PARALLELIZING GEMM AND LU FACTORIZATION

Fabian Bösiger, Georg Streich, Bartlomiej Frydrych, Benjamin Simmonds, Roger Csaky-Pallavicini

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

This report presents an investigation into the implementation of efficient parallel algorithms for general matrix multiplication and LU factorization. We focus on utilizing parallel programming models to achieve high performance on multicore systems for both tasks. Additionally, we employed various single-core optimization techniques such as loop unrolling and cache blocking to enhance the efficiency of our algorithms.

In order to evaluate the performance of our parallel implementations of matrix multiplication and LU factorization algorithms, we conducted benchmarks on ETH's Euler system with a variety of configurations and core counts. We compare the runtime and scaling characteristics of our implementations with that of popular linear algebra libraries.

1. INTRODUCTION

Motivation. General matrix multiplication (GEMM) and LU factorization (LUDCMP) are fundamental linear algebra operations that are widely used in many scientific and engineering applications, such as computer graphics, machine learning, and scientific simulations. Due to the limited growth in single core performance, it becomes increasingly important to have scalable parallel implementations of these algorithms which can efficiently use the rapidly growing number of cores available both on consumer machines and HPC clusters. It is worth noting that there exist a variety of highly efficient and well-established parallel implementations for both matrix multiplication and LU factorization algorithms.

PolyBench is a popular benchmark suite of over 30 numerical computations with static control flow [1]. The aim of this project is to create parallel versions for the LUDCMP and GEMM PolyBench kernels using popular parallel programming paradigms. We utilize OpenMP, an API that supports shared-memory multiprocessing [2], and MPI, an API for message passing in parallel computers [3].

Through benchmarking and profiling our code, we aim to improve CPU utilization and address scalability bottlenecks. We compare the performance of our results to the commonly used implementations of Basic Linear Algebra Subprograms (BLAS) [4] such as the Intel Math Kernel Library (MKL) [5] and OpenBLAS [6].

2. BACKGROUND

General Matrix Multiplication. Given matrices $A \in \mathbb{R}^{n,k}$, $B \in \mathbb{R}^{k,m}$ and $C \in \mathbb{R}^{n,m}$ general matrix multiplication computes a matrix $C' \in \mathbb{R}^{n,m}$ such that $C' = \beta C + \alpha AB$. A naive algorithm to compute it is given in algorithm 1. We note that computation is usually done in-place, meaning that C' is stored in the memory that was previously occupied by C.

Algorithm 1 Triple loop implementation of GEMM

```
for $i \leftarrow 0$ to $n$ do
    for $j \leftarrow 0$ to $m$ do
        $C[i][j] \leftarrow \beta C[i][j]$
        for $h \leftarrow 0$ to $k$ do
            $C[i][j] \leftarrow \alpha A[i][h]B[h][j]$
        end for
    end for
end for
```

LU Factorization. Given a matrix $A \in \mathbb{R}^{n,n}$ the LU factorization of A consists of a lower triangular matrix $L \in \mathbb{R}^{n,n}$ and an upper triangular matrix $U \in \mathbb{R}^{n,n}$ such that $A = LU$ is satisfied. We note that such a factorization is not always possible in the general case. Usually a permutation matrix P is introduced, and the factorization is changed into the following form $PA = LU$, which exists for any non-singular matrix. As handling pivoting introduces a lot of complexity, PolyBench/C restricts the problem to diagonally dominant matrices, e.g. matrices with the following property

$$|A_{ii}| \geq \sum_{j \neq i} |A_{ij}| \text{ for } i \in \{1, \ldots, n\}$$

As for any such matrix a LU factorization without pivots exists. In this case, one might compute the LU factorization as in algorithm 2. We note that here the computation is done in-place as well, namely L, U will be stored in the memory that was previously occupied by A.

Algorithm 2 Simple LU decomposition without pivoting

```
for $i \leftarrow 0$ to $n$ do
    for $j \leftarrow 0$ to $i$ do
        for $k \leftarrow 0$ to $j$ do
            $A[i][j] \leftarrow A[i][j] - A[i][k] * A[k][j]$
        end for
    end for
    for $j \leftarrow 0$ to $n$ do
        for $k \leftarrow 0$ to $i$ do
            $A[i][j] \leftarrow A[i][j] - A[i][k] * A[k][j]$
        end for
    end for
end for
```

PolyBench/C has kernels for both LU factorization and solving linear systems with LU factorization and back substitution. Our implementation handles both tasks, but since LU factorization dominates the running time (with a complexity of $O(n^3)$ versus $O(n^2)$ for back substitution), we primarily optimized this component. It's worth considering that for the distributed case, back substitution would ideally also be implemented in a distributed way, so that one can avoid gathering results on a single node.

