CS 420 Project Report

Parallel Cholesky Factorization with Shared and Distributed Parallelism

Songyuan Cui Sowmya Yellapragada

University of Illinois at Urbana-Champaign
May 12, 2023

Abstract

In this project, we investigate and implement parallel Cholesky factorization algorithms using both shared and distributed memory parallelism. The shared memory parallelism is achieved using OpenMP directives, while distributed memory parallelism is implemented by subdividing the symmetric positive definite (SPD) matrix in a circular fashion using MPI. We outline the strategies for parallel algorithmic design and report the performance in terms of strong scaling. We demonstrate that the parallel implementation of the Cholesky factorization algorithm provides significant speedup in comparison to the sequential implementation for both shared and distributed memory parallelism. Furthermore, we provide a comparative analysis of the performance of the shared and distributed memory parallelism strategies and discuss the advantages and limitations of each approach. Our results suggest that the proposed parallel Cholesky factorization algorithms can efficiently handle large-scale SPD matrices and can be used as a building block for various scientific and engineering applications.

1 Introduction

Efficient algorithms for factorizing and solving symmetric positive-definite (SPD) linear systems are of great interest to a large array of physical problems. Cholesky decomposition is an $\mathcal{O}(n^3)$ complexity factorization method that divides an SPD matrix into lower triangular and upper triangular parts. By leveraging the symmetric and positive-definite nature of the matrix, Cholesky factorization outperforms the robust LU-factorization by a factor of 2, and does not require pivoting. Generic Cholesky decomposition is designed for Hermitian positive-definite matrices; however, we only consider the cases of real matrices where Hermitian is equivalent to matrix symmetry, namely

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T \tag{1}$$

where $\boldsymbol{L}$ is the a lower triangular (diagonal inclusive) matrix and $\boldsymbol{L}^T$ is its transpose.

The Cholesky factorization algorithm [1] resembles that of the LU factorization, but it only operates on half of the original matrix due to its symmetry. The general algorithm is outlined in Algorithm 1. This algorithm uses a row-major order (and a column-major variation can be similarly devised), which loops over each row sequentially. In each iteration k, we call the diagonal element a_{kk} the pivot which is replaced by its square root. Then, the column below

the pivot is scaled by the pivot, similar to LU decomposition. Finally, we subtract from the sub-block ($k+1$ to n) the outer product of the scaled column with itself. This process repeats until we reach the last row of the SPD matrix.

Algorithm 1 Generic Cholesky factorization

```
for k = 0 : n − 1 do                                   ▷ Loop over rows
    a_kk = √a_kk
    for i = k + 1 : n − 1 do                           ▷ Scale current column
        a_ik = a_ik / a_kk
    end for
    for j = k + 1 : n − 1 do                           ▷ Subtract outer product of current column
        for i = j : n − 1 do
            a_ij = a_ij − a_ik a_jk
        end for
    end for
end for
```

We do not implement, but note for reference a variant of the Cholesky factorization, sometimes called the LDL factorization, where the SPD matrix is decomposed as $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$. Here, $\boldsymbol{D}$ is a diagonal matrix, and $\boldsymbol{L}$ has all diagonal entries equal to 1. The main advantage of this method is that it does not require the square root step in the classical Cholesky factorization.

This report is organized as the following. Section §2 outlines the Cholesky factorization algorithm and the parallelization strategies for both shared and distributed memory parallelism. Section §3 summarizes and analyzes the strong scaling results for both types of parallelism. In section §4 we point out potential means of improvements for this project. Finally, we conclude the discussion by reiterating important observations in section §5.

