Linear Equations Solver

ECE 688/588 Final Project

Group #1: Mohamed Ghonim, Phanindra Vemireddy,
Alexander Maso, Ahliah Nordstrom

TABLE OF CONTENTS

1.

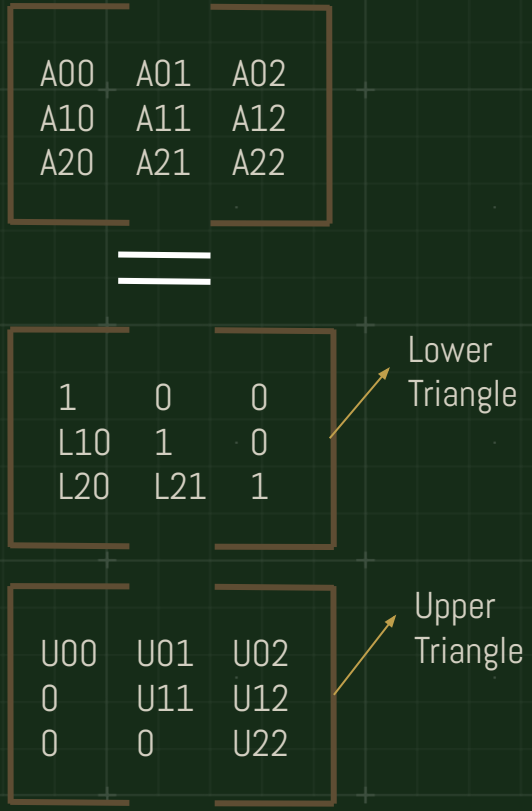
OBJECTIVE and BACKGROUND

PROJECT OBJECTIVE

Our objective:

- Develop and implement a parallel LU Decomposition algorithm using POSIX threads to enhance computational efficiency for solving linear system equations
- Investigate techniques such as partial pivoting to ensure numerical stability in the parallelized algorithm for reliable computational results
- Aim to leverage multicore processors effectively to achieve substantial reductions in computation time while maintaining numerical accuracy in the solution

LU DECOMPOSITION BACKGROUND



Linear System Equations

A set of equations that can be solved simultaneously to find the values of unknown variables (Gaussian Elimination/LU Decomposition)

LU Decomposition

A method used to factorize a matrix into lower and upper triangular matrices to simplify the solution of linear systems (Takes most time of the solver!)

Parallelization Importance

With the increasing size of systems, parallel computing becomes essential to accelerate the solution process by distributing the workload across multiple processors

II.

PARALLEL ALGORITHM

ALGORITHM DEVELOPMENT

MOTIVATION

Improve computational efficiency by leveraging multicore processors for faster solution of linear system equations

CHALLENGES

Synchronizing threads, load balancing, and ensuring numerical stability through techniques like partial pivoting

SOLUTIONS

Implementing efficient thread synchronization, load distribution strategies, and incorporating robust numerical stability measures in the algorithm

III.

PARALLEL IMPLEMENTATION

Code building blocks PARALLEL_SOLVER_WITH_PIVOTING.c FUNCTIONS

forwardSubstitution → - To solve the equation Ly = b for y

```
void forwardSubstitution(double** L, double* b, double* y, int n) {
    for (int i = 0; i < n; i++) {
        y[i] = b[i];
        for (int k = 0; k < i; k++)
            y[i] -= L[i][k] * y[k];
        y[i] = y[i] / L[i][i];
    }
```

backwardSubstitution → - To solve the equation Ux = y for x

```
void backwardSubstitution(double** U, double* y, double* x, int 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];
        }
        x[i] /= U[i][i];
    }
```

matrix_multiply → - To multiply two matrices

```
void matrix_multiply(double** matrix1, double** matrix2, double** product, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            product[i][j] = 0;
            for (int k = 0; k < n; ++k) {
                product[i][j] += matrix1[i][k] * matrix2[k][j];
            }
```

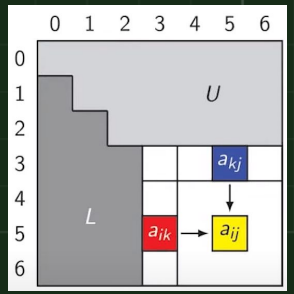
main →
- Read matrix from file
- Call defined functions for execution
- Record end time

