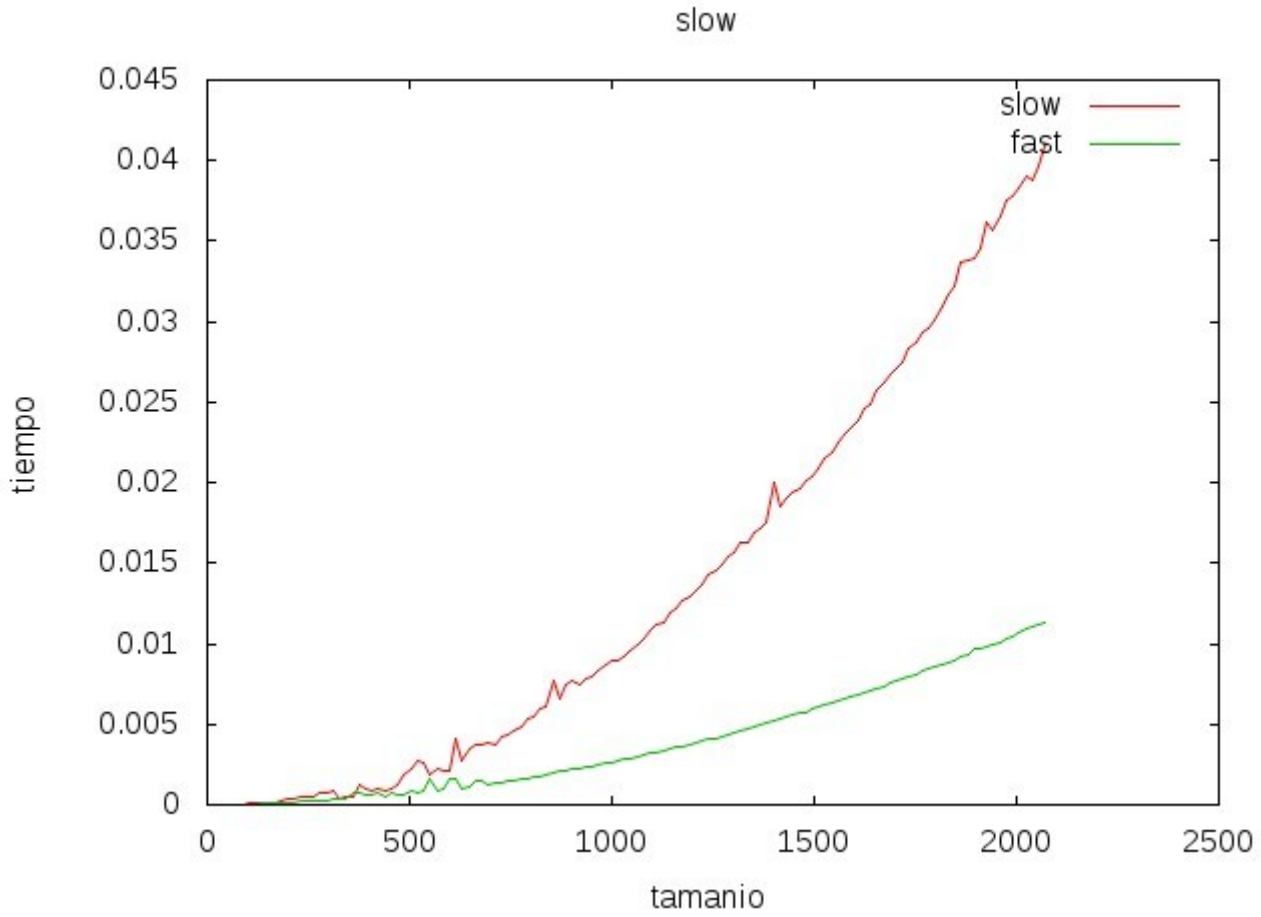
Memoria de la práctica 3 de Arquitectura de Ordenadores

Hecho por:
-Javier Gómez Martínez
-Carlos Li Hu

Ejercicio 0

Este ordenador contiene 3 niveles de cachés
En el nivel 1 de caché hay dos cachés separadas de datos e instrucciones. Siendo cada una de ellas asociativas de 8 vías. Ambas tienen un tamaño total de 32768 Bytes y 64 Bytes por linea.
En el nivel 2 hay una única caché también de 8 vías y 64 Bytes por linea, pero esta es más grande teniendo un tamaño de 262144 Bytes.
Por último, en el nivel 3, hay también una única caché que tiene un tamaño de 3145728 Bytes, 12 vías y 64 Bytes por línea.

