Informe de la tercera práctica

Juan Francisco Mier Montoto, UO283319

Programación Concurrente y Paralela, 10/11/2022

GIITIN, EPI Gijón

Índice

0. Introducción

Este trabajo se ha desarrollado a lo largo de varias semanas, desde la primera práctica hasta el día de la entrega, pasando por múltiples iteraciones de código, repetidos análisis de rendimiento y deducciones teóricas sobre el funcionamiento indicado. A continuación, se procede a detallar los diferentes análisis teóricos requeridos, resultados de las mediciones y discusiones sobre dichas mediciones.

Como nota adicional, se han utilizado las repeticiones originales (10, 8 y 6 en ese orden) ante la falta de otras indicaciones en la guía de esta práctica.

1. Análisis teórico de la primera práctica

1.1. Rendimiento teórico (TPP_{dp}) de los nodos de cálculo

Para obtener el rendimiento teórico de los nodos que se van a utilizar a lo largo de ambas prácticas, se aplica la fórmula vista en las diapositivas de teoría de la asignatura:

$$TPP_{dp} = chasis \times nodos_{chasis} \times sockets_{nodo} \times cores_{socket} \times clock_{GHz} \times \frac{nº flop}{ciclo}$$

Al rellenar la fórmula con la información de las máquinas, obtenidas a través del comando *lshw* y verificada gracias a la tabla de las diapositivas de teoría, el resultado es el siguiente:

- **ColaI3:** procesador I3-2100, 49.6 Gflop
- **ColaXeon:** procesador Xeon E5620, 153.6 Gflop
- **ColaGPU:** procesador Ryzen 7 3700x, 460.8 Gflop (al funcionar con el boost de frecuencia)

1.2. Tiempo por flop teórico de los nodos de cálculo

El tiempo por flop teórico (t_c) es la inversa del rendimiento teórico de cada máquina. Para obtener el tiempo por flop teórico, hay además que multiplicar por el número de sockets y de cores de cada máquina.

Procesador	**t_c secuencial**	**t_c paralelo**
Intel I3-2100	$\frac{2}{TPP_{i3}} = 4.0323e-11s$	$\frac{1}{TPP_{i3}} = 2.01613e-11s$
Xeon E5620	$\frac{8}{TPP_{Xeon}} = 5.2083e-11s$	$\frac{1}{TPP_{Xeon}} = 6.5104e-12s$
Ryzen 7 3700x	$\frac{8}{TPP_{Ryzen}} = 1.7361e-12s$	$\frac{1}{TPP_{Ryzen}} = 2.1701e-12s$

1.3. Complejidad temporal y espacial de $C = \beta C + \alpha AB$

Siendo el tamaño de las matrices A, B y C: "mxk", "kxn" y "mxn" respectivamente, y siguiendo el algoritmo desarrollado en C, se obtienen las siguientes complejidades:

- **Complejidad temporal:** *mn(2k + 3),* suponiendo que se utilizan 3 flops para asignar (una suma y dos multiplicaciones) y 2 flops para calcular cada producto y sumarlo.
- **Complejidad espacial:** se suma el tamaño de las tres gráficas, es decir: *mk + kn + mn*.

1.4. Complejidad temporal y espacial del subproblema transpuesta

El subproblema "transpuesta" recorre toda la matriz y la almacena en una matriz auxiliar, por lo que las complejidades son las siguientes:

- **Complejidad temporal:** mn
- **Complejidad espacial:** 2mn

2. Análisis teórico de la tercera práctica

2.1. Complejidad temporal y espacial de las soluciones propuestas

Hay que tener en cuenta que a todas las complejidades hay que sumar además la respectiva complejidad del subproblema de la transpuesta, puesto que se realiza dicha operación al comienzo de cada función para conseguir la mejora de rendimiento que esto conlleva.