Code building blocks PARALLEL_SOLVER_WITH_PIVOTING.c FUNCTIONS

parallel_portion → - To be executed by each thread to assign values to matrix A in parallel

```c
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;

    //printf("Thread %d: Computing step k: %d\n", id, k);
    int interation_per_thread = n - 1 - k;
    int start = (k + 1) + id * interation_per_thread / numThreads;
    int end = (k + 1) + (id + 1) * interation_per_thread / numThreads < n ? (k + 1) + (id + 1) *
 interation_per_thread / numThreads : n;
    for (int i = start; i < end; i++) {
        for (int j = k + 1; j < n; j++) {
            a[i][j] -= l[i][k] * u[k][j];
           // printf("Thread %d: A[%d][%d] = %f\n", id, i, j, a[i][j]);
        }
    }
    pthread_exit(NULL);
}
```



readMatrixFromFile → - To read a matrix from file

```c
void readMatrixFromFile(const char* filename) {
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        exit(EXIT_FAILURE); }
    fscanf(file, "%d", &n); // Read the size of the matrix from the first line
    a = (double**)malloc(n * sizeof(double*));
    a_duplicate = (double**)malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
        a[i] = (double*)malloc(n * sizeof(double));
        a_duplicate[i] = (double*)malloc(n * sizeof(double));
        for (int j = 0; j < n; j++) {
            fscanf(file, "%lf", &a[i][j]);
            a_duplicate[i][j] = a[i][j]; // Copy the value to a_duplicate
        }
    }
    // Assuming the vector b is in the last line after the matrix
    b = (double*)malloc(n * sizeof(double));
    for (int i = 0; i < n; i++) {
        fscanf(file, "%lf", &b[i]);
    }
    fclose(file);
}
```

Code building blocks PARALLEL_SOLVER_WITH_PIVOTING.c FUNCTIONS

performPartialPivoting → - To perform partial pivoting

```c
void performPartialPivoting(double** a, double** l, int* p, int k) {
    double max_value = 0.0;         // The maximum value in the column
    int max_index = k;              // The index of the maximum value in the column
    for (int i = k; i < n; i++) {   // Find the maximum value in the column
        double absolute_value = fabs(a[i][k]); // Get the absolute value of the element
        if (absolute_value > max_value) {
            max_value = absolute_value;
            max_index = i;
        }
    }

    if (max_value == 0.0) {
        fprintf(stderr, "This is a singular matrix, LU Decomposition is not possible\n");
        exit(EXIT_FAILURE);
    }

    // Swap elements in p
    int temp_p = p[k];
    p[k] = p[max_index];
    p[max_index] = temp_p;

    // Swap rows in a
    double* temp_a = a[k];
    a[k] = a[max_index];
    a[max_index] = temp_a;

    // Swap rows in l for the elements before diagonal (k)
    for (int i = 0; i < k; i++) {
        double temp_l = l[k][i];
        l[k][i] = l[max_index][i];
        l[max_index][i] = temp_l;
    }
}
```

ludecomp_verify → - To calculate the residual norm to verify the correctness of LU decomposition

```c
double ludecomp_verify(double** P, double** A, double** L, double** U, double** residual, int n) {
    // Compute the residual matrix: (PA - LU)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            residual[i][j] = 0;
            for (int k = 0; k < n; k++) {
                residual[i][j] += P[i][k] * A[k][j];          // Adding product of P and A matrices
                residual[i][j] -= L[i][k] * U[k][j];          // Subtracting product of L and U matrices
            }
        }
    }

    // Calculate the Frobenius norm of the residual matrix
    double norm = 0;
    for (int i = 0; i < n; i++) {
        double column_norm = 0;
        for (int j = 0; j < n; j++) {
            column_norm += residual[j][i] * residual[j][i]; // Summing squares of each element in a column
        }
        norm += sqrt(column_norm); // Adding the square root of the summed squares of each column
    }
    return norm;                   // Return the Frobenius norm as the measure of accuracy
}
```

Print → - To print a matrix

freeUpMemory → - To deallocate memory

main → - To perform partial pivoting

UTILITY SCRIPTS

matrix_generator.py

Generates random matrix of given dimension and saves to a file

plot_benchmarks.py

Automate graphing speedup data from benchmark tests

makefile

To run the program

run_benchmarks_linux.pl

Runs program across a range of threads then calculates execution time and speedup for each on linux

run_benchmarks_mac.pl

Runs program across a range of threads then calculates execution time and speedup for each on Mac

DATA DEPENDENCIES/ ALGORITHM CORRECTNESS

Meticulous Management

Mapping out dependencies to guide parallel execution and employing barrier synchronization allowed each computational step to occur in the correct sequence, despite concurrent operations

Employing these strict strategies with extensive testing and validation against the original sequential algorithm ensuring algorithm accuracy and reliability

Strategies

Maintain Algorithm Integrity

Dynamic Scheduling

Dynamic Synchronization

Barrier Synchronization

Verification Correctness

IV.

