Accelerating Linear System Solutions: Parallel LU Decomposition with POSIX Threads

Mohamed Ghonim, Phanindra Vemireddy, Alexander Maso, Ahliah Nordstrom
Department of Electrical and Computer Engineering, Portland State University.

***Abstract*—Solving linear systems of equations is a fundamental task in scientific computing, pivotal for numerous applications across engineering, physics, and finance. This project focuses on enhancing the computational efficiency of solving such systems, particularly through the parallelization of the LU Decomposition algorithm. Leveraging POSIX threads, we have developed two parallel programs: one optimized for speed by eschewing partial pivoting, and another incorporating pivoting for increased numerical stability. Our findings demonstrate significant speed improvements on multicore systems, without compromising the accuracy of the solutions. These results underline the potential of parallel computing to tackle large-scale computational problems efficiently, offering a scalable solution that leverages the capabilities of modern multicore processors.**

Keywords—LU Decomposition, Parallel Computing, POSIX Threads, Pthreads, Linear Systems of Equations, Numerical Stability, Performance Optimization.

I. Introduction

Linear systems of equations are fundamental to numerous scientific and engineering endeavors, providing a vital mathematical framework for modeling and solving a broad spectrum of practical problems. When faced with large and complex systems, the challenge of efficiently finding solutions becomes pronounced. In the quest for an effective algorithm to tackle these challenges, we initially explored several numerical methods, including Gaussian Elimination and LU Decomposition. Our exploration led us to choose LU Decomposition for its superior numerical stability and its inherent potential for parallelization, making it particularly suited for handling large-scale problems.

LU Decomposition, a method that splits a matrix into lower (L) and upper (U) triangular matrices, simplifies the solution process to the simple task of performing forward and backward substitutions. This efficiency, however, comes at a cost as the size of the system increases, leading to prohibitive computational demands. In response to these challenges, parallel computing emerges as a leading approach. By dividing computational tasks across multiple processors, parallel computing offers a pathway to significantly faster execution times. This paradigm shift holds the promise of revolutionizing the way large linear systems are solved, optimizing the workload distribution of LU Decomposition across multiple cores to enhance computational efficiency.

The primary objectives of this report are twofold: first, to delve into the development and implementation of a parallelized LU Decomposition algorithm utilizing POSIX threads, and second, to assess the algorithm's performance benefits and its implications for numerical stability. This investigation particularly emphasizes the transformative impact of parallel computing on the efficacy of solving linear systems of equations. It explores the delicate balance between computational speed and solution accuracy, providing insights into optimizing LU Decomposition for contemporary high-performance computing environments through a comprehensive evaluation involving various matrix sizes and processor counts.

By embarking on this exploration, our aim is to underscore the critical role of numerical methods in advancing scientific and engineering solutions and to highlight the significant potential of parallel computing strategies in overcoming the limitations posed by large-scale computational problems.

II. Background and Preliminaries

A. Overview of Linear Systems of Equations

Linear systems of equations form the backbone of numerous scientific and engineering disciplines, providing essential frameworks for modeling, and solving a broad spectrum of problems. These equations, typically represented in the form $Ax=b$, where A is the coefficient matrix, and both x and b are vectors, (the former for the variables in our system and the later for constants) encapsulate the relationships between variables in a system. The solutions to these equations are pivotal for understanding and predicting the behavior of complex systems across various fields, from fluid dynamics to electrical circuits. We explored different fast algorithms and techniques to implement in our solver program, including Barnyard math guessing solutions, corradiated randomness algorithms from [1] as well as Gaussian elimination from and different algorithms that can be parallelized in a multicore system [2], [3].

B. Basics of LU Decomposition

LU Decomposition is a key numerical method used to solve linear systems efficiently. This technique decomposes matrix A into two simpler matrices: a lower triangular matrix L and an upper triangular matrix U, such that $A=LU$. This decomposition simplifies the solution process into two sequential stages of triangular systems: solving $Ly=b$ for y using forward substitution, and then solving $Ux=y$ for x using backward substitution. LU Decomposition stands out for its applicability to a wide range of matrices, including those that are not necessarily symmetric or positive definite [4].

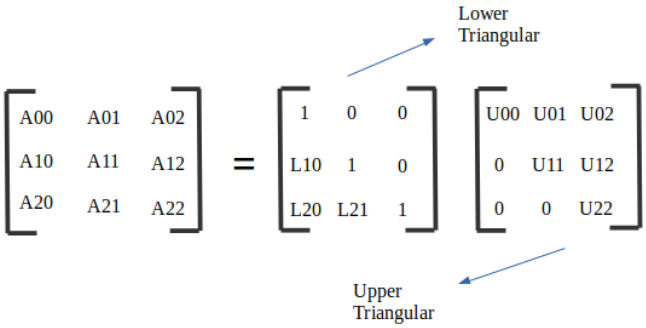

Fig. 1. Example of L and U matrices

C. *Importance of Parallelization in numerical methods*

As the scale and complexity of computational problems grow, the limitations of serial computing become increasingly apparent. Parallel computing emerges as a crucial strategy to overcome these challenges, distributing computational tasks across multiple processors or cores to achieve significant reductions in execution time. In the context of numerical methods such as LU Decomposition, parallelization offers a pathway to tackle large-scale linear systems more efficiently. By leveraging multicore processors and advanced parallel computing techniques, it is possible to enhance the performance of numerical algorithms, making it feasible to solve larger problems within reasonable time frames. This approach not only accelerates computation but also opens up new possibilities for research and application in areas that demand high computational power.

