

Linear Algebra Optimization in C using OpenMPI

Aidan Levy

Dr. Spickler

April 20, 2025

1 Introduction

In today's data driven world, there is a constant demand for faster and most efficient computing methods. Traditional sequential computing, where tasks are executed one after another by a singular processor, often falls short when tackling complex and large scale problems. While this model remains useful for certain tasks and small scale applications, it fails to meet the performance needs of large scale computations in modern research, analytics, and simulation based fields. Parallel computing addresses this limitation by distributing workload across multiple processors or cores, enabling simultaneous execution of sub tasks. This approach significantly reduces execution time and increases efficiency, especially for complex mathematical operations and data heavy computations. Parallelism has become integral in many different cases— from national defense and physics simulations to artificial intelligence and high frequency trading— where performance constraints dictate the outcomes of solutions (Barney & Frederick). The Open Message Passing Interface (OpenMPI) implementation has become a core element of solving this problem by facilitating communication and coordination among distributed processes. OpenMPI is widely used in both academic and industry settings due to its scalability, ease of use, and support for high performance clusters. This research project investigates the use of OpenMPI to optimize linear algebra algorithms written in C, which is a very low level language. Linear algebra remains a central pillar of computer science, with applications ranging from computer graphics and machine learning to economic modeling and engineering. One historic example includes Professor Leontief's use of linear equations in 1949 to model the structure of the United States economy, which demonstrated the importance of these systems (Lay et al., 2020). In this study, emphasis is placed on the efficient execution of operations such as matrix multiplication, row echelon form (REF), reduced row echelon form (RREF), and LU decomposition, which serve as critical tools in solving systems of linear equations. These algorithms are implemented in both sequential and parallel forms to allow for direct performance comparisons. Tim-

ing benchmarks collected during testing reveal the extent to which parallelism accelerates execution and under what conditions those gains are most significant. These performance results will be discussed in detail in later sections. In addition to algorithm optimization, the project involved constructing a custom parallel computing environment. The process of configuring and deploying a multi node OpenMPI cluster provided valuable learning experiences into the logistics and technical requirement of distributed computing. This included configuration of operating systems, secure shell (SSH) communication setup between nodes, implementation of shared storage, and fine tuning of the scripts. The resulting cluster not only enabled controlled benchmarking of parallel code but also served as a replicable model for lightweight high performance computing environments.

2 Mathematics and Algorithms

The implementation of parallelized linear algebra routines requires not only a theoretical understanding of the algorithms but also practical insight into how these operations work under distributed computation. This section outlines the core mathematical algorithms used in the project: matrix multiplication, REF, RREF, and LU decomposition, as well as comparing the differences between their sequential and OpenMPI forms.

Matrix multiplication is a foundational operation in linear algebra with applications ranging from linear transformations to machine learning. Given two matrices, A of size $m \times n$ and B of size $n \times p$, the product $C = A * B$ is defined as:

$$C_{i,j} = \sum_{k=1}^n A_{i,k} \cdot B_{k,j}$$

In the sequential implementation, three nested loops are used to iterate through each row i of A, each column j of B, and each inner dimension k , summing the partial products in $C_{i,j}$. The parallel implementation distributes the row range of matrix A across all MPI processes. Each process computes a subset of the output matrix C, then gets called by `MPI_Allreduce` to

consolidate the partial results across all nodes. Load balancing is managed by evenly dividing rows, and the implementation accounts for division with remainders by assigning an extra row to the first few ranks. This design allows each core to perform independent multiplication on the assigned rows, significantly reducing computation time while leveraging all available nodes. The correctness of results is guaranteed by the summation of `MPI_Allreduce`.

Additionally, the REF form of a matrix is another fundamental transformation in linear algebra, and is commonly used for solving systems of linear equations and preparing a matrix for further operations such as back substitution, RREF conversion, and LU decomposition. A matrix is said to be in row echelon form when all nonzero rows are above any rows of all zeroes; the leading coefficient (pivot) of a nonzero row is strictly to the right of the pivot in the row above it; and all entries below a pivot are zero. The sequential algorithm follows the Gaussian elimination process. The function iteratively scans for nonzero pivots, performs row swaps if needed, normalizes the pivot row so the leading entry becomes 1, and eliminates all entries below the pivot by subtracting a multiple of the pivot row from each target row. In the parallel implementation, the primary logic is preserved, but the elimination step is distributed across multiple MPI processes. Each process is responsible for updating a subset of rows. More specifically, rows are assigned based on their index modulus the total number of MPI processes:

```
if (i % p == rank) then process 'rank' handles row i
```

Each process waits until the pivot row is broadcast from rank 0, then applies the elimination step locally to its assigned rows. After completing a round of elimination, processes synchronize using `MPI_Barrier` and then broadcast the updated matrix using `MPI_Bcast` to ensure all nodes have a consistent view before moving to the next pivot row. This approach preserves correctness while achieving partial parallelism. Although some overhead exists due to the global communication at each iteration, the performance gains from dividing the row work outweigh the costs, especially on large matrices.

Building upon the REF, RREF represents a stricter standard that fully solves a system of

linear equations in a matrix form. In addition to the requirements for REF, RREF imposes two additional conditions: Each leading '1' is the only nonzero entry in its column; and each leading '1' is the first nonzero entry in its row. The RREF algorithm is typically used as a second phase after REF. Once the matrix is in upper triangular form, the algorithm proceeds in reverse, starting from the bottom row and moving upwards, eliminating entries above each pivot. In sequential form, the implementation iterates over the rows in reverse order. For each row, it locates the pivot, then eliminates all entries above that pivot by subtracting multiples of the pivot row from each row above. This using OpenMPI, the reverse pass is distributed similarly to REF. After converting the matrix to REF form, the backward elimination step assigns work to MPI processes based on row indices. Each process eliminates entries above pivots in rows it is responsible for, using the modulus rank pattern seen before. Once a set of updates is applied, all processes synchronize with `MPI_Barrier`, and the updated matrix is broadcast again using `MPI_Bcast` to maintain consistency. While this introduces communication overhead, particularly in the upper triangular back propagation phase, the division of labor across processors still yields speedup in large matrices.

References

Lay, D. C., Lay, S. R., & McDonald, J. J. (2020). *Linear algebra and its applications* (6th ed.). Pearson.

Barney, B., & Frederick, D. (n.d.). Introduction to parallel computing tutorial. Lawrence Livermore National Laboratory. Retrieved April 21, 2025, from <https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial>