HPPS - Exam 2022

Exam number 138

January

Contents

1 Introduction **2**

2 Runtimes of transpositions **2**

3 Runtimes of transpose_blocked() for various values of T **4**

4 Speedup of parallel vs sequential wrt. number of threads **5**

5 Runtimes of multiplications **6**

6 Explain the differences for multiplications **7**

7 Speedup of parallel matrix multiplication compared to sequential functions **8**

8 How was the algorithms parallelized? **8**

9 matmul_locality vs matmul_transpose **9**

10 Speedup of matrix multiplication as a function of utilized cores **9**

1 Introduction

This exam was all about designing different Matrix-based algorithms. Generally I think my implementations of the expected algorithms are quite good. I have, as you will see in this report, run quite a lot of tests, mostly getting the expected results. The main problems I can see are:

I have done most of the work on a 4-core laptop which made both benchmarking and multithreading suboptimal. Some of the test was executed by use of MODI and it will be clearly told when I have done so.

The `transpose_blocked_parallel` function is slower than I would have expected - this either points towards a wrongful implementation or a personal lack of intuition on what to expect. This is discussed in the corresponding section.

To analyse the performance I have written some functions which tests the executions and outputs the results as .csv files,[1] but due to time constraints these are more hard-coded than I would have liked.

Additional comments:
All timings are given in microseconds.
Running `omp_get_num_procs` from the `OpenMP`-library on my computer tells me I have 5 available cores, contradicting the 4 shown to me by my operating system. I have not been able to understand this discrepancy using the first pages of Google search results, so all 4-core `parallel` functions are to be seen with an asterisk.
To simplify the coding needed I only benchmarked using square matrices. I have tested the functions for NxM matrices (where N!=M), but they are not used in the experiments that can be seen in this report. Further work would have included jagged matrices.
I have used the `valgrind` command to minimize memory-leaks.
And lastly, for completeness sake, these are my CPU specifications:

$$\text{Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, 2601Mhz, 4 Core(s)}$$

2 Runtimes of transpositions

Firstly I had to find the optimal value of T. This resulted in the following plot, where I have tested multiple transpositions for NxN-matrices for $T = 1, 2, .., T/2$:

[1] Never send a human to do a machine's job.

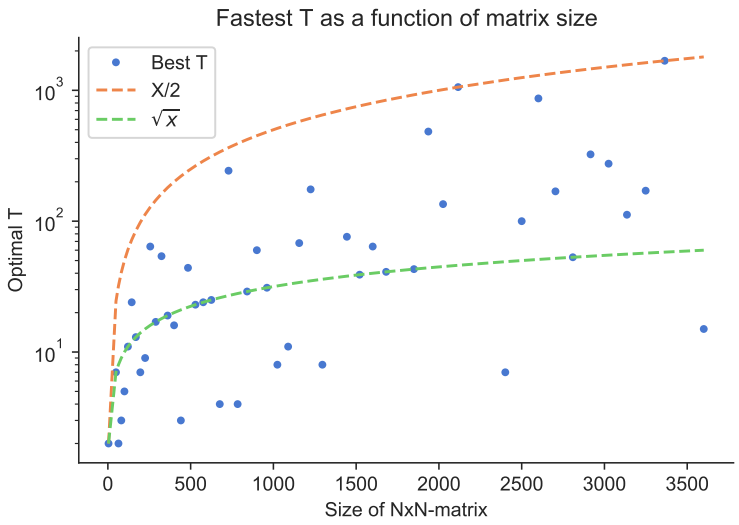

Figure 1: The optimal T-value for `transpose_blocked` as a function of matrix size.[2]

What I found was that the optimal T-value vaguely followed a square-root function, with the most points lying on this actual line. I suspect these are the matrices with square numbers as side lengths. Otherwise, the optimal T seems quite arbitrary, so I chose $T = 100$ semi-randomly[3] and went with that.