Complejidad temporal

A la hora de calcular la complejidad temporal de las dos primeras funciones, *MyDGEMM* y *MyDGEMMT*, simplemente hay que dividir entre el número de procesadores con los que se cuenta.

Para *MyDGEMMB,* es diferente, al tratar las matrices por bloques y reutilizar *MyDGEMM* para calcular los elementos dentro de los bloques. Además, hay que tener en cuenta que el tamaño del bloque es equivalente a la dimensión entre 10 (indiferente cuál, ya que se tratan solo matrices cuadradas).

- **Complejidad MyDGEMM y MyDGEMMT:** $\frac{mn}{p}(2k+3)$
- **Complejidad MyDGEMMB:** $\frac{mnk}{blk^3}\frac{blk^2}{p}(2blk+3) = \frac{mnk}{blk}p(2blk+3) = 10\frac{n}{p}\left(2\frac{n}{10}+3\right) = \frac{n^2}{p} + 30\frac{n}{p}$

Complejidad espacial

La complejidad espacial continúa siendo la misma para todos los problemas.

2.2. Speedup teórico para cada nodo

El speedup teórico para cada máquina es la división del tiempo por flop teórico en versión secuencial y paralela para cada máquina, que a su vez es la inversa del número de cores multiplicado por el número de sockets de cada uno, de modo que en el numerador siempre hay 1 (en el secuencial solo se utiliza un solo núcleo). Por ello:

- **ColaI3:** $\frac{1}{2} = 0.5$(número de cores: 2, número de sockets: 1)
- **ColaXeon:** $\frac{1}{8} = 0.125$(número de cores: 4, número de sockets: 2)
- **ColaGPU:** $\frac{1}{8} = 0.125$(número de cores: 8, número de sockets: 1)

3. Tablas de rendimiento (resolución del anexo)

Se presupone que lo que varía entre versiones secuenciales y paralelas es la cantidad de información a procesar, es decir, el número de flops, debido a la diferencia entre complejidades temporales ya definidas. Por lo tanto, el tiempo por flop se considera igual, puesto que, de lo contrario, se dividiría varias veces entre el número de núcleos disponibles en cada nodo.

También se ha eliminado la columna de t_c, puesto que contiene información repetida. El tiempo por nodo ya se ha definido en un punto anterior de este informe. Esto tiene relación con el párrafo anterior.

3.0. Python

Talla	I3	Xeon	Ryzen
1000x1001x999	0,978929043	2,40721035	0,367322683
2000x2001x1999	5,873624086	14,70052433	2,152242899
3000x3001x2999	14,60516167	36,62869835	5,326950312

3.1. MyDGEMM

i3 secuencial						
m	n	k	nº flop	Teórico	Empírico O0	Empírico O3
1000	1001	999	2,00E+09	8,08E-02	57,2686527	5,999754906
2000	2001	1999	1,60E+10	6,46E-01	417,583189	36,06150413
3000	3001	2999	5,40E+10	2,18E+00	954,555811	96,01594019

i3 paralelo						
m	n	k	nº flop	Teórico	Empírico O0	Empírico O3
1000	1001	999	1,00E+09	4,04E-02	29,4396508	4,86014533
2000	2001	1999	8,01E+09	3,23E-01	212,220077	26,054306
3000	3001	2999	2,70E+10	1,09E+00	484,11513	73,3114617

Xeon secuencial						
m	n	k	nº flop	Teórico	Empírico O0	Empírico O3
1000	1001	999	2,00E+09	1,04E-01	109,407039	6,799163818
2000	2001	1999	1,60E+10	8,34E-01	769,471961	92,02682328
3000	3001	2999	5,40E+10	2,81E+00	1842,49228	227,2121468

Xeon paralelo						
m	n	k	nº flop	Teórico	Empírico O0	Empírico O3
1000	1001	999	2,50E+08	1,30E-02	14,9531775	1,76423478
2000	2001	1999	2,00E+09	1,04E-01	101,127729	18,9001794
3000	3001	2999	6,75E+09	3,52E-01	238,53827	43,0494165

