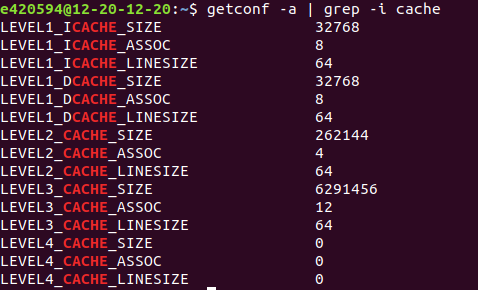
**Arq0 Practice 3**

**By Daniel Varela & Guillermo Martín-Coello**

**Exercise 0:**

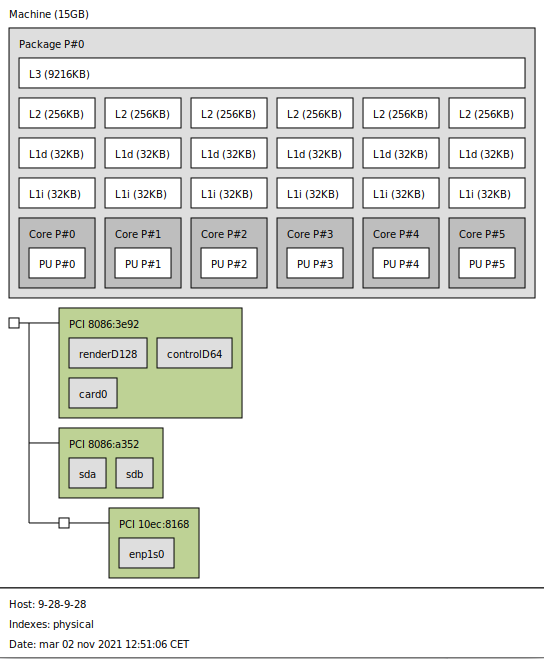


The labs computers has three cache levels:

1-On the first one we can see a not unified cache divided in two with a size of 32768 B, an associativity of 8 lines and a line size of 64B

2- On the second one we have a unified cache of 262144 B, an associativity of 4 lines and a line size of 64 B

3- On the third one we have a unified cache if 6291456 B, an associativity of 12 lines and a line size of 62 B

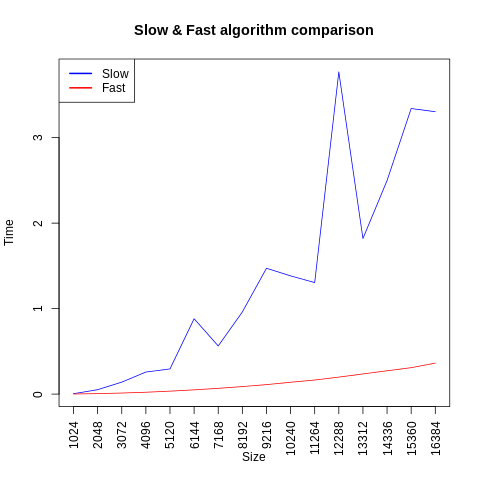


As we can see the computer we used in the labs has six different cores with three caches each. Then it has a last level shared cache much bigger than the others.

**Exercise 1:**

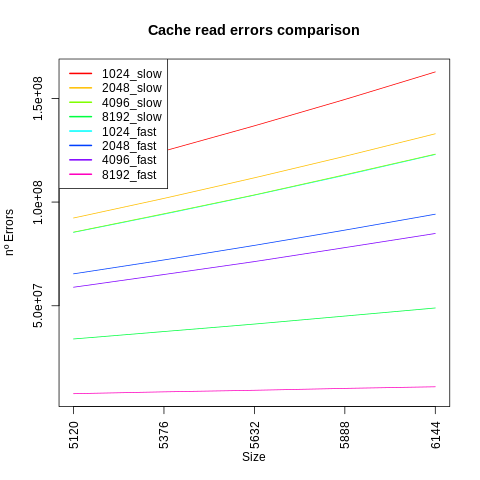
2) The reason why performance measurements need to be taken multiple times is the concurrence of programs in the processor, which can make the execution time vary depending on the usage and the OS politics of preference. When we take multiple interleaved measurements we are able to mitigate the momentary peaks of execution time (caused by the processor’s need to execute other programs at the same time) thus obtaining a more realistic value without abnormal data.

3)

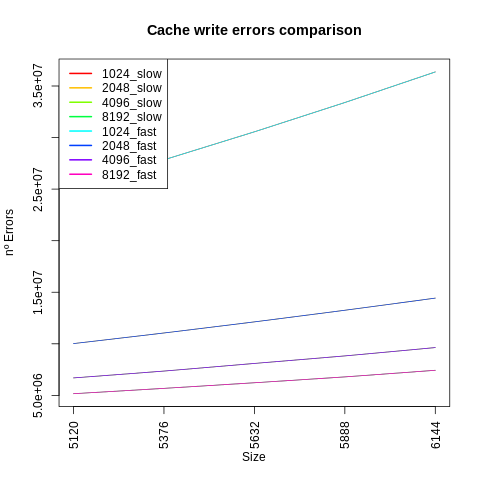


5) The method we use to take the results consists of executing two different algorithms for matrix addition. On the first algorithm **(Slow)** the different elements of the column are visited first and then when we finish the column, we go to the next one. On the other hand, the second algorithm **(Fast)** visits the elements row by row. When a matrix is stored in the memory, it’s stored in rows saving all the elements in the first row then second and so on… This makes a difference in the execution times of both methods because while with the Slow method we need to jump all the elements in a row to access to the next element therefore not using most of the elements charged in the block, with the Fast method, we access all the elements in the block because they are stored contiguous row by row. This difference may not be too significant in small matrices but once the size increases, the difference keeps getting greater (as seen in the previous graphic), this is due to the size of the row being greater thus making the Slow method waste more time to make the necessary jumps.

