Linear Algebra Optimization in C using OpenMPI

Aidan Levy
Dr. Spickler

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. Timing 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 Terminology and MPI Overview

To aid in understanding the parallel implementations discussed in this study, this section defines key terms and MPI (message passing interface) functions that appear throughout the algorithms discussed. MPI, message passing interface, is a standardized and portable message passing system that allows processes to run on distributed memory systems. The CPUs are physically separated from each other, but still communicate and transmit data between themselves. According to Spickler (2025), the following diagram displays how distributed memory systems look:

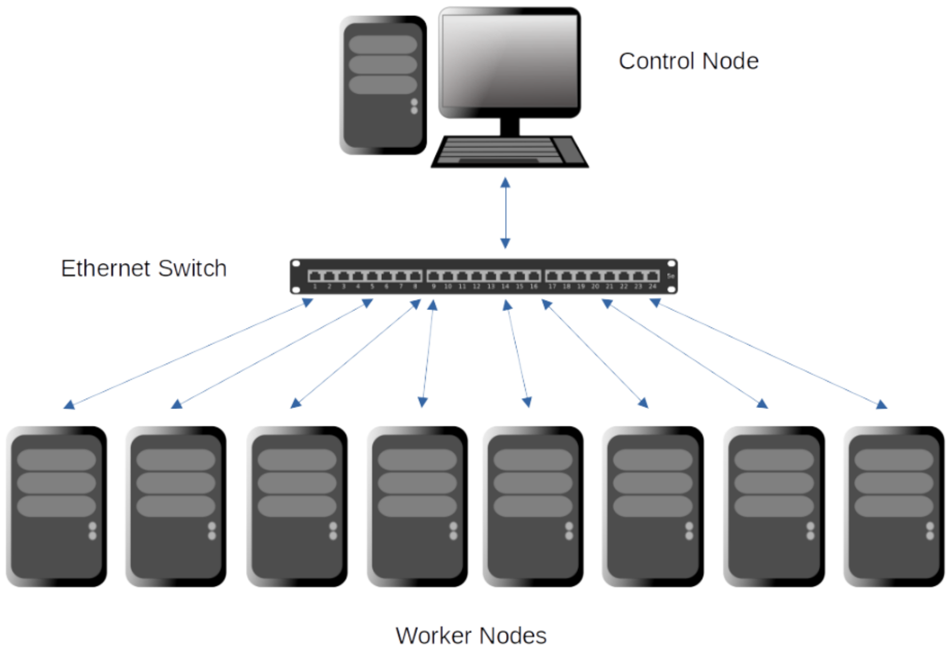

Figure 1: Data Flow in OpenMPI Cluster

An MPI process is a single instance of a running program in an MPI application. Each process has a unique rank, which is an integer associated with that process, separating it from the other processes. Rank 0 is typically the root process, the one that initializes the program. MPI_Bcast is a broadcast function that is used to send data from one process, usually the root, to all other processes in a communicator. This ensures that all processes have the same updated data before continuing. Without the usage of MPI_Bcast, the data would now be shared across all nodes. The following diagram displays how MPI_Bcast distributes memory:

