Maseeh College of Engineering and Computer Science
PORTLAND STATE UNIVERSITY

ECE 688: WINTER 2024
ADVANCED COMPUTER ARCHITECTURE II

PARALLEL LINEAR EQUATIONS SOLVER

FINAL LOG REPORT

MOHAMED GHONIM
PHANINDRA VEMIREDDY
AHLIAH NORDSTROM
ALEXANDER MASO

03/15/2024

Table of Contents

Getting Started:

First, we started a git repo and created directories with file structures that could be useful in our project. We may not end up using all those files, but this structure will be helpful in enforcing modularity into our program.

```
Parallel Equation Solver
│
├── src/                    # Source files
│   ├── main.c              # Main program entry point
│   ├── lu_serial.c         # Serial LU Decomposition implementation
│   ├── lu_serial.h         # Header for serial LU Decomposition
│   ├── lu_parallel.c       # Parallel LU Decomposition implementation using Pthreads
│   ├── lu_parallel.h       # Header for parallel LU Decomposition
│   └── utilities.c         # Utility functions for matrix operations and timing
│
├── include/                # Header files
│   ├── matrix.h            # Definitions and functions for matrix operations
│   └── timing.h            # Timing and performance measurement utilities
│
├── docs/                   # Documentation files
│   ├── setup.md            # Setup instructions
│   ├── usage.md            # Usage instructions
│   └── development.md      # Notes on development decisions and project structure
│
├── tests/                  # Test files
│   ├── test_serial.c       # Tests for serial implementation
│   └── test_parallel.c     # Tests for parallel implementation
│
├── benchmarks/             # Benchmark scripts and results
│   ├── benchmark_script.sh # Script to run benchmarks
│   └── results.md          # Benchmark results and analysis
│
├── Makefile                # Makefile for building the project
└── README.md               # Project overview and general instructions
```

Before we start any actual coding, we made sure to review how the actual LU Decomposition works, and just see how in general a set of linear equations is transferred into a matrix that's to be decomposed and solved using LU Decomposition.

Step-by-step walk through of the algorithm.

(Making sense of the program we're making): Those examples will be used to code our initial algorithm and verify its working as expected.

Example (1)

Step1: System of Linear Equations

Let's start with the following system of equations:

$$
\begin{aligned}
2x_1 + 3x_2 - x_3 &= 5 \\
4x_1 + x_2 + 2x_3 &= 6 \\
-2x_1 + 2x_2 + 3x_3 &= 8
\end{aligned}
$$

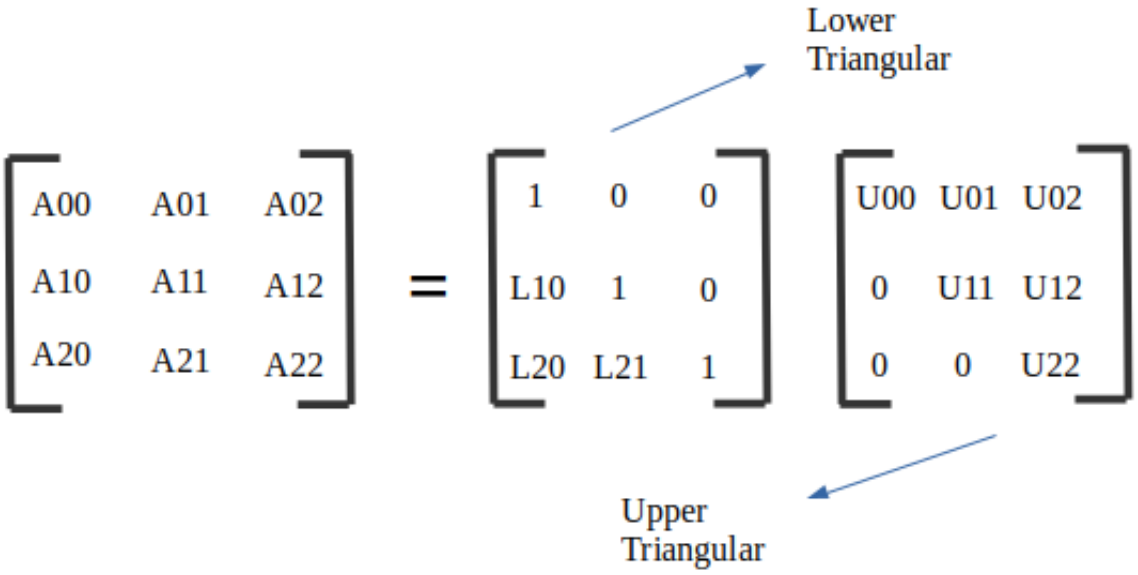
Step2: Transforming into Matrix Form:

This system can be represented in matrix form as Ax=b, where:

$$
A = \begin{pmatrix} 2 & 3 & -1 \\ 4 & 1 & 2 \\ -2 & 2 & 3 \end{pmatrix}, \qquad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \qquad b = \begin{pmatrix} 5 \\ 6 \\ 8 \end{pmatrix}
$$

Step3: LU Decomposition

Next, we decompose A into L and U, where L is a lower triangular matrix and U is an upper triangular matrix.



In our example:
(Step by step derivations of the LU matrices in the appendix)

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 2 & 3 & -1 \\ 0 & -5 & 4 \\ 0 & 0 & 6 \end{pmatrix}$$

Step4: Solving Ly = b for y

Given L and b, we solve for y (which is an intermediate vector, not the final solution).
Using <u>forward substitution.</u>

$$Ly = b, \qquad \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ 8 \end{pmatrix}$$

$y_1 = 5$ (from the first row of L and b)
$2y_1 + y_2 = 6, \qquad 2 \times 5 + y2 = 6, \qquad y2 = -4$
$-y1 - y2 + y3 = 8, \qquad -5 - (-4) + y3 = 8, \qquad y3 = 9$

So

$$\therefore \quad y = \begin{pmatrix} 5 \\ -4 \\ 9 \end{pmatrix},$$

Step5: Solving Ux = y for x

Now, with U and y, we can solve for x using <u>backward substitution.</u>

$$Ux = y, \qquad \begin{pmatrix} 2 & 3 & -1 \\ 0 & -5 & 4 \\ 0 & 0 & 6 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ -4 \\ 9 \end{pmatrix}$$

$6x_3 = 9, \qquad \therefore x_3 = \frac{3}{2}, \qquad$ (from the last row of U and y)
$-5x_2 + 4x_3 = -4, \qquad -5x_2 + 4 \times \left(\frac{3}{2}\right) = -4, \qquad x_2 = 2$
$2x_1 + 3x_2 - x_3 = 5, \qquad 2x_1 + 3 \times 2 - \frac{3}{2} = 5, \qquad x_1 = \frac{1}{4}$

$$\therefore \quad x = \begin{pmatrix} \frac{1}{4} \\ 2 \\ \frac{3}{2} \end{pmatrix}, \qquad \therefore x_1 = \frac{1}{4}, \qquad x_2 = 2, \qquad x_3 = \frac{3}{2}$$

Step6: Verifying the answers:

$$2x_1 + 3x_2 - x_3 = 5$$

$$2\left(\frac{1}{4}\right) + 3(2) - \left(\frac{3}{2}\right) = 5, \qquad \#$$

$$4x_1 + x_2 + 2x_3 = 6$$

$$4\left(\frac{1}{4}\right) + (2) + 2\left(\frac{3}{2}\right) = 6, \qquad \#$$

$$-2x_1 + 2x_2 + 3x_3 = 8$$

$$-2\left(\frac{1}{4}\right) + 2(2) + 3\left(\frac{3}{2}\right) = 8, \qquad \#$$

Example (2):