3. PROPOSED METHOD

General Matrix Multiplication. To speed up the initial triple loop implementation used by PolyBench we applied many of the techniques that are commonly used to implement optimized matrix multiplication routines. Namely, we added blocking to improve cache performance [7], register blocking to increase instruction level parallelism and added vectorization.

OpenMP. In a shared memory setting, we can parallelize GEMM by simply distributing iterations of the outermost loop to the available processors. With OpenMP this can easily be achieved with the `#pragma omp parallel for` directive.

MPI. Input Matrix A and C is divided into rows using Row-Wise Block-Striped Matrix Decomposition method [8]. Adjacent rows of A and C are assigned to each MPI process p. Rows are scattered evenly across all processes when p divides N evenly. If not, rows are scattered evenly across $p - 1$ process and last process p receives reminder of rows. A copy of matrix B is broadcasted to all processes p. This method of decomposition allows processes to compute it's own part of C independently without a need of interprocess communication during computation.

LU Factorization. Literature research showed that traditional LU factorization is not well-suited for performance-sensitive applications [9]. We hence directly implemented the decomposition as block LU factorization. In this procedure, the described factorization $A = LU$ is executed in the form of block submatrices

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix} \quad (1)$$

In this equation, submatrices L_{11} and L_{22} are unity lower-triangular matrices, while U_{11} and U_{22} are upper-triangular. L_{21} and U_{12} are rectangular.

The respective decomposition can be found in four steps: First, the submatrix A_{11} is LU-factorized, resulting in L_{11} and U_{11}. Second, L_{11} is inverted and

$$U_{12} = L_{11}^{-1} A_{12} \quad (2)$$

is calculated. Next, U_{11} is inverted to calculate

$$L_{21} = A_{21} U_{11}^{-1} \quad (3)$$

Last, we define

$$A'_{22} = A_{22} - L_{21} U_{12} \quad (4)$$

with $A'_{22} = L_{22} U_{22}$.

Using the same steps as described above, the blocking strategy can now be used recursively to find A'_{22}. Matrix A should ideally be split up into submatrices until the separate blocks of data fit into one cache. The algorithm was implemented using variable block sizes, which allowed us to efficiently find the ideal block size for our application and infrastructure.

Block Size. To make optimal use of cache characteristics of the target system, the algorithm can be used with different block sizes. The block size determines the size of the submatrix A_{11}, as well as the block size for the matrix multiplications in the computation.

OpenMP. The approach described in section 3 has the additional advantage that it enables us to easily reuse the matrix optimizations described in section 3 for the computation of equations 2, 3 and 4. OpenMP's loop parallelization can be used on the outermost loops. Furthermore, the computations for equations 2, 3 are entirely independent and can be done in parallel. In the following sections, the algorithm using OpenMP is referred to as `ludcmp-openmp`.

AVX Intrinsics. To further achieve performance improvements, 256-bit Advanced Vector Extensions SIMD instructions can be used to take advantage of data level parallelism.

Reusing GEMM Kernel. Our approach for LU factorization computes matrix multiplications for equations 2, 3 and 4. For these steps, we can reuse our already optimized GEMM kernel.

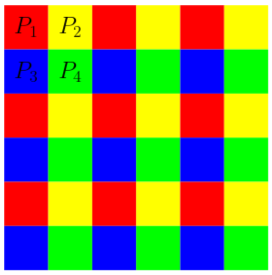

Fig. 1. Block Cyclic Data Distribution
The matrix is split up into small blocks, which are then assigned to processors in a cyclic manner. Here processors 1 gets blocks in red, processors 2 yellow ones, etc.

Further Optimizations. We use loop unrolling to reduce the overhead of loop control computations compared to actual computations. This also increases pipelining efficiency.

To further reduce data dependencies, multiple accumulators are used for reduction operations, for example summing up data over a loop.

When multiplying triangular matrices, multiplying with zero entries can be avoided, which effectively cuts the number of operations in the matrix multiplication in half.

MPI. Our initial approach in creating a parallel implementation of our algorithm involved utilizing MPI to compute the independent equations 2 and 3 in parallel. However, we quickly realized that this approach would not be scalable, as these sub-problems only accounted for a fraction of the overall problem size.

In order to achieve scalability, it is essential to ensure that work is evenly distributed among all processes, as per Amdahl's law. One way to accomplish this for LU factorization and other linear algebra routines is through the use of a block cyclic distribution, as shown in figure 1. This method of distributing data ensures that all processors have a consistent amount of work throughout the execution of the algorithm.

By matching the block size of the block cyclic distribution to the size of A_{11} of our blocking LU decomposition we can adapt our algorithm to a distributed version which is described under algorithm 3. We note that while we use assignments in the pseudocode, updates to B can be performed in place. As this new version is quite similar to our previous implementation, we were able to reuse optimization techniques we have already described. In the following sections, the algorithm using MPI is referred to as `ludcmp-mpi`.

Interleaving Communication and Computation. Communicating between MPI ranks can have a negative impact on performance due to high latency. A commonly used technique to mitigate this issue is to utilize MPI's non-blocking methods, which allow for independent computation to be performed during the time that data is being transferred over the network. Initially, it may seem that this approach is not suitable for our algorithm. However, upon further examination of the data dependencies within our algorithm, we discovered that in the first step, LU_{11} only depends on the upper left block of the previous iteration's result. Similarly, L_{21}, U_{12} only depend on specific parts of $B^{(k)}$. To illustrate this, we have included a dataflow graph in figure 2. By delaying the computation of $B^{(k)}_{21}$, $B^{(k)}_{12}$ and $B^{(k)}_{22}$, we are able to utilize the time required for broadcasts to perform independent computation, improving overall performance.

Algorithm 3 Distributed LU factorization

```
$B^{(0)} \leftarrow$ BlockCyclic(A)
**for** $k \leftarrow 0$ to number of blocks **do**
    **if** $A_{11}$ belongs to us **then**
        Compute $LU_{11}$ as the LU factorization of $B^{(k)}_{11}$
        Broadcast $LU_{11}$
    **end if**
    **if** parts of $A_{21}$ or $A_{12}$ belong to us **then**
        $L_{21} \leftarrow L^{-1}_{11} B^{(k)}_{21}$
        $U_{12} \leftarrow B^{(k)}_{12} U^{-1}_{11}$
        Broadcast $L_{21}$ to ranks sharing our row
        Broadcast $U_{12}$ to ranks sharing our column
    **end if**
    $B^{(k+1)} \leftarrow B^{(k)}_{22} - L_{21} U_{12}$
**end for**
```

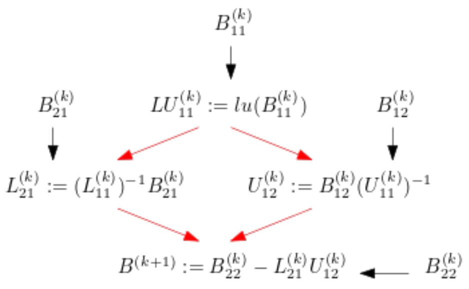
$B^{(k)}_{11}$
$B^{(k)}_{21}$
$LU^{(k)}_{11} := lu(B^{(k)}_{11})$
$B^{(k)}_{12}$
$L^{(k)}_{21} := (L^{(k)}_{11})^{-1} B^{(k)}_{21}$
$U^{(k)}_{12} := B^{(k)}_{12} (U^{(k)}_{11})^{-1}$
$B^{(k+1)} := B^{(k)}_{22} - L^{(k)}_{21} U^{(k)}_{12}$
$B^{(k)}_{22}$

Fig. 2. Dataflow depiction of distributed LU factorization. Edges marked in red denote communication between MPI ranks.

4. EXPERIMENTAL RESULTS

Experimental setup.

All of our benchmarks are run on ETH's Euler computer on EPYC 7763 processors. The EPYC 7763 processor has a 2.45 GHz nominal and a 3.5 GHz peak clock rate, and a L3 cache size of 256 MB.

