100 millones



Utilizando la teoría mencionada en el punto anterior sobre Amdahl y Gustafson se puede observar como se tiene un speedUp mejor que en el punto anterior y esto es debido a la cantidad de paralelización que tiene el programa desarrollado, donde la mayor parte de operaciones matemáticas se encuentran paralelizadas. Sin embargo, se puede observar como el % de error crece conforme aumentan los hilos

y esto es posible que se deba a que se tiene un error acumulado, donde en el programa al dividir la tarea que genera un pequeño error y replicarse en la suma acumulada de los hilos se incrementa el % de error final. Asimismo, este problema puede deberse a que también existe una mayor competencia por los recursos entre más hilos compiten por la información.

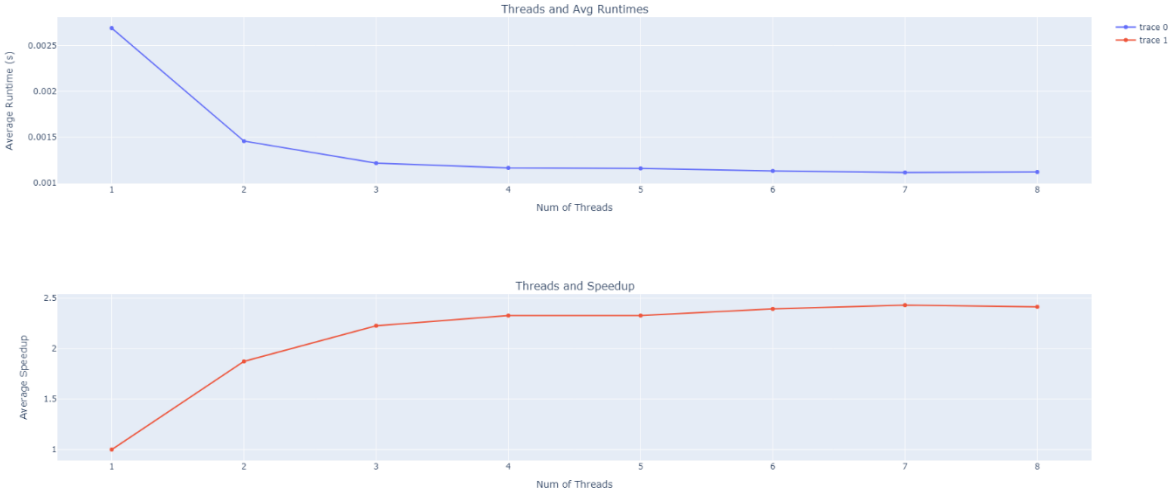
En cuanto al N utilizado para el calculo de los valores de e, inicialmente se pensó el valor que garantizara la mayor precisión por lo cual se eligió un valor grande que realizara la ejecución relativamente rápido en el orden de los milisegundos. Asimismo, este valor de N=10Millones, ofrecía un costo-tiempo mejor que otros valores y a su vez reducía el % de error obtenido. En si la elección de este N se basó en su mayoría en el obtener un punto medio entre la precisión el rendimiento y la utilización de los recursos del sistema.

En cuanto al speedUp y el tiempo de ejecución de e para este programa, se observa en la grafica como se obtiene un comportamiento similar al del calculo de pi, sin embargo, un mayor speedUp que probablemente se deba a la complejidad del calculo y al modo de paralelización utilizado.

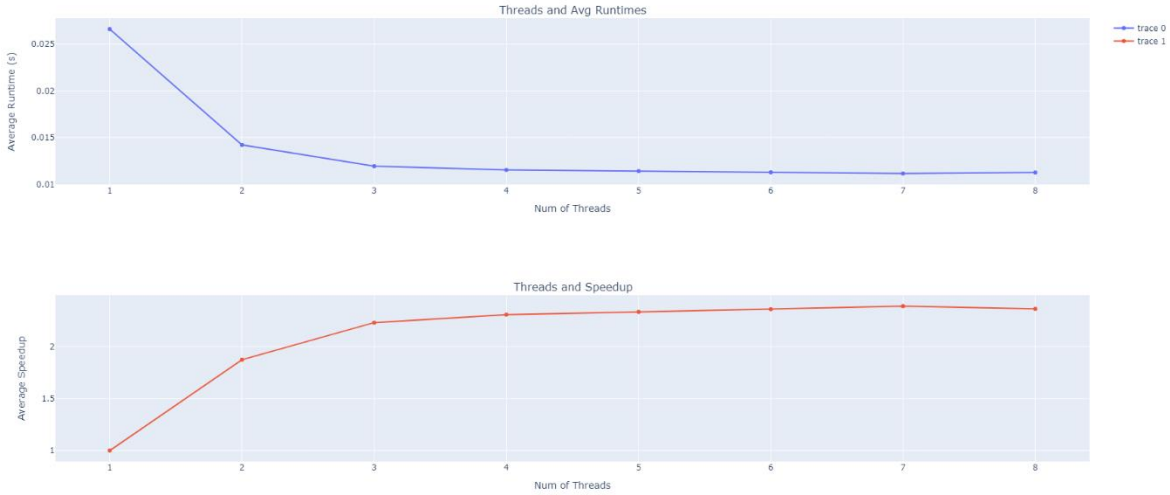
Finalmente, respecto a este análisis de la implementación del algoritmo de Taylor, se observa como los tiempos de ejecución empiezan a variar cada vez menos conforme crecen la cantidad de hilos, lo cual comprueba esta estabilización mencionada por Amdahl donde existe un limite en cuanto a la cantidad de ganancia de rendimiento o tiempo que se pueda lograr al implementar múltiples núcleos/hilos.