2 Methods

Here we outline the design and implementation details of the serial and parallel Cholesky factorization algorithm. First, we consider the generation of an SPD matrix. We generate a random, non-singular matrix $\boldsymbol{M} \in \mathbb{R}^{n\times n}$, and subsequently generate an SPD matrix by multiplying $\boldsymbol{M}$ with its transpose. Its symmetry can be trivially proven, and the positive definiteness can be shown as

$$\boldsymbol{u}^T(\boldsymbol{M}^T\boldsymbol{M})\boldsymbol{u} = (\boldsymbol{M}\boldsymbol{u})^T(\boldsymbol{M}\boldsymbol{u}) > 0,$$

where $\boldsymbol{u}$ is any non-zero vector in $\mathbb{R}^n$. In our implementation, we further scale the entire matrix $\boldsymbol{A} = \boldsymbol{M}^T\boldsymbol{M}$ by its first diagonal entry to ensure most entries in $\boldsymbol{A}$ is $\mathcal{O}(1)$ for double-precision numerical conditioning purposes. We then store the lower triangular part (diagonal inclusive) of the resulting SPD matrix in a 1D array.

A serial implementation of the Cholesky factorization is programmed according to Algorithm 1, with corrected tested by comparison with the 'cholesky' function in SciPy linear algebra package. Next, we parallelize the serial code using OpenMP to achieve shared memory parallelism. The OpenMP [2] implementation uses the generic `#pragma omp for` while separating it with thread creation and destruction to minimize overhead. Actual implementation can be found in the `omp_cholesky()` function within `utils/cholesky.cpp`.

For distributed memory parallelism using MPI [3], we first elucidate the partitioning of the matrix into different processes/ranks. The matrix is partitioned by rows, with each row assigned

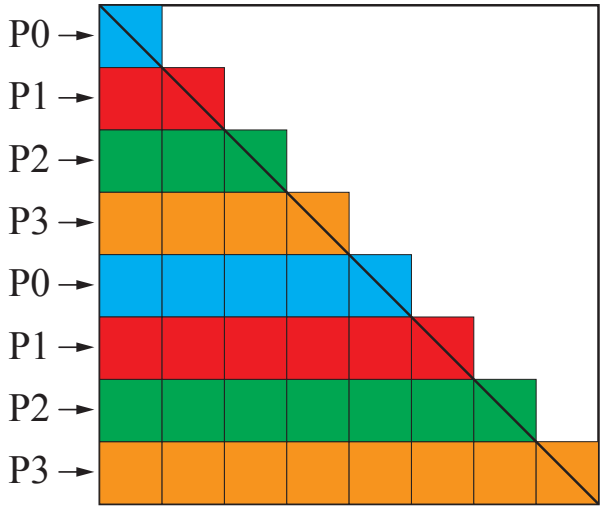

Figure 1: Distributed memory circular partitioning of a sample 8×8 SPD matrix.

to a particular process. For load-balancing purposes, this is achieved in a circular fashion, where the nth row is assigned to rank $n \pmod{np}$, where np is the total number of processes (Fig. 1). Consequently, the workload on each process is expected to be approximately equal. In our implementation, this assignment is achieved using a standard template library map (`std::map`) between the row index and the data contained on that particular row.

After the assignment is complete, each process follows Algorithm 2 to complete Cholesky factorization for the rows it is assigned. We note that two parts in this algorithm require collective communication, specifically broadcast. First, at each row iteration (k) we determine which process contains the current pivot a_{kk}, update it to its square root, and broadcast it to all processes. This is mandatory since the column below the pivot ($\boldsymbol{c}$) must be scaled, but is distributed to multiple processes. Second, after scaling $\boldsymbol{c}$ on each process individually, they are again broadcast to all processes so that every rank knows the entire scaled column. When all iterations are complete, each process sends its assigned rows to the root process (process 0 in our case), which collects the rows and arranges them back into a lower triangular matrix. Detailed implementation can be found in the `mpi_cholesky()` function within `utils/cholesky.cpp`.

3 Results

3.1 Shared Memory Parallelism (OpenMP)

Here we show the benchmark results of our Cholesky factorization on the UIUC Campus Cluster. The results for the shared memory parallelism (OpenMP) implementation of Cholesky factorization show that the execution time decreases as the number of processors is increased. We conduct strong scaling experiments on a single node for two different matrix dimensions: 4096 and 8192. The results are shown in Fig. 2, with optimal linear scaling shown in black dashed line for reference.

