Práctica 3

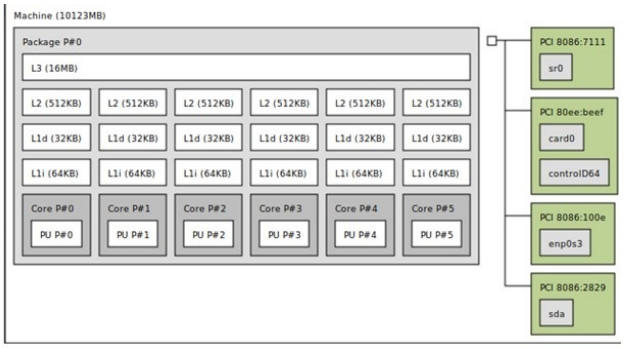
Javier Delgado del Cerro y Javier López Cano

Ejercicio 0

Al analizar con `getconf -a | grep -i cache` en la terminal las cachés de los ordenadores del laboratorio, vemos que estos tienen una cache multinivel de 3 niveles con separación entre instrucciones y datos en el nivel uno y con los niveles 2 y 3 unificados. La caché de nivel 1 es una caché de 8 vias, la de nivel 2 de 4 vias y la nivel 3 de 12 vias y todas ellas tienen un tamaño de linea de 64 bits. Por tanto se trata de memorias set-associative, la primera de 8, la segunda de 4 y la tercera de 12 vias. Tambien podemos observar que las cachés nivel 1 tanto de instrucciones como de datos tienen un tamaño de 32KB, la de nivel 2 un tamaño de 256KB y la de nivel 3 dispone de 6MB de capacidad.

En el caso de el equipo personal, segun los datos que nos proporciona `lstopo` se trata de un equipo con 6 cores, cada uno de los cuales tiene 2 niveles de caché propios y un tercer nivel que es común a todos ellos, por tanto dispone de una cache multinivel con 3 niveles. Las caches de nivel 1 tienen separación entre datos e instrucciones, la de instrucciones dispone de 64KB mientras que la de datos tan solo tiene 32 KB, el nivel 2 no tiene separación entre instrucciones y datos y tiene 512KB para cada core. Por ultimo la caché de nivel 3 que es común a todos los cores tampoco tiene separación entre datos e instrucciones y tiene 16MB. Si ademas de esto lo analizamos con `getconf -a | grep -i cache` podemos observar que las caches son de tipo set-associative, las de nivel 1 de instrucciones con 4 vias, las de datos con 8 vias, las de nivel 2 tambien con 8 vias, y la cache común de nivel 3 con 16 vias. Todas ellas con 64bits de tamaño de linea.