Ryzen secuencial						
m	n	k	nº flop	Teórico	Empírico O0	Empírico O3
1000	1001	999	2,00E+09	3,48E-02	29,2678385	1,452595711
2000	2001	1999	1,60E+10	2,78E-01	195,361936	23,51412225
3000	3001	2999	5,40E+10	9,38E-01	600,31209	59,0908978

Ryzen paralelo						
m	n	k	nº flop	Teórico	Empírico O0	Empírico O3
1000	1001	999	2,50E+08	4,35E-03	4,43943548	0,50391293
2000	2001	1999	2,00E+09	3,47E-02	27,0003476	6,51340437
3000	3001	2999	6,75E+09	1,17E-01	81,2216575	15,7132413

3.2. MyDGEMMT

m	n	k	nº flop	Teórico	Empírico O0	Empírico O3
1000	1001	999	2,00E+09	3,48E-02	9,04867959	0,5090487
2000	2001	1999	1,60E+10	2,78E-01	59,63180661	6,39097929
3000	3001	2999	5,40E+10	9,38E-01	170,6219535	16,99869537

3.3. MyDGEMMB

Ryzen secuencial						
m	n	k	nº flop	Teórico	Empírico O0	Empírico O3
1000	1000	1000	1,03E+06	1,79E-05	30,45365787	1,784814835
2000	2000	2000	4,06E+06	7,05E-05	191,5909388	9,9225142
3000	3000	3000	9,09E+06	1,58E-04	491,7733474	29,73748899

Ryzen paralelo						
m	n	k	nº flop	Teórico	Empírico O0	Empírico O3
1000	1000	1000	1,29E+05	2,24E-06	4,83173275	0,70913196
2000	2000	2000	5,08E+05	8,81E-06	27,7963927	2,37241006
3000	3000	3000	1,14E+06	1,97E-05	67,7686226	5,35058761

4. Discusión de los resultados obtenidos

Una vez realizados todos los experimentos necesarios para completar las tablas anteriores y las preguntas de índole teórica, se pueden analizar varios aspectos a modo de conclusión y discusión de la práctica.

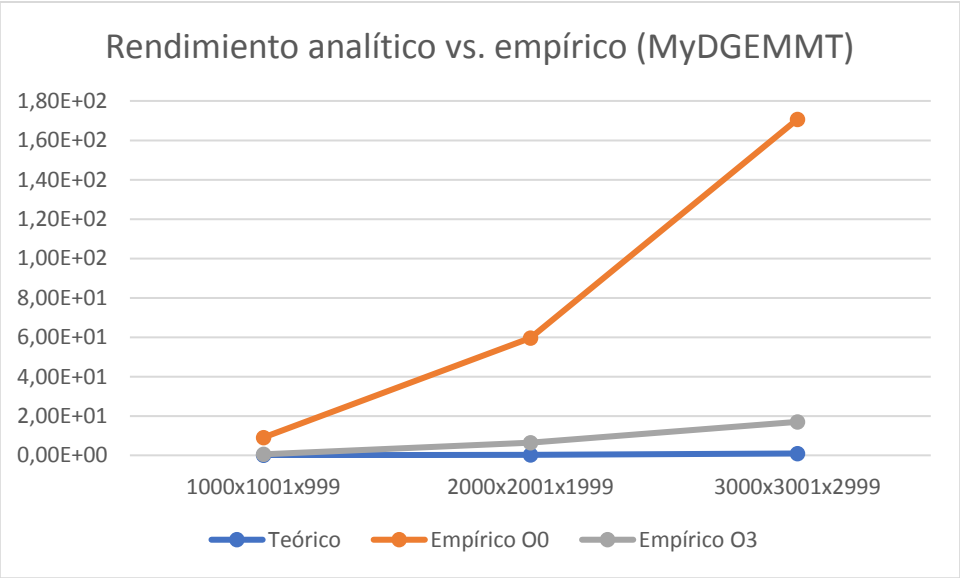
4.1. Diferencias entre tiempos analíticos y resultados empíricos

Esto es lo primero que sorprende al analizar los datos obtenidos: la enorme diferencia entre el cálculo teórico de los tiempos de ejecución y los tiempos reales obtenidos. Obviamente, los cálculos teóricos son muy sencillos y tan solo sirven como orientación de las mejoras que se pueden obtener entre utilizar diferentes métodos y diferentes métodos de ejecución (secuencial vs. paralelo).