Step1: System of Linear Equations

$$x_1 + x_2 - x_3 = 4$$
$$x_1 - 2x_2 + 3x_3 = -6$$
$$2x_1 + 3_{x2} + x_3 = 7$$

Step2: Transforming into Matrix Form:

$$\begin{pmatrix} 1 & 1 & -1 \\ 1 & -2 & 3 \\ 2 & 3 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ -6 \\ 7 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 1 & -1 \\ 1 & -2 & 3 \\ 2 & 3 & 1 \end{pmatrix}, \qquad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \qquad b = \begin{pmatrix} 4 \\ -6 \\ 7 \end{pmatrix}$$

Step3: LU Decomposition

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & -\frac{1}{3} & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 1 & 1 & -1 \\ 0 & -3 & 4 \\ 0 & 0 & \frac{13}{3} \end{pmatrix}$$

Step4: Solving Ly = b for y

Given L and b, we solve for y (which is an intermediate vector, not the final solution).
Using forward substitution.

$$\text{Ly} = \text{b}, \qquad \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & -\frac{1}{3} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 4 \\ -6 \\ 7 \end{pmatrix}$$

$y_1 = 4$ (from the first row of L and b)
$y_1 + y_2 = -6, \qquad y2 = -10$
$2y_1 - \frac{1}{3}y_2 + y_3 = 7, \qquad y_3 = -\frac{13}{3}$

So

$$\therefore \quad \text{y} = \begin{pmatrix} 4 \\ -10 \\ -\frac{13}{3} \end{pmatrix},$$

Step5: Solving Ux = y for x

Now, with U and y, we can solve for x using backward substitution.

$$\text{Ux} = \text{y}, \qquad \begin{pmatrix} 1 & 1 & -1 \\ 0 & -3 & 4 \\ 0 & 0 & \frac{13}{3} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ -10 \\ -\frac{13}{3} \end{pmatrix}$$

$\frac{13}{3}x_3 = -\frac{13}{3}, \qquad \therefore x_3 = -1, \qquad$ (from the last row of U and y)
$-3x_2 + 4x_3 = -10, \qquad x_2 = 2$
$x_1 + x_2 - x_3 = 4, \qquad x_1 = 1$

$$\therefore \quad \text{x} = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}, \qquad \therefore x_1 = 1, \qquad x_2 = 2, \qquad x_3 = -1$$

Step6: Verifying the answers:

$$x_1 + x_2 - x_3 = 4$$
$$x_1 - 2x_2 + 3x_3 = -6$$
$$2x_1 + 3_{x2} + x_3 = 7$$

$$x_1 + x_2 - x_3 = 4$$

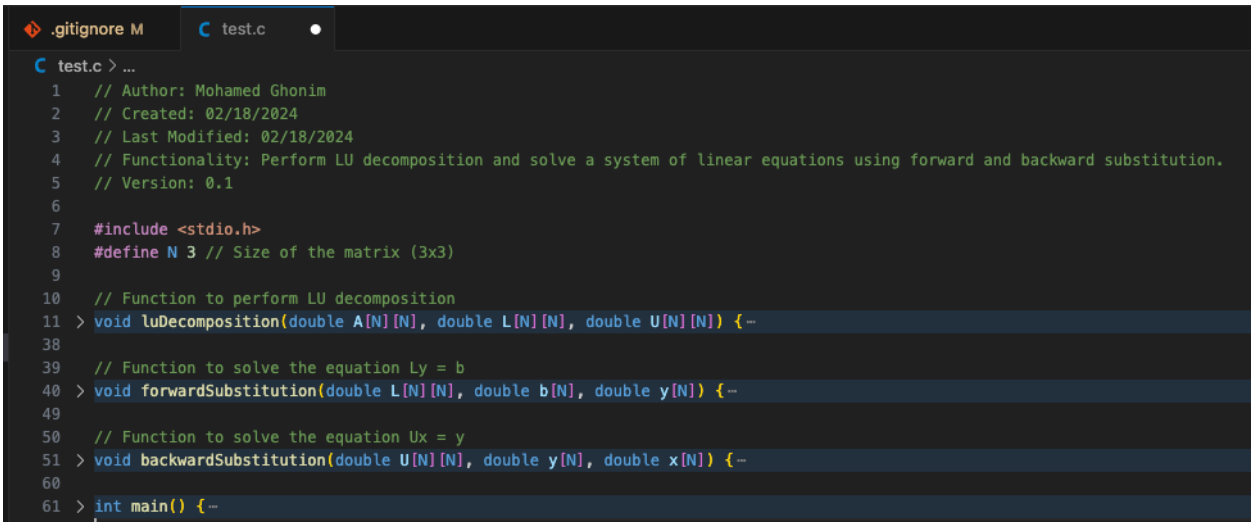
$$(1) + (2) - (-1) = 4, \qquad \#$$

$$x_1 - 2x_2 + 3x_3 = -6$$
$$(1) - 2(2) + 3(-1) = -6, \qquad \#$$

$$2x_1 + 3_{x2} + x_3 = 7$$
$$2(1) + 3(2) + (-1) = 7, \qquad \#$$

Coding a simple serial program to do this calculation:

To get started, we will code a simple C code implementing this algorithm, then we'll use those same examples above to verify the correctness of that algorithm. Next, we'll improve the algorithm to work on bigger matrices, and will use pthreads.
Code structure:



The matrix is defined in the main here, as a next step, we'll have It defined in a separate file or function. (The complete code is in appendix 2).

```
> gcc -o bin/simple src/simple.c
> ./bin/simple
Solution:
x[0] = 0.250000
x[1] = 2.000000
x[2] = 1.500000
```

Testing the program:

Version 0.1:

First example:
The first example we tried was this matrix:

$$A = \begin{pmatrix} 2 & 3 & -1 \\ 4 & 1 & 2 \\ -2 & 2 & 3 \end{pmatrix}, \qquad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \qquad b = \begin{pmatrix} 5 \\ 6 \\ 8 \end{pmatrix}$$

And the answer was:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} \frac{1}{4} \\ 2 \\ \frac{3}{2} \end{pmatrix} = \begin{pmatrix} 0.25 \\ 2 \\ 1.5 \end{pmatrix}$$

When we give the program this matrix:

```c
int main() {
    double A[N][N] = {{2, 3, -1}, {4, 1, 2}, {-2, 2, 3}};
    double b[N] = {5, 6, 8};
```

We get:

```
> ./bin/simple
Solution:
x[0] = 0.250000
x[1] = 2.000000
x[2] = 1.500000
```

Which is the expected solution.

Second example:

$$A = \begin{pmatrix} 1 & 1 & -1 \\ 1 & -2 & 3 \\ 2 & 3 & 1 \end{pmatrix}, \qquad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \qquad b = \begin{pmatrix} 4 \\ -6 \\ 7 \end{pmatrix}$$

And the answer was:

$$x = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix},$$

When we give the program this matrix:

```c
    double A[N][N] = {{1, 1, -1}, {1, -2, 3}, {2, 3, 1}};
    double b[N] = {4, -6, 7};
```

We get:

```
> gcc -o bin/simple src/simple.c
> ./bin/simple
Solution:
x[0] = 1.000000
x[1] = 2.000000
x[2] = -1.000000
```

Which is the expected solution.

