Pablo Izaguirre y Paula Samper
Arquitectura de Ordenadores
Grupo 1312

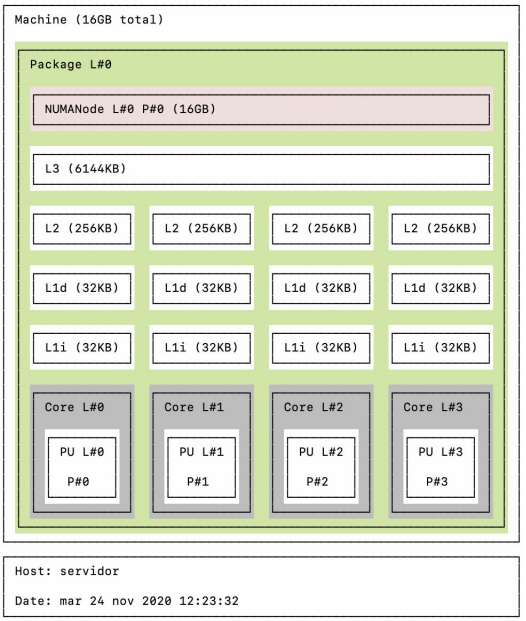
PRÁCTICA 3

Ejercicio 0

Para esta práctica, utilizaremos tanto el cluster como un ordenador propio.

Ordenador propio:

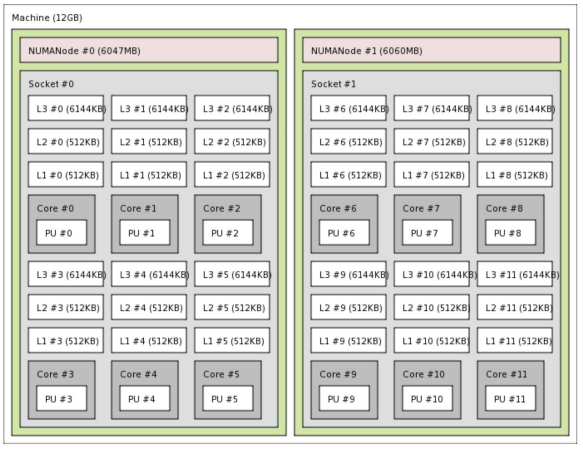
```
pablo@servidor:~$ sudo getconf -a | grep -i cache
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  6291456
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```



La caché de este ordenador tiene tres niveles distintos, todos ellos asociativos. Además, el nivel 1 se divide en dos distintos, uno para datos, y otro para instrucciones. Como podemos ver, cada core tiene una parte de las cachés de nivel 1 y 2, mientras que la caché de nivel 3 es común para todos ellos. La caché de nivel 3 tiene una capacidad de 6144 KB, la de nivel 2 de 256 KB, y la de nivel 1, tanto para datos como para instrucciones, tienen cada una 32 KB.

Cluster:

```
[arqo37@labomat36 P3]$ getconf -a | grep -i cache
LEVEL1_ICACHE_SIZE                 65536
LEVEL1_ICACHE_ASSOC                2
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 65536
LEVEL1_DCACHE_ASSOC                2
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  524288
LEVEL2_CACHE_ASSOC                 16
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  10485760
LEVEL3_CACHE_ASSOC                 96
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