For both 4096 and 8192 matrix dimension, we observe that the execution time decreases linearly as the number of OpenMP threads is increased up to 16 threads, indicating good strong scaling. Overall, our results demonstrate that the shared memory parallelism (OpenMP) implementation of Cholesky factorization achieves significant speedup with an increasing number of threads for

Algorithm 2 MPI process-wise Cholesky factorization for process n

```
Initialize pivot p, scaled column c
for k = 0 : n − 1 do                                          ▷ Loop over rows
    if n == k mod np then                                     ▷ Pivot is on process n, update pivot
        p = √a_kk
        a_kk = p
    end if
    Broadcast pivot p to all processes (MPI_Bcast)
    for row i in (All rows assigned to process n) do
        a_ik = a_ik/p                                         ▷ Scale current column
        c_i = a_ik                                            ▷ Populate the scaling column as much as possible
    end for
    for i = k + 1 : n − 1 do
        Broadcast scaled column element c_i to all processes (MPI_Bcast)
    end for
    for row i in (All rows assigned to process n) do
        for j = k + 1 : i do
            a_ij = a_ij − c_i c_j                             ▷ Subtract outer product of scaled column
        end for
    end for
end for
```

both matrix dimensions. However, due to hardware limitations, shared memory benchmark tests with more than 16 OpenMP threads are not conducted. This limitation is overcome by utilizing distributed memory parallelism as we shall subsequently discuss.

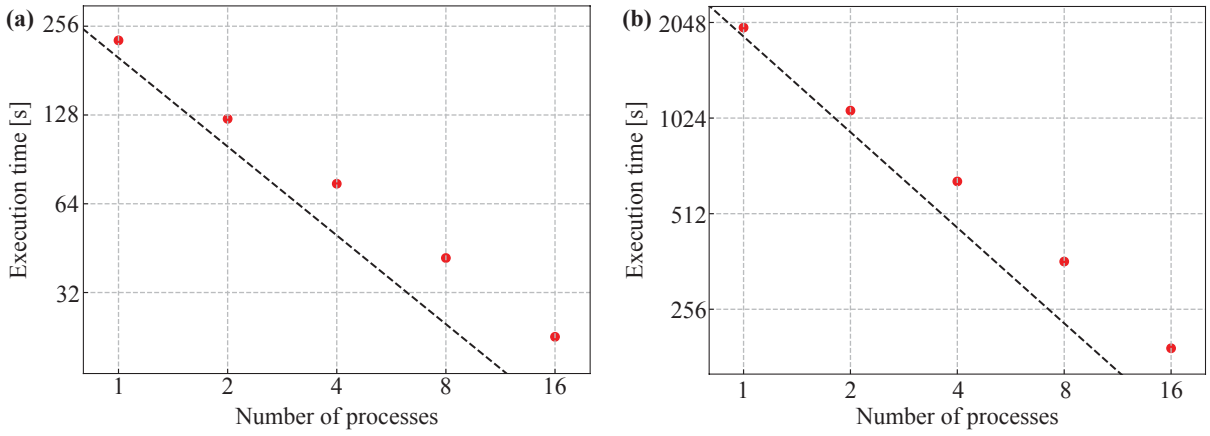

Figure 2: Strong scaling of Cholesky factorization using OpenMP on a single node. Execution time is plotted against a number of OpenMP threads, with optimal strong scaling shown in the black dashed line for reference. Results are shown for matrix dimensions (a) 4096, and (b) 8192.

3.2 Distributed Memory Parallelism (MPI)

Next we test the MPI implementation across multiple nodes and show strong scaling behavior for both matrix dimensions: 4096 and 8192. Benchmark tests are performed across multiple nodes where each node host a maximum of 16 MPI processes, up to a total of 8 nodes (128 processes in total). The results are shown in Fig. 3, with the optimal strong scaling (linear

scalability) plotted in black dashed line for reference.