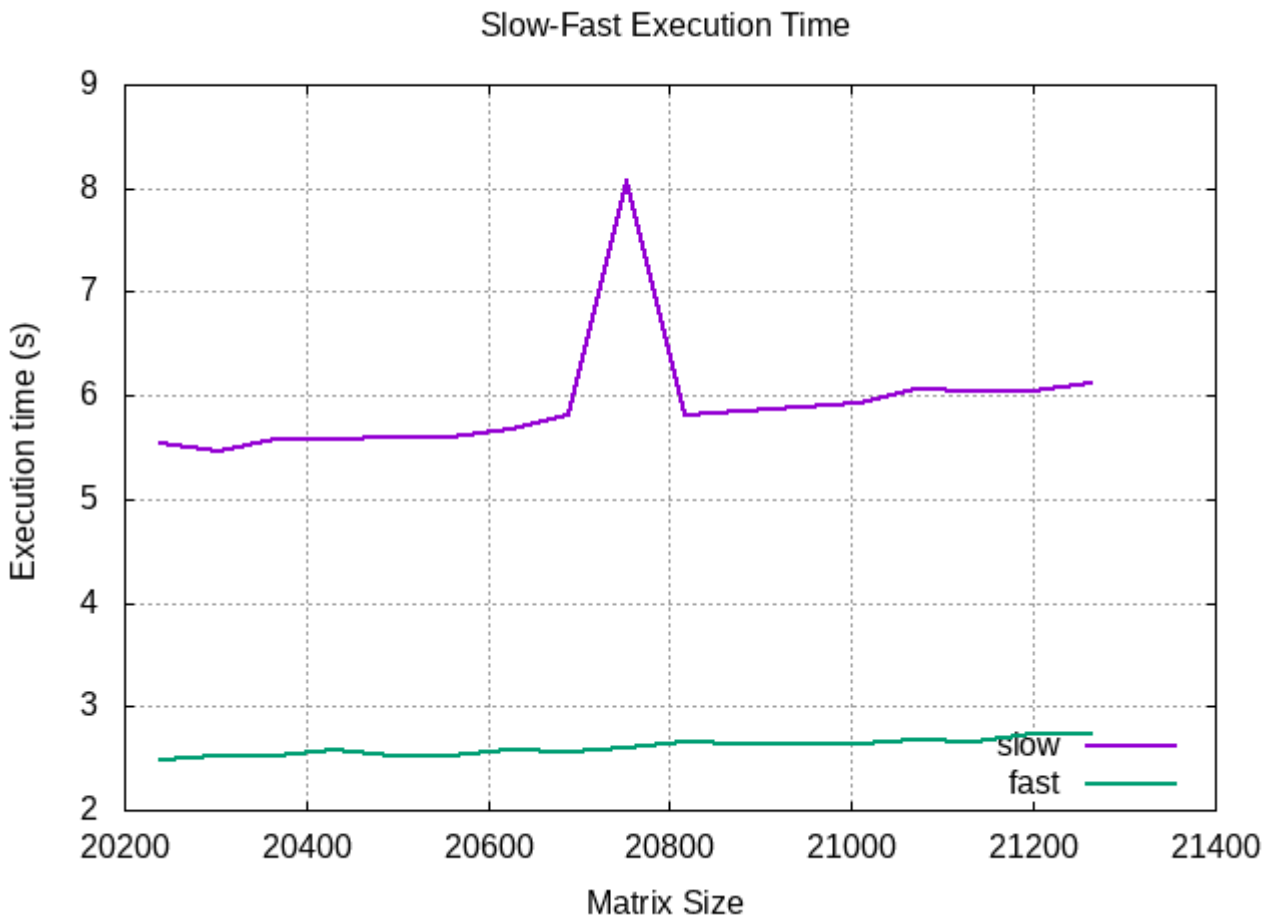
```
javi@javivm1:~/Escritorio$ getconf -a | grep -i cache
LEVEL1_ICACHE_SIZE                 65536
LEVEL1_ICACHE_ASSOC                4
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  524288
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  16777216
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```



Ejercicio 1

Dado que durante la ejecución del programa, en el ordenador corren múltiples procesos de forma simultánea, los tiempos de ejecución de ambos programas para una misma N varian. Así, al tomar la media de varias ejecuciones para la misma N, de forma desordenada, para evitar patrones en el acceso a la memoria caché, conseguimos tener una estimación de el tiempo que tarda en ejecutarse.

Para obtener los datos sin tener el cuenta la mejora del rendimiento de la memoria caché debida a la repetición de patrones de acceso, hemos hecho las ejecuciones de los programas de forma desordenada, ejecutando en primer lugar *slow* con todas las *N*, y posteriormente *fast* con todas las *N*. Estos resultados los hemos almacenado en dos arrays, de forma que repetimos esta prueba 20 veces, guardando la suma de tiempos para cada uno de los dos programas y cada N en dichos arrays, y calculamos posteriormente la media para guardarla en el fichero *slow_fast_time.dat* y luego generar el gráfico *slow_fast_time.png*.