For compilation, we use gcc version 6.3.0 with the flags `-O3 -march=native` as they have shown to provide the best performance for the final executable. For our benchmarks, we used up to 32 cores, and a single physical node. These numbers arose from the usage limits on Euler. In most cases, we performed 10 runs per configuration, as this usually reduced variance to an acceptable level. Sometimes, a run would contain large outliers, which we attribute to contention with other jobs running on the system. In these cases, we would rerun the job to get more reasonable measurements.

To evaluate the performance of our implementations, we compare them with both the Intel MKL and OpenBLAS libraries. For MKL, we used Euler's `intel/19.1.0` module. As Euler did not have a version of OpenBLAS that supported multiple cores, we compiled OpenBLAS version 0.3.21 ourselves targeting the ZEN microarchitecture using `icc 19.1.0` with `-O3` compilation flags. It is worth noting that our implementation of the LU factorization algorithm did not include pivot selection, whereas MKL and OpenBLAS do. As a result, it is not entirely fair to directly compare the performance of our implementation to that of these libraries. They are solving a more complex problem. That being said, with diagonally dominant matrices, MKL and OpenBLAS should not need to apply any pivoting. This seems to also be the case in practice when one checks the computed pivot indices.

We start timing the implementations after the data has been initialized. In the distributed case, we make sure that processors are synchronized with a call to `MPIBarrier` before starting the timer.

Step by Step Optimizations. Figure 3 shows the impact of the optimizations described in section 3, where the runtime was measured after applying every subsequent optimization. `ludcmp-polybench` refers to the naive implementation from PolyBench. `ludcmp-blocking` implements the blocking approach described in section 3. `ludcmp-blocking-openmp` adds OpenMP parallelization. `ludcmp-blocking-openmp-fma` additionally utilizes vector instructions. For this benchmark, 16 cores were available, however we note that `ludcmp-polybench` and `ludcmp-blocking` only utilize a single core.

MPI Ranks. In figure 4 we show the weak scaling behavior of our MPI LUDCMP implementation when running with a varying number of MPI ranks. That is, we split the available processors among different number of MPI ranks. We note that due to how our algorithm works, this number has to be square. What we can observe is that in order to efficiently utilize more cores, it is beneficial to introduce a larger number of MPI ranks. We were somewhat surprised by this result, as these benchmarks are performed on a shared memory system. MPI is not strictly necessary in such a setting, and seemingly only adds additional overhead. Through profiling the implementation with `perf` we can attribute the difference in performance to higher CPU

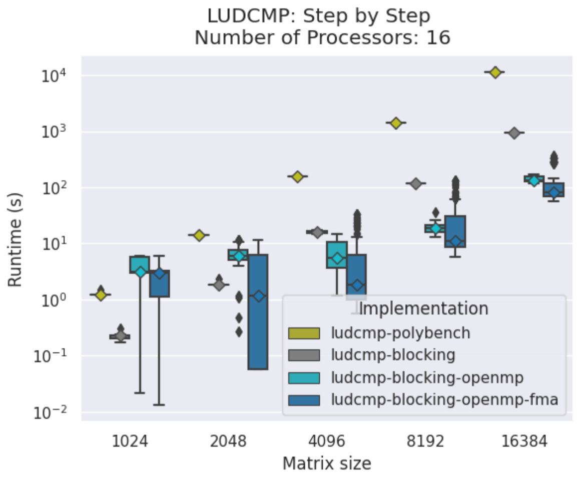

Fig. 3. Step by step improvements of the LU factorization algorithm using OpenMP.

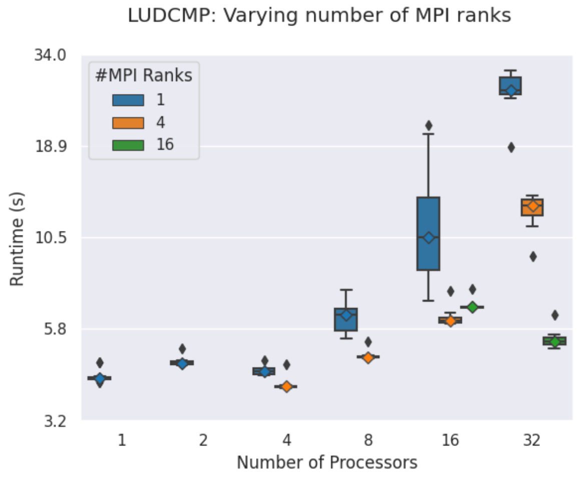

Fig. 4. Runtime compared to the number of processors and MPI ranks.

utilization for configurations with a larger number of ranks. We note that by taking more care in implementing the algorithm, similar results could likely also be achieved with OpenMP only. For our benchmarks, use the best performing number of ranks for every processor count.