**Exercise 2:**



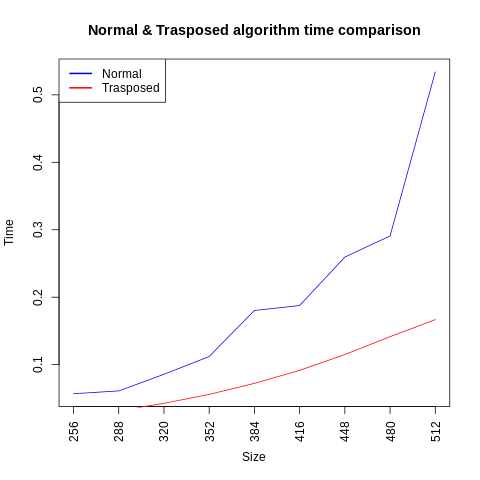
The first plot represents the errors generated while reading the cache (which means that the requested element of the matrix is not in the cache when requested). The information we obtain from the graphic is that, as we expected, the **Slow** executions generate more reading errors than the **Fast** executions due to the algorithm they use (explained in answer 1.5) being that **Fast** only gets reading errors after visiting as many elements as fit in the cache, and **Slow** method getting a reading error for each element unless the cache can store more than one whole row in which case it will get a reading error once every number of rows that fit in the cache. The other thing that should be highlighted is that when the cache size is increased, fewer errors occur, this is because if the cache is bigger, more elements can be stored in it reducing then the times that we encounter reading errors (elements requested that are not in the cache). We also need to clarify that the slope keeps getting lower and lower when we increase the cache just because we are increasing the size of the cache much more than we are increasing the size of the matrix thus making the matrix increment of size seem more insignificant when we increase the cache size at the same time.



In this case we can see a graph that shows the errors generated while writing in the cache. It’s visible at first sight that the amount of errors is much fewer than the ones encountered when reading the cache, this is because the algorithms presented in slow and fast are designed to just write on one single variable the result of a sum but must access a big amount of positions in memory therefore it’s significantly more probable to encounter a reading error than a writing one. It’s also remarkable that in the graph we can only see one line for each type of cache, this is because they are solapated because both algorithms slow and fast at the same amount of writing errors since they access the memory for writing the same amount of times. Finally, we can say that the bigger the cache is, the fewer errors we get, this is due to the same reasons as the explanation of the previous graphic, the bigger the cache is, the less probable it is to encounter a failure of these kinds.

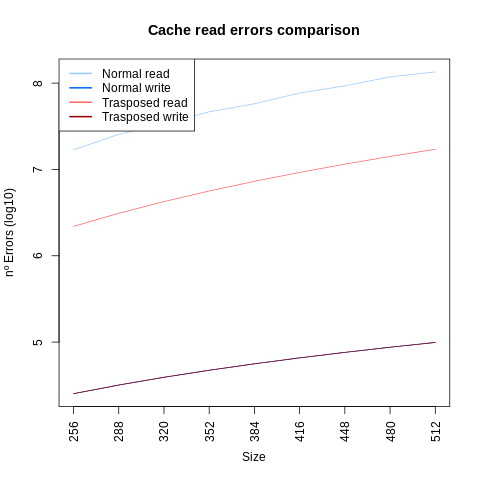
**Exercise 3:**

Here we have the plot generated with the times of executing a normal matrix multiplication and a transposed one. We’ve executed these with the default cache configuration of the UAM cluster:



As expected, the normal matrix multiplication is slower than the transposed one. We can also point out that the growth of the function’s time is proportional to the matrix size, because if we increase the size, we increase the amount of operations to make hence increasing the execution time.

Next, we can see a plot of the cache misses of the same programs as the previous plot. It’s important to highlight that the plot’s y axis is in logarithmic scale:



The misses in the plot are divided in write and read misses, so let’s analyze the **Write misses** first:

We can see that the Write misses count from the two programs are almost the same, being the ones in the transposed slightly higher than the ones in the normal multiplication. This is due to the transposed code needing to make more writes (first the transposed of the matrix and then the multiplication result) than the normal multiplications (which just needs to store the multiplication result).