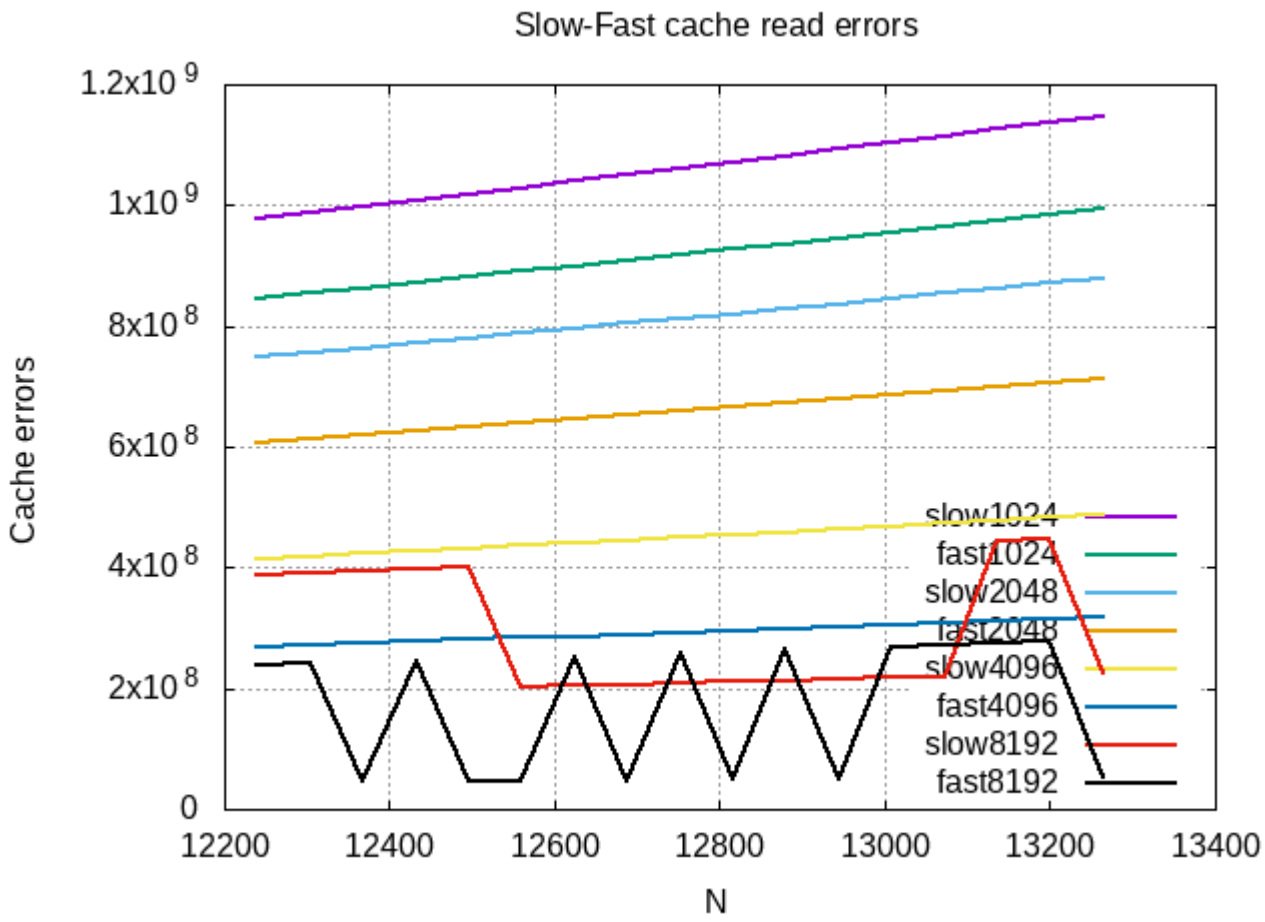


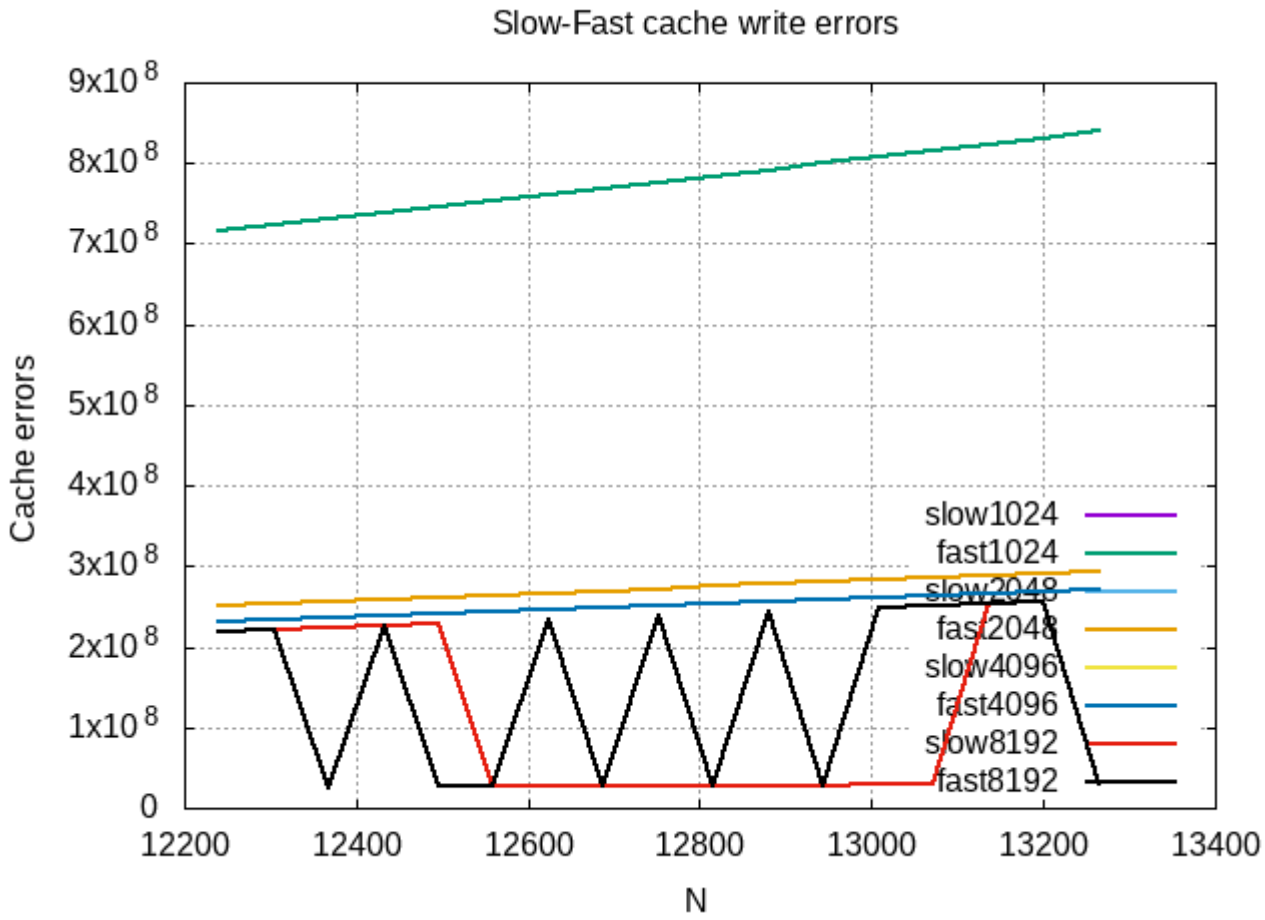
Se ve claramente como *slow* es considerablemente más lento que *fast*, y además, al aumentar el tamaño de las matrices también lo hace la separación entre ambas gráficas. Esto se debe a que en *slow*, la suma se hace por columnas, mientras que en *fast* se hace por filas. De esta forma, como en C las filas se almacenan de forma contigua, al cargar un elemento de una fila se cargan los siguientes también, pues se leen bloques enteros, mientras que al cargar un elemento de una columna es posible que el siguiente se tenga que cargar también, pues están en distintas filas. Por tanto, el programa *slow* requiere de muchos más accesos a memoria, y estos accesos aumentan al incrementarse el tamaño de la matriz (pues es menos probable que en un bloque quepa una fila entera para poder leer un elemento y su siguiente).

En el ordenador usado para la práctica, aparece un pico en la ejecución de *slow* para N=20752, y este pico se puede aprecir también al ejecutar *slow* para dicha N de forma aislada. Lo hemos consultado con el profesor de teoria y parece que es normal, y que por características de la memoria caché, este tamaño causa un número muy grande de fallos, aumentando así el tiempo de ejecución.