Además de la imprecisión de los cálculos teóricos, hay que remarcar la poca precisión de los datos obtenidos: pese a realizar muchas pruebas, es inevitable que el código funcione de maneras distintas, en este caso de manera mucho más lenta. Solo así se explicarían los casi 40 minutos de ejecución en secuencial en las máquinas Xeon de las pruebas de *MyDGEMM*, resultados repetidos para eliminar errores aleatorios.

Gracias a la ejecución de la versión de Python se puede corroborar además que estos errores no solo son debidos a una posible mala implementación, o a una ejecución de manera incorrecta en las colas, puesto que la librería *numpy* tampoco se acerca a los valores analíticos pese a contar con numerosas optimizaciones y formar parte vital de un lenguaje tan importante.

La siguiente gráfica denota la diferencia entre el rendimiento esperado y el obtenido al analizar las mediciones de la función que utiliza tareas para paralelizar (ejecutado en una máquina Ryzen):

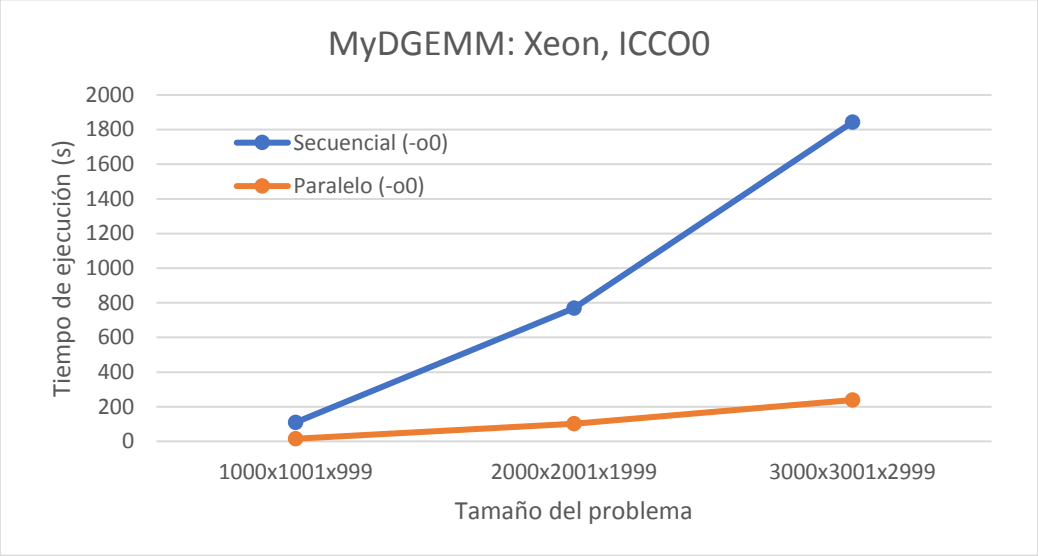


4.2. Diferencias entre nodos de ejecución

Las diferencias entre los nodos de ejecución son, probablemente, lo más coherente y fácilmente verificable de estos resultados.