TESTING and REFINING

DEBUGGING/ VERIFYING SOLUTIONS

- For smaller matrices [3x3, 4x4, and 5x5] we compared the calculated solution to our hand calculations.
- For bigger matrices, we compared that LU = A the original matrix decomposed.
- For even bigger matrices, where visual comparison is not practical, we implemented a Frobenius norm of the residual matrix verification. [0 and ~0 means correct!]

We implement debug statements to see which thread is working on which matrix element and in which k step.

```
> cc -lpthread src/parallel_solver_debug_mode.c -o bin/mac/parallel_solver_debug
> ./bin/mac/parallel_solver_debug matrices/py_generated/5000x5000.txt 11
Starting LU Decomposition with 11 threads for 5000x5000 matrix.
Completed LU Decomposition with 11 threads for 5000x5000 matrix.

 Frobenius norm of the residual matrix: 0.000002

Time taken: 17.958659000 seconds
```

```
printf("Starting LU Decomposition with %d threads for %dx%d matrix.\n", numThreads, n, n);
```

```
printf("Thread %d: Computing step k: %d\n", id, k);
```

```
printf("Thread %d: A[%d][%d] = %f\n", id, i, j, a[i][j]);
```

```
printf("Completed LU Decomposition with %d threads for %dx%d matrix.\n", numThreads, n, n);
```

```
printf("\n\nPA matrix:\n");
print(PA, n);
printf("LU matrix:\n");
print(LU, n);
```

FRAMEWORK

LU Decomposition was adapted for parallel execution by dividing the process into concurrent tasks across multiple cores, using Pthreads and BSP model for structured computation and synch. (Bulk synchronous parallel)

INTEGRATION

To enhance numerical stability, partial pivoting was integrated into the algorithm, involving a pre-decomposition phase for pivot selection and row swaps. However, this slowed down the program.

SYNCHRONIZATION

Our initial Parallel LU Decomposition suffered from a race condition due to poor parallel algorithm with multiple barriers. We fixed the synchronization issue by adapting a more systematic BSP approach.

PERFORMANCE

Extensive testing and performance analysis assessed the parallel LU Decomposition speed improvements and accuracy, confirming the algorithms effectiveness and maintaining high standards of numerical stability.

V.

PERFORMANCE ANALYSIS and RESULTS

EXPERIMENTAL RESULTS

Parallel LU Decomposition implementation showcased its performance and scalability across different hardware environments in order to provide a comprehensive assessment of the algorithms scalability and efficiency

- Linux systems with Intel Xeon E312xx processors (32 cores)
- MacBook Pro with M3 Pro chip (11 cores)

Evaluation

Leveraging all available processors on Linux resulted in significant speedups, proving scalability and efficiency. Macbook Pro showed speedups despite having only 11 cores compare to linux, thus highlighting adaptability.

Speedup and Efficiency

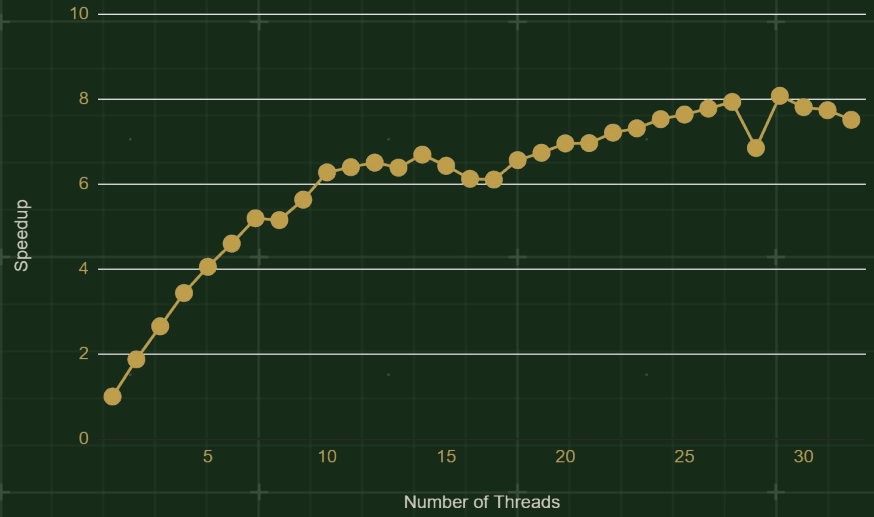
Various processor counts indicate performance improvements are directly linked to # of processors used, with diminishing thread count approaches systems core limit.