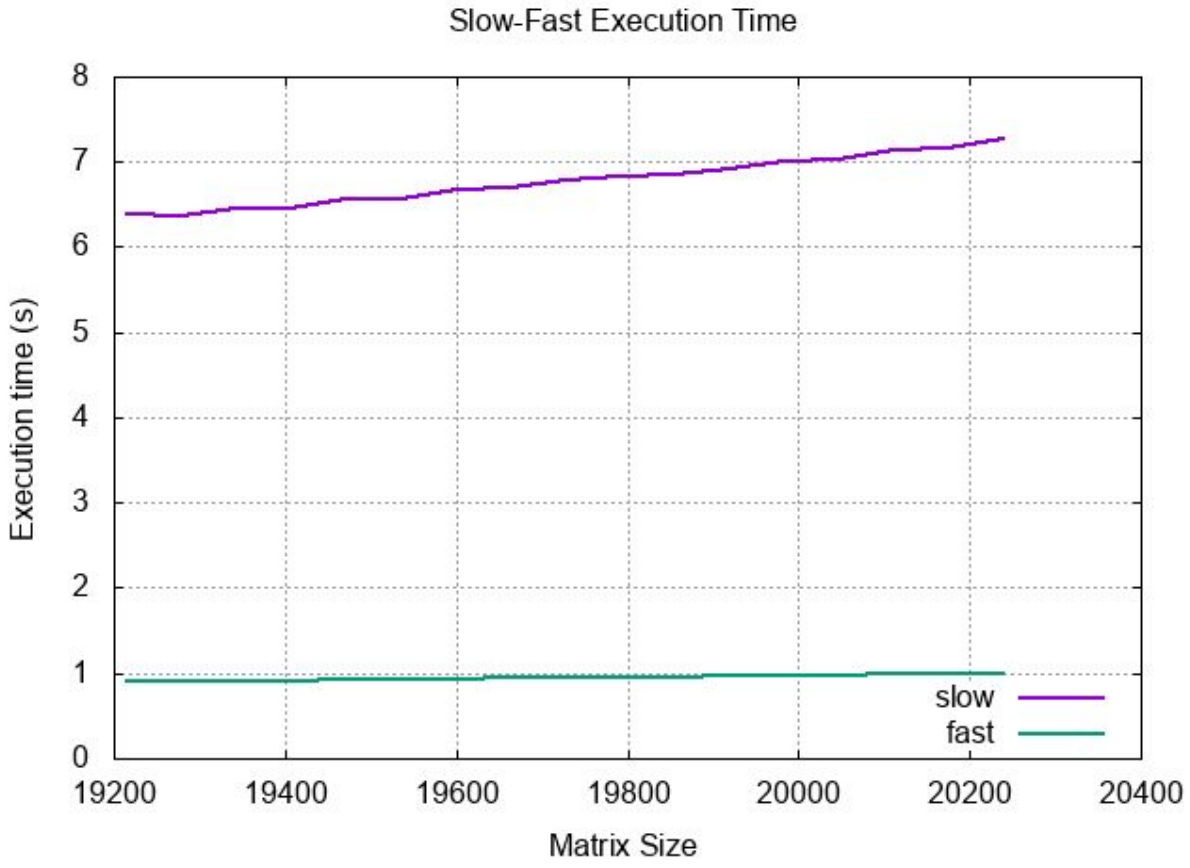


La caché de este ordenador tiene tres niveles distintos, el nivel 1 se divide en dos distintos, uno para datos, y otro para instrucciones. La caché de nivel 3 tiene una capacidad de 10 MB, la de nivel 2 de 512 KB, y la de nivel 1, tanto para datos como para instrucciones, tienen cada una 64 KB.

Ejercicio 1

Las pruebas de este ejercicio se han realizado en nuestro ordenador.
Para este ejercicio hemos tomado 20 medidas para cada tamaño de matriz. Hemos comprobado que con este número las curvas se suavizan lo suficiente. Es necesario tomar varias medidas y luego hacer la media entre ellas ya que el ordenador puede estar realizando otras tareas al mismo tiempo que se ejecutan los programas, y por lo tanto el tiempo de ejecución puede ir variando de una ejecución a otra. Tomando la media tenemos una mejor estimación del tiempo que tardan en ejecutarse los programas.

La gráfica del tiempo de ejecución con respecto al tamaño de la matriz es la siguiente:



Para obtener los datos de los tiempos de ejecución hemos creado un script de bash que nos permite ejecutar de manera alternada los programas slow y fast 20 veces cada uno por cada uno de los tamaños de la matriz. Así creamos dos archivos auxiliares con todos los datos, y a partir de estos archivos elaboramos el fichero slow_fast_time.dat, en el que incluimos las medias de los tiempos de ejecución para cada uno de los tamaños de matriz.

La diferencia entre los dos programas de prueba radica en que el algoritmo “slow” lee la matriz por columnas, mientras que el algoritmo “fast” la lee por filas. Por lo tanto el motivo por el cual para tamaños de matrices pequeños ambos programas dan el resultado en

tiempos similares es que la matriz entra en pocos bloques de memoria y por lo tanto no hay que realizar muchos accesos a memoria, y en este caso ambos programas actúan de manera similar, obteniendo los datos de la matriz de la caché.

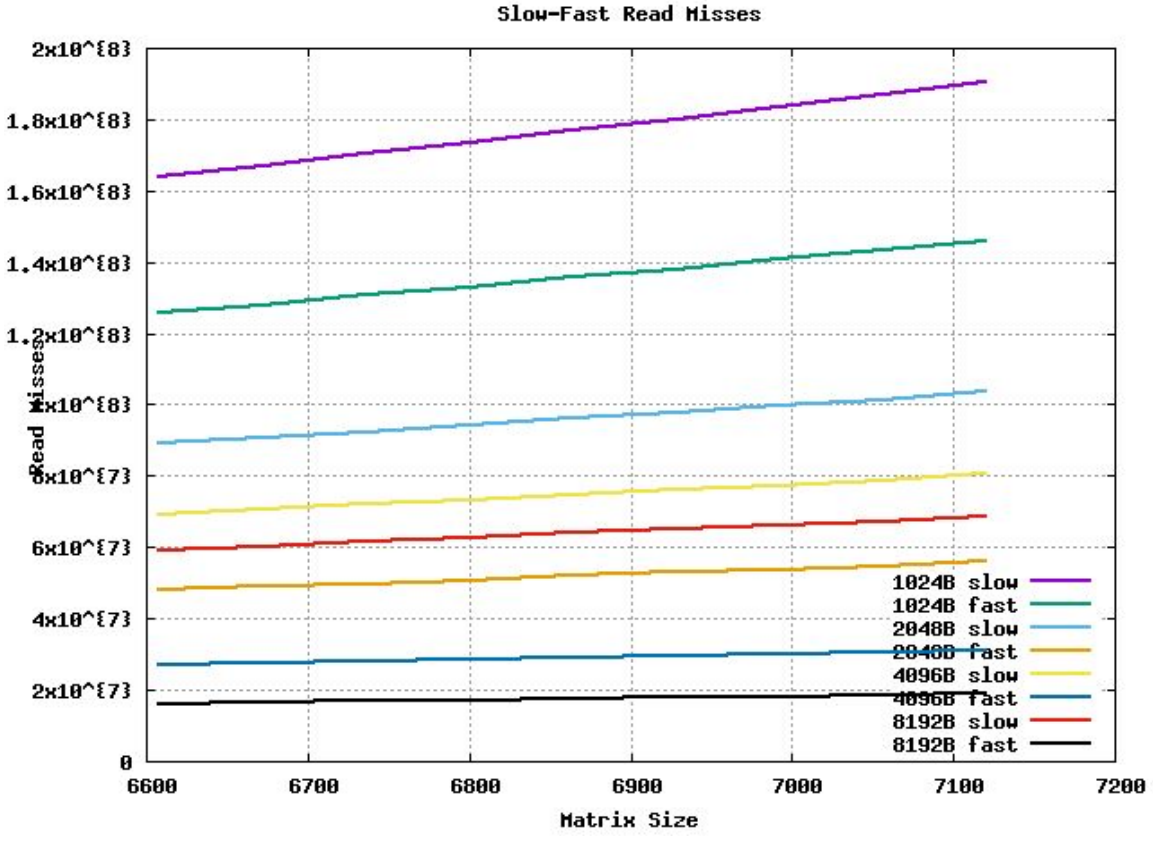
Cuando la matriz es más grande, deja de entrar entera en la caché y es necesario acceder a la memoria más veces, y en esto es más eficiente el algoritmo “fast”. Las matrices se guardan en memoria por filas, por lo tanto este programa irá accediendo de bloque en bloque ordenadamente, haciendo un acceso a memoria por bloque. Por otro lado, en tamaños grandes de matriz cada elemento de una columna estará en bloques distintos, por lo tanto cada vez que el algoritmo “slow” accede a un dato de la matriz tendrá que buscarlo en un bloque distinto, que habrá que traer a cache, reemplazando uno que ya esté ahí en caso de que no esté en caché y ésta se encuentre llena. De esta manera el programa “slow” realizará muchos más accesos a memoria que el programa “fast”.

Ejercicio 2

Las pruebas de este ejercicio se han realizado en el cluster.

En nuestro caso, P = 19 mod(7) + 4 = 5+4 = 9, por lo que el tamaño de nuestras matrices variará entre Ninicio = 2000+512*P = 6608, y Nfin = 2000+512*(P+1) = 7120, con incrementos de 64 unidades.

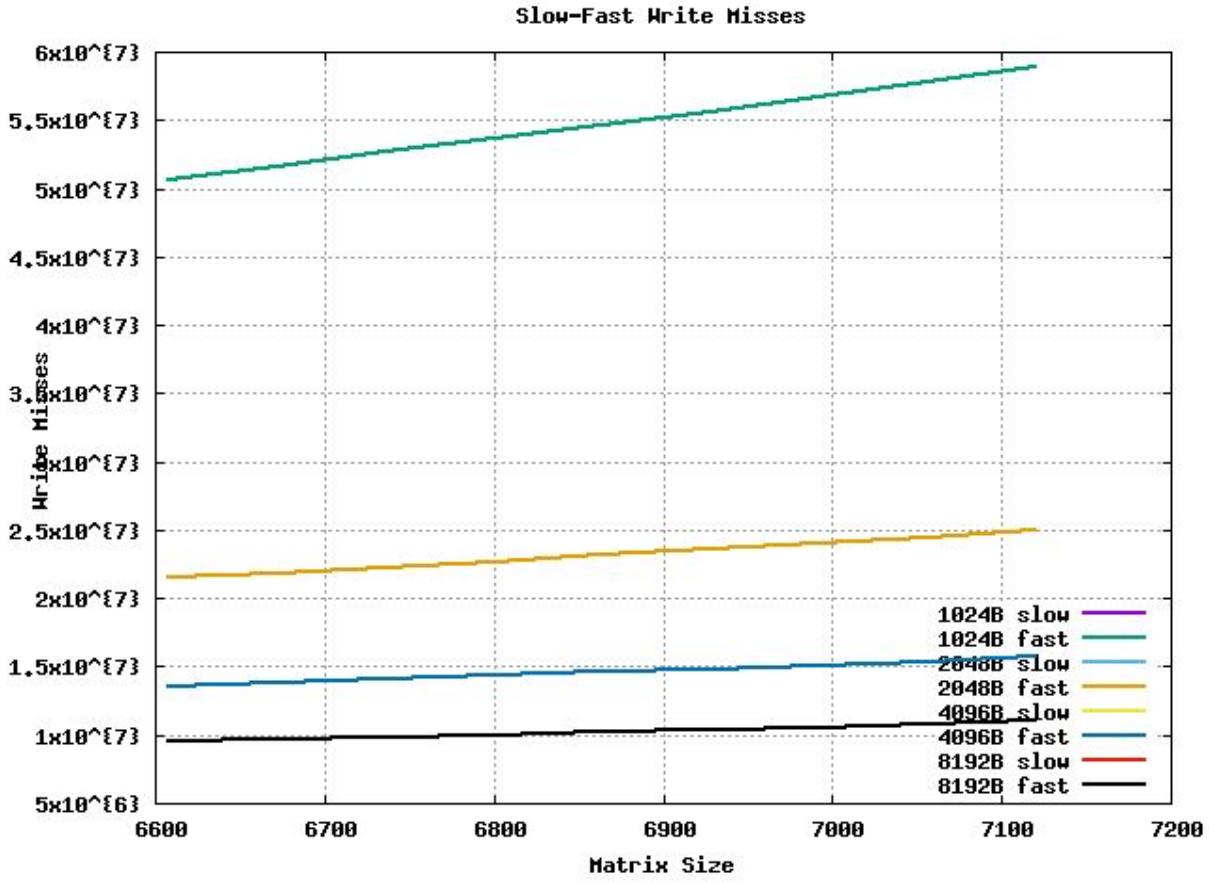
Tras ejecutar el script de bash, obtenemos las dos gráficas, una para los fallos de lectura de los datos, y otra para los fallos de escritura.