Ejercicio 2

Para llevar acabo este ejercicio, hemos creado el script `slow_fast_cache.sh`, que genera los ficheros y gráficas provenientes de ejecutar con `cachegrind` los programas *slow* y *fast*.


Slow-Fast cache read errors
Cache errors
1.2x10 9
1x10 9
8x10 8
6x10 8
4x10 8
2x10 8
0
12200
12400
12600
12800
13000
13200
13400
N
slow1024
fast1024
slow2048
fast2048
slow4096
fast4096
slow8192
fast8192


Slow-Fast cache write errors
Cache errors
9x10 8
8x10 8
7x10 8
6x10 8
5x10 8
4x10 8
3x10 8
2x10 8
1x10 8
0
12200
12400
12600
12800
13000
13200
13400
N
slow1024
fast1024
slow2048
fast2048
slow4096
fast4096
slow8192
fast8192

Al observar estas gráficas y ficheros obtenidos y ver los fallos de caché producidos se puede ver que dependiendo del tamaño de la caché simulada, y de si se ejecuta el *slow* o el *fast* se producen variaciones en los fallos de caché producidos. En primer lugar, si nos fijamos en la gráfica de los fallos de lectura podemos ver de forma clara que, tanto para el programa *slow* como para el programa *fast* se cumple que, cuanto mayor es el tamaño de la caché simulada, menor es el número de fallos de lectura que se producen, esto se debe a que cuanto mayor sea el tamaño de la caché simulada, mayor cantidad de datos se pueden almacenar en ella y por tanto no será necesario realizar tantos accesos a otras memorias para encontrar los datos que el programa necesita pues la mayoría estarán en esta caché. Cabe destacar del mismo modo que, en general se observa que cuanto mayor es el N mayor es el número de fallos de lectura (si mantenemos el tamaño de la caché fijo), esto es razonable ya que el N representa el tamaño de las matrices que se calculan en el programa y, cuanto mayor sea este tamaño, mayor cantidad de datos necesitará el programa para ejecutarse, al igual tamaño de caché, cuantos mas datos sean necesarios, más veces habra que acceder a memorias distintas a la caché puesto que habrá mas datos fuera de esta. Sin embargo podemos observar un caso especial en el caso de las gráficas que representan a *slow* y *fast* con cachés de tamaño 8192 bytes, en estos casos se puede observar que se poducen numerosos picos a lo largo de la gráfica, esto puede deberse a que, con una caché tan grande, practicamente todos los datos que necesita el programa se encuentran en la caché y por tanto, en general, el número de fallos es muy próximo a cero, y, en las ejecuciones en las que se requiera acceder a la memoria principal para obtener un dato concreto, la gráfica genera un pico pues la diferencia será muy notable. Si en esta misma gráfica (manteniendo fijo el tamaño de la caché) comparamos la tendencia de los programas *slow* y *fast*, observamos que, por como están programados estos, en general *fast* tiene un menor número de fallos de lectura que *slow*.