Now, having implemented the different transposition algorithms and making sure they were working correctly, I had to benchmark them. I designed the main experiment as follows: As we are looking at $T = 100$ for the transposition algorithm, I created matrices with multiples of this number as side lengths. I then ran a modified version of the benchmarking functions from the hand-out, doing ten repetitions each time and taking the average. The final results can be seen here:

[3] I wanted a number that was not too small as to have no effect, but also not so large that there were few multiples in the ranges I would be looking at.

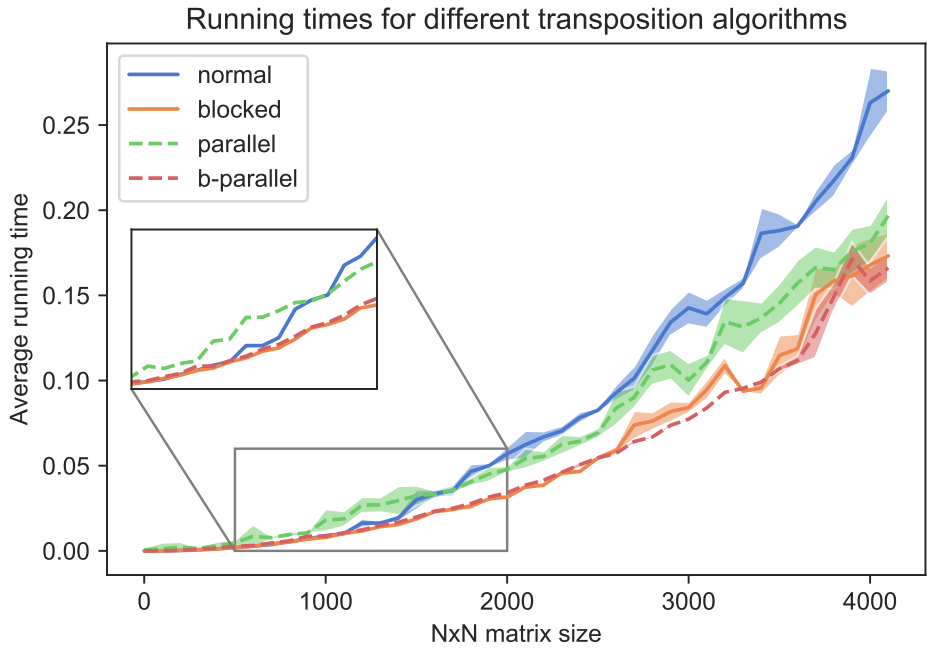

Figure 2: Parallel implementations are dotted, sequential are filled.

Okay - a couple of things to note: I ran the experiment ten times for each size, taking the average. The uncertainty[4] is also displayed as the semi-transparent outline. I am mostly not surprised with the results: For larger matrices, the naive implementation is slowest, beaten by the parallelization with the 'blocked'-algorithms being the quickest. What I did not expect was that the `blocked_parallel` is slower than its sequential counterpart. As this discrepancy was consistent across multiple runs and T-values, it makes me question my implementation of the parallelization.

3 Runtimes of transpose_blocked() for various values of T

If you are wondering how I found the optimal value for T in figure 1, an example will follow: I did so by transposing matrices of consistent size[5] multiple times using every legal value of T and then finding the quickest. I have a background in physics, so seeing a plot without uncertainties makes my eyes twitch. Taking the matrix and for every legal T making 5 runs of 10 transpositions allows me to estimate the uncertainty for every run, even though the error seems underestimated for some T-values. An example run can be seen here:

[4]Taken as the empirical standard deviation

[5]The size in this example was chosen to be 5040, as it is both a 'highly composite number' meaning it has many divisors and in the right order of magnitude to make testing straightforward and telling.

