1. Amdahl's law

$$Speedup = \frac{1}{1 - p + \frac{p}{n}}, \tag{1}$$

where p is the percentage of the program that can be parallelized, and n the number of processors that run the parallel part of the program.

a) Let $p = 0.9999$, $n = 200000000$, then the speedup is:

$$S = \frac{1}{1 - 0.9999 + \frac{0.9999}{200000000}} = 9999.5 \tag{2}$$

The maximum speedup ($\lim_{n\to\infty}$) for $p = 0.9999$ is:

$$S = \frac{1}{1 - 0.9999 + 0} = 10000, \tag{3}$$

as $\lim_{n\to\infty} \frac{p}{n} = 0$.

The speedup for different amounts of cores can be seen in Table 2.

Cores	Speedup
20	19.962
200	196.098
2000	1666.806
200000	9523.855
2000000	9950.254
200000000	9999.500

Table 1: Speedup for $p = 0.9999$ and different amount of cores n.

b) Communication operation O_c, that scales proportionally with the number of cores as $0.01n$. The serial portion therefore is $s = 0.01 + 0.01n$ and the parallel portion becomes $p = 1 - s = 0.99 - 0.01n$. The speedup then becomes:

$$S = \frac{1}{1 - (0.99 - 0.01n) + \frac{0.99 - 0.01n}{n}} = \frac{1}{0.01(n+1) + \frac{0.99 - 0.01n}{n}} \tag{4}$$

In order to find the maximum we derive the speedup Equation 4 and set it equal to zero:

$$\frac{dS}{dn} = \frac{d}{dn}\left(\frac{n}{0.01n^2 + n + 0.99 - 0.01n}\right) = \frac{d}{dn}\left(\frac{n}{0.01n^2 + 0.99n + 0.99}\right) = \frac{9900 - 100n^2}{(n^2 + 99n + 99)^2} = 0 \tag{5}$$

Solving for n:

$$\begin{aligned} 0 &= \frac{9900 - 100n^2}{(n^2 + 99n + 99)^2} \\ 100n^2 &= 9900 \\ n^2 &= 99 \\ n &\approx 9.95 \end{aligned} \tag{6}$$

When pluggin in the numbers we find out that 10 is in fact the optimal number of cores, leading to a speedup of $S = 5.025$. For $O_c = 0.001n$ we have:

$$\begin{aligned} S &= \frac{1}{0.01(1+0.1n) + \frac{0.99-0.001n}{n}} \\ \frac{dS}{dn} &= \frac{990000 - 1000n^2}{(n^2+9n+990)^2} \\ n &\approx 31.464 \end{aligned} \tag{7}$$

The maximum speedup then is $S = 13.901$ and $n = 31/32$, which we confirm by calculating the speedup for all values. The results for the speedup with $O_c = 0.01n$ and $O_c = 0.001n$ can be seen in Figures 1 and 2 respectively.

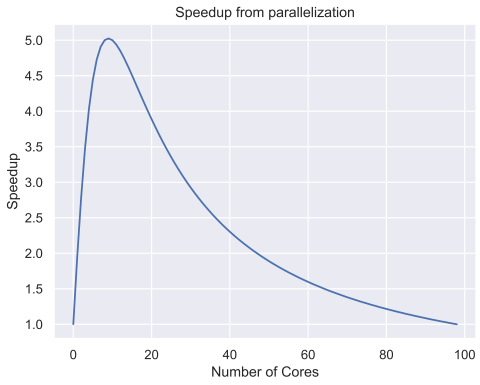

Figure 1: Change of speedup in dependence of number of cores for $O_c = 0.01n$.