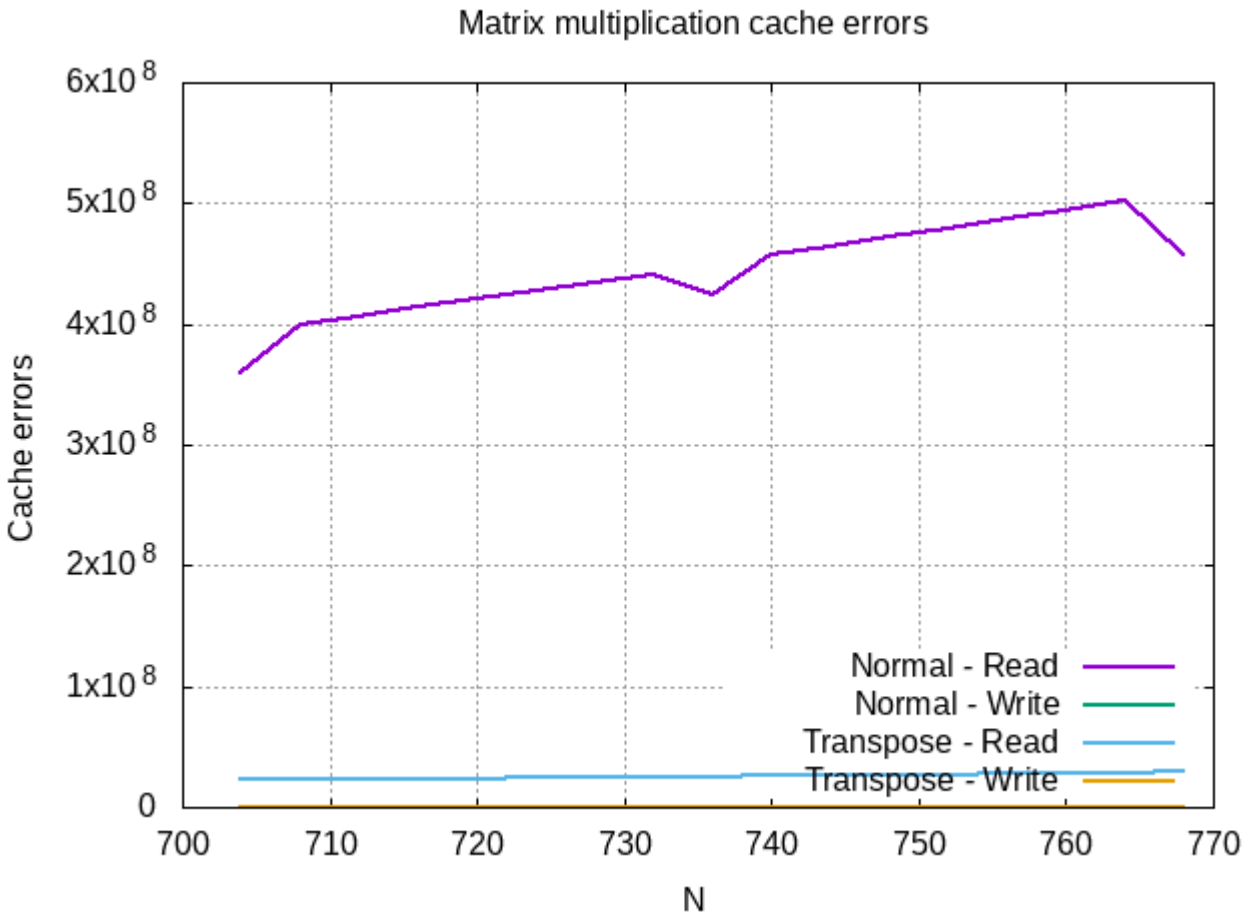
Si nos fijamos en la otra gráfica otenida, que representa los fallos de escritura en caché, observamos un fenómeno parecido, cuya explicación es análoga a la del caso de los fallos de lectura. En este caso, sin embargo, se observa una diferencia a la hora de comparar (manteniendo constante el tamaño de caché) el número de fallos de escritura en los programas *slow* y *fast*, que en este caso son iguales. Esto se debe a que, a pesar de ue ambos programas calculan las matrices de formas distintas y, por lo tanto, requieren una cantidad diferente de datos (por lo que los fallos de lectura si varían), los datos que deben escribir son los de la matriz resultante, luego la cantidad de datos que deben escribir son los mismos, por lo que la cantidad de fallos de escritura son equivalentes en ambos programas.