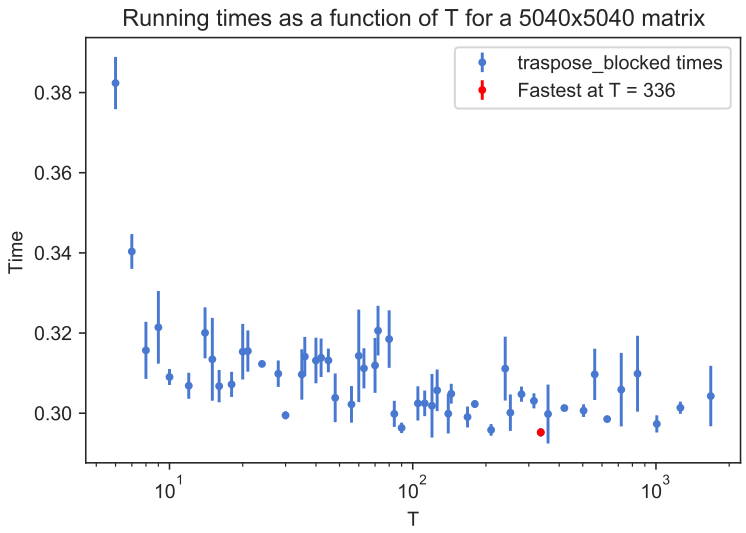

Figure 3: Note that the errorbars seem too small sometimes. I believe my use of statistics should be accurate, but denial is the most predictable of all human responses, as they say.

I am supposed to answer *Why does T affect performance?* From what I can visualize, the beauty of blocking comes from an improved temporal locality. This happens as the same 'block' is queried multiple times before moving on to gathering entries from the matrices. This corresponds to T needing to be in a sweet spot where it is neither too large as to negate the effects of having sub-matrices and not so small as to needlesly complicating the transposition algorithm: For T = 1, nothing has changed from non-blocking, for example.

4 Speedup of parallel vs sequential wrt. number of threads

In our case, the speedup is simply defined as $S = \frac{\text{sequential}}{\text{parallel}}$. This means that a speedup above 1 corresponds to the parallel implementation being quicker. In the following plot the speedup at different matrix sizes can be seen for 2 and 4 cores respectively:

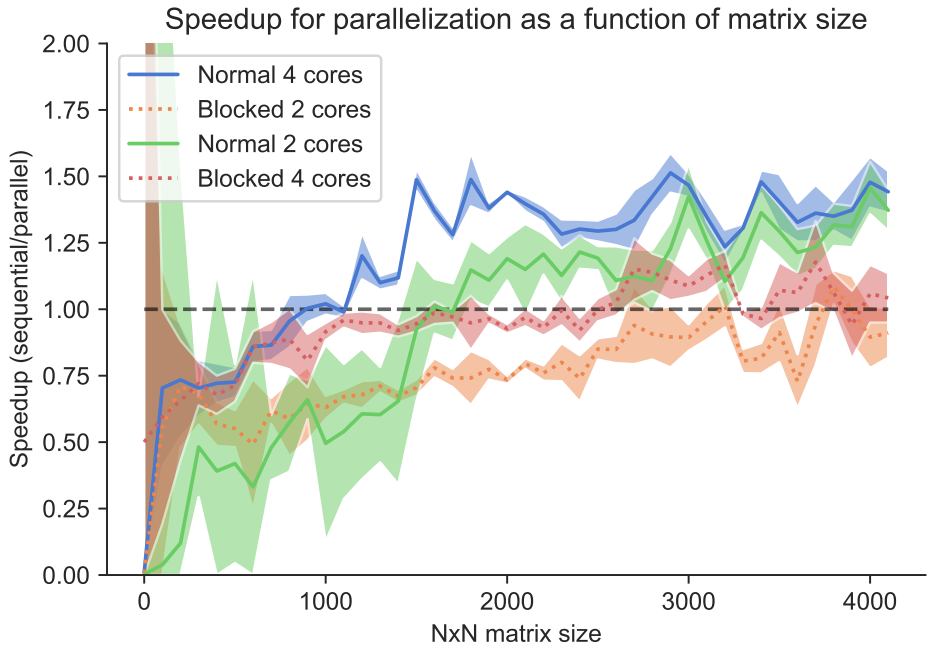

Figure 4: The speedups as a function of matrix size. Note that dotted lines correspond to the `blocked` implementations. The displayed errors are of course correctly propagated from the individual uncertainties.