Version 0.2:

We then made the program more structured but adding the matrices in the /matrices directory as text files passed to the program using the command-line terminal as the first argument.

For example:

```
> ./bin/simple matrices/3x3_1.txt
Solution:
x[0] = 0.250000
x[1] = 2.000000
x[2] = 1.500000
```

and

```
> ./bin/simple matrices/3x3_2.txt
Solution:
x[0] = 1.000000
x[1] = 2.000000
x[2] = -1.000000
```

Version 0.3:

Now we parameterized the size of the matrix, such that it's passed to the program from the first line in the matrix txt file.

```
// Version: 0.3 : readin the matrix from a file passed as an argument to the program
//              : added a function to free the allocated memory
//              : the matrix size is parameterized, and passed as the first line in
the matrix file
```

We tested the program on multiple 3x3 and 4x4 matrices which we know the answers to previously as a way to debug the code. The program solved all the matrices correctly.
To get and test much bigger matrices, we wrote a python script <utilities/matrix_generator.py> which takes in an argument of the dimension of nodes, and generates a matrix which gates saved in the <matrices/py_generated> directory with $matrix_dimension.txt tile.

```
# matrix_generator.py
# Author: Mohamed Ghonim
# Created: 02/18/2024
```

```python
# Last Modified: 02/18/2024
# Function: This script generates a random matrix of a given dimension and saves it to
a file in matrices/py_generated
# Usage: python matrix_generator.py <dimension>
# version: 0.1

import numpy as np
import sys
import os

def generate_matrix_and_save(dimension):

    # Define the target directory
    target_directory = os.path.join(os.path.dirname(__file__),
"../matrices/py_generated")
    #target_directory = "../matrices"

    # Ensure the target directory exists
    os.makedirs(target_directory, exist_ok=True)

    # Generating a dimension x dimension matrix with random integers between -10 and
10
    matrix = np.random.randint(-10, 10, size=(dimension, dimension))

    # Generating the final row with random integers between -10 and 10
    final_row = np.random.randint(-10, 10, size=(dimension,))

    # Preparing the string representation and defining the file path
    matrix_str = f"{dimension}\n" + "\n".join(" ".join(map(str, row)) for row in
matrix) + "\n" + " ".join(map(str, final_row))
    file_name = os.path.join(target_directory, f"{dimension}x{dimension}.txt")

    # Saving to file
    with open(file_name, 'w') as file:
        file.write(matrix_str)

    print(f"Matrix saved to {file_name}")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python script.py <dimension>")
        sys.exit(1)

    try:
        dimension = int(sys.argv[1])
        if dimension <= 0:
            raise ValueError
        generate_matrix_and_save(dimension)
```

```
    except ValueError:
        print("Please provide a valid positive integer for the matrix dimension.")
```

To further enhance the code, we are not allowing for the user to specify an output file where the solution will get saved, if no output file is specified, the solution gets printed on the terminal.

```
// main.c
// Author: Mohamed Ghonim
// Created: 02/18/2024
// Last Modified: 02/18/2024
// Functionality: Perform LU decomposition and solve a system of linear equations
using forward and backward substitution.
// Version: 0.3 : readin the matrix from a file passed as an argument to the program
//              : added a function to free the allocated memory
//              : the matrix size is parameterized, and passed as the first line in
the matrix file

// Version: 0.4 : allow write the solution to a file passed as an argument to the
program
//              : if no file is passed, the solution will be printed to the standard
output
//              : example usage: matrices/py_generated/5000x5000.txt
matrices_solution/5000x5000.txt
```

Running the program serially:

Currently, we have a functional serial program that takes in a matrix of dimensions N and solves it using LU decomposition and saves the solution to a text file or prints it on the screen.

Parallel Algorithm and Implementation

To speed up the program such that we can get the answer vector [x] quickly, especially in bigger matrices we decided to parallelize the program. Initially, we were hoping to parallelize the entire program, including reading the file, solving the LU Decomposition, and perform the forward and backward decomposition.
We soon realized how intricate it is to try to parallelize I/O operations, and we also realized that, given the sequential nature of the backward and forward substitution operations, it didn't make sense to try to parallelize them. From our research, it seemed that there are some parallelization techniques that we could consider for this funtions, but a ROI of our time and overhead would not be guaranteed. Given our limited time on

this program, we decided to focus on parallelizing the LU Decomposition, which is the most time-consuming part of our program anyway.

Parallelizing LU Decomposition

To enhance the performance of LU decomposition on large matrices, we adapted a parallel approach using POSIX threads (pthreads). This method divides the matrix into submatrices or blocks, assigning each part to a separate thread. This division allows simultaneous execution of the LU decomposition steps on different parts of the matrix, significantly reducing the overall computation time. The question becomes how we can effectively devise/find a parallel algorithm for the LU Decomposition.

Devising a Parallel Algorithm for LU Decomposition

The quest for parallelizing LU decomposition starts with the recognition of the computational intensity and data dependencies inherent in the algorithm. A pivotal strategy involves breaking down the matrix into manageable submatrices or blocks that can be processed concurrently. This approach necessitates a thorough understanding of how data flows through the algorithm and where parallelism can be introduced without violating data dependencies.
Initial Parallel Code Challenges
The first attempt at parallelization exposed several critical issues:

- **Race Conditions**: The premature parallel algorithm led to race conditions, where multiple threads attempted to read and write shared data simultaneously without proper synchronization, leading to unpredictable outcomes. Every time we ran the program, we got a different solution to the matrix, except when we specified a number of threads = 1.
- **Barrier Synchronization**: Implementing an effective barrier synchronization mechanism was crucial to ensure that all threads reached a certain point in the computation before any thread proceeded further. This was necessary to maintain the sequential dependencies of the LU decomposition steps. Given that our parallel algorithm was immature and not really-parallel in-nature, the implementation of the rigorous barrier synchronization was still ineffective.

Refinement Using BSP Model

The turning point came from applying the insights from Rob Bisseling's "Parallel Scientific Computation, a structured approach using BSP" which advocates for a backwards design of parallel algorithms. This approach emphasizes starting with the desired final step of the computation and working backwards to determine the necessary data and computations needed at each step. Such a structured approach aids in identifying the most efficient data distribution and communication patterns required for parallelization.

Implementation Details and Challenges

Data Distribution and Communication

A Cartesian matrix distribution scheme was identified as optimal for distributing the workload across processors while minimizing communication overhead. This scheme distributes matrix rows and columns cyclically across processors, aligning with the computation's data access patterns.
However, challenges remained:

- **Balancing Load**: Ensuring an even distribution of work across processors to avoid idle time and achieve maximum utilization was non-trivial. The cyclic distribution helped, but careful tuning was needed to balance the load effectively across all stages of the computation.
- **Minimizing Communication**: While the Cartesian distribution minimized the need for data transfer, optimizing the communication pattern to reduce the bandwidth and latency impacts was crucial. Techniques like two-phase broadcasting were explored to enhance efficiency.

Overcoming Data Dependencies

The inherent data dependencies in LU decomposition required innovative solutions to maintain the algorithm's correctness in a parallel setting. Implementing a dynamic scheduling mechanism that could adapt to the computation's flow and data dependencies on-the-fly was part of the solution to this challenge.