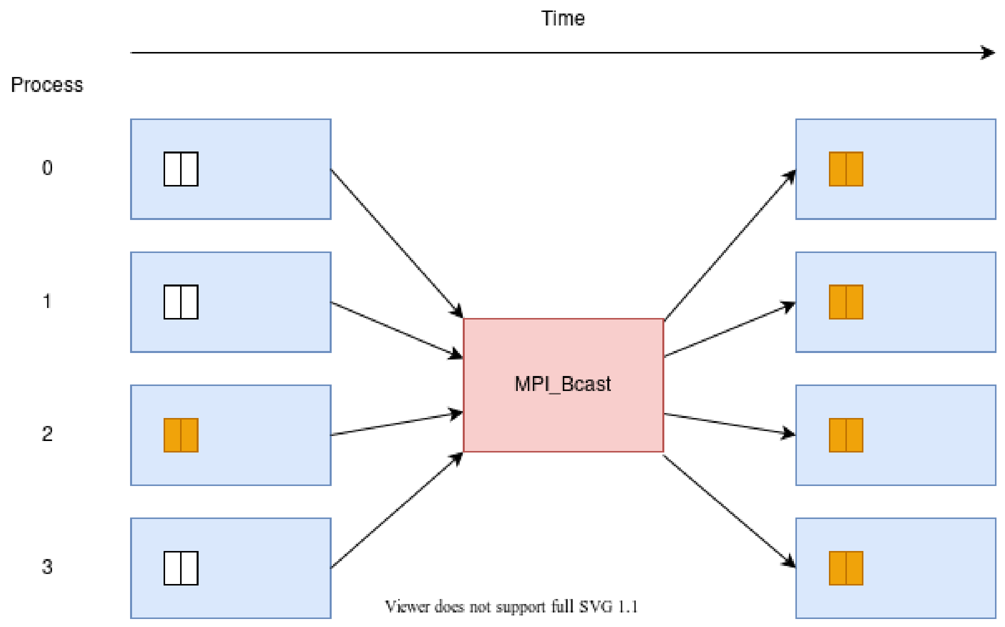

Figure 2: Data Flow using MPI_Bcast

Additionally, MPI_Barrier is a synchronization function that blocks all processes until every process in the communicator has reached the barrier. This makes sure that all previous operations are complete across all processes before proceeding. While this tool is necessary, large overhead can occur because it pauses all processes until every one is complete. In large matrices, this build up can accumulate over time. The following diagram demonstrates how MPI_Barrier waits for all processes to complete:

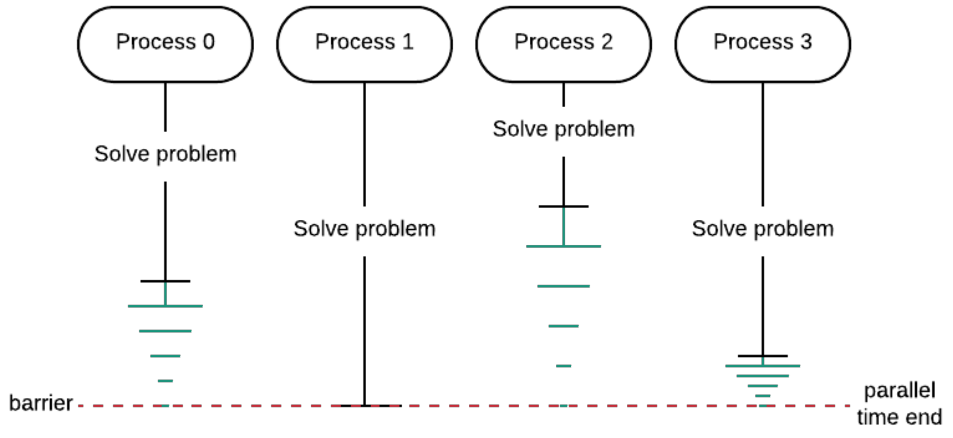

Figure 3: Data Flow using MPI_Barrier

MPI_Allreduce is a collective communication operation that sums values from all running processed using a specified operation, and distributes the result back to all processes. This command is used to pull all results together and re-send data back out to all processes. Here is a diagram displaying how MPI_Allreduce works:

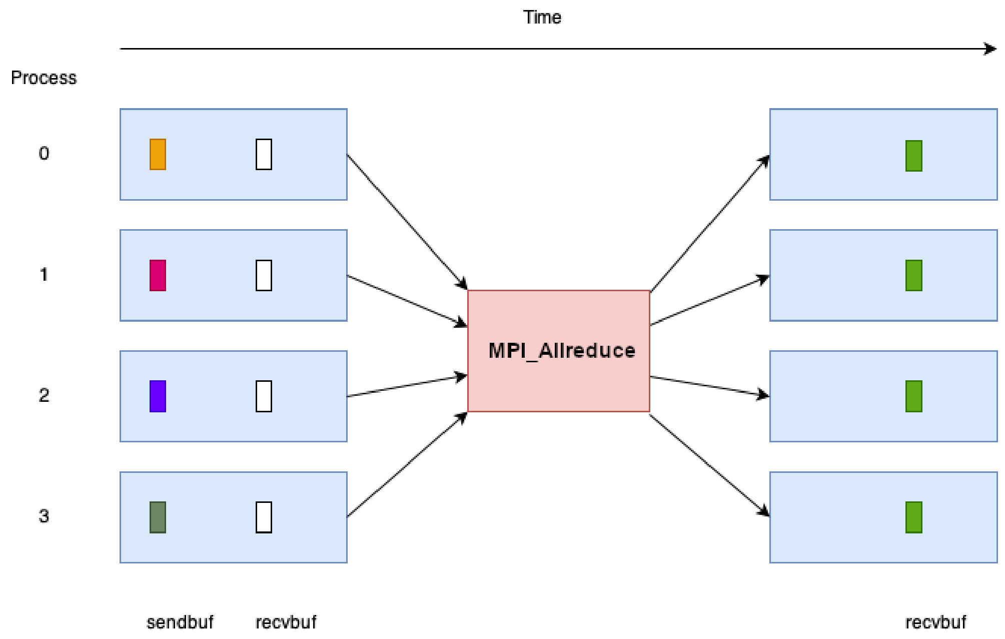

Figure 4: Data Flow using MPI_Allreduce

3 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 x n and B of size n x p, the produce 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 Ci,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 implementations 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:

$$\text{If } i \bmod p = \text{rank}, \text{ then process } \texttt{rank} \text{ handles row } i.$$

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 iterations 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 back substitution stage ensures that each pivot is isolated. In the parallel implementation using OpenMPI, the reverse pass is distributed similarly to REF. After converting the matrix to REF form, the backward elimination steps 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 broadcasted 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.

Furthermore, LU decomposition is another rudimentary matrix operation used to repeatedly solve systems of linear equations, especially when the coefficient matrix remains constant. The process factors a square matrix, A, into two triangular matrices– a lower triangular matrix L and an upper triangular matrix U such that A = L * U. In the sequential algorithm, Gaussian elimination is used to progressively zero out entries below the pivots, simultaneously creating the L matrix with the multipliers used during the elimination process and assigning the resulting upper triangle to U. This involves searching through each pivot position, calculating scalars for the rows below, and updating both matrices in place. In the parallel OpenMPI version, the decomposition is distributed across multiple processes by partitioning the matrix rows and assigning each subset to a different rank. During each iteration, the pivot row is broadcast from the root process to all other using MPI_Bcast, and each process computes updates for its assigned rows independently. As with REF and RREF, synchronization is required after each iteration using MPI_Barrier to ensure consistent progress. The challenge in the parallel LU decomposition lies in how the managing data dependencies between the L and U matrices are handled.

4 Results

To evaluate the performance of sequential versus parallel implementations, a series of timing benchmarks were conducted on square matrices of varying sizes. Each algorithm– matrix multiplication, REF, RREF, and LU decomposition– was tested using both a single process mode and a multi processes OpenMPI mode on the constructed cluster. Matrix sized ranged from 10 x 10 to 2,000 x 2,000 and execution times were recorded for each operation. The results revealed that parallel matrix multiplication, REF, and RREF achieved significant speedups compared to their sequential counterparts, particularly as matrix size increased. For example, on a REF test of a 2,000 x 2,000 matrix, the sequential algorithm took roughly 9.5 seconds, whereas the parallel process with 8 processes took just around 2.0 seconds to compute, which is a 4.75x speedup. Similarly, the parallel RREF and matrix multiplication implementations showed substantial time reductions on larger matrices, although some communication overhead was observed during pivot synchronization. However, the parallel LU decomposition algorithm was almost consistently slower than the sequential version across all tested matrix sizes. As discussed earlier, the method in which the algorithm has to be built and the frequent use of MPI_Bcast and MPI_Barrier caused the communication and synchronization overhead to dominate execution time, negating the benefits of multiprocessing.

5 Conclusion

The study demonstrated that parallel computing using OpenMPI can significantly accelerate linear algebra operations that naturally allow independent work distribution, such as matrix multiplication, REF, and RREF. Timing benchmarks confirmed that parallel implementations of these algorithms achieved substantial speedup compared to sequential versions, particularly as matrix sizes increased. However, not all algorithms benefited equally from parallelization. LU decomposition, which has strong data dependencies between elimination steps, experienced performance degradation in the parallel implementation due to communication and synchronization overhead. This highlighted a key limitation in naive parallelization strategies– operations that frequently require inter process communication may not scale well without more sophisticated techniques or algorithms. Overall, this project provided valuable insights into both the potential and challenges of parallel computing. Future work could explore more advanced decomposition methods, such as block LU decomposition, or hybrid models that combine MPI with multithreading to better balance computation and communication.

References

References

[1] 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`

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

[3] Spickler, D. (2025). *COSC 420: High-Performance Computing*. Salisbury University.