Ejercicio 1



El método empleado para obtener los datos ha sido un script que, en un bucle, guarda los outputs que los programas imprimen por pantalla en dos cadenas de caracteres distintas, sometidas a las operaciones básicas de shell scripting correspondientes para que únicamente representen los datos que nos interesan (los tiempos de ejecución). Estas operaciones son un grep para obtener la linea de interés y un cut para separar el dato que interesa del resto de la línea.
Una vez obtenidas las cadenas correspondientes, se vuelcan a un fichero con los datos (organizados en columnas) mediante un echo. Sobre ese fichero, se ejecuta el script de gnuplot adjunto en la carpeta “ejercicio1” para generar la gráfica presentada más arriba.

Para responder al resto de preguntas, utilizaré el siguiente esquema para poder explicar parte de los conceptos básicos.

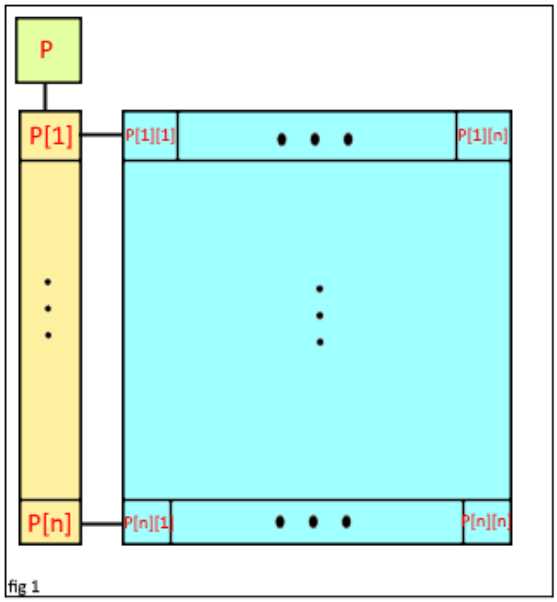


fig 1

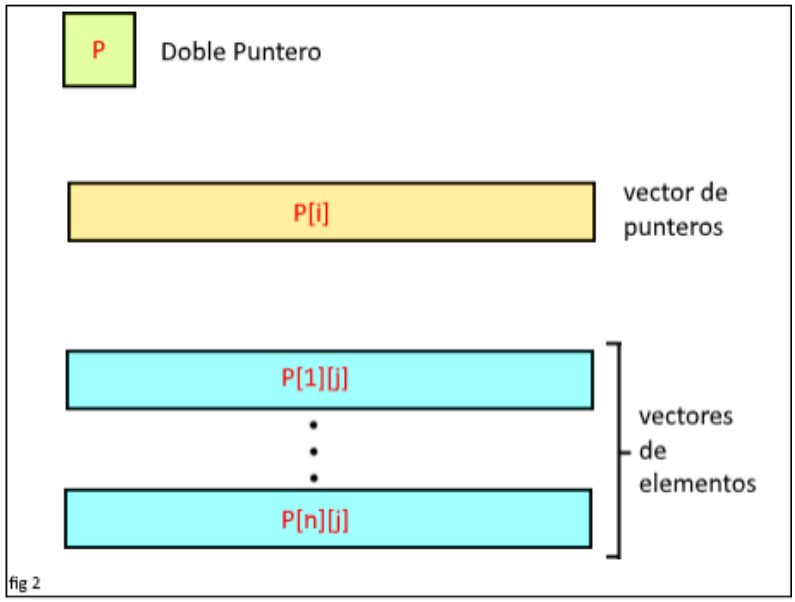


fig 2

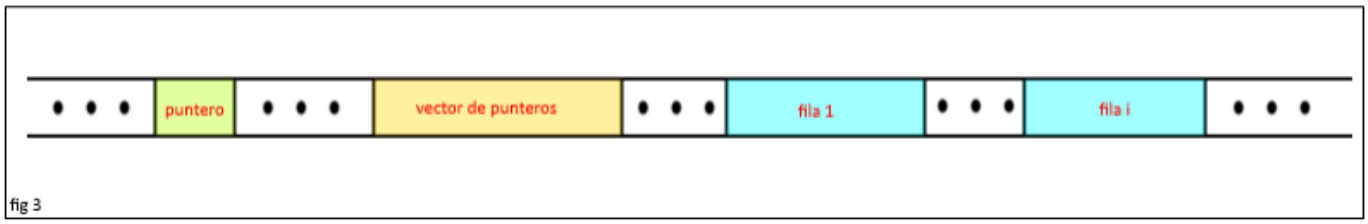


fig 3

En la figura 1, hemos representado la matriz de forma abstracta, como un puntero que apunta a una cadena (o vector) de punteros que, a su vez, cada uno de ellos apuntan a un vector de elementos.
En la figura 2 hemos reorganizado lo explicado en la figura 1 conforme al principio de localidad (porque se reserva memoria para cada vector en una misma llamada a malloc), es decir, el puntero P está “aislado” en una parte de la memoria, de la misa forma que lo están los vectores (tanto de punteros como de elementos), sin embargo, todos los elementos de un mismo vector están juntos. Nótese que los vectores de elementos son las filas de la matriz.
En la figura 3, hemos dibujado como podría estar guardada la matriz en memoria. La memoria en si está representada como la “cinta” y cada uno de los objetos del programa (punteros y vectores) como rectángulos.