In our pursuit of optimizing LU Decomposition for large matrices, we developed two parallel programs: a faster implementation without pivoting and a slightly slower one that incorporates partial pivoting for increased robustness. The no-pivoting version offers speed advantages, particularly suitable for matrices where stability is not a concern. On the other hand, the partial pivoting version, although marginally slower, ensures greater numerical stability, making it a preferable choice for a broader range of matrices where accuracy and reliability are paramount.

Code Structure and Implementation

Parallel LU Decomposition without pivoting

The core of our parallel LU Decomposition implementation leverages the Bulk Synchronous Parallel (BSP) model, following the strategy outlined by Rob Bisseling. This approach divides the LU Decomposition process into discrete steps, ensuring that data dependencies are maintained, and that parallel execution does not compromise the algorithm's correctness.

Parallel Execution Framework

- We use POSIX threads (**pthreads**) to enable parallel computation across multiple cores. Each thread is responsible for a portion of the matrix, working concurrently to perform the decomposition.
- A **struct thread_data** is defined to pass necessary information to each thread, including the thread ID and the current step (**k**) in the decomposition process.

Partial Pivoting and Thread Synchronization

- To accommodate matrices that might lead to numerical instability, we implemented a version with partial pivoting. This approach involves selecting the largest pivot in each step to minimize rounding errors, a feature especially crucial when dealing with ill-conditioned matrices.
- Synchronization points (**Barrier**) ensure that no thread proceeds to the next step of the decomposition until all threads have completed their current step. This barrier mechanism is critical for preserving the sequential dependencies inherent in the LU Decomposition algorithm.

Key Functions

- **parallel_portion(void* thread_data)**: The function executed by each thread. It updates the matrix **A** based on the current decomposition step **k**, ensuring that computations are spread across threads to leverage parallelism effectively.
- **forwardSubstitution()** and **backwardSubstitution()**: Functions for solving the linear systems once **L** and **U** matrices are obtained. These are executed serially after the parallel decomposition phase.

BSP Strategy for LU Decomposition

- For every decomposition step **k**, the matrix is updated in parallel across threads. Each thread computes a subset of the **L** and **U** matrices and updates the corresponding elements of **A**.
- The implementation follows the BSP model by calculating each **k** step in a manner that maintains data dependency. This approach ensures that the decomposition process logically mirrors its sequential counterpart, albeit executed in parallel for efficiency.

Memory Management and Performance Measurement

- Dynamic memory allocation is used for matrix storage, with careful attention to deallocating memory post-computation to avoid leaks.
- Performance measurement is integral to our implementation, with timing captured at the start and end of the decomposition process to evaluate the effectiveness of parallelization.

This sophisticated parallelization of the LU Decomposition, particularly with the incorporation of partial pivoting, demonstrates a significant advancement in numerical computing. By meticulously adhering to the BSP model and leveraging POSIX threads,

we've achieved a balance between speed and accuracy, making our implementation suitable for a wide array of scientific and engineering applications.

Incorporating Pivoting into Parallel LU Decomposition

Our approach to enhancing the robustness of LU decomposition involves integrating partial pivoting into the parallel execution framework. Partial pivoting is essential for handling matrices that may lead to numerical instability during decomposition, ensuring our solution's accuracy and reliability.

Pivoting Mechanism

The pivoting process iterates over each decomposition step, identifying the maximum element in the current column (from the diagonal element downwards) to use as the pivot. This strategy mitigates the risk of dividing by numbers close to zero, which could amplify rounding errors and lead to inaccurate results. The **performPartialPivoting** function encapsulates this logic, swapping rows in the matrix **A**, the lower triangular matrix **L**, and the permutation vector **p** to reflect the chosen pivot row. This step is crucial in maintaining the numerical stability of the decomposition.

Parallel Implementation with Pivoting

- Each thread, participating in the decomposition process, first waits for the pivot selection and row swaps to complete. This synchronization ensures the decomposition operates on the correctly pivoted matrix at each step.
- Post pivoting, the threads proceed to update their assigned sections of the matrix **A**, computing the elements of **L** and **U** in parallel. This phase leverages the benefits of parallel processing, significantly speeding up the computation without sacrificing the decomposition's accuracy due to numerical instability.

Thread Synchronization and Data Integrity

- To ensure data integrity and correctness, synchronization points are strategically placed throughout the computation. Before updating the matrix **A** in parallel, threads synchronize to confirm the completion of row swapping, ensuring all threads operate on the same, updated matrix state.
- The **parallel_portion** function, executed by each thread, is responsible for computing and updating the matrix elements based on the current decomposition step, reflecting the parallel algorithm's core logic.

Memory Management and Computational Efficiency

- Dynamic memory allocation is employed for matrices and vectors, including the permutation matrix that embodies the row swaps performed during pivoting. Post-computation, a systematic deallocation of memory prevents leaks, showcasing our commitment to computational efficiency and resource management.

Verification and Performance Measurement

- To verify the correctness of the LU decomposition with pivoting, we multiply the permutation matrix by the original matrix and compare the result against the product of the computed **L** and **U** matrices. This step confirms the decomposition's accuracy.
- Performance is measured by recording the computation time before and after the decomposition process. This metric highlights the efficiency gains achieved through parallel processing and the impact of incorporating pivoting on the algorithm's overall performance.

Results and Observations

The mature parallel algorithm, inspired by Bisseling's work and tailored through iterative refinement, demonstrated significant performance improvements. The implementation showcased not only the raw speedup achieved through parallelization but also highlighted the importance of a well-thought-out data distribution strategy and efficient communication mechanisms.

Lessons Learned

- **Understanding Data Flow**: A deep understanding of the algorithm's data flow is essential for effective parallelization. This insight guides the distribution of data and tasks across processors.
- **Iterative Refinement**: The transition from an initial, flawed parallel attempt to a successful implementation underscores the importance of iterative refinement, guided by theoretical insights and practical experimentation.
- **The Power of Models**: Applying structured models like BSP can dramatically simplify the design and analysis of parallel algorithms, providing clear pathways to overcoming challenges such as data dependencies and communication overhead.

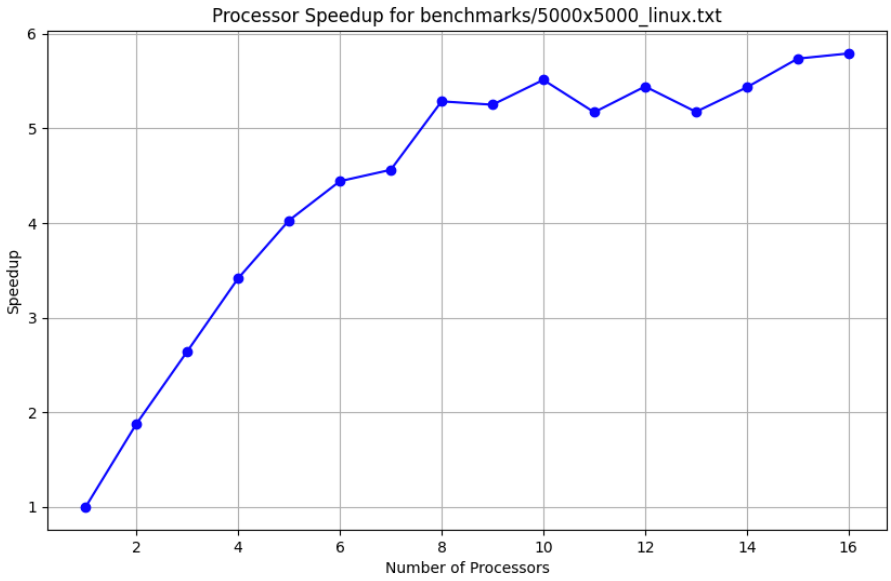
In conclusion, the journey from an initial failed attempt to a successful parallel implementation of LU decomposition serves as a compelling case study in the challenges and strategies of parallel algorithm design. This experience emphasizes the critical role of data distribution, communication optimization, and synchronization in achieving efficient parallel computations.