III. Developing the Parallel LU Decomposition Algorithm

A. *Motivation for Parallelization*

Solving increasingly complex linear systems necessitated a transition from conventional sequential solvers to a parallel computational strategy. Sequential solvers, effective for smaller matrices, encounter performance bottlenecks with larger matrices, resulting in impractical computation times for real-world applications. Parallelization overcomes these limitations by distributing computational tasks among multiple processors, significantly reducing execution time, and facilitating the processing of larger matrices more efficiently. This is particularly beneficial for LU Decomposition, where the operation's complexity grows with the matrix size. Initially, our project began with a serial LU Decomposition algorithm. This sequential framework served as a baseline, from which we explored parallelization of various components, notably the LU Decomposition process itself [5].

B. *Initial Challenges and Iterative Solution*

Transitioning to a parallel LU Decomposition algorithm introduced significant initial challenges. Primary among these was the emergence of race conditions, where concurrent threads' attempts to access shared data led to inconsistent and incorrect outcomes. Moreover, our initial parallel algorithm struggled with maintaining the sequential dependencies crucial to the LU Decomposition process, often resulting in incorrect computations outside of single-threaded execution.

To address these challenges, we embarked on an intensive iterative debugging and refinement process. We incorporated copious synchronization mechanisms, like barriers and mutex locks. These were added in effort to ensure all threads completed their computation steps before progressing, thus preserving essential operation order. Initially, to an attempt to rectify the race condition, we utilized an excessive number of barriers. Our hope was to achieve a functional parallel program that we could then refine by removing unnecessary barriers. Despite these efforts, the race conditions persisted, highlighting fundamental flaws in our parallelization algorithm.

C. *Application of the Bulk Synchronous Parallel Model for Parallel Computing*

A significant breakthrough came with the adoption of the Bulk Synchronous Parallel (BSP) model. Utilizing a structured parallel algorithm design, this approach emphasized understanding the algorithm's data flow and computational steps, enabling a more efficient distribution of tasks among processors [6].

1) Integrating Bulk Synchronous Parallel for Enhanced Parallelization: The BSP model informed our division of the LU Decomposition algorithm into phases executable in parallel while respecting inherent data dependencies. It required strategic matrix division into blocks, allowing concurrent but cohesive work on different matrix parts by each processor. This approach minimized communication overhead among processors and ensured balanced workload distribution, critical in avoiding inefficiencies.

2) Practical Implementation of Bulk Synchronous Parallel Model: Implementing our parallel LU Decomposition within the BSP framework involved a few critical steps: dividing the input matrix into smaller blocks for concurrent processing, managing data dependencies meticulously, and identifying and optimizing performance bottlenecks. This systematic approach significantly improved the algorithm's performance, as demonstrated by substantial computation time reductions across various matrices and computational environments.

3) Demonstrating Performance Improvements: Adopting the BSP model not only resolved race condition issues but also led to performance improvements. These outcomes underscore parallel computing's transformative potential in numerical methods, particularly for complex, large-scale problems.

4) Code Reference: Our parallel LU Decomposition implementation, leveraging POSIX threads, is encapsulated in a C program. Here's a snippet demonstrating the BSP model's application in structuring parallel execution within a POSIX thread framework:

```
// cc -lpthread src/parallel_solver.c -o bin/parallel_solver -std=gnu11
...
void* parallel_portion(void* thread_data) {
   struct thread_data* my_data;
   my_data = (struct thread_data*) thread_data;
   int id = my_data->id;
   int k = my_data->k;
   ...
}
...
int main(int argc, char *argv[]) {
   ...
   for (int k = 0; k < n; k++) {
      ...
      for (int i = 0; i < numThreads; i++) {
         pthread_create(&threads[i], NULL, parallel_portion, (void*)&thread_data_array[i]);
      }
      ...
   }
   ...
}
```

Fig. 2. Bulk Synchronous Parallel model code structure for parallel execution within POSIX framework

IV. Data Distribution and Efficient Communication

A. Strategy for Workload Distribution in a Parallel Environment

Effective workload distribution is crucial in parallel computing to maximize utilization of available computational resources. For our parallel LU Decomposition algorithm, we adopted a strategy that involves partitioning the matrix into submatrices or blocks that could be processed independently. This method not only facilitated parallel computation but also allowed for a more balanced distribution of work. Each processor was assigned a portion of the matrix that was roughly equal in computational load, ensuring that all processors contributed equally to the overall task. This approach helped avoid scenarios where certain processors remained idle while others were overloaded, ultimately leading to inefficiencies in the computation process.

B. Minimizing Communication Overhead for Efficiency

One of the key challenges in parallel computing is the communication overhead that arises when processors need to exchange data. Excessive communication can significantly hinder the performance gains achieved through parallelization. To minimize this overhead in our parallel LU Decomposition algorithm, we carefully designed the data distribution scheme and computational steps to reduce the need for inter-processor communication.

We employed a Cartesian matrix distribution scheme, which allocates matrix rows and columns cyclically across processors. This scheme was chosen because it aligns with the data access patterns of the LU Decomposition, allowing most of the necessary data to be locally available within each processor. When communication was unavoidable, we optimized the communication patterns to make them as efficient as possible, employing techniques such as collective communication operations and two-phase broadcasting. These strategies were instrumental in reducing the bandwidth and latency impacts of data exchange, thereby maintaining the high computational efficiency of the parallel algorithm.

Furthermore, synchronization points were strategically placed to ensure that communication occurred only when necessary, and all processors maintained a consistent state in the computation process. This careful management of data distribution and communication not only minimized the overhead but also played a significant role in enhancing the overall performance and scalability of the parallel LU Decomposition algorithm.

V. Addressing Data Depenedencies and Algorithm Correctness

Maintaining algorithmic integrity and ensuring correctness in a parallel setting, especially for complex numerical methods like LU Decomposition, requires careful consideration of data dependencies. Data dependencies are critical in ensuring that the computation proceeds correctly and yields accurate results. In the parallel LU Decomposition algorithm, we adopted several strategies to address these dependencies effectively.