¿Por qué para matrices pequeñas los tiempos de ejecución de ambas versiones son similares, pero se separan según aumenta el tamaño de la matriz?
Para matrices pequeñas, sucede que el tamaño de las memorias caché de los computadores de los laboratorios es suficientemente grande como para almacenar las matrices por completo (o al menos gran parte de éstas). De forma que, en ambos casos fast y slow, los fallos de caché son escasos y se accede pocas veces a memoria principal. Por tanto, los tiempos resultan similares

Sin embargo, a medida que aumentamos el tamaño de las matrices, sucede que slow tarda mucho más que fast (es mucho menos efectivo).
Esto es debido a que, para tamaños grandes, las matrices no caben enteras en caché (incluso que apenas cabe unas cuantas filas de la matriz) y por tanto se producen más fallos.
Fast produce pocos fallos porque accede a los elementos de la matriz por filas y como hemos explicado más arriba, las matrices se guardan en memoria por filas, es decir, los elementos de las filas están adyacentes los unos a los otros. Dado que al acceder a memoria principal se guarda en caché tanto el dato leído como los adyacentes, sucede que en este caso en caché suelen guardarse datos que serán utilizados más adelante (tras acceder al elemento 1 de una fila, los siguientes estarán almacenados en caché y por tanto el acceso a los mismo será más rápido)
Por el contrario, slow produce muchos más fallos en caché porque accede a los elementos de la

matriz por columnas, en otras palabras, accede a elementos que están a n*(tamaño del elemento) bytes de distancia. Es decir, los datos no están compactos en memoria sino lo contrario. Por tanto, en pocas ocasiones (si es que acaso sucede en alguna para tamaños muy grandes) ocurrirá que el dato siguiente a leer estará presente en la memoria caché.
Es por esto que a medida que se aumenta el tamaño de las matrices, slow produce más fallos en caché que fast y por tanto acaba siendo más lento. Esto se ve reflejado en que el gráfico de slow se distancia cada vez más y más del de fast, tomando valores más elevados.

¿Cómo se guarda la matriz en memoria, por filas (los elementos de una filas están consecutivos) o bien por columnas (los elementos de una columna son consecutivos)?

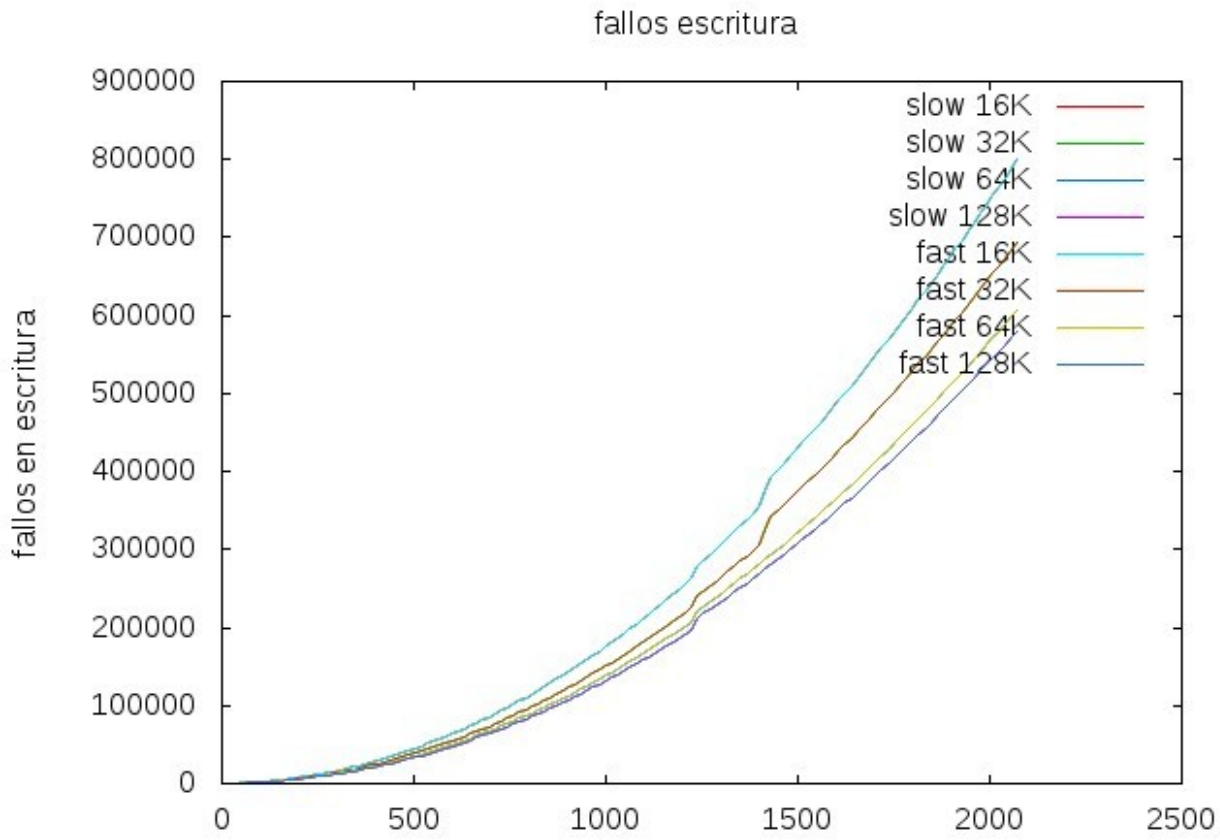
Tal y como hemos comentado más arriba, los elementos de la matriz se guardan en memoria según filas y no según columnas. Este hecho es la razón por la que existe esa diferencia de rendimiento entre slow y fast vista en la gráfica.