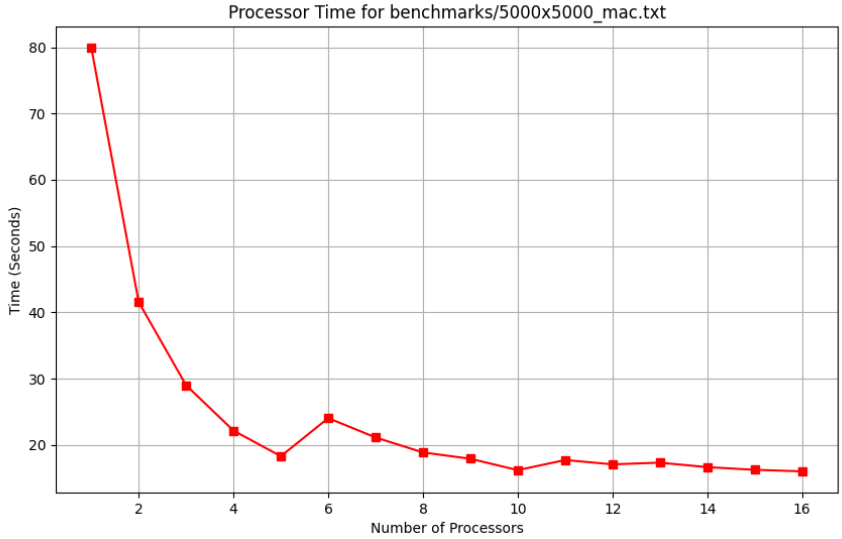
Performance Analysis and Results

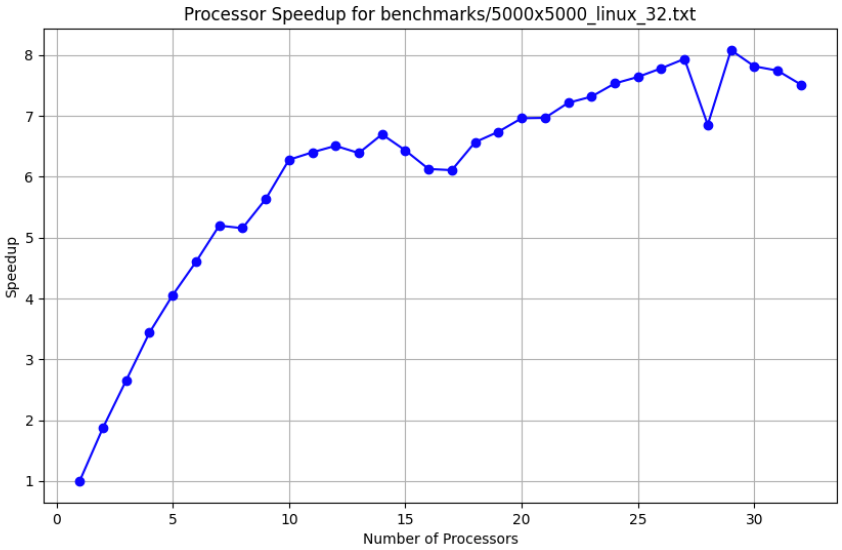
The performance and results of our parallel LU Decomposition program were meticulously evaluated across various matrix sizes and processor counts, primarily focusing on matrices ranging from 3x3 to 10000x10000. This analysis was conducted on two distinct systems: a MacBook Pro with an M3 Pro chip which has 11 cores, and a Linux system equipped with an Intel Xeon E312xx (Sandy Bridge, IBRS update) processor featuring 32 CPUs. Our primary objective was to gauge the efficacy of parallelizing the LU Decomposition process, hence we timed this specific part of the computation to ascertain the speedup achieved.

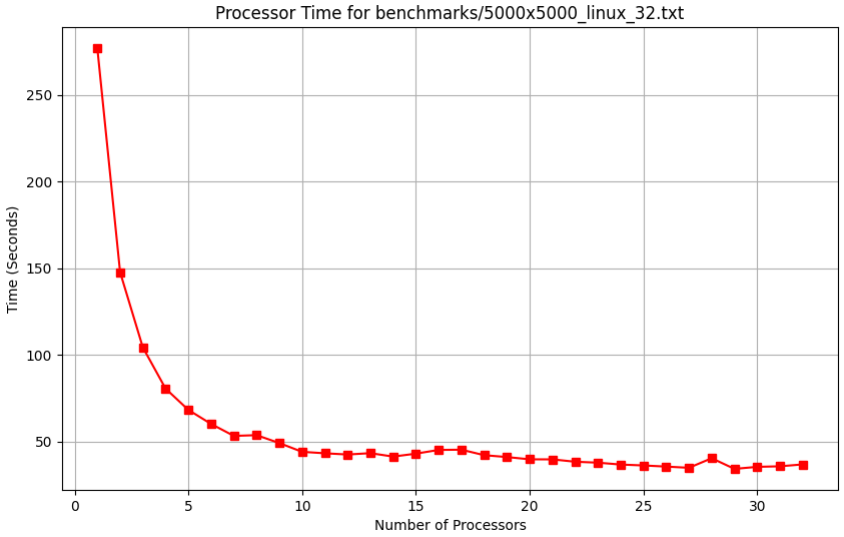
Speedup Achieved on Linux Systems

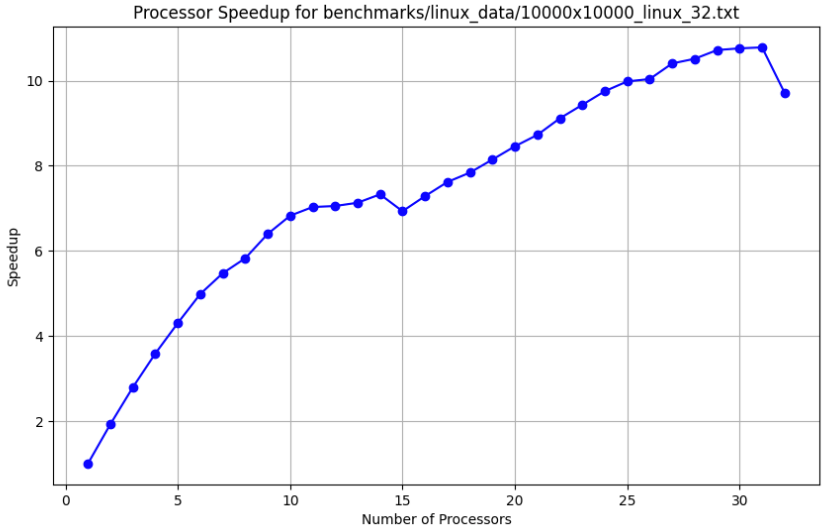
On the Linux system, leveraging 16 processors for a 5000x5000 matrix, we observed a significant decrease in computation time as the number of threads increased, peaking at a speedup of 5.792209 with 16 threads. Interestingly, similar experiments with 32 processors for the same matrix size also demonstrated substantial speedups, reaching up to 7.934753 with 27 threads. This trend was consistent for a larger 10000x10000 matrix, where the speedup continued to improve as more threads were employed, peaking at a remarkable 10.781217 with 31 threads.