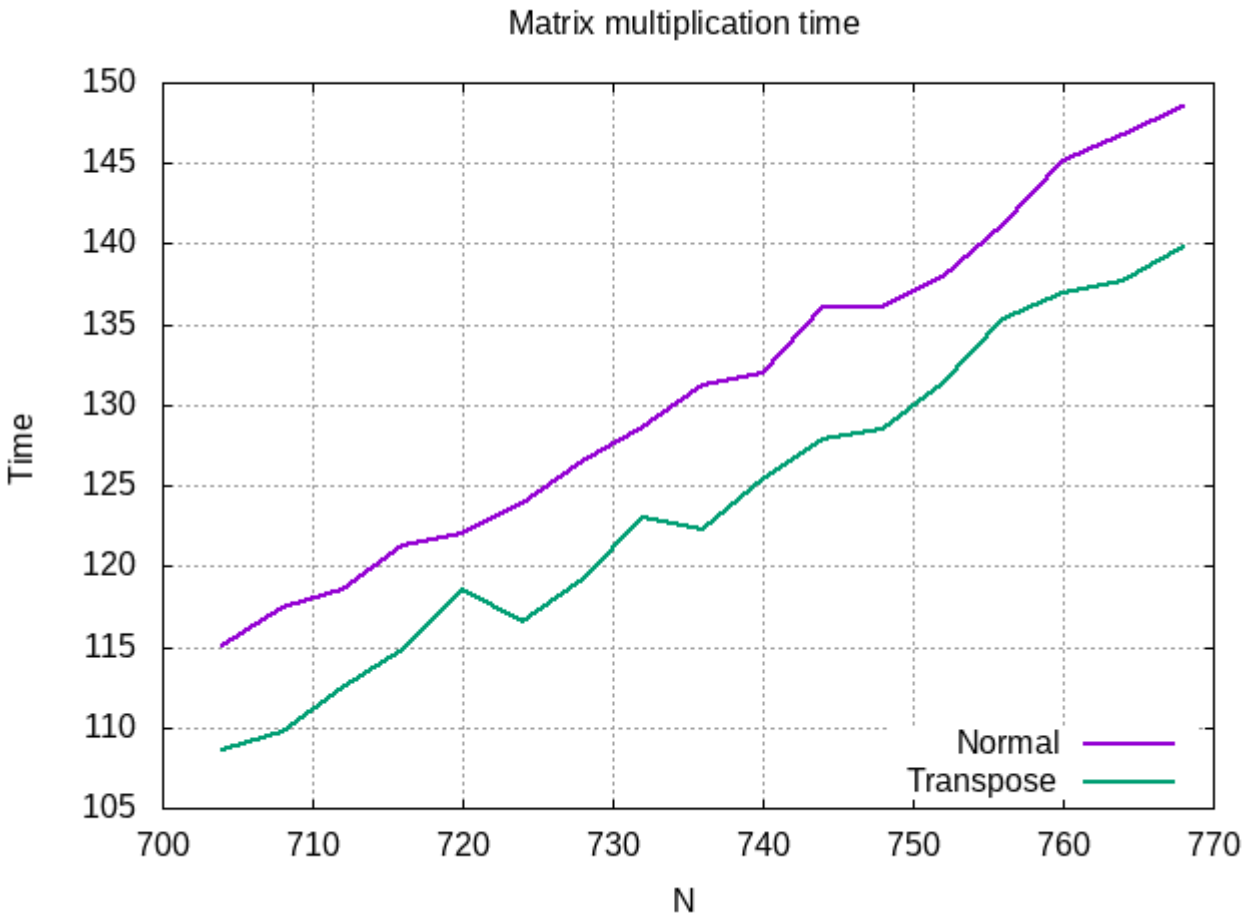
Ejercicio 3

Una vez programamos la multiplicacion de matrices normal, y la traspuesta en los archivos *matrix_mult.c* y *matrix_mult_trans.c* respectivamente, haciendo unas pequeñas modificaciones al script del ejercicio anterior, conseguimos un script que mide el tiempo y los fallos de caché (usando la caché por defecto del ordenador, que está especificada en el ejercicio 1) de cada uno de los dos tipos de multiplicación, para las *N* especificadas en el enunciado. Este escript es `mult.sh`.

Cabe mencionar, eso sí, que hemos tenido que cambiar los valores N para los cuales se ejecuta el script para obtener tiempos de ejecución relativamente razonables, pues con los indicados en el enunciado tardaba más de tres horas por cada una de las N. Además, dado que de por sí el script requiere bastante tiempo de ejecución, hemos decidido medir el tiempo que tarda el programa una única vez para cada N, y en su ejecución con valgrind, y no en una independiente, pues aunque hay variaciones, estas son iguales en ambas multiplicaciones y por tanto no afectan a la comparación de las gráficas. Aún así, la ejecución ha llevado más de dos horas.