A. Maintaining Algorithmic Integrity

One of the primary concerns in parallelizing the LU Decomposition was preserving the sequential logic inherent in the algorithm, despite the need for concurrent execution of operations. The key to maintaining algorithmic integrity lay in understanding and respecting the data dependencies within the decomposition process. To achieve this, we employed a structured approach that mapped out the dependencies in the algorithm and used this map to guide the parallel execution.

B. Dynamic Scheduling and Synchronization

To address the challenge of data dependencies, dynamic scheduling of computations was implemented. This approach allowed the algorithm to adapt to dependencies dynamically. Each step in the LU Decomposition was carefully synchronized to ensure that no operation proceeded before a prerequisite computation was completed. This synchronization was achieved with the use of barriers, which were added to stop threads at specific locations until all threads reached that same point. This strategy ensured that the updated elements of the matrix were correctly computed in parallel, adhering to the required sequential order of operations.

C. Barrier Synchronization

Barrier synchronization was particularly crucial in ensuring the veracity of the parallel algorithm. Before any thread could proceed to the next step in the decomposition, it had to wait at a barrier until all threads had completed the current step. This synchronization mechanism was essential in preserving the sequential dependencies inherent in the LU Decomposition algorithm, allowing for accurate parallel computation of the L and U matrices.

D. Verification of Correctness

To verify the parallel LU Decomposition, we conducted extensive testing and validation. The computed L and U matrices were used to reconstruct the original A matrix, and the results were compared against the original. This step-by-step verification process confirmed the accuracy of the parallel decomposition, ensuring that the algorithm maintained its integrity and produced reliable results despite the parallel execution.

By carefully managing data dependencies and employing dynamic scheduling and synchronization strategies, we were able to maintain the algorithmic integrity of the LU Decomposition in a parallel setting. This careful approach to parallelization ensured that the algorithm remained correct and efficient, making it a robust solution for solving large systems of linear equations.

VI. Implementation Details

The development of a parallel LU Decomposition algorithm for solving large systems of linear equations, with improved efficiency and enhanced numerical stability, was a multi-faceted process. This section explores the nuances of our implementation, focusing on parallel execution adaptation, partial pivoting integration for stability, and a thorough approach to debugging and verification.

A. Parallel LU Decomposition Framework

The implementation began with the division of the LU Decomposition process into tasks that could be executed concurrently, leveraging the computational power of multiple cores. Each thread was assigned a portion of the matrix to work on, allowing for simultaneous calculation of the L and U matrices. The implementation utilized POSIX threads (pthreads), enabling a structured and efficient approach to parallel computation. The algorithm was structured around the Bulk Synchronous Parallel (BSP) model, which facilitated a clear division of labor and synchronization points among the threads.

To manage the threads and their respective tasks, a **struct** containing thread metadata (such as thread ID and current step of decomposition) was used. This struct was instrumental in orchestrating the parallel execution, ensuring that each thread knew its role in the decomposition process and could synchronize with the others appropriately.

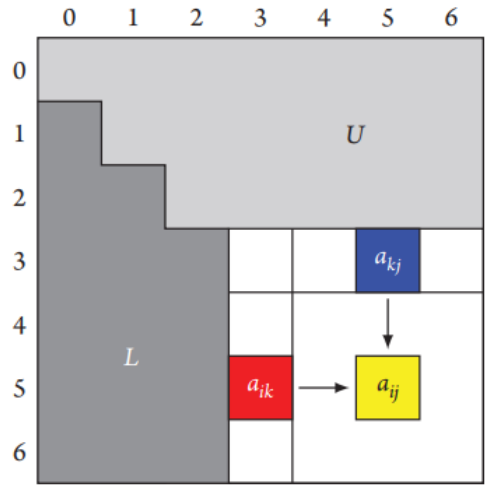

Fig. 3. LU Decomposition of a matrix from [6]

B. Integration of Partial Pivoting

Recognizing the importance of numerical stability in the LU Decomposition, particularly for ill-conditioned matrices, we incorporated partial pivoting into the parallel algorithm. Partial pivoting involves selecting the largest pivot element in each column (from the current pivot row downwards) to be used in the decomposition process. This approach minimizes the impact of numerical errors and enhances the stability of the solution.

Implementing partial pivoting in a parallel setting posed additional challenges, particularly in synchronizing the row swaps across all threads. To address this, the algorithm included a pivoting phase before the main decomposition at each step. During this phase, the algorithm identified the maximum pivot for the current column across the matrix and performed the necessary row swaps before proceeding with the decomposition. This ensured that all threads worked with the correctly pivoted matrix, maintaining numerical stability throughout the process.

C. Synchronization for Numerical Stability

The incorporation of partial pivoting necessitated rigorous synchronization mechanisms to ensure the integrity of the matrix at each step of the decomposition. Before any thread could update its portion of the matrix, it had to ensure that the pivoting process was complete. This synchronization was achieved through barriers, where threads paused until the pivot selection and row swaps were finalized, ensuring consistency across the matrix and preserving the accuracy of the decomposition.

D. Advanced Debugging Tools and Verification

To ensure algorithm correctness and numerical stability, we employed several advanced debugging tools and verification methods:

1) Matrix Print Function: A debugging tool that prints any given matrix. It was invaluable for small to medium-sized matrices, allowing us to visually inspect the correctness of the LU decomposition process and the resulting L and U matrices.

2) Serial Matrix Multiplication for Validation: We developed a serial matrix multiplication function to verify the accuracy of our LU decomposition. By multiplying the L and U matrices, we could compare the product to the original matrix A, ensuring the decomposition's integrity.