W/o Pivoting

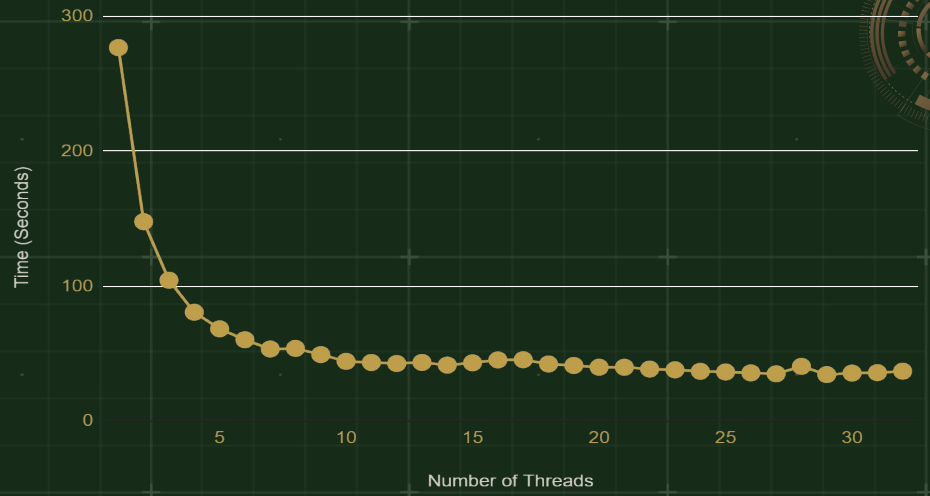
While partial pivoting introduced a slight performance overhead, its offset by considerable gains in numerical stability and accuracy. Can be worthwhile option for solving ill-conditioned matrices.

5000x5000 32-LINUX RESULTS

Processor Speedup



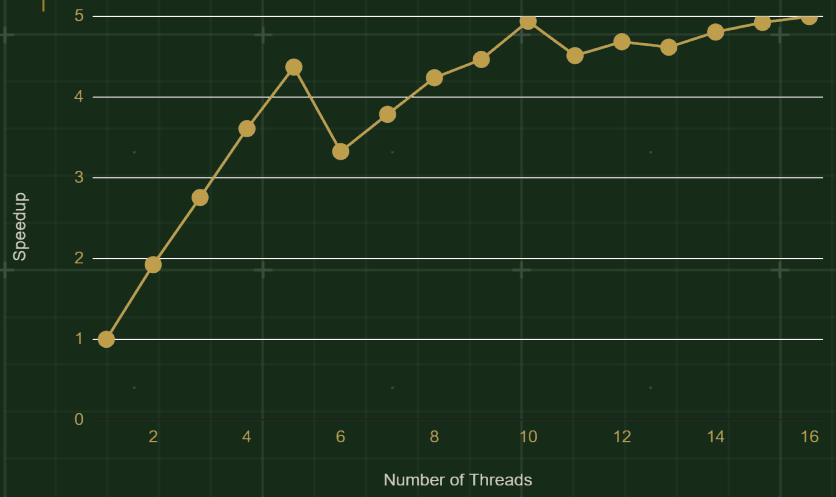
Processor Time



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.