Strong and Weak Scaling. For both our GEMM (figure 5) and LUDCMP (figure 6) implementations, we provide runtime of our algorithms with respect to the number of processors used. Additionally, we performed weak scaling analysis for both GEMM (figure 7) and LUDCMP (figure 8) to show how our algorithms handle a growing amount of work while adding resources to the system in the form of additional processors. As both of our algorithms run in $O(n^3)$, we use input sizes given by the following function to keep the work per core constant

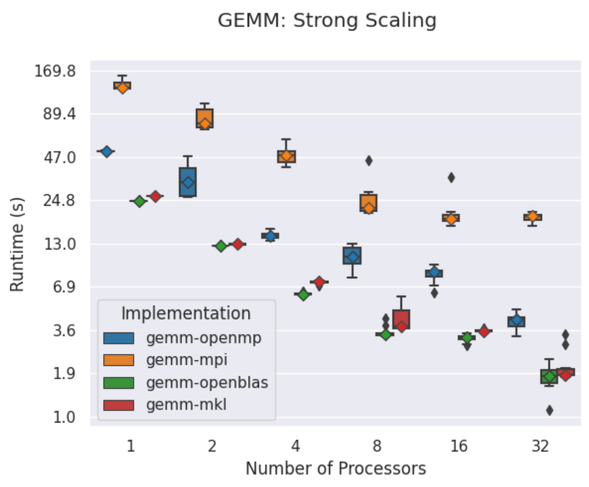

Fig. 5.

$$N = \lceil (P \cdot N_0^3)^{\frac{1}{3}} \rceil \text{ with } N_0 = 5000$$

where P is the number of processors used.

For GEMM, we use the corresponding `cblas_dgemm` function from the MKL and OpenBLAS libraries as a reference. Our GEMM implementations were not able to outperform the BLAS implementations. We can further observe that the OpenMP implementation scales better than the MPI implementation with additional workload and resources. This can be attributed to the fact that MPI has to do additional work in decomposing and communicating the data across process, as well as gathering partial matrices into one. On the other hand, OpenMP has a shared memory model, eliminating the need for data communication across processes.

We compare our LUDCMP implementations to an implementation that uses the `LAPACKE_dgetrs` to solve a system of linear equations using the LU factorization computed by `LAPACKE_dgetrf`, again using both OpenBLAS and MKL as the backing BLAS implementation. At 32 cores, we were able to outperform OpenBLAS which seems to have trouble scaling to more than a few cores. MKL is still noticeably faster across the board, though, our MPI implementations matches or even outperforms its scaling. However, we note again that `LAPACKE_dgetrf` accounts for pivoting, while our algorithm does not, this most likely makes it a lot easier to produce a scalable algorithm.

We further note that our LUDCMP MPI implementation performs better than the OpenMP implementation, even with only single core available. We attribute this to the fact that while implementing the algorithm using MPI, we have taken into account the findings from implementing the algorithm with OpenMP and improving on it.

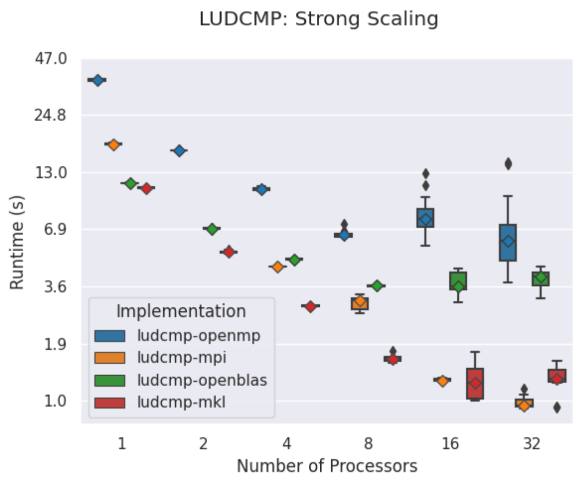

Fig. 6.