3) Residual Function: For larger matrices, where visual inspection was impractical, we implemented a residual function. This function computes the difference between the original matrix A and the product of the L and U matrices (LU), then calculates the Frobenius norm of the resulting matrix. This quantitative measure provided a precise evaluation of the LU decomposition's accuracy, with values near zero indicating high fidelity.

4) Performance and Verification Testing: The parallel LU Decomposition, with and without partial pivoting, underwent extensive performance analysis and correctness testing. This involved comparing parallel algorithm outputs against known solutions and serial implementations, under various matrix sizes and computational conditions. Performance metrics, especially decomposition time, were rigorously recorded to assess parallelization benefits.

This multi-pronged approach to debugging and verification played a pivotal role in refining our parallel LU

Decomposition algorithm. It ensured not only speed enhancements but also upheld the stringent accuracy and stability standards necessary for computational mathematics [7].

VII. Tooling and Debugging Strategies

In the development of our parallel LU Decomposition algorithm, a suite of utility programs and scripts played a pivotal role in enhancing the efficiency of the debugging process, facilitating systematic performance evaluation, and optimizing the algorithm's implementation. These tools not only underscored the methodical approach taken towards the project's challenges but also serve as a repository of practical solutions that could be leveraged by other researchers and practitioners in their computational endeavors. Below, we detail these tools and their contributions to the project's success.

A. Matrix Generator Script

Our matrix generation script, implemented in Python, automated the creation of test matrices of specified dimensions. This utility utilized the numpy library to generate matrices and accompanying vector b, ensuring a wide variety of test cases for comprehensive algorithm testing. The script's capability to produce matrices systematically allowed for efficient and consistent evaluation across different scales of complexity.

B. Data Visualization Script

To analyze and present the performance data derived from our algorithm's execution, we developed a data visualization script, also in Python, leveraging the matplotlib library. This script parsed benchmark data files to plot performance metrics, such as execution time and speedup, against varying numbers of processors. The visualizations provided intuitive insights into the algorithm's scalability and efficiency, aiding in the identification of optimization opportunities.

C. Benchmark Collection Scripts

A series of Perl scripts were created to automate the execution of the parallel solver across a range of thread counts, collecting execution times and calculating speedups. The outputs were structured into organized files for easy analysis. These scripts streamlined the collection of performance benchmarks, allowing for a systematic evaluation of the algorithm under various computational loads and thread configurations.

D. Makefile for Compilation and Management

To manage the compilation and maintenance of the different solver versions (sequential, parallel, with and without pivoting), a Makefile was employed. This tool simplified the build process, ensuring consistent compilation parameters and facilitating rapid iterations during development. The Makefile also supported clean-up operations, maintaining an organized development environment.

These tooling and debugging strategies were instrumental in navigating the complexities of parallel algorithm development. By providing a framework for generating diverse test cases, visually analyzing performance data, systematically collecting benchmarks, and streamlining compilation tasks, these tools significantly contributed to the project's methodical approach towards achieving computational efficiency and algorithmic accuracy.

VIII. Numerical Stability and Accuracy

In the pursuit of developing a parallel LU Decomposition algorithm, maintaining numerical stability emerged as a paramount concern, given its critical role in ensuring the algorithm's correctness and reliability. This section delves into the considerations that guided our decision to integrate partial pivoting, the challenges encountered in preserving numerical stability in a parallel computing context, and the methodologies employed to validate the algorithm's accuracy.

A. Decision to Implement Partial Pivoting

Our decision to incorporate partial pivoting into the parallel LU Decomposition algorithm was motivated by its well-documented advantages in enhancing numerical stability, especially for ill-conditioned matrices. Partial pivoting, by selecting the largest pivot element in each column to minimize rounding errors during the decomposition process, significantly reduces the risk of numerical instability. However, integrating this feature in a parallel setting introduced additional complexities, primarily related to synchronizing row swaps across all processing threads. Addressing these challenges required meticulous planning and implementation strategies to ensure that the benefits of increased stability were not offset by potential performance trade-offs.

B. Challenges of Numerical Stability in Parallel Computing

Maintaining numerical stability in a parallel computing environment presented a unique set of challenges. The concurrent execution of decomposition steps, while efficient, necessitated a robust framework to manage dependencies and ensure the integrity of calculations across multiple threads. The inherent asynchronicity of parallel processing raised concerns about the potential amplification of numerical errors, making the implementation of partial pivoting and synchronization mechanisms all the more critical. Overcoming these hurdles was essential to preserving the algorithm's accuracy without compromising its parallel efficiency.

C. Validation of Algorithm Correctness

To verify the correctness and numerical stability of our parallel LU Decomposition algorithm, we employed several validation methods:

1) *Hand Calculations for Small Matrices:* For matrices of small dimensions, we conducted hand calculations to derive the expected L and U matrices and solution vectors. These manual verifications served as a benchmark, ensuring that our algorithm produced accurate decompositions and solutions for basic test cases.

2) *Residual Function:* For larger matrices, where hand calculations were impractical, we developed a residual function. This function calculates the difference between the product of the computed L and U matrices (LU) and the original matrix A, subsequently computing the Frobenius norm of this

residual matrix. A Frobenius norm approaching zero indicated a high level of accuracy in the LU decomposition, affirming the effectiveness of our parallel algorithm and the implementation of partial pivoting in maintaining numerical stability across varied test cases.

These methodologies underscore the complexity and importance of ensuring numerical stability and accuracy in numerical methods, particularly when adapted for parallel computing frameworks. Through careful implementation, rigorous validation, and strategic integration of partial pivoting, we successfully addressed the challenges associated with numerical stability, reinforcing the reliability and efficacy of our parallel LU Decomposition algorithm.