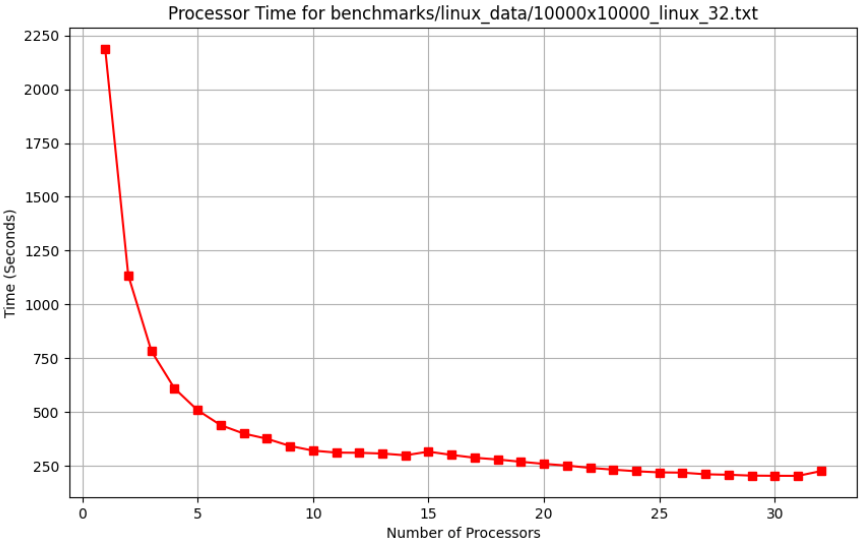



Processor Time for benchmarks/5000x5000_mac.txt
Time (Seconds)
Number of Processors
80
70
60
50
40
30
20
2
4
6
8
10
12
14
16


Processor Speedup for benchmarks/5000x5000_linux_32.txt
Speedup
Number of Processors
8
7
6
5
4
3
2
1
0
5
10
15
20
25
30

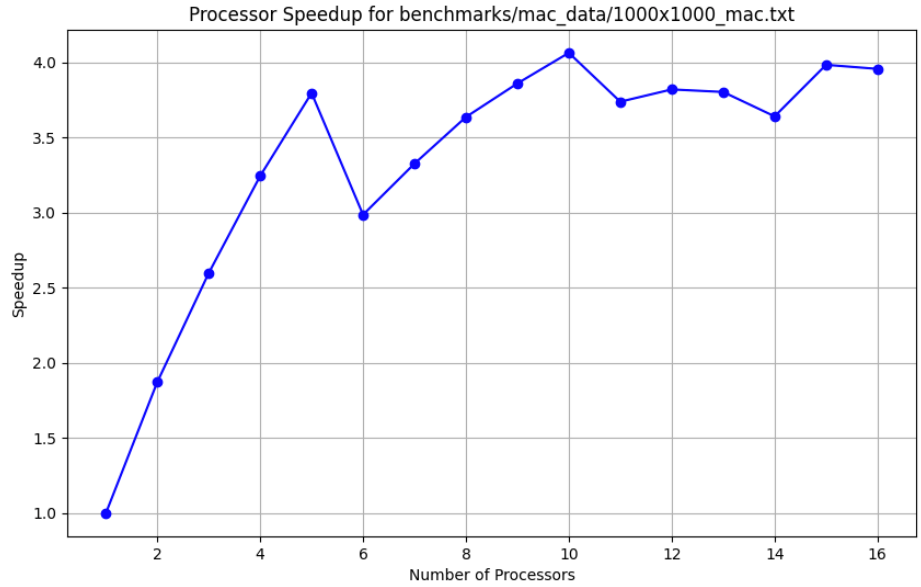

Processor Time for benchmarks/5000x5000_linux_32.txt
Time (Seconds)
Number of Processors

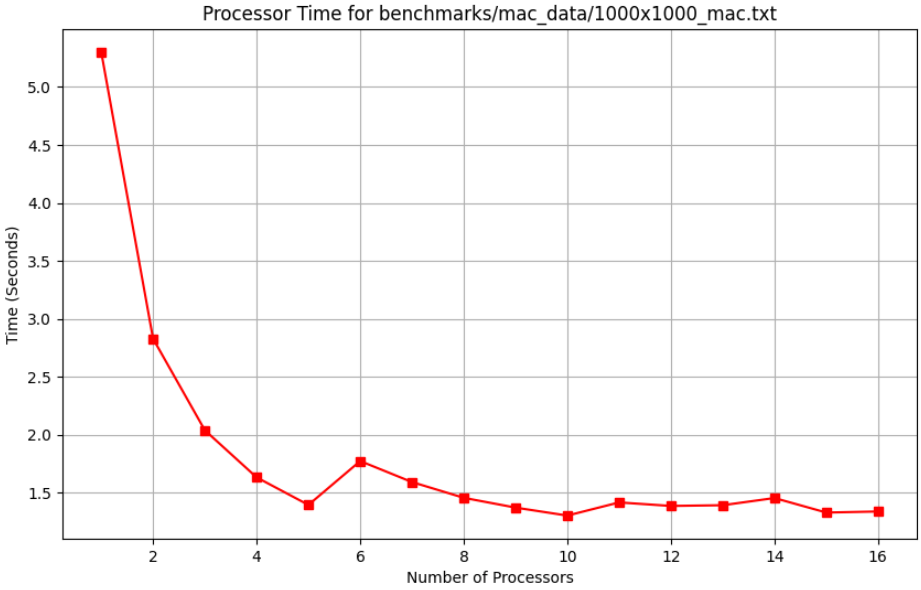

Processor Speedup for benchmarks/linux_data/10000x10000_linux_32.txt
Speedup
Number of Processors

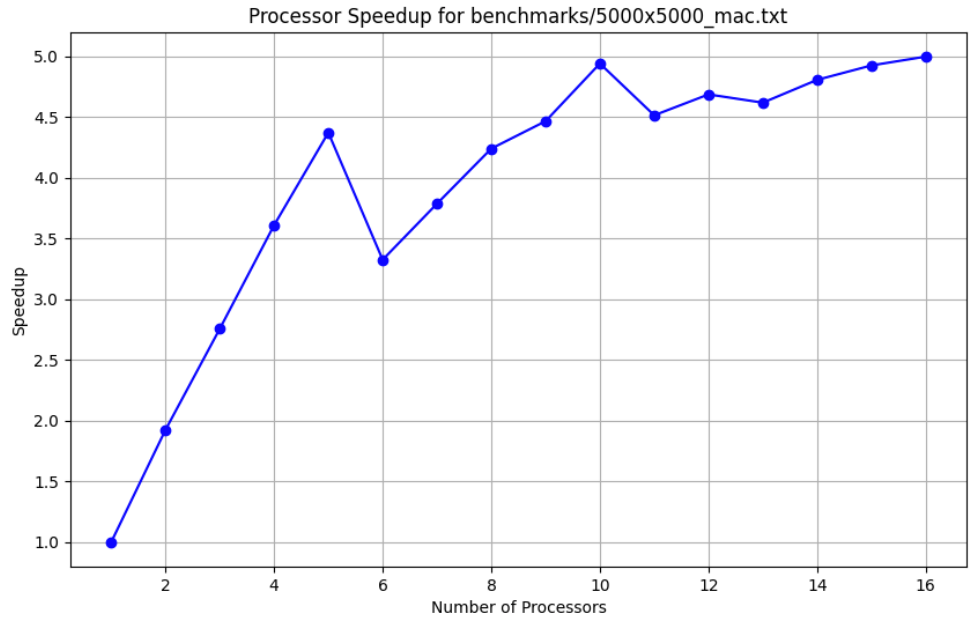

Processor Time for benchmarks/linux_data/10000x10000_linux_32.txt
Time (Seconds)
Number of Processors

