Análisis de Paralelismo

Presentado por:

Jorge Iván Villada Lizarazo

Johan Felipe Marín

Felipe Andrés Bravo Giraldo

Presentado a:

Ramiro Andrés Barrios

Universidad Tecnológica de Pereira

Facultad de Ingenierías

High Performance Computing

Pereira

2019

El computador sobre el cual se corrieron los algoritmos tiene las siguientes características:

System Manufacturer: LENOVO

System Model: 81F4

Processor: Intel(R) Core(TM) i3-8130U CPU @ 2.20GHz (4 CPUs), ~2.2GHz

Memory: 4096MB RAM

Operating System: Linux (Ubuntu 18.04.2 LTS)

------------------------

Disk & DVD/CD-ROM Drives

------------------------

Drive: C:

Free Space: 720.2 GB

Total Space: 786.3 GB

File System: NTFS

Model: Intel Optane+932GBHDD

Drive: D:

Free Space: 9.2 GB

Total Space: 10.5 GB

File System: NTFS

Model: Intel Optane+932GBHDD





















**Gráfica 1**



**Gráfica 2**

**ANÁLISIS**

Comparando los tiempos de ejecución de las multiplicaciones de las matrices Vs el tamaño de la matriz, se puede llegar a varias conclusiones dependiendo el método empleado de solución.

Los tamaños que se utilizaron para las pruebas fueron de: **50, 100,200,300,400,500,600,700,800,900,1000,1200,1400,1600.**

**Secuencial:**

La primera implementación que se hizo fue la multiplicación clásica de matrices, donde se utilizan 3 ciclos **for** y se multiplica cada una de las filas de A con todas las columnas de B. La implementación se hizo utilizando la librería **vector** y la matriz se declaró como un vector de vectores.

Como era de esperarse, los resultados en tiempo para matrices de tamaños considerablemente grandes fueron bastante elevado.



Esto se debe a que no se estaba aprovechando toda la utilidad de la memoria y a nivel de código, la complejidad de 3 ciclos anidados es de N^3 que no es nada buena. La gráfica correspondiente a los resultados se encuentra en la figura 2 y es de color azul oscuro.

**Threads:**

La segunda implementación fue usando Threads (hilos) y con la librería vector. Se hizo un hilo por cada columna de la matriz. Los resultados para matrices pequeñas, en teoría el modo secuencial debería ser más rápido ya que no tiene que hacer el cambio del contexto y se hace todo más rápido ya que para el procesador no sería una tarea tan pesada, pero viendo los resultados, los hilos siempre estuvieron por debajo del tiempo del modo secuencial.

Al hacerlo con hilos se quita un ciclo **for** en la implementación lo que reduce la complejidad un poco más. Si se comparan los resultados con los del secuencial, se nota que están aproximadamente por mitad.

De forma gráfica se puede ver en la **Gráfica 2** de color naranja.



Debido a que los hilos son utilizados para paralelizar el trabajo de la multiplicación, no se debe esperar que se haga una cosa a la vez, sino que al mismo tiempo se están ejecutando varias multiplicaciones de columnas, lo que acelera considerablemente el trabajo.

La aceleración conseguida por esta implementación se calculó dividiendo los tiempos de la implementación secuencial por la implementación paralela (threads) y el resultado fue de **2,208033082X** y se puede ver gráficamente en la **gráfica 1** de color azul oscuro.

Se puede notar que hasta un tamaño de 500 la aceleración fue subiendo y luego se estabilizó y cayó un poco pero después del tamaño 1000 volvió a incrementar y eso se debe a que la implementación secuencial en esos tamaños se demoró más del doble de tiempo que la implementación en paralelo, así que la relación en los tiempos hace que en la gráfica se vea la elevación y por ende hubo más aceleración en esa parte.

**Procesos**

A la hora de trabajar con procesos se debe tener en cuenta que cada proceso (proceso padre) una vez se pone a funcionar crea otro proceso (proceso hijo) y este nuevo proceso al estar en ejecución crea a su vez un nuevo proceso (proceso nieto). Para comparar la implementación por procesos y con hilos, se tomó la decisión de hacerlo también por columnas.

Si no se tiene conciencia de cuántos procesos se generan una vez se ponga a ejecutar el algoritmo, la memoria se puede llenar fácilmente ya que por fórmula se generan **2^n** procesos donde en este caso **n** son la cantidad de columnas de la matriz, así que para una matriz de **3x3** se generan **8** procesos.