5000x5000 Mac RESULTS

Processor Speedup

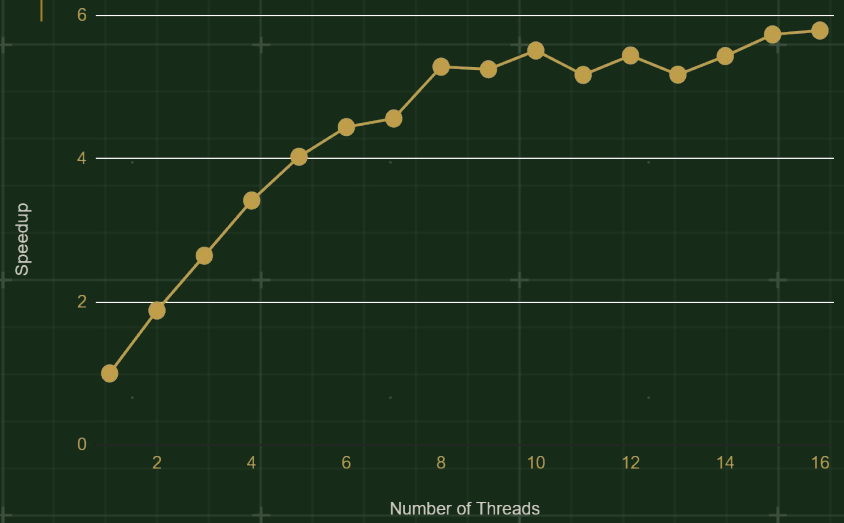


Processor Time

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.

5000x5000 16-LINUX RESULTS

Processor Speedup

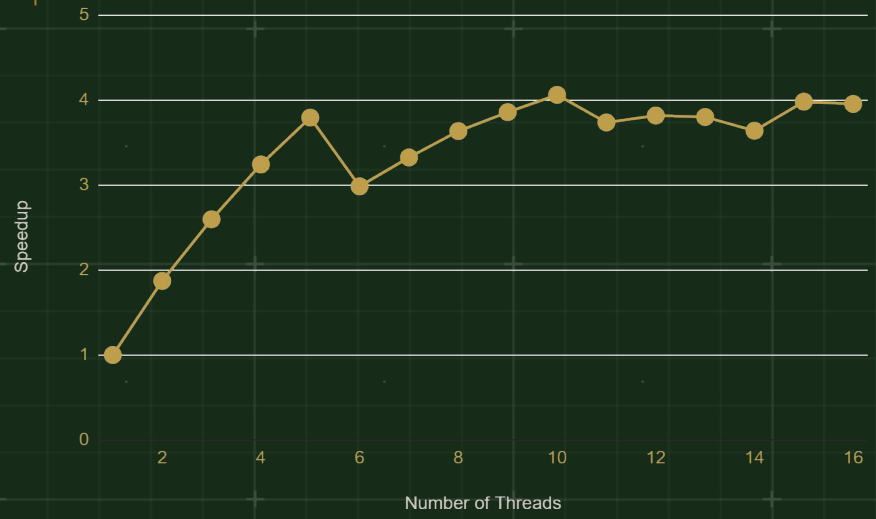


Processor Time

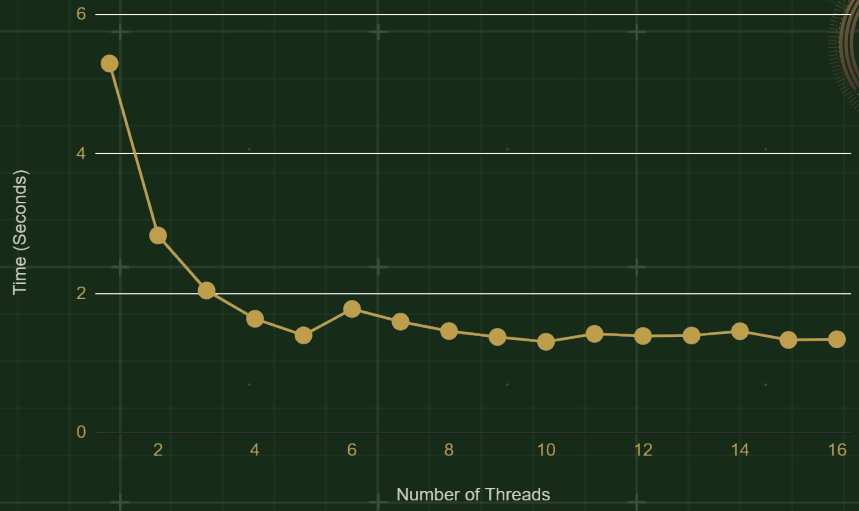
Speedup trends exhibited diminishing returns as the thread count approached the physical limit of the CPU cores, indicating an optimal range for thread utilization that maximizes performance without incurring excessive overhead.