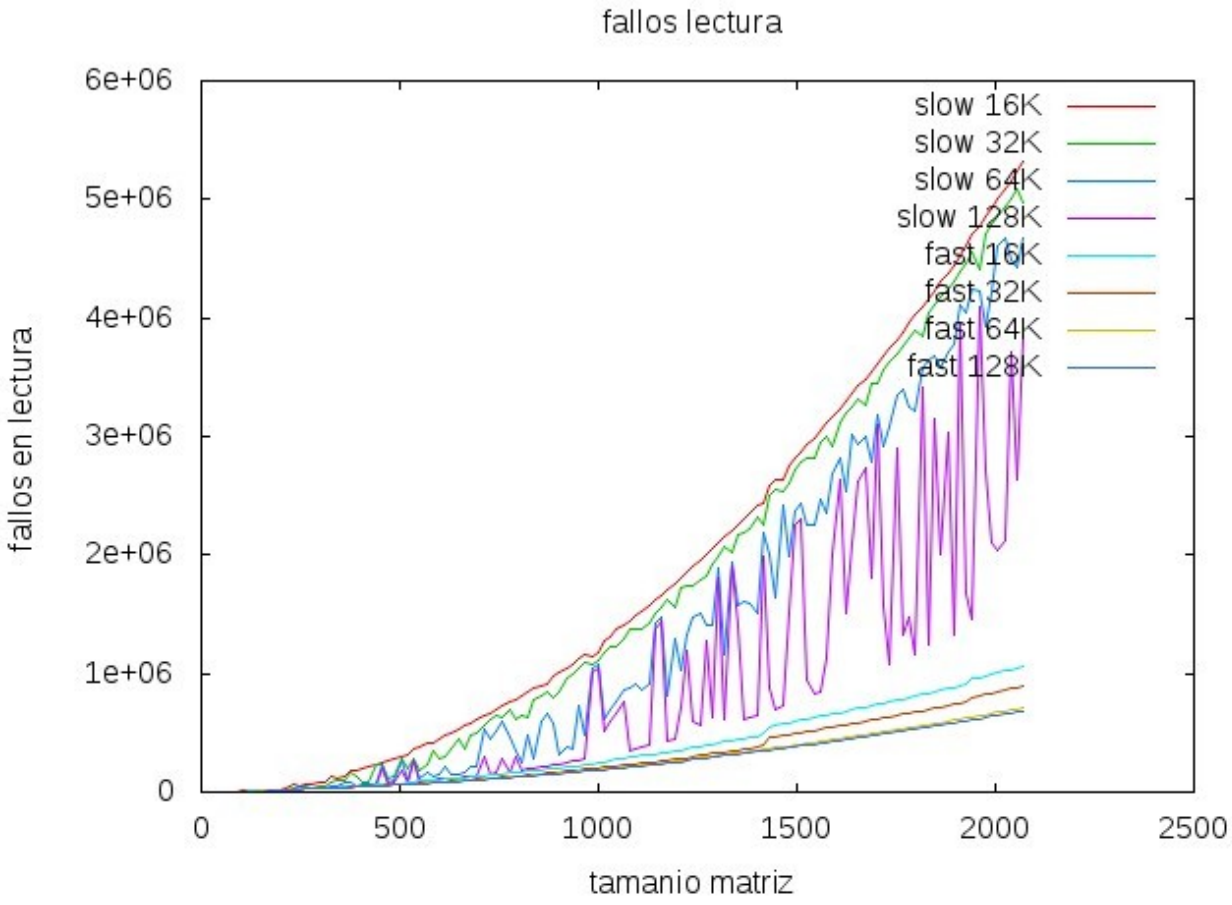
¿qué efecto tiene el tamaño de bloque de la cache?