Se puede pensar que 8 procesos son poco, pero para matrices de tamaños grandes la cantidad de procesos son muchos, por ejemplo, para una matriz de **1000x1000** la cantidad de procesos serían **1,07150860718626732094842504906e+301.** Toda esta cantidad de procesos saturarían la memoria. Así que en la implementación se tuvo que pensar en una manera de impedir este crecimiento en cantidad de procesos y se llegó a la conclusión de crear un solo proceso por columna y esto se pudo lograr solamente dejando que el proceso padre pudiera tener **n** procesos hijos y que los hijos no pudieran crear más procesos, entonces con esto para una matriz de **3x3** solo habrá **3** procesos en vez de **8.**

Una vez se pone a correr y se sacan los resultados de tiempo, se puede notar que los resultados solo están un poco por debajo del resultado de la implementación secuencial; así que sin importar que se estuviera haciendo de forma paralela los resultados nos fueron nada prometedores; esto se debe a que los procesos una vez se le acaba el tiempo de proceso en el procesador, se crea una copia en memoria de toda la información del proceso y todo eso lo hace muy pesado en cuestión de tiempo y capacidad de memoria.

Cuando se crean las copias del proceso padre, los resultados no se guardan en la misma dirección de memoria, así que se debe tener en cuenta a nivel de código, separar espació de memoria para que todos los procesos que estén corriendo puedan compartir esa memoria.

Esta implementación se hizo con un doble puntero ya que con un vector de vectores es más difícil hacer la separación de memoria ara compartirla.



De forma gráfica se puede ver en la **Gráfica 2** y se ve que está por debajo del secuencial, pero al final lo sobrepasa y eso se debe a un dato bastante alto que apareció en la séptima iteración de la matriz de 1600x1600, aquí el tiempo subió aproximadamente 100000 ms, esto no se sabe qué pudo haber sido, tal vez fue un proceso aparte del computador que se empezó a ejecutar y retrasó el tiempo de ejecución o tal vez fue porque el computador en cierto momento se bloqueó y a la hora de desbloquearlo pudo ocupar tiempo de procesamiento.

La aceleración gráficamente se puede ver en la **Gráfica 1** de color naranja. Se puede notar que hasta un tamaño de 700 se estaba acelerando, y después comenzó a bajar la aceleración drásticamente y se cree que es por la cantidad de procesos que se ejecutan con tamaños superiores de 700x700.

La aceleración que hubo fue de **1,090601228X**

Se puede concluir que la forma de paralelización por procesos no es nada fiable debido a que la aceleración es poca.

Luego de mirar estos dos tipos de paralelización se decidió hacer ciertas cosas para acelerar más el algoritmo.

No se tenía en cuenta al principio el Cache line y llegó a la conclusión que, en vez de hacer la multiplicación de fila por columna, era mucho más rápido si se hacía fila por fila, entonces, a la matriz **B** se le hizo la transpuesta y se multiplicó por filas.

**Secuencial con Matriz Transpuesta**

Se hizo casi la misma implementación de algoritmo secuencial, pero esta vez se sacó la transpuesta de la segunda matriz y se multiplicó fila por fila.



Los resultados fueron mucho mejores a la implementación secuencial inicial, todo por tener en cuenta el cache line; cabe aclarar que acá no se hizo ningún tipo de paralelización, solo se tuvo en cuenta que a nivel de memoria se podía acelerar más el algoritmo.

También se usó la librería vector en esta implementación.

En la **Gráfica 2** se puede ver de color amarillo cómo se comportan los tiempos y ve la mejoría que hubo en comparación a la implementación secuencial normal.

En la **Gráfica 1** se puede ver de color gris la aceleración que hubo, es poca, pero hace notar que con solo pensar en la memoria y saber trabajar con ella se puede mejorar el tiempo considerablemente.

La aceleración fue de **1,455991527X,** mejor que la aceleración por procesos.

Después de un tamaño de 1000x1000 se ve que la aceleración incrementa y se debe al cache line, ya que no se necesita mover entre las diferentes memorias para traer cada uno de los valores de las columnas, sino que todo o gran parte se encuentran en las filas así que se necesitan hacer menos saltos para traer todos los valores, lo que ayuda a la aceleración.

**Threads con Matriz Transpuesta**

Al igual que con el secuencial con matriz transpuesta, se hizo exactamente lo mismo, se sacó la transpuesta de la matriz **B** y se multiplicó por filas. Como era de esperarse, al transponer la matriz se aceleraría y usando hilos mucho más.

Al ver los resultados se ve que mermaron mucho en comparación de todas las demás implementaciones, incluso mucho más rápida que uso usando hilos.



En la **Gráfica 2** de color verde se puede ver cómo mejoró y que incluso está por debajo de la otra implementación con hilos.

En la **Gráfica 1** la aceleración se ve de color amarillo y se puede notar que incrementó mucho en comparación a todas, después del tamaño de 1000x1000 también ocurre lo mismo que con el secuencial, por cuestiones del cache line la aceleración incrementa.

La aceleración obtenida en este caso fue de **2,838632068X,** aunque en la gráfica se puede notar que su máxima aceleración fue de **3,1X.**

Se sabe que usando arreglos se podía acelerar mucho más ya que la librería vector es un poco más pesada y los arreglos ayudan más con el trabajo en cuestiones de memoria, pero en este caso solo se trabajó con la librería vector a excepción de la implementación de procesos.