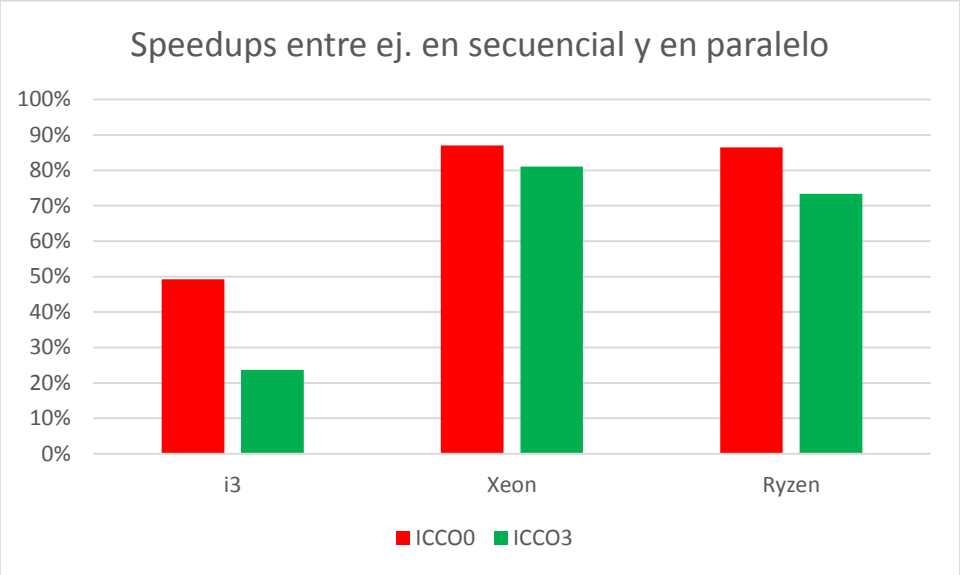
- Las máquinas I3 son lentas en general, tanto en secuencial como en paralelo, puesto que cuentan con una memoria caché media y tan solo dos núcleos, salvado solo por la frecuencia de 3.1GHz, lo que los hace más rápidos que la máquina Xeon en modo secuencial.

- La máquina Xeon es increíblemente lenta en modo secuencial, tanto que una sola medición con la talla más grande que se pedía analizar tardaba cerca de 30 minutos. Sin embargo, y gracias a contar con 2 sockets y 4 procesadores, el speedup entre la ejecución en secuencial y en paralelo es la mejor de todas las máquinas. Con un speedup del **87%** entre dichas ejecuciones (utilizando el compilador sin mejoras) es claramente mejor que el I3 (≈**50%**) y ligeramente mejor que el Ryzen (**86%**) pese a tener la misma cantidad de núcleos y estar separado en dos sockets.