En esta primera gráfica, la de lectura, podemos ver que:

Fallos caché 1024B > Fallos caché 2048B > Fallos caché 4096B > Fallos caché 8192B
Esto se debe a que los fallos de capacidad de la caché disminuyen a mayor tamaño. Los fallos de capacidad de la caché son aquellos causados porque un bloque que ha sido reemplazado es accedido más tarde. Esta situación se dará más a menor capacidad de la caché, ya que los bloques tendrán que ser reemplazados más a menudo.

Dentro de los fallos en una caché de un mismo tamaño, podemos distinguir dos casos distintos: cuando se usa *slow* y cuando se usa *fast.* Esto se debe a que, como ya hemos explicado en el ejercicio 1, las matrices se guardan por filas en memoria, por lo que *fast* irá accediendo en orden de bloque en bloque, haciendo como mucho un acceso a la memoria principal por bloque, mientras que slow accede cada vez a un bloque distinto, por lo que como máximo hará N accesos por bloque a memoria principal. Por ello, los fallos de lectura de *slow* serán mayores a los de *fast*.



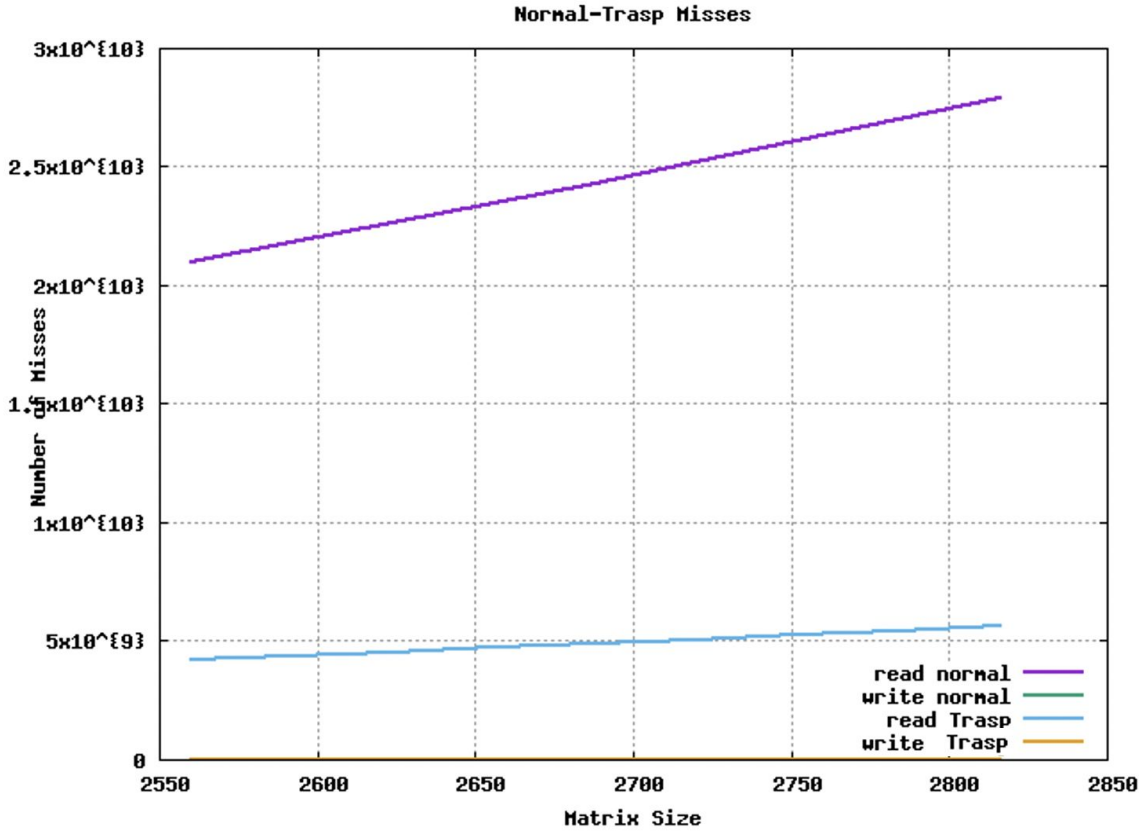
En la segunda gráfica, de escritura, podemos apreciar que se repite la secuencia Fallos caché 1024B > Fallos caché 2048B > Fallos caché 4096B > Fallos caché 8192B, por las mismas razones que en la gráfica anterior.
Sin embargo, en este caso no distinguimos diferencia entre los casos *slow* y *fast* para cachés del mismo tamaño. Esto se debe a que tanto slow como fast escriben en memoria el mismo número de veces (el número de elementos de la nueva matriz, NxN), por lo que tendrán aproximadamente el mismo número de fallos de escritura.