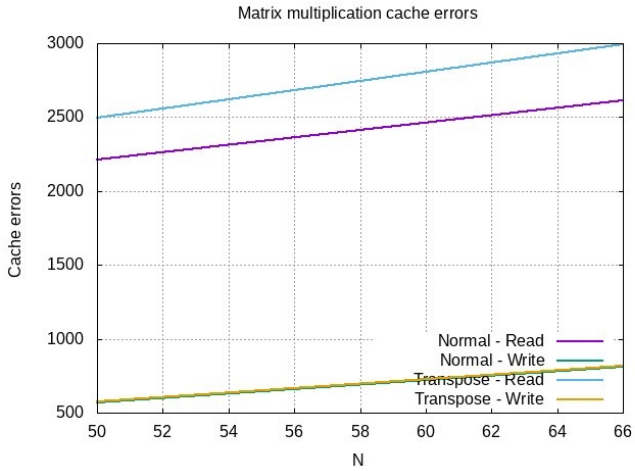
Así, guardamos los datos en el fichero *mult.dat* con el formato pedido, y obtenemos las dos gráficas siguientes:

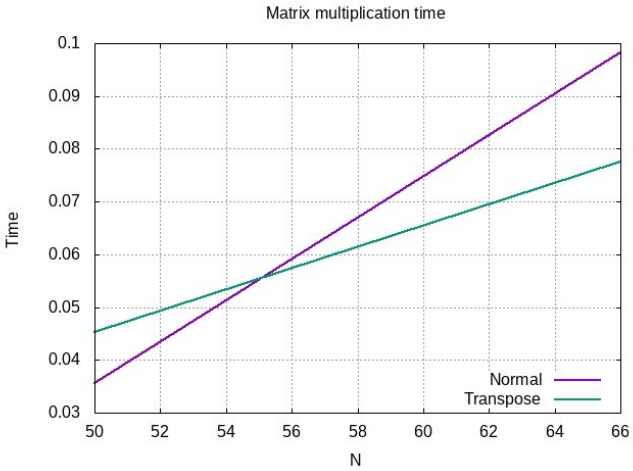

Matrix multiplication cache errors
Cache errors
6x10^8
5x10^8
4x10^8
3x10^8
2x10^8
1x10^8
0
700
710
720
730
740
750
760
770
N
Normal - Read
Normal - Write
Transpose - Read
Transpose - Write


Matrix multiplication time
Time
150
145
140
135
130
125
120
115
110
105
700
710
720
730
740
750
760
770
N
Normal
Transpose

Se puede ver perfectamente como tanto el tiempo como los fallos de caché aumentan al incrementar el tamaño de la N, sin embargo, ambos parámetros crecen con mayor velocidad en la multiplicación normal de matrices. Esto se debe a que, como hemos explicado en el ejercicio 1, las matrices se almacenan por filas dentro de la caché, y cómo la multiplicación normal de matrices accede a una matriz por filas y a la otra por columnas, mientras que la multiplicación traspuesa solo accede a las filas, esta última produce muchísimos menos fallos de caché (unas dieciseis veces menos).

Se podría intentar discutir sin embargo que en la trasposición también hay que cargar bloques, pero esta trasposición solo requiere N^2 operaciones, mientras que la multiplicación requiere N^3, con lo que en cuanto la matriz es relativamente grande, los fallos de trasponer mas multiplicar por filas son mucho menores que los de multiplicar por filas y columnas. En nuestro caso, al usar desde el principio matrices tan grandes, la multiplicación traspuesta siempre es más rápida, pero si usamos tamaños alrededor de 50, se puede apreciar muy bien el cambio:





En la gráfica de los fallos de caché, los fallos de escritura de ambos programas se solapan, pues son bastante similares y son mucho menores que los fallos de lectura, sin embargo, dado que en el enunciado de la práctica se comenta expresamente que incuyamos estas gráficas, se aporta el archivo *mult.dat* con la estructura pedida para apreciar las diferencias entre ambos programas.