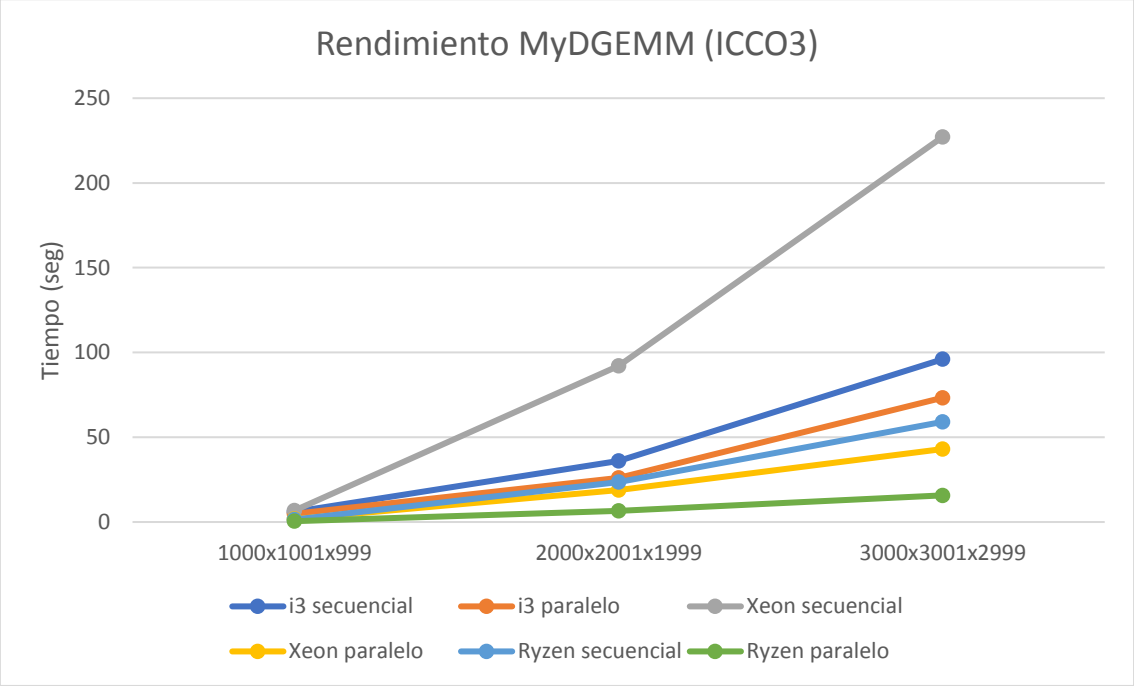


- Las máquinas Ryzen son obviamente mejores en todas las ejecuciones, tanto en secuencial como en paralelo. Nada sorprendente ni a destacar, puesto que en el papel es claramente mejor que el resto de alternativas.

Todas las máquinas obtienen las aceleraciones esperadas al cambiar de secuencial a paralelo: el I3 mejora casi un 50% (pasa de 1 a 2 núcleos) y las otras dos máquinas mejoran muy cerca del ideal 87.5% (pasan de 1 a 8 núcleos). El gráfico obtenido es sorprendentemente cercano a lo esperado:



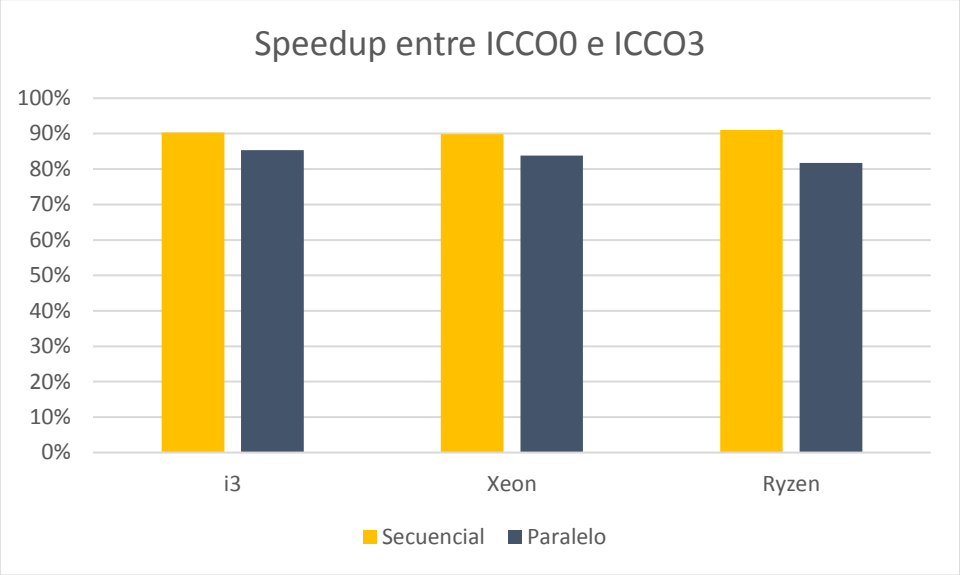
Dicho todo esto, esta es la gráfica final de comparación entre los nodos, utilizando el compilador con mejoras de optimización:



Esta gráfica sirve solo como comparación a simple vista del rendimiento de cada nodo. No se incluyen otros datos, como el rendimiento en Python ni los tiempos teóricos. Como punto a destacar, la máquina Ryzen, incluso en secuencial, es tan solo superada por la máquina Xeon ejecutando en paralelo.