As can be seen in figure 2, It is only for matrices larger than 1000x1000 that the parallelization has any positive effect on the naive algorithm. It only happens for the `blocked` algorithm at the edge of my experiment. Given more time (or a better computer) I would have loved to run the tests for even bigger matrices. Without inducing a déjà vu I would like to comment again that this makes me question the parallelization of the `blocked`-algorithm. It is clear that an more available cores corresponds to a better performance for the same matrix size which shows evidence of weak scaling.

To get a comprehensive result, I should have run this on MODI giving me access to more than the 4 cores on my computer. In section 7 it can be seen that a thread increase almost guaranties a speed increase. Here the scaling of the problems will also be discussed further.

5 Runtimes of multiplications

Now, to show the runtimes of all my matrix multiplication functions I did about the same as for the transpositions:

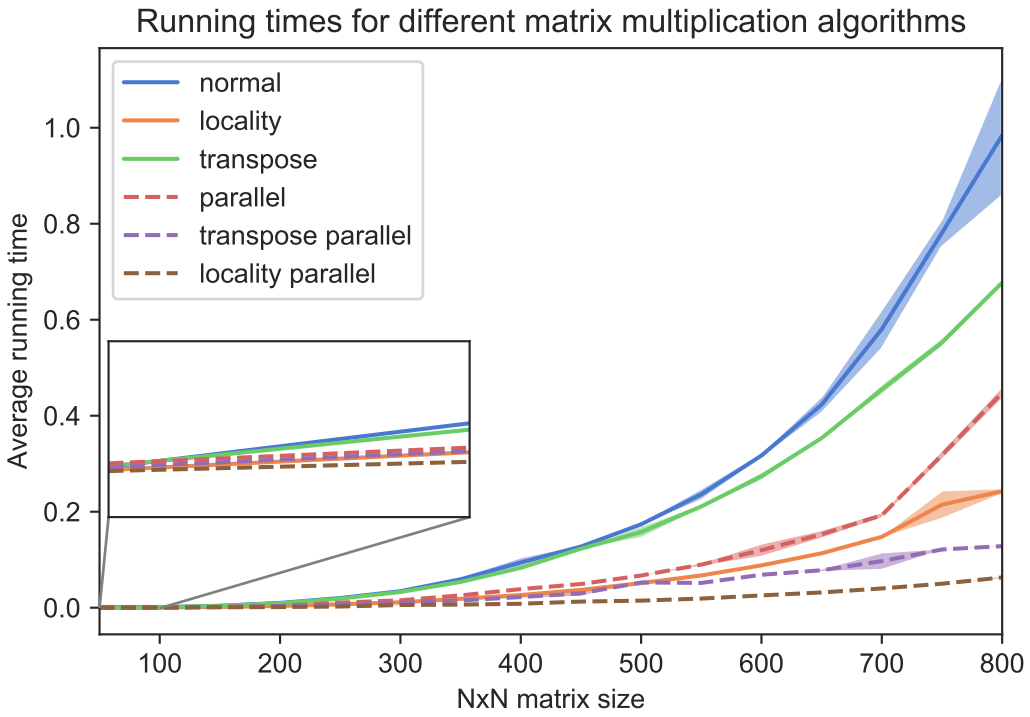

Figure 5: A summary plot like figure 2. Again, dotted lines corresponds to `parallel` implementations.

The results of the `locality` are oddly impressive - you really weren't lying to us!

The three parallel implementations follow the same trends; parallelizing wins about a factor five in speed (for larger matrices, at least). The simple solution is slow, the transposition is better, and `locality` is clearly the best.