Ejercicio 3

Estas pruebas se han realizado en el cluster.

En este ejercicio, creamos dos programas para multiplicar matrices. El primero, *mult.c*, genera dos matrices y luego las multiplica, mientras que el segundo, *mult_transpuesta.c*, transpone la segunda matriz antes de multiplicarlas. Tras ejecutar el script, obtenemos dos gráficas.

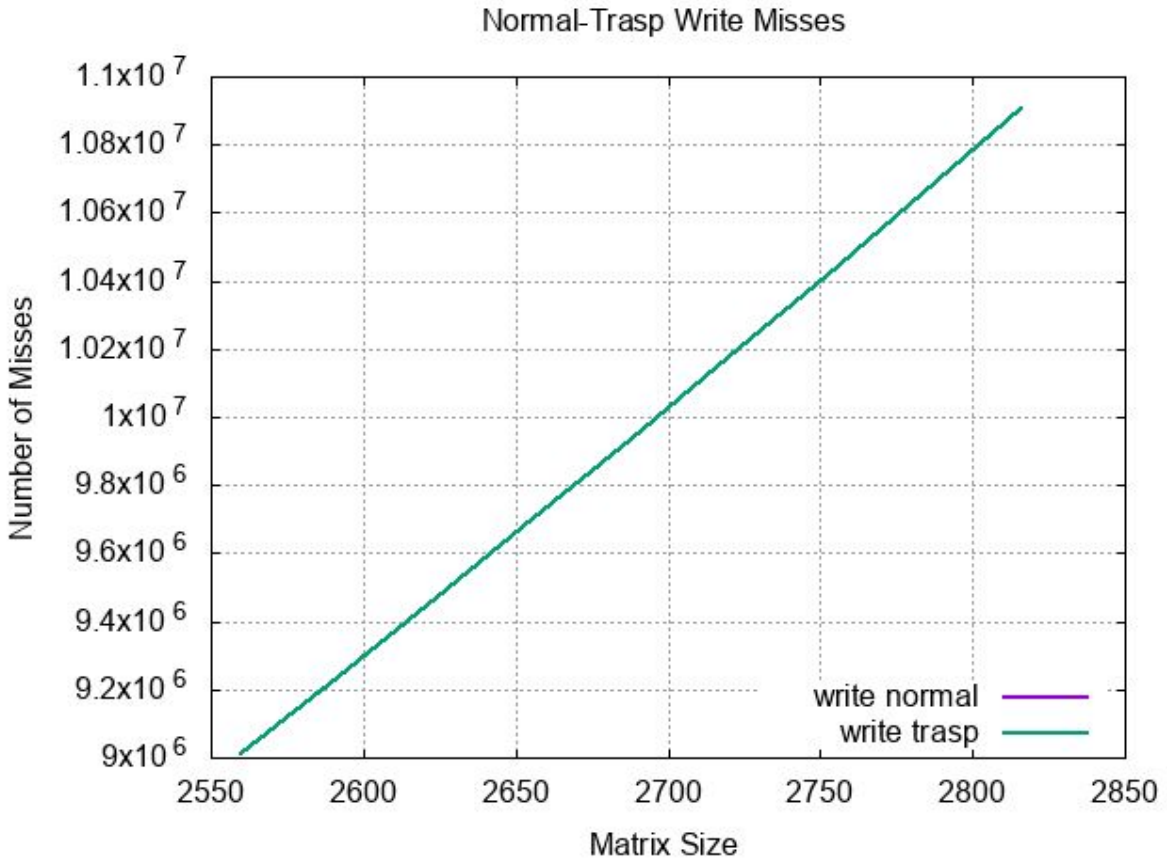
La primera representa los fallos de caché, distinguiendo entre los fallos de lectura y escritura en *mult.c* (normal), y entre los fallos de lectura y escritura de *mult_transpuesta.c* (trasp).