IX. Performance Evaluation and Optimization

A. Experimental Setup

Our experimental setup was meticulously designed to evaluate the parallel LU Decomposition algorithm's performance across a spectrum of matrix sizes and computational environments. We employed matrices ranging from small (3x3) to large scales (up to 10000x10000), encompassing both sparse and dense configurations to thoroughly test the algorithm under varied conditions. The experiments were conducted on two distinct hardware platforms: a Linux system equipped with an Intel Xeon E312xx (Sandy Bridge) processor featuring 32 CPUs and a MacBook Pro with an M3 Pro chip, housing 11 cores. This dual-platform approach allowed us to assess the algorithm's adaptability and performance across different computational resources.

B. Selection Criteria for Matrices and Hardware

The choice of matrices for our experiments was guided by the need to cover a broad range of scenarios, from simple to complex, ensuring a comprehensive evaluation of the algorithm's capabilities. Similarly, the selection of hardware platforms was driven by the desire to examine the algorithm's performance across varying levels of computational power and architecture. The contrasting characteristics of the Intel Xeon and M3 Pro processors provided valuable insights into how different hardware configurations impact the algorithm's speedup and efficiency.

C. Analysis of Performance Data

Our performance analysis focused on measuring execution time and calculating speedup achieved through parallelization. Speedup, defined as the ratio of execution time on a single core to that on multiple cores, served as a primary metric for evaluating the algorithm's scalability. Our findings revealed significant speedups, particularly on the Linux system with its higher CPU count, demonstrating the algorithm's ability to leverage multicore architectures effectively. However, the analysis also highlighted the diminishing returns on speedup as the number of threads exceeded the physical core count, underscoring the importance of optimizing thread usage and workload distribution.

Furthermore, we explored optimization strategies aimed at enhancing performance, such as minimizing communication overhead between threads and balancing the computational load more evenly across processors. These optimizations contributed to more efficient parallel execution, reducing bottlenecks and improving overall computation time.

Through detailed experimental setup, careful selection of test matrices and hardware configurations, and thorough performance analysis, we gained a deeper understanding of the factors influencing our parallel LU Decomposition algorithm's efficiency and scalability. This comprehensive evaluation not only confirms the benefits of parallelization but also highlights potential areas for future optimization, setting the stage for further advancements in parallel computing applications in numerical linear algebra [8].

X. Experimental Results

The experimental evaluation of our parallel LU Decomposition implementation focused on its performance across different computational environments, specifically contrasting its execution on Linux systems equipped with Intel Xeon E312xx processors and a MacBook Pro featuring an M3 Pro chip. These systems provided a diverse range of hardware capabilities, allowing for a comprehensive assessment of the algorithm's scalability and efficiency.

A. Performance Evaluation on Different Systems

The Linux platform, boasting 32 CPUs, offered a robust environment for assessing the scalability of our parallel solution. For a 5000x5000 matrix, leveraging all available processors yielded a notable decrease in computation time, achieving a maximum speedup of 7.934753 with 27 threads for this matrix size. This trend was even more pronounced with a 10000x10000 matrix, where the speedup improved significantly, peaking at 10.781217 with 31 threads, showcasing the parallel algorithm's capacity to efficiently utilize extensive computational resources.

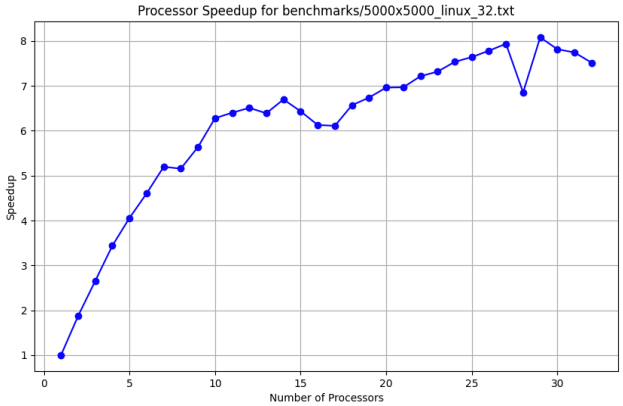

Fig. 4. Processor speedup of 5000x5000 matrix on 32-core Linux machine

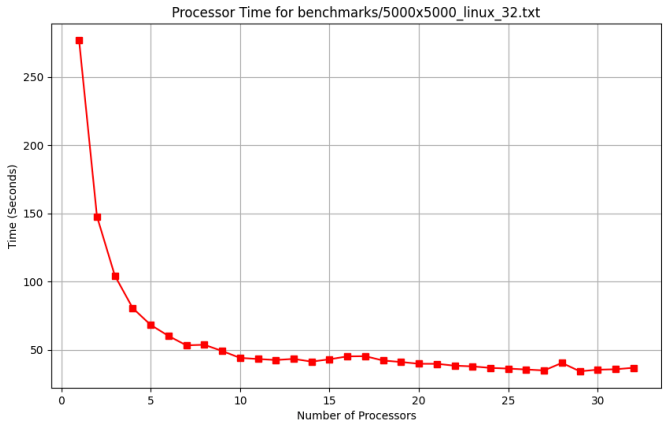

Fig. 5. Processor time of 5000x5000 matrix on 32-core Linux machine

In contrast, the MacBook Pro, constrained by its 11-core architecture, illustrated the algorithm's adaptability to hardware limitations. Despite the core saturation, significant speedups were observed; for a 5000x5000 matrix, the highest speedup recorded was 4.940405 with 10 threads. This efficiency, even within a more restricted hardware context, underscores the parallel algorithm's effectiveness across varying system architectures.

