Parallel LU Decomposition for Linear Algebra

Abigail Pitcairn

December 7, 2025

1 Introduction

This project runs an LU decomposition using parallel algorithms from OpenMP and MPI [1–3] .

This project explores the use of parallel programming techniques to accelerate LU decomposition, a fundamental operation in linear algebra commonly used for solving systems of equations, inverting matrices, and computing determinants. We investigate and compare three implementations of LU decomposition:

- **Serial Implementation**: A single-threaded approach that serves as the baseline for performance and correctness.
- **OpenMP Parallelization**: Uses shared-memory parallelism by distributing workload across multiple CPU cores within a single node, utilizing OpenMP tools for efficient thread management.
- **MPI Parallelization**: Uses distributed-memory parallelism, splitting the computation across several processes potentially running on different physical nodes, enabling scalability for large matrix sizes.

Our goals include analyzing the speedup and efficiency gains enabled by parallelization, identifying practical challenges in implementing parallel LU decomposition, such as workload distribution and numerical stability, and verifying the accuracy of the results. The project benchmarks the different approaches using matrices of various sizes and investigates how well each implementation scales as the computational resources and problem size increase.

The report documents a careful assessment of computation time, correctness, and parallel efficiency, as well as a discussion of the considerations involved in applying OpenMP and MPI to linear algebra problems. Additionally, this study provides insights into the benefits of parallelism, and comparison between parallelism methods and tools.

2 Methodology

The LU decomposition code takes a square matrix and breaks it down into two simpler matrices: a lower triangular matrix (L) and an upper triangular matrix (U). This process makes it easier and faster to solve systems of linear equations.

We implemented three different versions of the LU decomposition method:

- **Serial:** This basic version runs on a single processor and works through the matrix row by row, performing the decomposition step by step. This serves as our baseline for comparison.
- **OpenMP:** To speed things up, we used OpenMP for parallel computing on a shared-memory machine (like a normal multi-core computer). The code splits the tasks between multiple CPU cores, allowing them to work at the same time. OpenMP organizes, starts, and manages these parallel tasks efficiently.
- **MPI:** For even larger matrices, we used MPI (Message Passing Interface) to distribute the work across different computers or processes over a network. The matrix is divided between several processes, each handling a part of the computation. The processes communicate with each other to share results and keep the decomposition correct. This approach allows the code to handle very large matrices by spreading the work across multiple processors.

For all versions, the code first sets up the matrix, filling it with a specific pattern. For parallel versions, the software divides the computation so that multiple threads (OpenMP) or processes (MPI) perform calculations at the same time. After the LU decomposition, the code checks the results for accuracy and measures how long everything took to run.

All implementations aim to produce the same output, but the OpenMP and MPI versions do it much faster on larger matrices by taking advantage of parallelism.

3 Results

We evaluated the performance and accuracy of the LU decomposition implementations using several test cases of varying matrix sizes and parallelization configurations. The main results are summarized below. All tests report the decomposition status, the maximum numerical error compared between the original matrix and the matrix after reconstruction, as well as execution time.

Test Configuration

- Matrix sizes tested (all square): 10, 100, 1000, 2500, and 5000
- Matrix pattern: diagonal entries = 4.0, off-diagonal entries = 1.0
- Serial, OpenMP (8 threads), MPI (1 or 4 processes)
- Accuracy tolerance: 1e−10

Results

Matrix Size	Method	Threads	Status	Max Error	Time (s)
1000	Serial	1	Success	5.33×10^{-15}	0.185
	OpenMP	8	Success	5.33×10^{-15}	0.18
	MPI	1	Success	5.33×10^{-15}	0.21
2500	Serial	1	Success	6.66×10^{-15}	1.66
	OpenMP	8	Success	6.66×10^{-15}	1.37
	MPI	1	Success	6.66×10^{-15}	1.85
5000	Serial	1	Success	9.77×10^{-15}	14.96
	OpenMP	8	Success	9.77×10^{-15}	14.67

All implementations produced correct results, well within the allowed error tolerance.

Result Details

1000 x 1000 Matrix

- **Serial:** Execution time: 0.185 s; Maximum error: 5.33×10^{-15}
- **OpenMP (8 threads):** Execution time: 0.18 s; Maximum error: 5.33×10^{-15}
- **MPI (1 process):** Execution time: 0.21 s; Maximum error: 5.33×10^{-15}

2500 x 2500 Matrix

- **Serial:** Execution time: 1.66 s; Maximum error: 6.66×10^{-15}
- **OpenMP (8 threads):** Execution time: 1.37 s; Maximum error: 6.66×10^{-15}
- **MPI (1 process):** Execution time: 1.85 s; Maximum error: 6.66×10^{-15}

5000 x 5000 Matrix

- **Serial:** Execution time: 14.96 s; Maximum error: 9.77×10^{-15}
- **OpenMP (8 threads):** Execution time: 14.67 s; Maximum error: 9.77×10^{-15}

Discussion

The results demonstrate:

- All implementations (Serial, OpenMP, MPI) complete with high accuracy on all tested matrix sizes up to 2500×2500.
- OpenMP grows faster than serial as the matrix size increases, with nearly identical accuracy.

- For the largest matrix, 5000×5000, only Serial and OpenMP were tested, both taking nearly 15 seconds and yielding very high floating-point accuracy. MPI methods were not run on this size matrix due to high runtime cost.

The results show that the OpenMP implementation is the fastest for all large matrix sizes tested, followed by the serial implementation. The MPI implementation is the slowest for larger matrices, but it is still able to complete the decomposition in a reasonable time for sizes under 5000 by 5000.

Conclusion

The results show that the OpenMP implementation is the fastest for all larger matrix sizes tested, followed by the MPI implementation, and with the serial implementation being the slowest. For square matrices smaller than 1000 by 1000, the serial implementation was the fastest, with MPI and OpenMP methods taking significantly longer

The results also show that the OpenMP implementation is able to achieve a speedup of up to 15-20% over the Serial implementation for the same number of threads.

References

[1] P. D. Michailidis and K. G. Margaritis, "Implementing parallel lu factorization with pipelining on a multicore using openmp," in *2010 13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 2010, pp. 132–139. [Online]. Available: https://www.researchgate.net/publication/220775793_Implementing_Parallel_LU_Factorization_with_Pipelining_on_a_MultiCore_Using_OpenMP

[2] J. Dongarra and A. Hinds, "Parallel numerical linear algebra," in *Handbook of Parallel and Distributed Computing*, M. J. Atallah, Ed. McGraw-Hill, 2000, pp. 571–586. [Online]. Available: https://dl.acm.org/doi/10.5555/582787.823016

[3] V. Eijkhout, *Parallel Programming*. University of Texas at Austin, 2024. [Online]. Available: https://ftp.utcluj.ro/pub/users/civan/CPD/3.RESURSE/10.Book_2024_Parallel%20Programming_Eijkhout.pdf