Como podemos observar, los fallos de escritura de *normal* y *trasp* son iguales, mientras que los fallos de lectura de *normal* son mucho mayores que los de *trasp*. Esto se debe, a que, como hemos explicado previamente, las matrices se almacenan en memoria por filas, por lo que cuando multiplicamos una matriz de forma normal, accedemos a las filas de la primera matriz y a las columnas de la segunda.

En *trasp*, como hemos traspuesto la segunda matriz, en realidad estamos accediendo a las filas de la segunda matriz, por lo que no estaríamos accediendo a ninguna matriz por columnas.

Hemos generado adicionalmente un gráfico extra de los write misses para estudiarlos más a fondo.

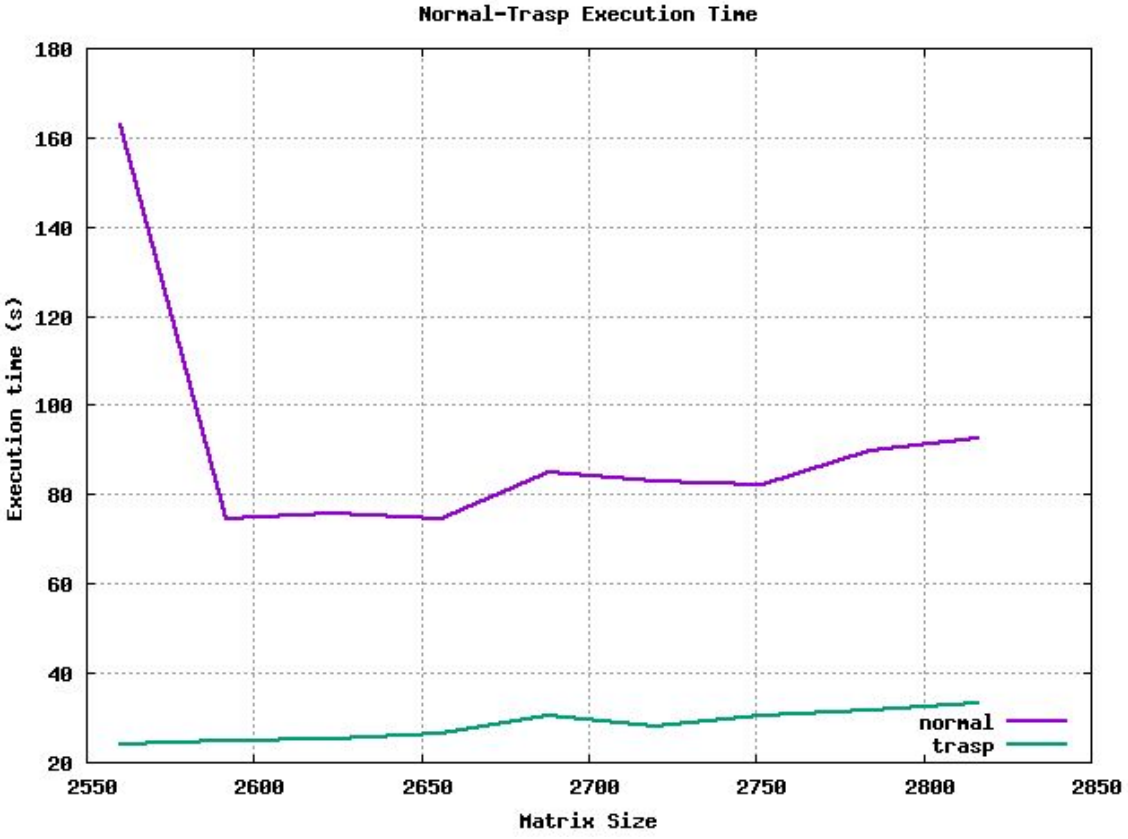


Como podemos ver, el número de fallos de escritura de *normal* es el mismo que el de *trasp*. En un primer momento, pensamos que habría algún error en nuestro código, ya que creíamos que debería haber más fallos en *trasp* al transponer la matriz.

Sin embargo, nos dimos cuenta de que como estábamos sobreescribiendo la matriz B, al transponer los elementos, estábamos escribiendo en una posición que había sido leída una o dos instrucciones antes, por lo que el dato ya estaría en la caché a la hora de escribir en él.