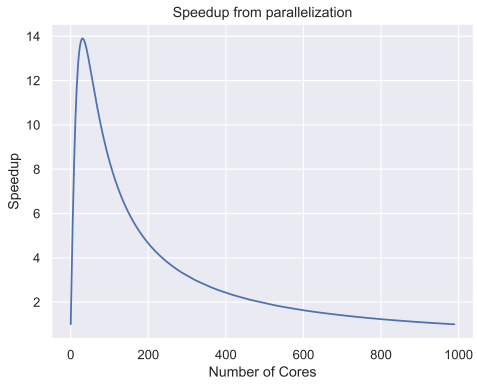

Figure 2: Change of speedup in dependence of number of cores for $O_c = 0.001n$.

c) The total execution time is $T(n) = 10t_1 + \frac{1000}{n}t_1$. For $n = 10$ cores, the execution time is $T(10) = 10t_1 + 100t_1 = 110t_1$. The speedup gained from the parallelization is $S(10) = \frac{T_s)}{T_p(10)} = \frac{1010t_1}{110t_1} \approx 9.18$.

We assume the word "more" means additional and not a redistribution over the $1000t_1$ in the problem statement, therefore, the total number of parallizable operations becomes ≥ 1000. For a load of 1.5 and 3 on one core we have the following speedups respectively:

$$\begin{aligned} S &= \frac{1060t_1}{10t_1 + 150t_1} \approx 6.63 \\ S &= \frac{1210t_1}{10t_1 + 300t_1} \approx 3.90 \end{aligned} \tag{8}$$

2. Linear Algebra Operations

a) The row-major implementation is faster because we have better memory access patterns. This only comes into effect after a certain size of the matrix. The gap between the row-major and column-major implementation becomes bigger as the dimension $N \times N$ of the matrix increases. For one we can optimize the row-major implementation and save constants in the outer loop, secondly we also have more cache hits thanks to the better access pattern. Noticeable is that there is a jump around 512×512, which is where our problem doesn't fit fully into the (L3) cache anymore.

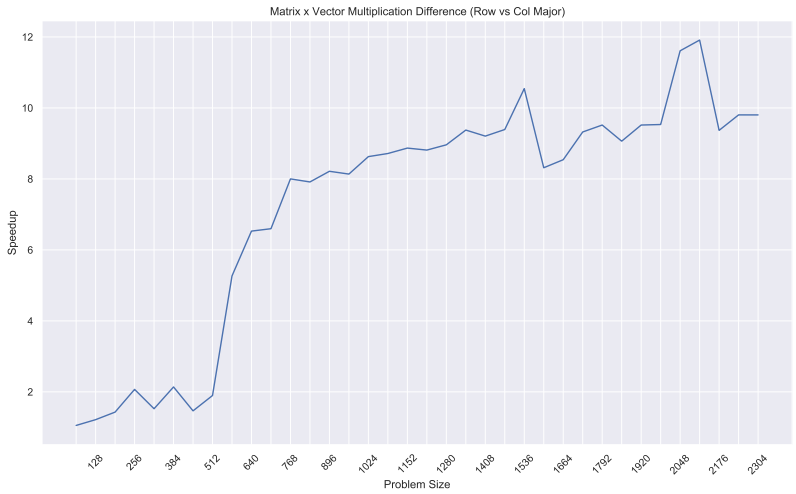

Figure 3: Speedup of row major versus column major implementation for matrix $\times$ vector multiplication.

b) When running the matrix transpose we see that there is an optimal block size, namely size 8. If we choose smaller or bigger blocks than that, we'll have worse performance. The performance of the different block sizes can be seen in Figure 4. The result is not surprising, as storing the blocks of size 8 in the cache will use $2 * 8^2 * sizeof(double)$ space, which is 1 KiB. Even though a bigger block size would still fit in cache, we assume that the reason this block size is better in performance is because of the compiler optimizations (we compile with $-O3$), like prefetching.

c) When running the matrix matrix multiplication we see that there is an optimal block size, namely size 16. If we choose smaller or bigger blocks than that, we'll have worse performance. The performance of the different block sizes can be seen in Figure 5. The result is not surprising, as storing the blocks of size 8 in the cache will use $3 * 16^2 * sizeof(double)$ space, which is ≈ 6 KiB. Even though a bigger block size would still fit in cache, we assume that the reason this block size is better in performance is because of the compiler optimizations (we compile with $-O3$), like prefetching. As we expected, there is an increase in performance when we store the second matrix in column-major instead of row-major. The advantage of this is that we can then access the matrix elements as contiguous memory. This leads to better access patterns for the cache and therefore fewer cache-misses compared to the naive implementation or the regular block matrix matrix multiplication.