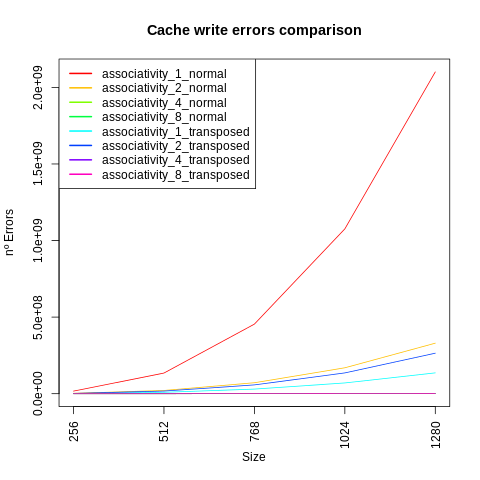
Now let's analyze the Read errors. Here we have a more significant distance between the amount of misses that the normal and the transposed make, generating the normal one more misses than the transposed. The cause is that while in the normal algorithm we visit for each first matrix’s row all the columns of the second matrix, in the second one we just visit each column one time to make the transposed and after that we access both matrix by rows, this, as explained in exercise 2, means a larger amount of misses in the normal algorithm and also a bigger difference of misses when we increase the size of the matrix.

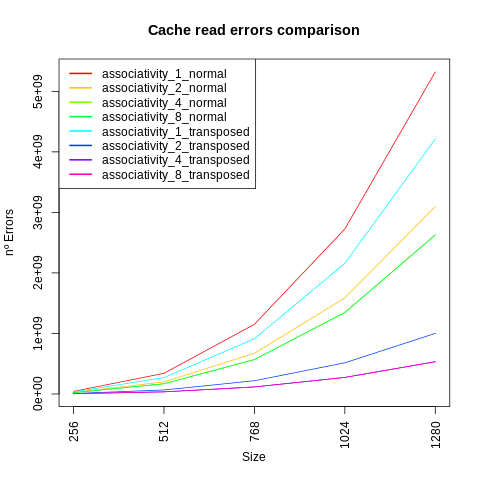
**Exercise 4:**

This exercise is about making experiments to see the different variations of time and cache errors of the matrix multiplication algorithms of exercise 3 when changing the properties of the cache used. We developed two different experiments to check the results:

Experiment 1:

For the first experiment we decided to change the associativity of a matrix that increases in size while maintaining the cache size. Our hypothesis for the result of this experiment is that as we increase the cache associativity the read and write misses will lower

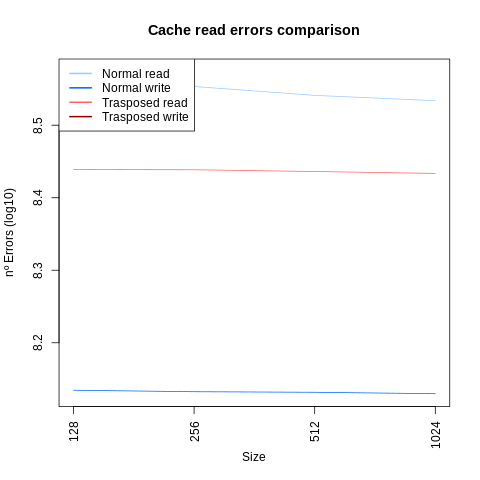


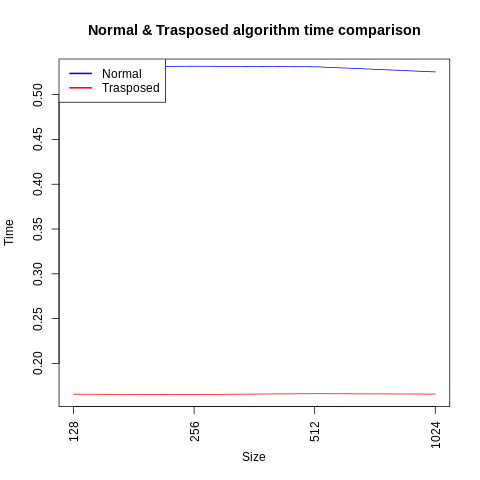


As we can see, as the cache associativity increases the reading and writing misses rate decreases. As we use n-way associative caches for this experiment, the more entries the cache has the more common it is for the block to be in the set, so there are less misses in general. We can also tell (as indicated in previous exercises) that the bigger the matrix, the more errors we get but it seems that when we increase the associativity, we are able to reduce those errors exponentially

Experiment 2:

For the second experiment, we decided to change the cache size while maintaining the same matrix size. Our hypothesis is that the bigger te cache, the less misses we will get and the more time it will take to execute because we will be able to store more elements of the matrix but it would be harder to find them (as explained in exercise 2)





As it’s possible to see by the graphic, our hypothesis was not entirely correct. We can observe that while we increase the cache size (x index), the amount of reading errors decreases slightly, nevertheless, both these and the time seem to stay stable through the cache modifications. We think this could be due to the sizes of the cache that we are testing, since due to time constraints, we decided to reduce them to go from 128B to 1024B. It would be interesting to retry the experiment with bigger caches and a bigger difference between them to obtain the real result.

**Conclusions:**

With this practice it’s possible to understand the way caches work and the way our processors use them. It’s also possible to learn ways to optimize code so the cache is more efficient in terms of misses or time.