Assignment 1: High Performance Dense Linear Solver

Christian Wiskott, 01505394.

April 19, 2020

Part 2: Unblocked Right-looking LU Factorization

In this part of the assignment, the goal was to write an efficient algorithm for calculating an unblocked right-looking LU-Factorization *without pivoting* for a randomly generated, non-singular matrix $A \in \mathbb{R}^{n \times n}$. With this factorization, A is decomposed into an upper triangular matrix U and a lower triangular matrix L, such that $A = LU$. The main idea is to apply Gauß elimination to A, to eliminate the elements below the main diagonal, which transforms A into the upper triangular matrix U. The elements of the matrix L are the factors computed during the Gauß elimination.

The matrix A was generated using a uniform distribution featuring randomly chosen values within the interval of $[90, 100)$.

Performance Evaluation

In order to evaluate the performance of the written code, the run-time was measured, such that only the actual computation of the resulting matrix U was included in the measurement. This computation required three explicit for-loops which scale with $\mathcal{O}(n^3)$ and can be examined in the provided code.

During the computation only one matrix U was used, instead of creating the two matrices U and L. This was made possible by overwriting the given matrix A and adapting the last for-loop, to not overwrite the coefficients otherwise being in L. This choice was motivated by reducing the necessary memory during the computation, since only one matrix had to be stored, but had no significant effect on the run-time of the algorithm.

Fig. 1 shows the run-time analysis of the LU-factorization for problem sizes within the range of [100, 10000]. The data has been fitted with a cubic function, which shows (with high correlation) a cubic increase w.r.t the matrix dimension, which coincides with the expected $\mathcal{O}(n^3)$-scaling. Due to this scaling, the required computation time increased very rapidly and reached more than 300 seconds for size 10k $\times$ 10k. This led to the termination of the evaluation, since higher dimensions would result in exceptionally long run-times.

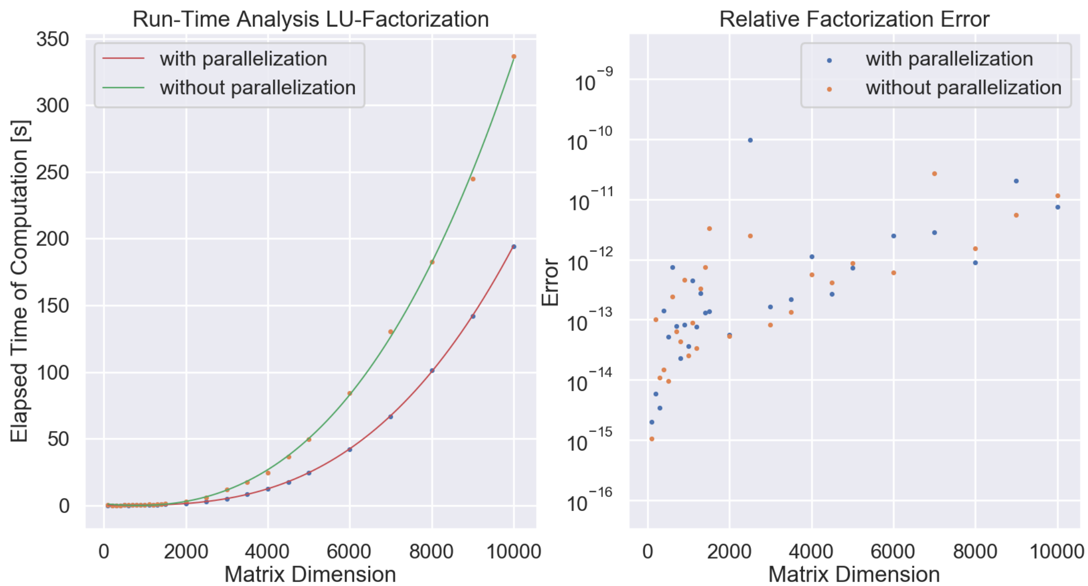

Figure 1: Run-time and Error Analysis

To minimize the run-time of the algorithm, the Python module $<$ *numba* $>$, which translates the code into much faster machine code, was used. The code was adapted, such that it was possible to use parallelization (via the parallel=True argument and the prange-function, which indicates *numba* where to apply parallelization) and thus, utilizing all cores of the CPU instead of just one. This resulted in a constant CPU-load of 100% during the computation and increased the computation speed by a factor of approx. 1.5 . The comparison between running the code with and without parallelization, is visualized in fig. 1.

Fig. 1 also shows the relative factorization error of the computed solutions w.r.t the matrix dimension. Both, the parallelization and the non-parallelization method, feature comparable factorization errors. At the maximum dimension of 10000, the results still feature relative factorization errors of $\sim 10^{-11}$. Since the accuracy of both methods are approximately equal, the parallelization method is superior, due to its lower run-time and more efficient use of CPU-capacity.

In order to provide a reference value for the run-time, the code was executed on the server the university provided. Fig. 2 showcases the results. Since the server doesn't feature the module *numba*, the non-parallelization method was evaluated and completed the task for size 5000×5000 in around 340 seconds. The direct comparison of run-times of the non-parallelization method of the server and the user-case (which included the *numba*-module and needed 49 seconds), shows that it completed the task approx. seven times faster. Since two different hardware-setups were used, this value should only be taken as a approximation of the optimization of *numba*, but can be seen as an improvement since other evaluations of identical code, have shown that the user-setup was generally only twice as fast as the server.

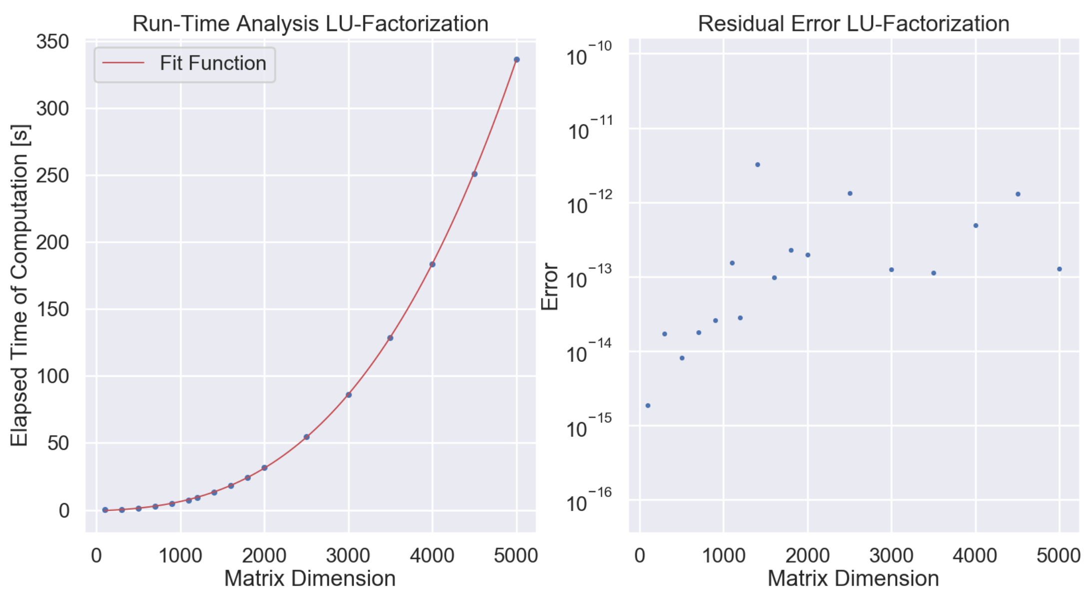

Figure 2: Run-time and Error Analysis using the provided university server