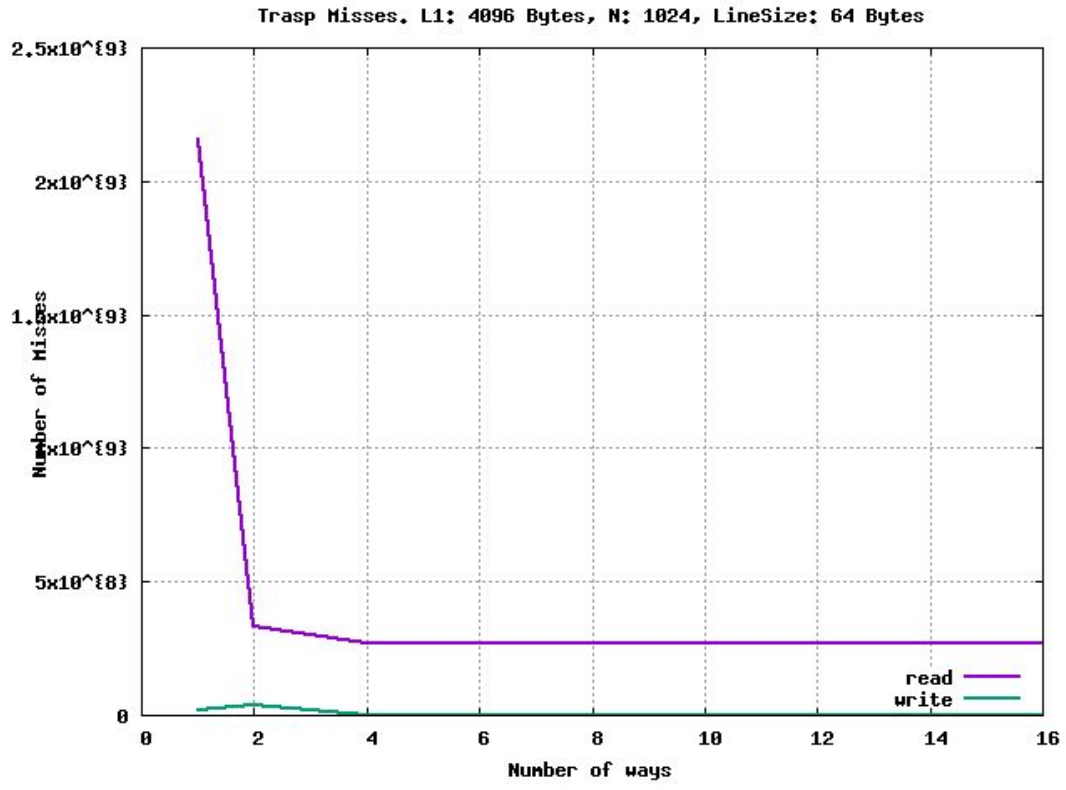
Al multiplicar las matrices, ambos programas van escribiendo de igual forma en una matriz nueva, por lo que los errores de escritura serán los mismos.

La segunda gráfica que se genera con nuestro script, refleja el tamaño de las matrices frente al tiempo de ejecución. Se observa que el tiempo de ejecución de *normal* es mayor que el de *trasp*. Esto se debe a que, como hemos explicado antes, *normal* tiene bastantes más fallos de caché que *trasp*, lo que causa que tenga que emplear tiempo extra en acceder a la memoria principal a buscar los datos.



Ejercicio 4

En primer lugar, queremos ver cómo afecta la variación del número de vías al número de fallos.



Confirmamos, que como hemos visto en teoría, cuantas más vías, menor número de errores.

Por último,vamos a ver cómo afecta la variación del tamaño de los bloques al número de fallos.

Al aumentar el tamaño de los bloques, debería reducirse también el número de fallos, ya que solemos acceder a información cercana en memoria. Sin embargo, bloques más grandes implica que caben menos en la caché, lo que aumenta la competición entre bloques. Los bloques más grandes también implican más contaminación por datos innecesarios en la caché.

Las desventajas de tener bloques grandes superan a las ventajas, lo que se ve reflejado en la gráfica.