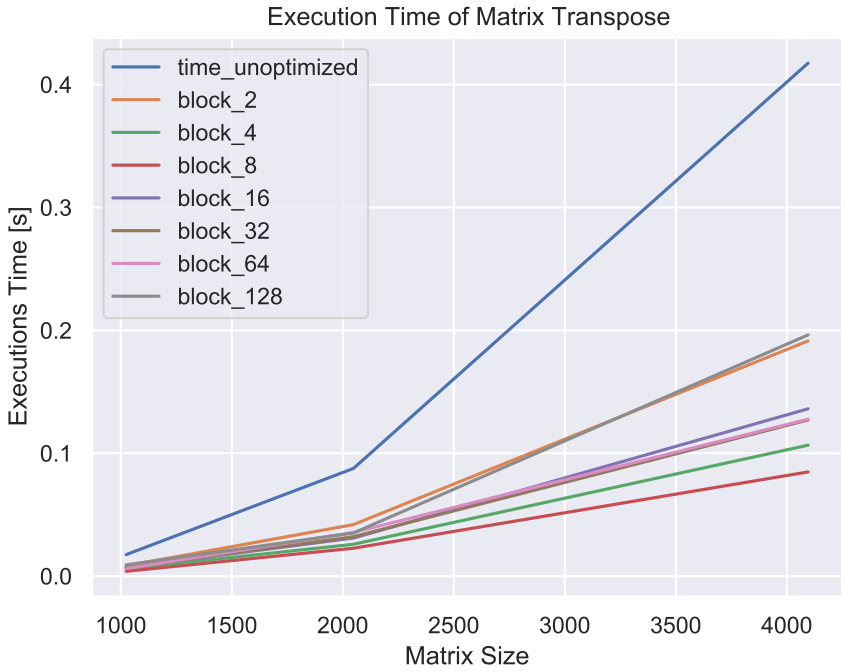

Figure 4: Execution time of matrix transpose for naive transpose and matrix blocks of various sizes. Measurements are taken for matrix sizes 1024, 2048 and 4096.

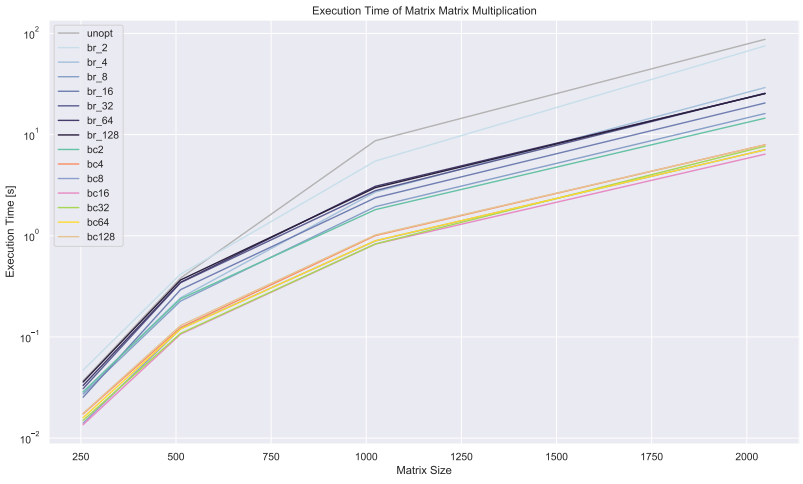

Figure 5: Execution time of matrix matrix multiplication for naive multiplication and matrix blocks of various sizes. $unopt$ is the measurement for the naive implementation, bc are measurements for $A * B^T$, and br are measurements for $A * B$. Measurements are taken for matrix sizes 256, 512, 1024 and 2048.