10000x10000 Mac RESULTS

Processor Speedup



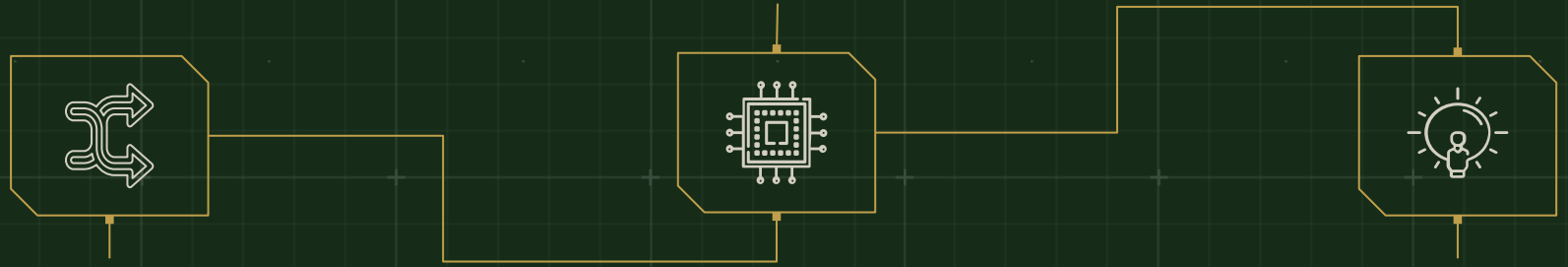
Processor Time



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.

The effectiveness of the algorithm depends on adapting to hw specifics, achieving significant speedups on high-CPU systems (Linux) and notable efficiency on limited-core systems (Mac).
It's worth noting that even with only 11 cores, the mac outperformed the intel 32 cores systems.

HW LIMITATIONS



COMPARATIVE ANALYSIS

Comparing the algorithms implementation with and without partial pivoting highlighted trade-offs between speed and numerical stability:

- Non-pivoting: Faster but can suffer from instability
- Partial Pivoting: Slightly slower but greater stability for matrices with elements close to 0

INSIGHTS

The Bulk Synchronous Parallel (BSP) model [by Leslie Valiant (1989)] and iterative refinement were key in structuring parallelization, ensuring task division and synchronization that maintained LU Decomposition integrity and addressed initial challenges like race conditions.

VI.

CONCLUSION and LESSONS LEARNED

CONCLUSION

OBJECTIVES AND FINDING RECAP:

- Aimed to develop a parallel LU Decomposition algorithm to utilize multicore processors for faster computation in solving large systems of linear equations
- Implementations with and without partial pivoting showed significant speedups, particularly on high-CPU systems, and improved numerical stability (w/ partial pivoting), accommodating a wider variety of matrices

CONTRIBUTIONS:

- Development of a parallel algorithm that efficiently distributes tasks across threads [using pthreads]
- A comprehensive performance evaluation demonstrating scalability [on intel/linux and apple silicon/mac]
- An analysis of speed vs scalability highlighting partial pivotings benefits
- Insights into overcoming challenges in parallelizing numerical methods

LESSONS LEARNED

Parallelizing LU Decomposition marks a substantial advancement in solving large linear systems more efficiently, showcasing the transformative potential of parallel computing in numerical methods and significantly accelerating scientific discovery and innovation without sacrificing accuracy

FUTURE DIRECTIONS:

- Exploring advanced parallelization strategies for enhanced performance and scalability
- Applying the algorithm to real-world problems to test its utility
- Integrating the algorithm into high-performance computing frameworks for broader access
- Extending its applicability to sparse matrices and special structures

This project lays a foundational step towards innovative research and application in parallel computing and numerical linear algebra.

ACKNOWLEDGMENTS

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 and substantive Quanta Magazine moderates comments to facilitate an informed, “New algorithm breaks speed limit for solving linear equations,” Quanta Magazine, https://www.quantamagazine.org/new-algorithm-breaks-speed-limitfor-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. doi:10.1016/0377- 0427(94)90302-6

[3] W. by: Q. Chunawala, “Fast algorithms for solving a system of linear equations,” Baeldung on Computer Science, 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. doi:10.1016/j.amc.2004.01.027

[5] E. E. Santos and M. Muralcetharan, “Analysis and Implementation of Parallel LU-Decomposition with Different Data layouts,” University of California, Riverside, Jun. 2000.