6 Explain the differences for multiplications

Okay, here goes: `matmul()` is written expressing complete naïvité. This is the baseline implementation. `matmul_locality()` is more sophisticated, as it uses the ideas taught to us by Troels about locality. Concisely: We make sure that for every iteration in the inner loop, the data we are accessing lies relatively close to each other in physical hardware space. Apart from the sequentially local data, the same variable is used multiple times allowing this to be kept in the cache for longer, minimizing cache misses, improving temporal locality. It can be seen that this grants a very notable improvement in performance.

The transposition has an initial cost (of having to transpose the matrix first, obviously), but after that, has an improved locality, ultimately making it consistently faster than the baseline, which is especially apparent for larger matrices.

For a vizualisation, see the speed-up plot in the next section.

7 Speedup of parallel matrix multiplication compared to sequential functions

As there will be a more focused analysis of the core-dependency in a later sub-question, I simply plot the latency speedup for using my 4 available cores:

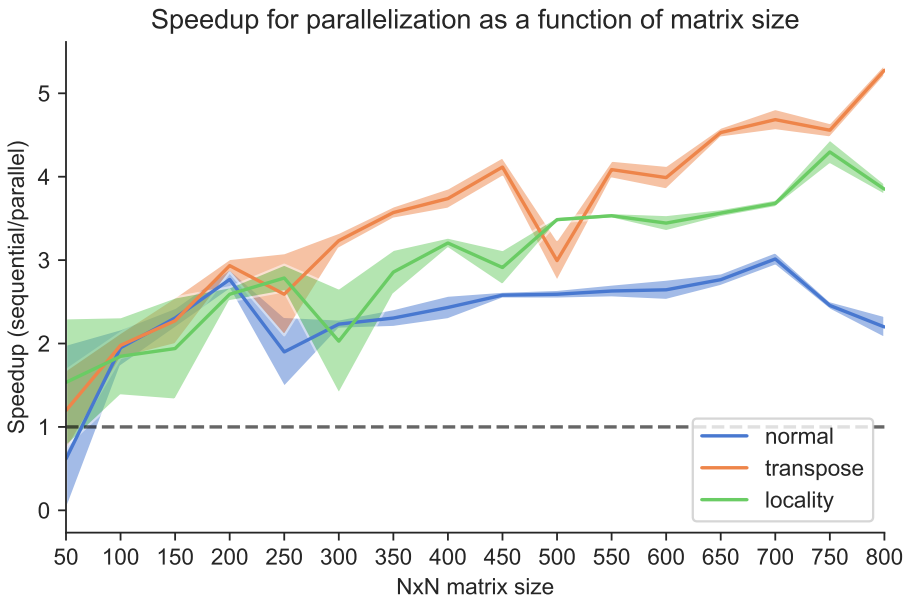

Figure 6: Caption

While the transposition algorithm has a weird quirk at around N=500, this looks about as expected. The reasoning for my expectations will be discussed in section 9.

8 How was the algorithms parallelized?

This feels stupid to say, but all I did was parallelize the outer loops of each implementation using `#pragma omp parallel for`. I really wanted to something more advanced, so I tried implementing the `#pragma omp parallel for reduction (+: sum)` in the `matmul_parallel`, but it only worsened my results.

I also see no reason to do anything dynamically and gained no additional speedup from doing so, so this was not implemented. But that might just be me who is still not confident in understand the inner workings of multiprocessing. Ignorance is bliss, I guess.

I was very aware of avoiding race conditions, but it did not seem that that would be a problem in the way I applied the parallelizations.

9 matmul_locality vs matmul_transpose

From my findings I would say, that `transpose` can take advantage of the parallelism more than the `locality`. This makes sense, as the difference between `transpose` and `locality` is, that `transpose` utilizes improved locality by use of doing additional loops. While this usually means it is less optimized, the loops are more ripe for parallelization, granting a higher speedup. I believe my data in figure 6 could suggest that this happens in practice.