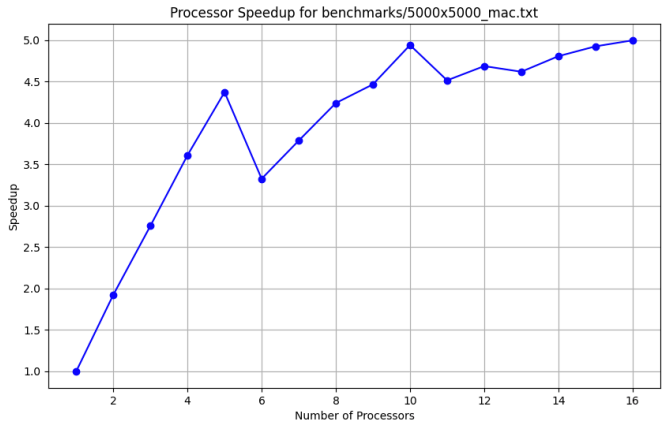

Fig. 6. Processor speedup 5000x5000 matrix on 32-core Linux machine

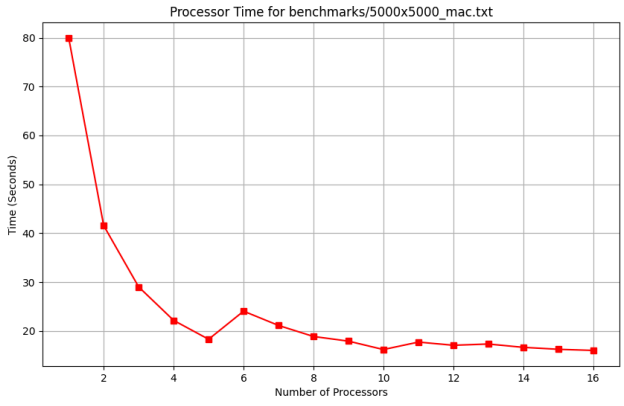

Fig. 7. Processor time 5000x5000 matrix on 11-core MacBook Pro

B. *Analysis of Speedup and Efficiency with Varying Processor Counts*

The experiments clearly demonstrated that the speedup and efficiency of our parallel LU Decomposition algorithm are directly influenced by the number of processors (threads) utilized. On both Linux and Mac systems, an increase in the number of threads generally resulted in better performance, up to a certain point where the overhead of managing additional threads outweighed the benefits of parallel execution.

On the Linux system, the speedup trends exhibited diminishing returns as the thread count approached the physical limit of the CPU cores, indicating an optimal range for thread utilization that maximizes performance without incurring excessive overhead. Similarly, on the MacBook Pro, speedups plateaued beyond a certain number of threads, highlighting the importance of tailoring parallel execution strategies to the specific hardware characteristics of the execution environment.