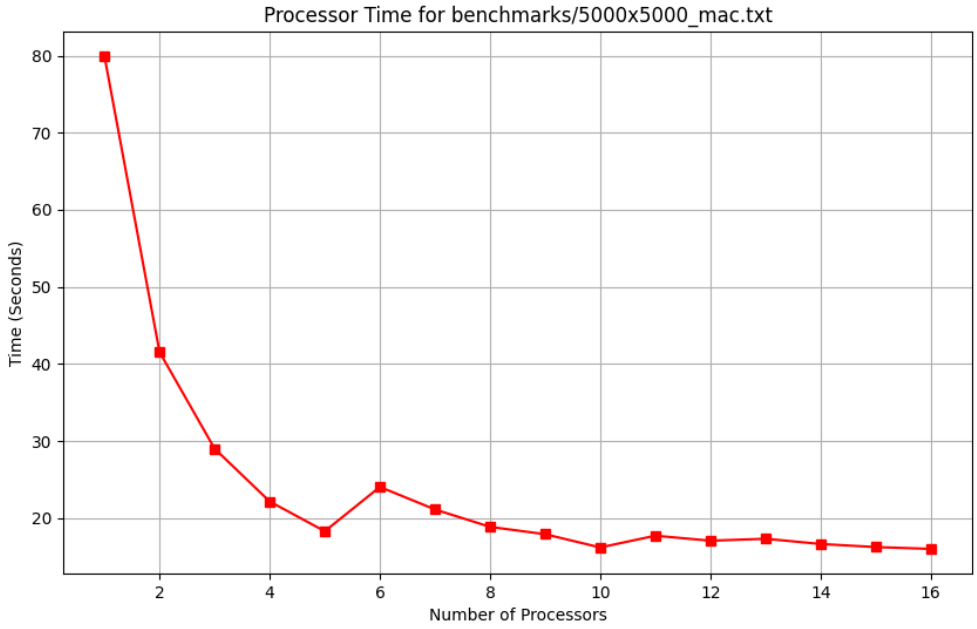
Speedup Achieved on MacBook Pro

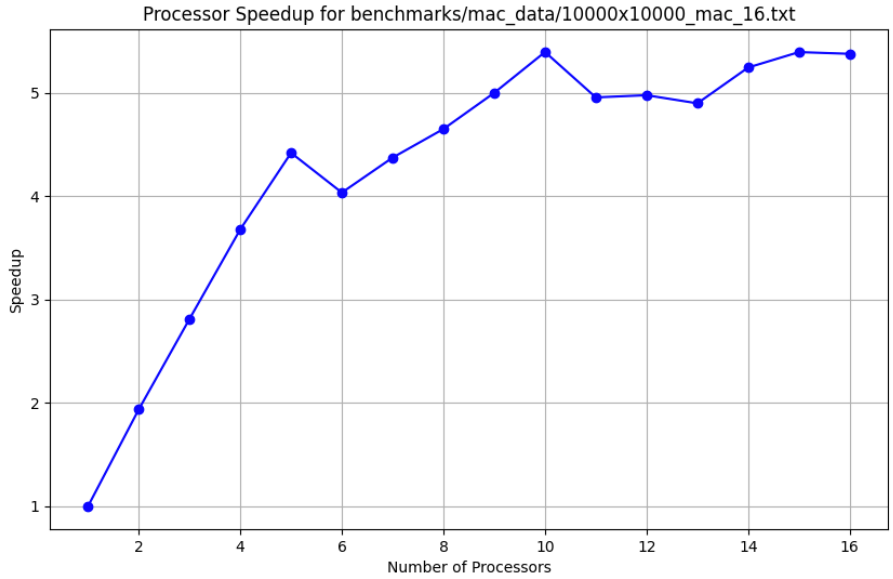
On the MacBook Pro system, the saturation of cores became evident due to the hardware limitation of having only 11 cores. Despite this, we observed a notable speedup in performance as the number of threads increased for a 5000x5000 matrix. The maximum speedup recorded was 4.940405 with 10 threads, which is an impressive achievement given the hardware constraints. As we extended our testing to larger matrices, the trend of achieving higher speedups with an increasing number of threads continued, illustrating the efficiency of our parallel LU Decomposition algorithm.

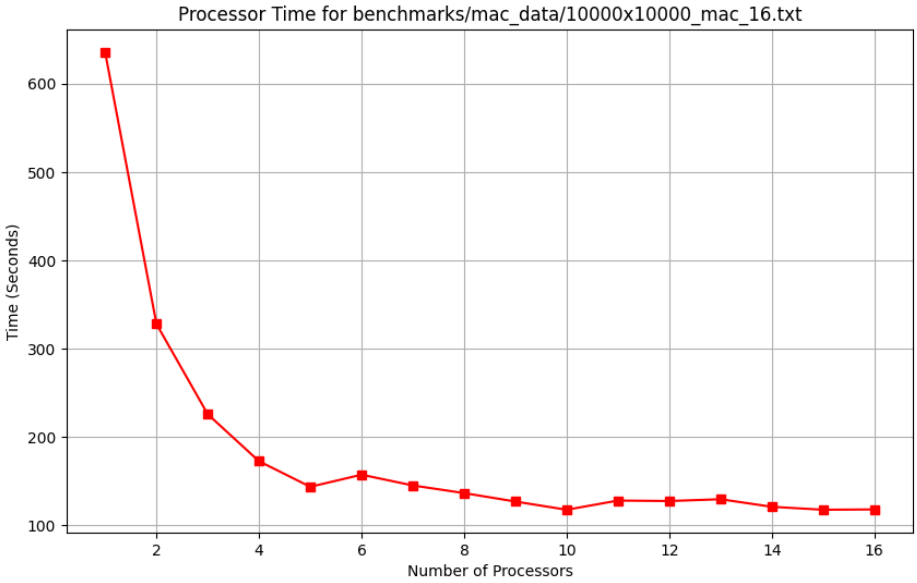





Processor Speedup for benchmarks/5000x5000_mac.txt
Speedup
Number of Processors
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
2
4
6
8
10
12
14
16


Processor Time for benchmarks/5000x5000_mac.txt
Time (Seconds)
Number of Processors
20
30
40
50
60
70
80
2
4
6
8
10
12
14
16





Discussion

These results underscore the effectiveness of our parallel approach to LU Decomposition, particularly in leveraging multicore processors to achieve substantial reductions in computation time. The data from the Linux system, with its more robust CPU capabilities, displayed significant speedups that underscore the scalability of our algorithm. Meanwhile, the MacBook Pro's performance, albeit constrained by its core count, still showed considerable improvements, highlighting the versatility and adaptability of our implementation across different hardware configurations.
The observed speedup across different systems and configurations illustrates the potential of parallel computing in tackling large-scale computational problems

efficiently. Our implementation effectively distributes the workload across available processors, optimizing resource utilization and significantly reducing overall computation time.