El efecto que tiene el tamaño del bloque en la caché es esencialmente que, cuanto más grande sea, más datos caben en el bloque, más inusuales son los fallos de caché y, por tanto, más rápido funcionaría el programa.
Por otra parte, cuanto más grande sea el bloque de la caché, más tiempo malgastan los fallos, dado que se leen más datos de memoria principal para traerlos a caché (dado que leer de memoria principal es lento).

Si la caché de nivel 1 de datos de estos ordenadores tiene un tamaño de 32768 Bytes y el tamaño de sus bloques es de 64 Bytes, implica que la caché tiene 512 bloques, en los que caben 64 elementos de tipo double (que son los empleados en la práctica) en toda la caché, entonces, para matrices de tamaños mayores a 64 (8x8), ocurren fallos de caché.
En concreto, para fast, ocurren floor((n*n)/64) fallos de caché cuando no cabe la matriz entera en caché.
Y para slow, en el peor caso, habrá tantos fallos de caché como elementos tenga la matriz (n*n) Esto es cuando la fila ocupa un bloque entero o más (n >= 8).
Para matrices de tamaños menores, el número de fallos de caché es igual o parecido en ambos casos (porque caben las matrices enteras en caché).

Ejercicio 2



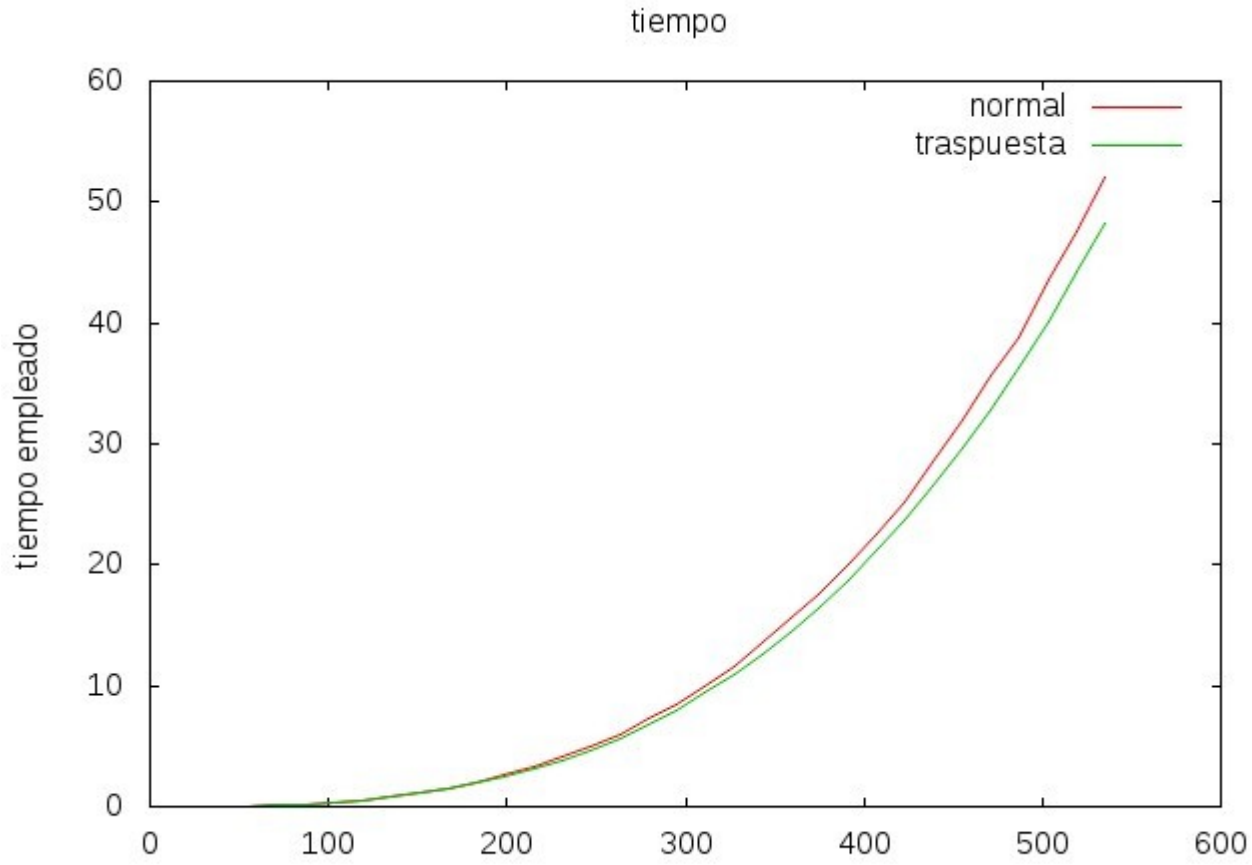


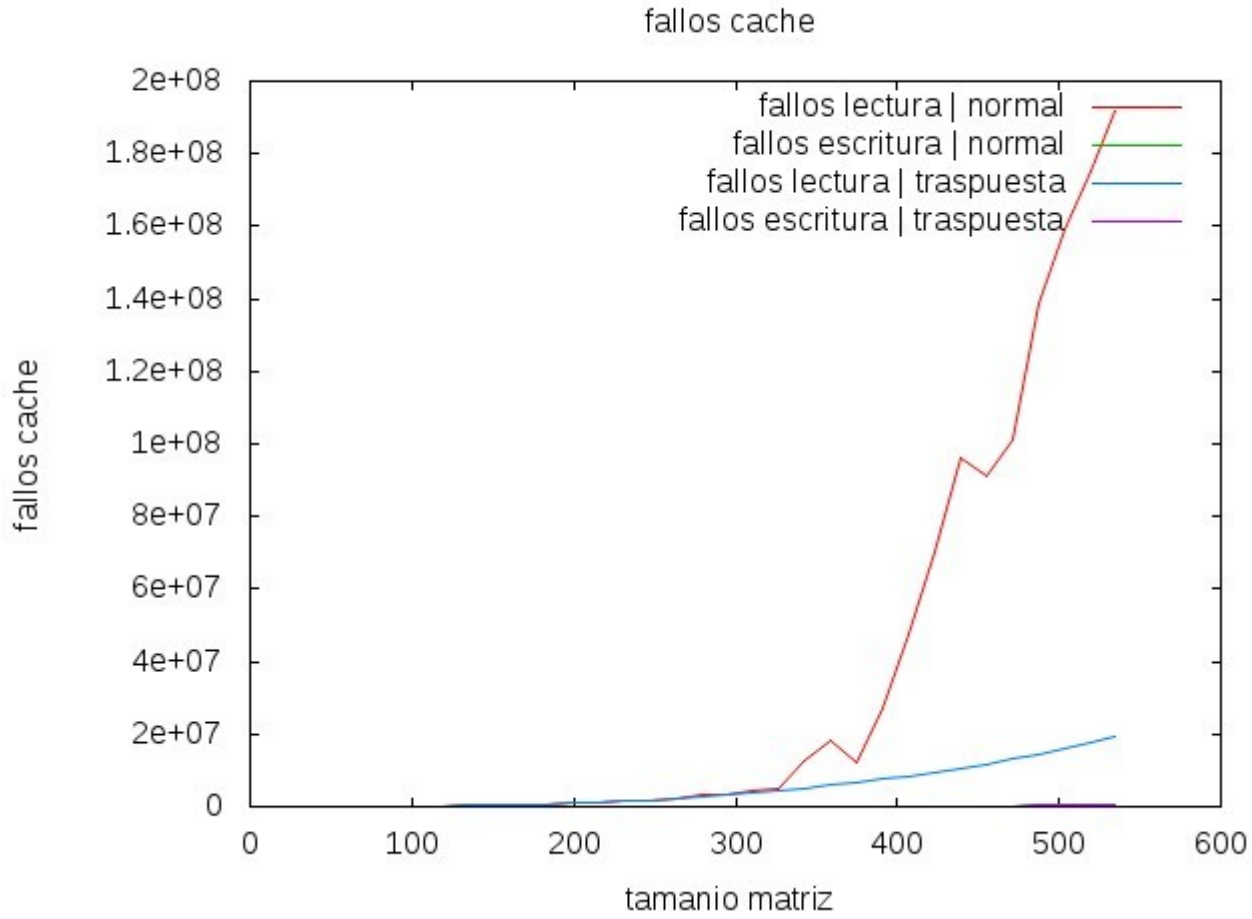
Respecto a los fallos de escritura en caché, se observan variaciones de fallos en los programas

conforme aumentamos el tamaño de la caché.
Dado que tanto en slow como en fast se “escribe” en la matriz del resultado de la misma forma, las diferencias entre las simulaciones de ambos programas para el mismo tamaño de caché son mínimas (y en las gráficas son casi indistinguibles).
Por otra parte, podemos observar que según aumentamos el tamaño de la caché, hay más fallos de escritura, tal y como era de esperar. Además de esto, no se produce ningún comportamiento anómalo, es decir, todas las gráficas siguen aproximadamente la misma distribución.