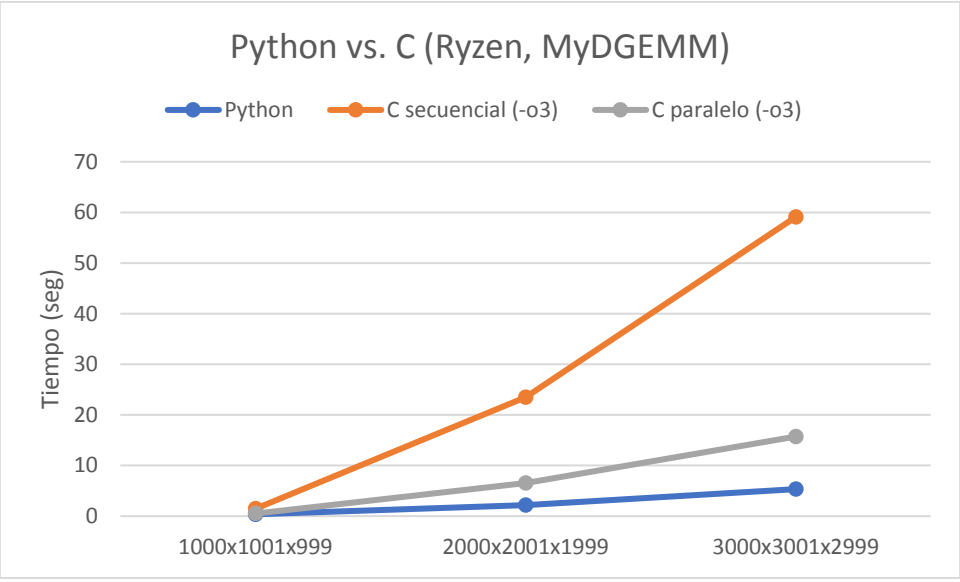
4.3. Diferencias entre opciones de compilación y lenguajes

No hace falta mirar muchos resultados para darse cuenta de que la mayor diferencia de tiempo entre funciones y modos de ejecución es gracias a las optimizaciones del compilador.



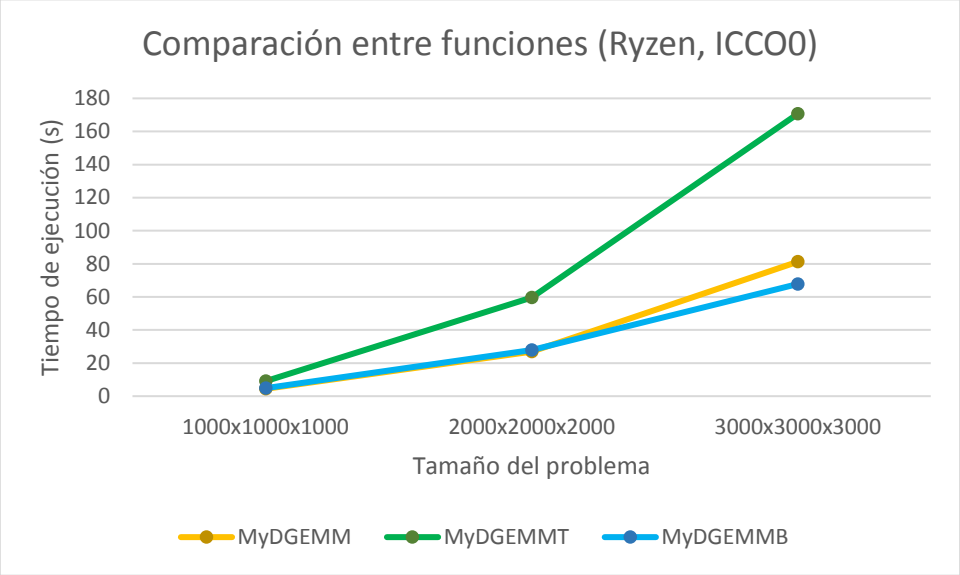
Como se puede apreciar en la gráfica, compilar utilizando *-o3* supone, de media, un **85%** de mejora con respecto al rendimiento original.