10 Speedup of matrix multiplication as a function of utilized cores

To show how the speedup changes with the number of threads used I uploaded a batch job to MODI and ran the `matmul_transpose` for multiple matrix sizes. The measurements can be seen here:

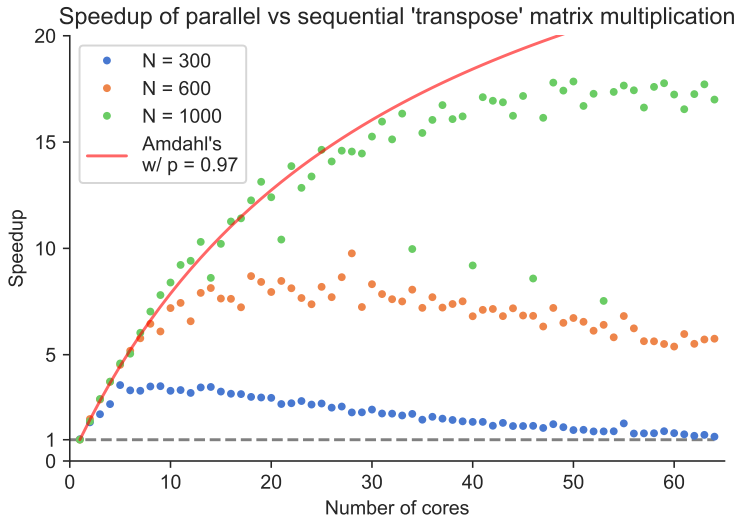

Figure 7:

I have chosen to plot an Amdahl-prediction along with the empirical evidence.

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}} \tag{1}$$

This is not a fit - I simply eyeballed the p-value. Apart from being an indication of the problem scaling strongly, a p of about 97% in Amdahl tells us something about the fact,

that pure matrix multiplication and transposition are very parallelizable![6]

I also wanted to test for weak scaling, so I designed an experiment where the throughput was consistent per number of used core. For doing the test (called `modi2` in my code), I scaled the number if entries with the number of cores. The smallest matrix was 200x200. The speedup between the parallel and sequential implementations can be seen here with Gustafson's law plottet on top:

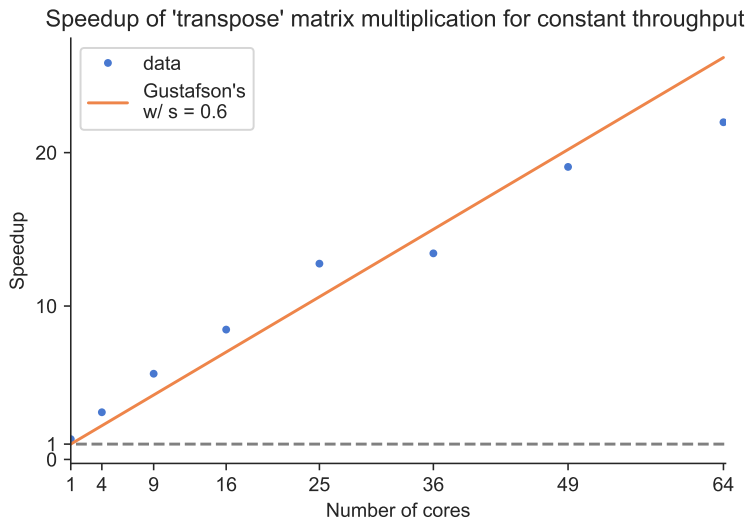

Figure 8:

Even though the results are not exactly linear, I believe we see weak scaling of the problem. Especially the first 5 data points are very consistent with this hypothesis. I would have loved to have created more data points, but I did not have the time. There is a difference between knowing the path and walking the path - and the latter is more time consuming, unfortunately.

[6] yes, that's a word