Sin embargo, en las gráficas de fallos de lectura observamos comportamientos que a priori parecen anómalos. En concreto, en las gráficas de la ejecución slow en memorias caché de 128 y 64 KB. Creemos que estos resultados son así debido a que la memoria caché del computador donde ejecutamos los scripts (uno de los ordenadores del laboratorio 8b) es más pequeña que los 128 y 64 KB que pedíamos a valgrind para la simulación, por tanto, los fallos en caché se ven afectados no solo por los fallos en caché de la simulación sino también por los fallos en caché del sistema en sí. De hecho, podemos ver, en especial en la gráfica slow 128 KB, que hay dos “tendencias” de valores entre los que la función fluctúa: el “suelo”, que se aproxima a los valores de las gráficas para fast, se asemejaría más a cómo debería resultar la simulación en caso de que la máquina realmente tuviera una caché de 128 KB, por otra parte, consideramos que el “techo” está formado por valores que se ven afectado por lo anteriormente comentado. De hecho, este “techo” se parece mucho a la gráfica de 32 KB para slow. Esto, junto con el hecho de que el tamaño total de las cachés del sistema es aproximadamente de 32 KB (tal y como hemos visto en el ejercicio 0), es lo que nos ha llevado a la conclusión de que la simulación se ve afectada por las características reales del ordenador.
Obviando las gráficas de slow 64 KB y 128 KB, podemos observar que el resto de casos se comporta como cabría a esperar. El motivo por el que las gráficas de slow se alejan tanto de las de fast según aumentamos el tamaño de las matrices es el mismo que el explicado en el ejercicio1