3. Cache size and cache speed

Running the command from the exercise sheet on the Euler login node we find the cache sizes of the CPU, which are listed in Table 2.

Cache	Size	Coherency Line Size
L1	32KB	64
L2	256KB	64
L3	6144KB	64

Table 2: Cache sizes and cache line size on Euler login node. Note that the L1 is twice the size, as there is one L1 cache for data and one for instructions.

When running our program for the different array access patterns, we get the operations per second shown in Figures 6 and 7. For the code compiled with $-O3$ it is noticeable that the random access has a "burn-in" phase, as we don't have any cache prefetching and start (presumably) with a cold cache. The behaviour when our array gets big in size seems very random and the only distinct trend we notice is that the number of operations decreases steadily for the sequential accesses. In the case of $-O0$ we have a more stable picture. The 4 byte strided access and the random access are both roughly equal over time and have a slight decrease in operations per second as the array size increases. There is not a very significant drop when the array size passes over a cache size and we have a smooth transition. We assume this is because for the bigger size we can still prefetch cachelines, which means that there will be a slightly higher fetching time, as we are in a higher level cache or in RAM, but mostly the operations will still have the same time as we have a lot of cache hits. The two methods are comparable, as we're frequently making cache hits with the random method. The 64 byte strided access has the expected sudden drops in performance when we exceed the sizes of the cache levels, so it is a sharp transition. The reason for this is that we have to fetch a new cache line every time we do an operation, so the increase in cache line fetching time will lead to an overall worse operations per second performance. The drops match the cache sizes we found in 2. Foremost the change from the L1 to L2 and L3 to RAM have large gaps as when our array doesn't fit in the according memory anymore. For very large N the random access will become continuously worse as we will get less cache hits. The same holds for the 64 byte strided access, as the bigger our array gets, the more memory we need and henceforth also will save our data in higher level memory. This will lead to worse performance as we need to search further in the memory hierarchy until we find our data.

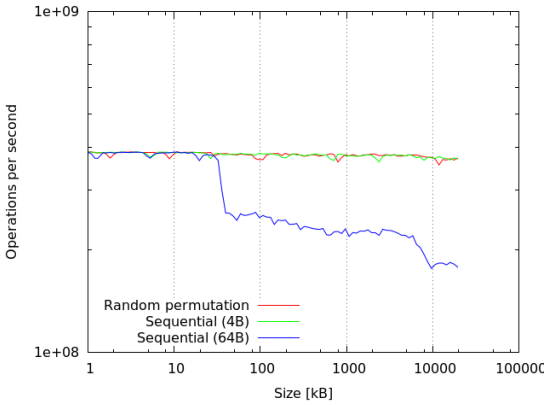

Figure 6: Number of operations for the three different access patterns on the Euler cluster in dependence of the array size. Compiler flag $-O0$.

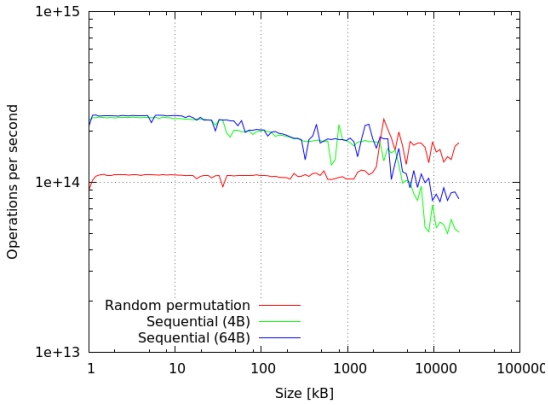

Figure 7: Number of operations for the three different access patterns on the Euler cluster in dependence of the array size. Compiler flag $-O3$.