For the 4096 matrix dimension, we observe that the execution time decreases linearly as the number of MPI processes is increased up to 16 processes. After 32 processes, the strong scaling deteriorates as the performance deviates from linear scalability. This trend continues until 64–128 MPI processes where the performance flattens. An obvious reason in this diminishing strong scaling is the collective communication within the MPI implementation, as pointed out previously in section §2. As the number of processes increase, the communication cost becomes increasly taxing upon the overall performance. A second reason is the distribution of ranks to multiple nodes. Beginning at 32 processes, the processes are distributed across multiple nodes. Therefore, it is plausible that the across-node communication, from the hardware's perspective, produces more overheads than distributed memory on a single node, and thus decrease the performance.

For the 8192 matrix dimension, we observe a similar trend in the execution time as the number of MPI processes is increased. However, it is worth noting that good strong scaling is preserved for larger number of processes (up to 32 processes) than in the 4096 matrix case. This implies that the MPI implementation may be increaslying beneficial for larger matrix sizes as computation cost on each process dominates over communication cost. Similar to the 4096 matrix case, the strong scaling deviates from optimum at 64 porcesses, where there is a sudden surge in execution time. The reason for this behavior is not clear, but it may be related to the aforementioned across node communication overheads.

In summary, our results demonstrate that the MPI implementation of Cholesky factorization achieves strong scaling behavior for both matrix dimensions. Additionally, we comment that with a single process (thread in OpenMP's case), the MPI implementation is slower than the OpenMP version. This is because of additional implementation details in MPI Cholesky factorization that is absent in serial and OpenMP implementations, such as extra buffer creation, copying, and MPI calls. However, MPI excels at scaling larger matrices to multiple nodes, which poses as a significant barrier for shared memory parallelism.

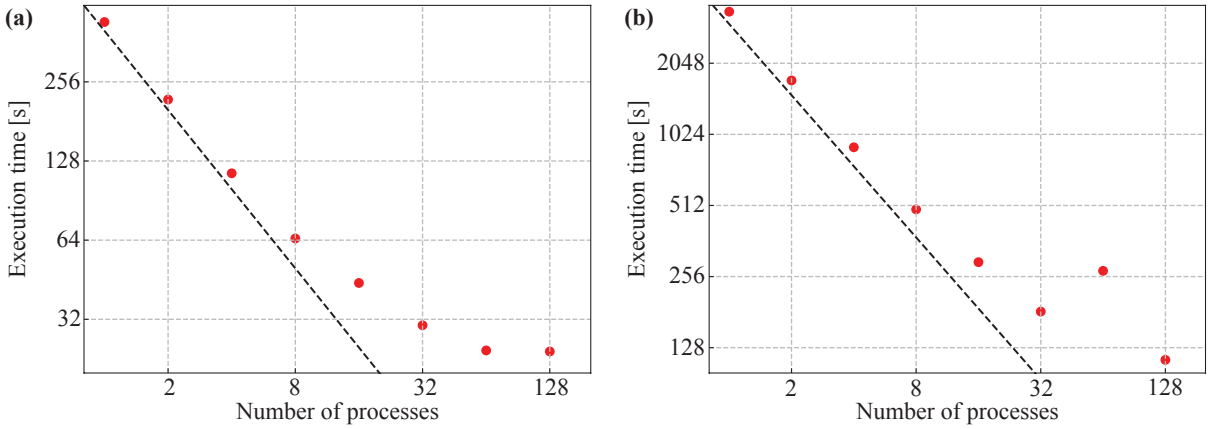

Figure 3: Strong scaling of Cholesky factorization using MPI. For the low number of processes, the benchmark is performed on a single node up to 16 processes per node. The higher number of MPI processes is achieved by requesting multiple nodes, with 16 processes on each node, up to a maximum of 8 nodes. Execution time is plotted against a number of MPI processes, with optimal strong scaling shown in the black dashed line for reference. Results are shown for matrix dimensions (a) 4096, and (b) 8192.