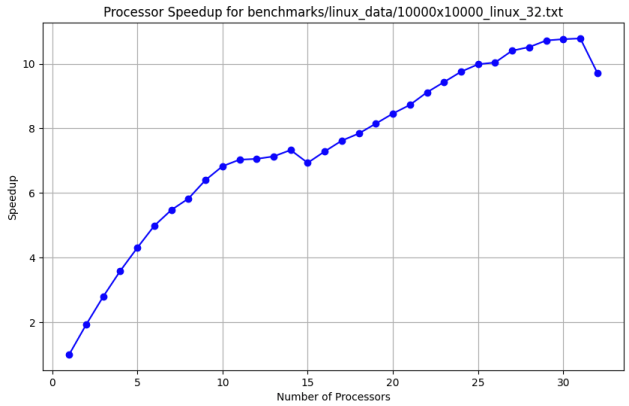

Fig. 8. Processor speedup5000x5000 matrix on 32-core Linux machine

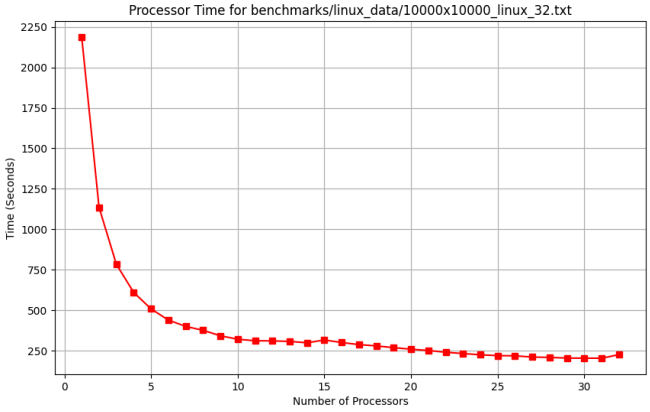

Fig. 9. Processor time 10000x10000 matrix on 32-core Linux machine

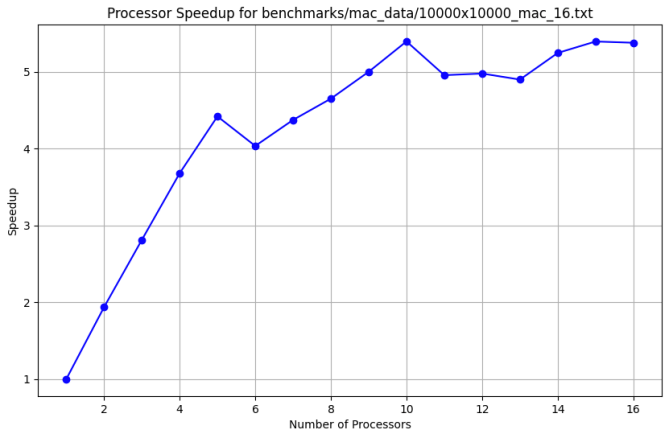

Fig. 10. Processor speedup 5000x5000 matrix on 32-core Linux machine

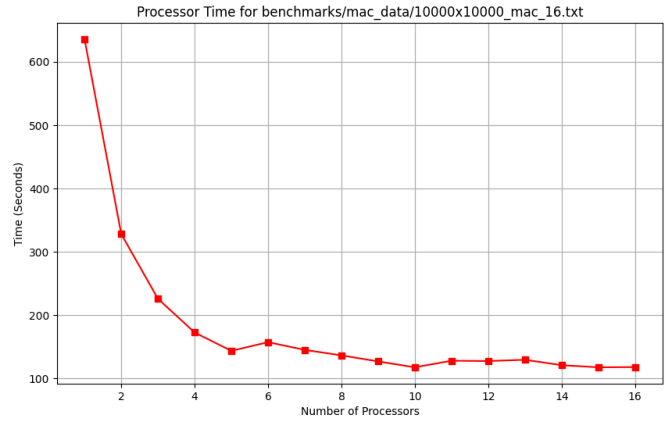

Fig. 11. Processor time 10000x10000 matrix on 11-core MacBook Pro

C. *Comparison of Performance with and Without Pivoting*

The incorporation of partial pivoting into the parallel LU Decomposition algorithm introduced an additional layer of complexity to the computation, potentially impacting performance due to the overhead of row swapping and synchronization. However, our experiments revealed that while the version with partial pivoting generally exhibited slightly longer computation times compared to the non-pivoting variant, the difference in performance was marginal.

This negligible performance trade-off is justified by the significant increase in numerical stability and accuracy provided by partial pivoting, especially for ill-conditioned matrices. Therefore, the choice between using partial pivoting or not can be made based on the specific requirements of numerical stability and the nature of the matrices being decomposed [10].

XI. Challenges and Lessons Learned

The journey to develop a parallel LU Decomposition algorithm for solving systems of linear equations efficiently has been both challenging and enlightening. This section reflects on the hurdles we encountered, the iterative debugging process, and the critical lessons learned throughout the project.

A. *Parallelization Challenges*

Parallelizing the LU Decomposition algorithm introduced several initial challenges, notably race conditions and maintaining the algorithm's integrity due to concurrent thread execution. Overcoming these required a deep understanding of the algorithm's underlying mathematics and the intricacies of parallel computing. The transition from a sequential to a parallel approach was a complex process that necessitated iterative debugging and a methodical testing strategy.

B. *Insights from Debugging*

Debugging played a pivotal role in refining our parallel algorithm. Through this process, we learned the importance of employing a range of tools and strategies, from simple print statements to sophisticated profiling techniques. The development of utility scripts for generating matrices and collecting performance data was invaluable. These tools not only facilitated a more efficient debugging process but also provided insights into optimizing our algorithm for better performance and stability.

C. *Importance of Numerical Stability*

Our project underscored the critical importance of maintaining numerical stability in parallel algorithms. The integration of partial pivoting to enhance stability introduced additional complexity but was necessary to ensure accurate results across all matrix sizes and types. This experience highlighted the delicate balance between optimizing for speed and preserving the integrity of the computational results. Furthermore, to quantitatively make sense of our results and analysis we referred to [9].

D. *Lessons for Future Projects*

1) Structured Parallelization Approach: Employing a structured approach to parallelization, such as the BSP model, can significantly improve the development process and result in more robust algorithms.

2) Debugging and Optimization Tools: Developing and utilizing a suite of debugging and optimization tools is crucial for identifying and addressing performance bottlenecks and ensuring algorithm correctness.

3) Focus on Numerical Stability: Ensuring numerical stability is as important as optimizing for performance in numerical algorithms. This often requires careful consideration of algorithm modifications and their implications.

Reflecting on these challenges and lessons learned offers valuable insights not only for the continuation of this project but also for the broader community engaged in parallel computing and numerical analysis. These experiences underscore the complexity of developing efficient parallel algorithms and highlight the importance of a thorough and methodical approach to algorithm design, debugging, and optimization.

XII. Discussion

The implementation and rigorous testing of our parallel LU Decomposition algorithm provided valuable insights into the optimization of numerical methods for solving systems of linear

equations on multicore processors. This section discusses the key takeaways from the development process, the impact of hardware limitations, and the comparative effectiveness of different parallelization strategies.

A. Insights from Implementation and Testing

The transition from a sequential to a parallel algorithm underscored the critical importance of understanding and managing data dependencies to ensure algorithmic correctness and efficiency. The use of the Bulk Synchronous Parallel (BSP) model facilitated a structured approach to parallelization, enabling a clear division of tasks and synchronization points that preserved the integrity of the LU Decomposition process across multiple threads. The iterative refinement of our parallel algorithm, guided by both theoretical insights and empirical observations, was instrumental in overcoming initial challenges, such as race conditions and ineffective barrier synchronization.

B. Consideration of Hardware Limitations and Their Effects

Our experiments across diverse hardware platforms highlighted the significance of tailoring parallel algorithms to the specific characteristics of the execution environment. On systems with a high number of CPUs, such as the Linux server equipped with an Intel Xeon processor, our parallel LU Decomposition algorithm achieved substantial speedups, demonstrating its ability to leverage extensive computational resources effectively. Conversely, on the MacBook Pro, with a more limited core count, the algorithm still realized significant efficiency gains, albeit within the constraints imposed by the hardware. This adaptability is crucial for the development of scalable and efficient parallel algorithms that can operate optimally across a range of computational environments.

C. Comparative Analysis of the Parallel Approaches

The implementation of our algorithm with and without partial pivoting offered an opportunity to compare these two parallel approaches in terms of performance and numerical stability. The non-pivoting version, while slightly faster due to the absence of row swapping overhead, was more susceptible to numerical instability for certain types of matrices. The partial pivoting variant, although marginally slower, provided enhanced stability by selecting the optimal pivot at each step of the decomposition process. This comparative analysis underscores the trade-offs between computational speed and accuracy inherent in parallel numerical algorithms and highlights the necessity of choosing the appropriate strategy based on the specific requirements of the task at hand.