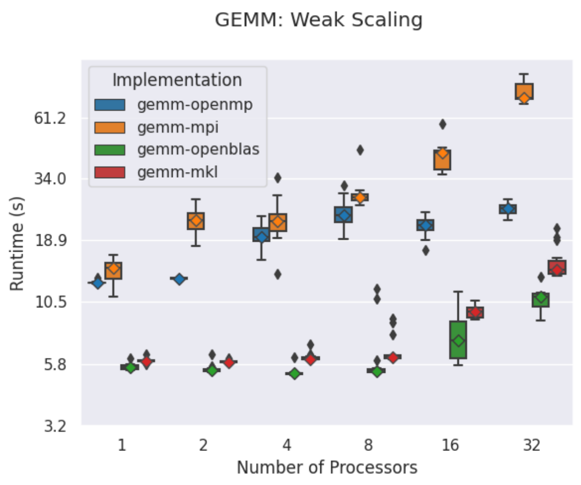

Fig. 7.

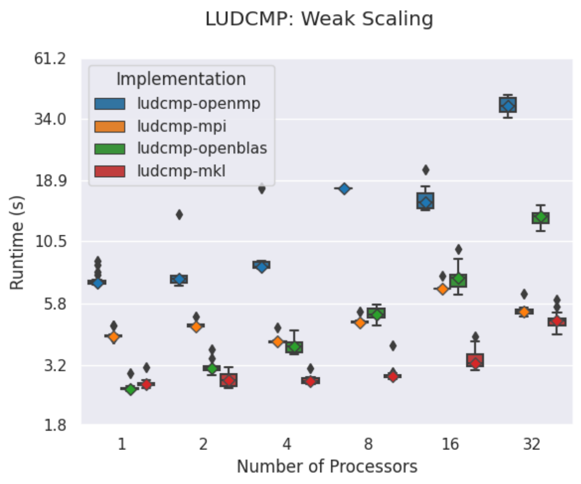

Fig. 8.

5. CONCLUSIONS

In this report, we detail the implementation of efficient parallel algorithms for PolyBench's GEMM and LUDCMP kernels using both shared memory and message passing paradigms. We evaluated our implementations on ETH's Euler System and compared the results to those obtained using Intel MKL and OpenBLAS. We found that both libraries had higher single core performance than our implementations. However, in both cases we managed to be within 50% of the respective library implementations.

For GEMM, we weren't able to match the scaling characteristics of MKL, but still managed to get a reasonable performance. We were able to reuse the resulting implementation for our LUDCMP algorithm.

We have developed a fully distributed version of LU factorization[1], which allows for handling very large problem sizes. Although we have only been able to evaluate this distributed LU factorization on a single physical node, the results were promising, with good scalability to multiple MPI ranks. Taking into account the limitations outlined in Section 4, our LUDCMP implementation appears to be faster than OpenBLAS when utilizing a larger number of cores, and we expect the implementation to also be efficient when running on multiple physical nodes.

Through benchmarking and profiling our code, we were able to improve CPU utilization and address scalability bottlenecks. This guided us to an implementation that delivers good performance and scalability for both problems.

6. REFERENCES

[1] "Polybench/c," `https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/#documentation`, Accessed: 2023-1-10.

[2] Leo Dagum, "Openmp: an industry standard api for shared-memory programming," vol. 5, pp. 46 – 55, 1998.

[3] Frank Nielsen, "Introduction to mpi: The message passing interface," pp. 21–62, 2016.

[4] "Blas (basic linear algebra subprograms)," `https://netlib.org/blas/`, Accessed: 2023-1-10.

[5] Bo Shen Guangyong Zhang Endond Wang, Qing Zhan, "Intel math kernel library," pp. 167–188, 2014.

[6] "Openblas an optimized blas library," `http://www.openblas.net/`, Accessed: 2023-1-10.

[7] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf, "The cache performance and optimizations of blocked algorithms," *SIGPLAN Not.*, vol. 26, no. 4, pp. 63–74, apr 1991.

[8] Michael Quinn, "Parallel programming in c with mpi and openmp," 2004.

[9] Jianping Chen, Kun Ji, Zhenguo Shi, and Weifu Liu, "Implementation of block algorithm for lu factorization," vol. 2, pp. 569–573, 2009.

[1]Again, we note that we did not implement back substitution in a distributed manner.