Análisis parte 4.B

Daxpy 10 millones



Daxpy 1 millon



Para empezar, es de resaltar la razón por la cual solo hay dos graficas cuando se esperan cuatro y esto es debido a que para valores pequeños de N se tenia el problema de que algunos tiempos daban cero, por lo cual al intentar posteriormente operar los datos se tenían problemas con las divisiones. Asimismo, el variar N en la misma escala no iba a garantizar un cambio significativo por lo cual se decidió solo realizar cambios de N en base 10, valores los cuales se muestran en las anteriores graficas.

Ahora bien, respecto a las graficas obtenidas, se observa un speedup aún menor y esto se debe a múltiples factores del código. Para empezar, hay una parte sustancial del código que es secuencial por lo cual se tiene un problema relativamente grande si nos queremos apegar a la teoría de Amdahl. Asimismo, en el caso del código desarrollado existe cierta sobrecarga, debido a la inicialización y finalización de los tiempos en cada una de las iteraciones. Por otra parte, se pueden realizar optimizaciones aún mayores relacionadas a la teoría y al hardware donde se esta ejecutando el programa. Por ejemplo, el sistema origen, trabaja con linea de cache de 64 bytes por lo cual se puede alinear los datos para realizar operaciones SIMD, asimismo, se pueden utilizar instrucciones de openMP como #pragma omp simd, para realizar vectorización de bucles. Por otra parte, también se puede intentar alinear los cálculos para que trabajen con los datos que entren a las líneas de cache completas, para de esta manera minimizar las lecturas y escrituras a memoria principal. O también, se podría utilizar optimizaciones del compilador para obtener un mejor desempeño.

Si bien, muchas de estas soluciones no fueron implementadas a la hora de desarrollar los códigos, las mismas si se tienen presentes para poder garantizar la mayor eficiencia y el mejor rendimiento a la hora de ejecutar tareas multi hilos o con múltiples unidades de procesamiento.

Finalmente, es importante recalcar que el bajo desempeño de la prueba daxpy se debe a las pocas optimizaciones realizadas al mismo que si bien se encuentra paralelizado, podría obtener mejores resultados con las optimizaciones anteriormente mencionadas. Asimismo, aun sin estar optimizado al máximo, el mismo presenta un comportamiento similar a los otros ejemplos presentados los cuales se encuentran respaldados teóricamente por las leyes de Amdahl, Gustafson y demás teoría que existe al respecto.

Referencias

Mattson, T., & Engineer, L. M. P. (s/f). *A "Hands-on" Introduction to OpenMP*. Openmp.org. Recuperado el 8 de octubre de 2023, de https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf

van der Pas, R. (s/f). *An Overview of OpenMP*. Openmp.org. Recuperado el 10 de octubre de 2023, de https://www.openmp.org/wp-content/uploads/ntu-vanderpas.pdf

Computer organization. (2018, junio 11). GeeksforGeeks. https://www.geeksforgeeks.org/computer-organization-amdahls-law-and-its-proof/

(S/f). Intel.com. Recuperado el 8 de octubre de 2023, de https://www.intel.com/content/www/us/en/products/sku/203473/intel-core-i310100f-processor-6m-cache-up-to-4-30-ghz/specifications.html

Singal, G., Gopalani, D., Kushwaha, R., Badal, T. (2019). Automatic Parallelization of C Code Using OpenMP. In: Somani, A., Ramakrishna, S., Chaudhary, A., Choudhary, C., Agarwal, B. (eds) Emerging Technologies in Computer Engineering: Microservices in Big Data Analytics. ICETCE 2019. Communications in Computer and Information Science, vol 985. Springer, Singapore. https://doi.org/10.1007/978-981-13-8300-7_25