XIII. Summary and Conclusions

This project embarked on the ambitious task of enhancing the computational efficiency of solving systems of linear equations through parallelizing the LU Decomposition algorithm. Our journey from conceptualizing a parallel approach to its successful implementation and rigorous testing across different computational environments has yielded significant insights and contributions to the field of numerical linear algebra and parallel computing. This section provides a recap of our objectives, findings, contributions, and contemplates future directions for extending this work.

A. Recap of Objectives and Findings

Our primary objective was to develop a parallel LU Decomposition algorithm that could leverage multicore processors to achieve substantial reductions in computation time for solving large systems of linear equations. Through the implementation of both non-pivoting and partial pivoting versions of the algorithm, we demonstrated the feasibility and efficiency of parallel computation in this domain. Our findings indicate that parallelizing the LU Decomposition can lead to significant speedups, especially on systems with a high number of CPUs. Moreover, incorporating partial pivoting into the parallel framework enhanced the numerical stability of the algorithm, making it suitable for a broader range of matrices.

B. Contributions

- The development of a parallel LU Decomposition algorithm that effectively distributes the workload across multiple threads while maintaining algorithmic integrity and efficiency.
- A comprehensive performance evaluation that demonstrates the scalability of the algorithm across different hardware platforms and matrix sizes.
- An analysis of the trade-offs between computational speed and numerical stability, highlighting the benefits of partial pivoting in certain contexts.
- Insights into the challenges of parallelizing numerical methods and strategies for overcoming these challenges, including data dependency management and synchronization.

C. Significance of Parallelizing LU Decomposition

Parallelizing LU Decomposition represents a significant advancement in our ability to solve large systems of linear equations more efficiently. This work underscores the potential of parallel computing to transform numerical methods, enabling researchers and engineers to tackle more complex problems within shorter time frames. By reducing computation time without compromising accuracy, parallel algorithms like the one developed in this project can significantly accelerate the pace of scientific discovery and innovation.

D. Future Directions

- Exploring more sophisticated parallelization strategies, such as dynamic scheduling and asynchronous communication, to further optimize performance and scalability.
- Applying the parallel LU Decomposition algorithm to real-world problems in engineering, physics, and other fields to validate its utility and adaptability.
- Investigating the integration of our parallel algorithm into high-performance computing frameworks and software libraries to make it more accessible to the broader scientific community.

- Extending the algorithm to support sparse matrices and specialized matrix structures, broadening its applicability.

In conclusion, this project has made a substantial contribution to the parallelization of numerical methods, offering a powerful tool for solving systems of linear equations more efficiently. The insights and methodologies developed through this work lay a solid foundation for future research and application in parallel computing and numerical linear algebra.

ACKNOWLEDGMENT

We express our gratitude to our ECE 688 instructor, Yuchen Huang, and Adjunct Support Faculty, Venkatesh Srinivas, for their invaluable guidance and support throughout this project.

REFERENCES

[1] K. Hartnett, "New algorithm breaks speed limit for solving linear equations," Quanta Magazine, *quantamagazine.org*, Mar. 8, 2021. [Online]. Available: https://www.quantamagazine.org/new-algorithm-breaks-speed-limit-for-solving-linear-equations-20210308. [Accessed Jan. 28, 2024].

[2] A. A. Abouelfarag, N. M. Nouh and M. ElShenawy, "Improving Performance of Dense Linear Algebra with Multi-core Architecture," 2017 International Conference on High Performance Computing & Simulation (HPCS), Genoa, Italy, 2017, pp. 870-874.

[3] T. J. Dekker, W. Hoffmann, and K. Potma, "Parallel algorithms for solving large linear systems," *Journal of Computational and Applied Mathematics*, vol. 50, no. 1–3, pp. 221–232, 1994.

[4] D. Kaya and K. Wright, "Parallel algorithms for LU decomposition on a shared memory multiprocessor," *Applied Mathematics and Computation*, vol. 163, no. 1, pp. 179–191, Apr. 2005.

[5] Q. Chunawala, "Fast algorithms for solving a system of linear equations," Baeldung, *baeldung.com*, Jun. 13, 2023. [Online] Available: https://www.baeldung.com/cs/solving-system-linear-equations. [Accessed Jan. 28, 2024].

[6] R. H. Bisseling, "LU Decomposition," in *Parallel Scientific Computation: A Structured Approach Using BSP*, Oxford, U.K.: Oxford University Press , 2020, ch. 2.

[7] R. Mahfoudhi, "High Performance Recursive LU Factorization for Multicore Systems," 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA), Hammamet, Tunisia, 2017, pp. 668-674

[8] E. E. Santos and M. Muralcetharan, "Analysis and Implementation of Parallel LU-Decomposition with Different Data layouts," University of California, *math.ucr.edu* Jun. 2000. [Online] Available: https://math.ucr.edu/~muralee/LU-Decomposition.pdf. [Accessed Jan. 28, 2024].

[9] A. Bushnag, "Investigating the Use of Pipelined LU Decomposition to Solve Systems of Linear Equations," 2020 International Conference on Computing and Information Technology (ICCIT-1441), Tabuk, Saudi Arabia, 2020, pp. 1-5.

[10] Cong Fu and Tao Yang, "Sparse LU Factorization with Partial Pivoting on Distributed Memory Machines," 1996 ACM/IEEE Conference on Supercomputing, Pittsburgh, PA, USA, 1996, pp. 31-31.