Summary and Conclusion

In this project we implemented a linear system of equations/matrix solver, with the big focus on parallelizing the LU Decomposition algorithm to improve its efficiency on multicore processors. Through detailed planning, algorithm development, and extensive testing across various systems, we have demonstrated the significant potential of parallel computing in numerical linear algebra.

Key Findings:

- The parallel LU Decomposition algorithm, both with and without partial pivoting, showcases notable performance enhancements across different matrix sizes, particularly evident in the execution times for matrices ranging from 3x3 to 10000x10000.
- Implementation on a Linux system with an Intel Xeon processor and on a MacBook Pro with an M3 Pro chip highlighted the algorithm's adaptability and efficiency. The Linux system, with its higher CPU capabilities, achieved remarkable speedups, demonstrating the scalability of our approach.
- The algorithm's design incorporates strategies from Rob Bisseling's work on parallel scientific computation, effectively managing data dependencies and workload distribution across threads. This structured approach ensures the maintenance of algorithmic integrity and correctness while enhancing computational speed.
- Testing on the MacBook Pro system, despite core saturation, revealed significant speedups, underscoring the parallel algorithm's capability to leverage available computational resources effectively.

Conclusions:

The journey from conceptualizing a parallel approach to the LU Decomposition to its successful implementation and testing has underscored the transformative power of parallel computing in dealing with computationally intensive tasks. Our findings confirm that with careful algorithm design, effective workload distribution, and consideration of hardware capabilities, substantial improvements in computational efficiency can be achieved.

The parallel LU Decomposition algorithm stands as a testament to the synergy between mathematical theory and computer science, opening new avenues for research and application in numerical analysis and beyond.

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] 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.

[3] 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].

[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] 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].

[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] 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.

[8] 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.

[9] 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.

[10] 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.

Appendix – 1 (Step by step solution of example 1 using LU Decomposition)

$$A = \begin{pmatrix} 2 & 3 & -1 \\ 4 & 1 & 2 \\ -2 & 2 & 3 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix} \begin{pmatrix} 2 & 3 & -1 \\ 4 & 1 & 2 \\ -2 & 2 & 3 \end{pmatrix} \quad \begin{matrix} II-(2)I \\ III-(-1)I \end{matrix}$$

$$= \begin{pmatrix} 1 & & \\ 2 & 1 & \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 3 & -1 \\ 0 & -5 & 4 \\ 0 & 5 & 2 \end{pmatrix} \quad III-(-1)II$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 3 & -1 \\ 0 & -5 & 4 \\ 0 & 0 & 6 \end{pmatrix}$$

L matrix U matrix

Solve $Ly = b$ for y

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}, \quad b = \begin{pmatrix} 5 \\ 6 \\ 8 \end{pmatrix}$$

$$\therefore \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ 8 \end{pmatrix}$$

① $y_1 = 5$

② $2y_1 + y_2 = 6$ $\quad \therefore y_2 = 6 - 2(5) = -4$

③ $-y_1 - y_2 + y_3 = 8$ $\quad \therefore y_3 = 8 + (5) + (-4) = 9$

$$\therefore y = \begin{pmatrix} 5 \\ -4 \\ 9 \end{pmatrix}$$

Solve $Ux = y$ for x

$$\begin{pmatrix} 2 & 3 & -1 \\ 0 & -5 & 4 \\ 0 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ -4 \\ 9 \end{pmatrix}$$

① $2x_1 + 3x_2 - x_3 = 5$ $\quad \therefore 2x_1 = 5 + \frac{3}{2} - 3(2) = \frac{1}{4}$ *

② $-5x_2 + 4x_3 = -4$ $\quad \therefore 5x_2 = 4 - 4\left(\frac{3}{2}\right), \; x_2 = 2$

③ $6x_3 = 9$ $\quad \therefore x_3 = \frac{3}{2}$

Now we know that $x_1 = x$, $x_2 = y$, $x_3 = z$

$x_1 = \frac{1}{4}$

$x_2 = 2$

$x_3 = \frac{3}{2}$

Let's substitute back in the equations.

① $2x_1 + 3x_2 - x_3 = 5$

$2\left(\frac{1}{4}\right) + 3(2) - \left(\frac{3}{2}\right) = 5$ #

② $4x_1 + x_2 + 2x_3 = 6$

$4\left(\frac{1}{4}\right) + (2) + 2\left(\frac{3}{2}\right) = 6$ #

③ $-2x_1 + 2x_2 + 3x_3 = 8$

$-2\left(\frac{1}{4}\right) + 2(2) + 3\left(\frac{3}{2}\right) = 8$ #

Appendix –2 (Basic Example code to solve a 3x3 matrix using LU Decomposition)

```c
// Author: Mohamed Ghonim
// Created: 02/18/2024
// Last Modified: 02/18/2024
// Functionality: Perform LU decomposition and solve a system of linear equations
using forward and backward substitution.
// Version: 0.1


#include <stdio.h>
#define N 3 // Size of the matrix (3x3)

// Function to perform LU decomposition
void luDecomposition(double A[N][N], double L[N][N], double U[N][N]) {
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            if (j < i)
                L[j][i] = 0;
            else {
                L[j][i] = A[j][i];
                for (k = 0; k < i; k++) {
                    L[j][i] = L[j][i] - L[j][k] * U[k][i];
                }
            }
        }
        for (j = 0; j < N; j++) {
            if (j < i)
                U[i][j] = 0;
            else if (j == i)
                U[i][j] = 1;
            else {
                U[i][j] = A[i][j] / L[i][i];
                for (k = 0; k < i; k++) {
                    U[i][j] = U[i][j] - ((L[i][k] * U[k][j]) / L[i][i]);
                }
            }
        }
    }
}

// Function to solve the equation Ly = b
void forwardSubstitution(double L[N][N], double b[N], double y[N]) {
    for (int i = 0; i < N; i++) {
        y[i] = b[i];
        for (int j = 0; j < i; j++) {
```

```
            y[i] -= L[i][j] * y[j];
        }
        y[i] = y[i] / L[i][i];
    }
}

// Function to solve the equation Ux = y
void backwardSubstitution(double U[N][N], double y[N], double x[N]) {
    for (int i = N - 1; i >= 0; i--) {
        x[i] = y[i];
        for (int j = i + 1; j < N; j++) {
            x[i] -= U[i][j] * x[j];
        }
        // No division by U[i][i] since U[i][i] = 1
    }
}

int main() {
    double A[N][N] = {{2, 3, -1}, {4, 1, 2}, {-2, 2, 3}};
    double b[N] = {5, 6, 8};
    double L[N][N] = {0};
    double U[N][N] = {0};
    double y[N] = {0};
    double x[N] = {0};

    luDecomposition(A, L, U);
    forwardSubstitution(L, b, y);
    backwardSubstitution(U, y, x);

    printf("Solution: \n");
    for (int i = 0; i < N; i++) {
        printf("x[%d] = %f\n", i, x[i]);
    }

    return 0;
}
```