Pyhton, además de ser el elemento que define si las funciones están bien implementadas o no (gracias a la diferencia entre resultados), es el tiempo “a batir”, puesto que se opera utilizando la librería *numpy*, que cuenta con numerosas optimizaciones y está bajo el escrutinio de múltiples expertos en el mundo puesto que se trata de una de las librerías de Python más utilizadas y además es *open-soruce*. Es debido a esto que los tiempos obtenidos están muy, muy lejos de llegar a alcanzar a dicha librería, pese que los cálculos son con las mismas tallas y el mismo número de repeticiones.



4.4. Diferencias entre funciones (normal, por tareas y por bloques)

La manera más fácil de analizar el rendimiento de las diferentes funciones es mediante una gráfica:



Es evidente que los resultados no son los esperados: *MyDGEMMT*, que es la versión pararelizada mediante tasks, debería ser similar al resto de funciones. Sin embargo, para tallas muy grandes, los tiempos de ejecución se disparan. Esto significa que la función cuyo funcionamiento se basa en el lanzamiento de tareas desde un solo hilo al resto es más lento que la función con un simple *#pragma parallel for*.

La otra función, *MyDGEMMB*, debería aprovechar mejor la localidad espacial y hacer mejor uso de la memoria caché, pero obviamente no funciona del todo o no supone una mejora clara del rendimiento con respecto a la función original. Como nota adicional, se recuerda que, en la implementación dada, esta función hace uso de la función base *MyDGEMM* para realizar las operaciones entre bloques.

5. Anexo (notas sobre la práctica y otras opiniones)

Es evidente que la implementación actual del código cuenta con ineficiencias. Mismamente, el código de la función *MyDGEMMB* funciona más lento al implementar paralelismo mediante OpenMP y tareas. Además, las tallas más grandes tardan mucho en ejecutarse en los nodos disponibles, especialmente en sus formas secuenciales.

A parte de esto, se ha tenido que escoger una metodología para obtener los tiempos teóricos, que reside en utilizar las expresiones de complejidad temporal en cada función e ignorar las diferencias en el tiempo por flop al utilizar un solo núcleo frente a utilizar varios, puesto que se obtendrían resultados divididos dos veces por el número de estos, algo que a priori es incorrecto.

Le he dedicado especial mimo a partes del trabajo que no se van a valorar ni a puntuar: el código Python y scripts de ayuda en Shell. Quizás esto haya sido una mala idea, pero me han ayudado a comprenderlo todo mucho mejor y, de paso, expandir mi conocimiento en varios lenguajes de programación a la vez. Además, las hojas de Excel también han sido muy prácticas y estoy muy contento con su resultado.

La ejecución en las colas de la asignatura ha sido ligeramente agridulce. Al haber una sola máquina Xeon, se han generado colas de horas al ejecutar scripts en dicho nodo, incluso varios días antes de la entrega del trabajo. Además, la manera de enviar y recibir información de y desde los servidores es muy incómoda. No me cabe duda de que están muy bien planteados técnicamente, pero no deja de ser extraño.

Han ocurridos fallos serios durante el desarrollo del código, pero ninguno tan serio como el aparente fallo de implementación de la función transpuesta. Se ha tenido que implementar una versión que devuelve la función transpuesta en forma de matriz, en vez de actualizar la matriz utilizando punteros y luego liberando la matriz auxiliar. Personalmente no tengo ni la más remota idea de qué podría estar mal, pero quizás esta función es uno de los motivos del mal rendimiento del programa. Aun así, la función solo se llama una sola vez al principio de cada función (en caso de que se le pase el tipo 2), por lo que no debería impactar mucho.