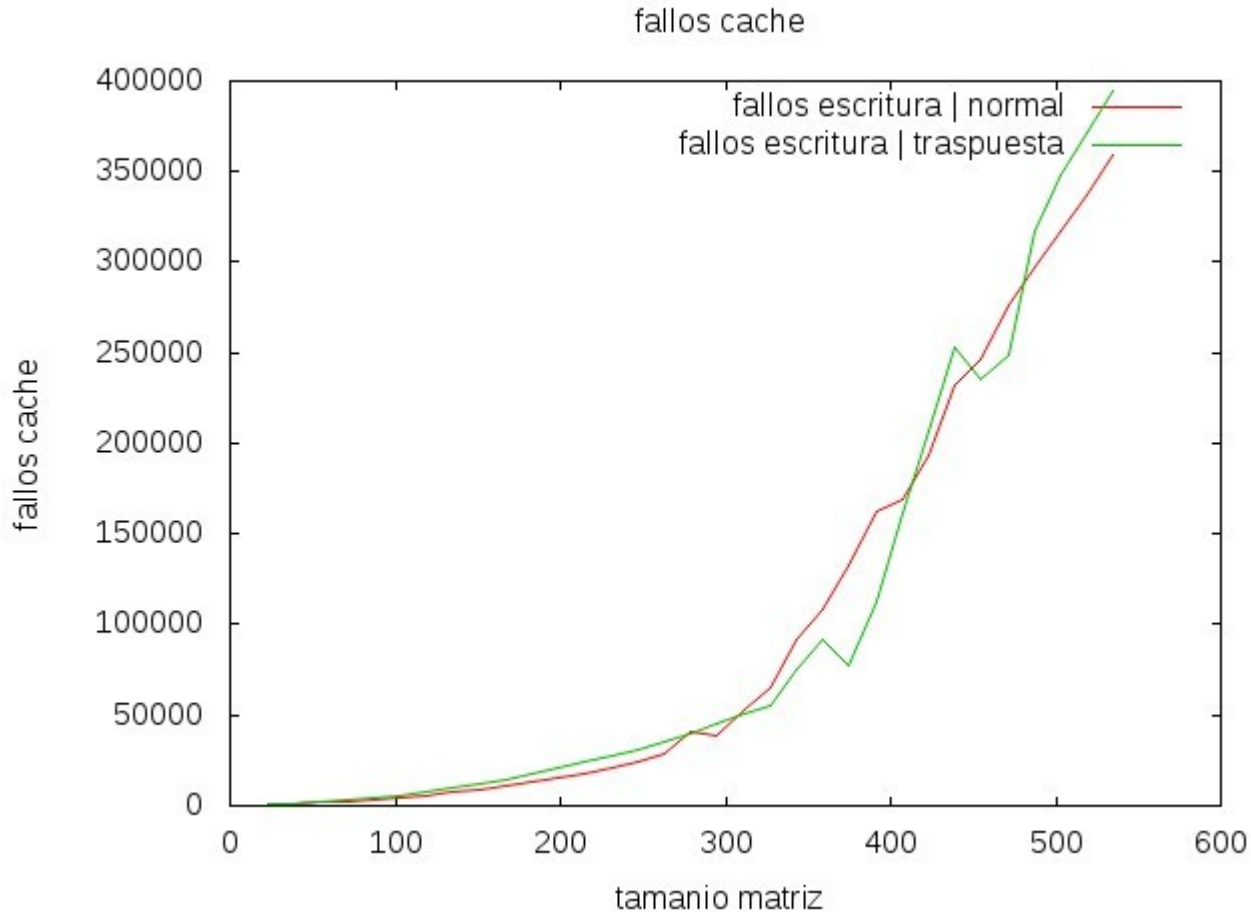
Ejercicio 3


tiempo
tiempo empleado
normal
traspuesta
0
10
20
30
40
50
60
0
100
200
300
400
500
600


fallos cache
fallos cache
fallos lectura | normal
fallos escritura | normal
fallos lectura | traspuesta
fallos escritura | traspuesta
0
2e+07
4e+07
6e+07
8e+07
1e+08
1.2e+08
1.4e+08
1.6e+08
1.8e+08
2e+08
0
100
200
300
400
500
600
tamanio matriz

Al aumentar el tamaño de las matrices se observa que, en los tiempos, la multiplicación de matrices sin trasponer a la larga es mucho más lenta (varios segundos más lenta para los tamaños más grandes) que la multiplicación de matrices trasponiendo una de ellas. Esto es debido a que los fallos de caché que se producen al no trasponer la segunda matriz ralentizan la ejecución.

Esto lo podemos corroborar con la gráfica de fallos de caché, en la que observamos que los fallos en lectura de la caché en la multiplicación normal se disparan en comparación con la multiplicación trasponiendo.
Esto es debido a la forma en la que se acceden a los elementos de la matriz:
Tomando como referencia la explicación dada en el ejercicio 1 de cómo están guardadas las matrices en memoria y cómo acceder a ellas por filas tiene mucho menos impacto que si se hiciera por columnas en lo que a fallos de caché se refiere, podemos deducir la razón de que las gráficas del ejercicio se comporten de esta forma:
En el caso en el que no transponemos ninguna de las matrices, para hacer la multiplicación de A por B, hemos de acceder a los elementos de B por columnas, causando así multitud de fallos de caché en lectura. Al transponer, sin embargo, los accesos que se hacen a la matriz A y los que se hacen a la matriz B^T son por filas y por tanto hay muchos menos fallos de caché en lectura.
Los fallos de caché en escritura son tan minúsculos que en comparación con los de lectura apenas son apreciables. Para analizarlos, hemos hecho otra gráfica aparte comparándolos:



Aquí podemos observar que en ambos casos los fallos de escritura son similares. Sin embargo, las causas de estos fallos son distintas.
En el caso de la multiplicación transponiendo, es evidente que los fallos se deben a que se escribe en la matriz transpuesta por columnas, generando bastante fallos de caché en escritura (siguiendo el razonamiento ya explicado de por qué los accesos por columnas a una matriz producen más fallos que por filas).

Sin embargo, para el caso de la multiplicación sin transponer, los fallos de escritura suceden como consecuencia de los fallos de lectura:
Al haber muchos fallos en lectura, los bloques de la caché deben ser sustituidos con mayor frecuencia, como consecuencia de esto, para cuando se va a hacer una escritura, el bloque correspondiente tiene altas probabilidades de haber sido sustituido y por tanto se produce un fallo en escritura.
En definitiva, los fallos de escritura al transponer se producen por la propia transposición de la matriz, mientras que los fallos de escritura al no transponer se producen como consecuencia de los frecuentes fallos de lectura.

En conclusión, transponer la matriz acaba resultando mucho más rápido que no hacerlo a la hora de multiplicar, debido al reducido número de fallos de caché en lectura. El tiempo que hay que emplear en transponer la matriz es despreciable en comparación.