3.3 Weak Scaling

Weak scaling, as opposed to strong scaling, involves increasing the problem size while keeping the workload per core constant. In other words, weak scaling measures how well a parallel program can handle larger problem sizes by distributing the workload to a larger number of cores. The ideal speedup under weak scaling achieved by adding more computation cores should remain constant as the problem size increases.

However, for our project we elect not to perform weak scaling analysis for the following reasons. Cholesky factorization is an algorithm with $\mathcal{O}(n^3)$ computational complexity, which implies that the total workload increase cubically with matrix size. In other words, if we double the matrix size, $8\times$ the number of processes / threads are required to maintain the same level of workloads on each process / thread. Thus, it is infeasible to perform a large scale test of weak scaling on Campus Cluster for Cholesky factorization. Furthermore, our MPI Cholesky algorithm distribute lower triangular rows to processes in a circular fashion, which means the workload is close to but not perfectly balanced. This issue is exacerbated with large number of processes which would manifest as poor weak scaling. Therefore, we only test Cholesky factorization for strong scaling, where the problem size is kept constant and the number of processes / threads is increased to reduce the overall computation time.

4 Optimization Strategies

There are several potential improvements that can be made to our parallel Cholesky Factorization implementation using shared and distributed memory parallelism.

One possible improvement is to combine MPI and OpenMP to exploit both shared and distributed memory parallelism simultaneously. This approach, known as hybrid parallelism, can provide better performance and scalability by allowing multiple threads to execute on each processor while also distributing the work across multiple processors. Our program implements the two separately, yet they can be combined via, for example, a funneled protocol on multiple nodes to further improve performance. This is particularly beneficial for larger matrices where the memory requirements require multiple nodes.

Another potential improvement is to test the algorithm with larger matrices. This can help to further evaluate the performance and scalability of the algorithm and identify any potential bottlenecks or limitations. Additionally, the use of larger matrices can help to test the limits of the hardware and software platforms and identify opportunities for optimization.

Finally, the I/O performance of the current implementation can be improved by using MPI-IO to distribute the input matrix file across multiple processes. Currently, the entire input file is read on one process and then distributed to other processes, which can lead to I/O bottlenecks and decreased performance. By using MPI-IO, the input file can be read and distributed in parallel, leading to faster I/O performance and improved scalability.

5 Conclusion

In this project, we investigated and implemented parallel Cholesky Factorization algorithms using shared and distributed memory parallelism. Our implementation leveraged OpenMP directives for shared memory parallelism and MPI for distributed memory parallelism, with the SPD matrix subdivided in a circular fashion. We presented the strategies for parallel algorithmic design and evaluated the performance in terms of strong scaling.

Our results demonstrate that both shared and distributed memory parallelism can significantly improve the performance of Cholesky factorization compared to the sequential implementation. For shared memory parallelism, our implementation exhibits good strong scalability as the execution time decreased linearly as the number of OpenMP threads increased up to 16 threads. For distributed memory parallelism, our implementation shows good strong scaling up to a certain number of processes, but suffers from communication overheads with more MPI processes.

In conclusion, our study presents effective parallel implementations of Cholesky factorization using shared and distributed memory parallelism and provides insights into their strong scaling behavior. The implementation of parallel Cholesky factorization algorithms can benefit a wide range of scientific and engineering applications that involve the solution of linear systems of equations, and can be used as a building block for various scientific and engineering applications.

6 Acknowledgements

This work made use of the Illinois Campus Cluster, a computing resource that is operated by the Illinois Campus Cluster Program (ICCP) in conjunction with the National Center for Supercomputing Applications (NCSA) and which is supported by funds from the University of Illinois at Urbana-Champaign.

7 Code availability

The code developed for this project can be accessed at https://github.com/sy-cui/cs420_project

References

[1] Michael T. Heath. *Scientific Computing.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 2018.

[2] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[3] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.