Parallelism in Four Numerical Computational Algorithms

Damian Franco
Dept. of Computer Science
University of New Mexico
dfranco24@unm.edu

***Abstract*—This paper discusses the importance of non-parallel and parallel numerical algorithms. Parallel processing techniques are necessary to improve computation time, efficiency, and scalability of numerical computations. The study focuses on implementing four different algorithms, including matrix-matrix multiplication, solving a linear system with Gaussian Elimination, LU Factorization with Doolittle's Algorithm, and approximating the largest eigenvalue with the Power Method, and comparing their performance and accuracy between parallel and non-parallel implementations. The importance of careful consideration of various factors, including load balancing, communication overhead, and hardware utilization, to ensure the correct implementation of parallel algorithms cannot be emphasized enough. The results demonstrate that the use of parallel algorithms can lead to significant performance improvements, but can also decrease performance and numerical accuracy. The goal for this paper is to highlight the need for careful design and optimization of parallel algorithms to maximize performance and scalability, while taking in account of the various aspects that can decrease or increase the performance of parallel numerical algorithms.**

***Index Terms*—parallel computation, numerical methods, algorithms, matrix-matrix multiplication, gaussian elimination, LU Factorization, Doolittle's algorithm, eigenvalues, eigenvectors, power method**

I. Introduction

Parallel and non-parallel numerical algorithms are essential in computer science, as many applications require parallel computation to efficiently achieve goals. Parallel processing can greatly improve computation time and efficiency by dividing problems into smaller sub-problems, increasing granularity, achieving a higher degree of concurrency, and becoming increasingly essential with the growth of data and complexity, distributed computing, and cloud computing, enabling high-performance computing and scalability in large-scale systems.

There are two very important aspects to numerical computation, accuracy and performance. In this paper, I will be focusing on the performance aspect of numerical computation while maintaining accuracy and stability of the algorithms implemented. Parallel processing techniques are used to improve the computation time of matrix, vector, and linear system operations. While parallel processing has been known to improve performance, incorrect implementation can lead to decreased performance time. The primary objective of this study is to correctly implement various algorithms ranging from simple matrix-matrix operations to advanced linear system operations, compare and contrast the accuracy and performance of parallel versus non-parallel implementations, and understand how scaling factors into performance. The goal of this paper is to demonstrate the effectiveness of parallel computing in numerical computational tasks and provide insights into the relationship between scaling and performance of implementations.

I will be discussing four algorithms in specific. The first is a simple matrix-matrix multiplication algorithm that takes in two matrices and multiply them to one another. The second algorithm that I will be investigating and implementing is the solving a linear system with Gaussian Elimination with the input of a matrix and solution vector. The third will be performing a lower-upper (LU) Factorization of a input matrix using Doolittle's Algorithm. The last algorithm that will be discussed in this paper is approximating the largest eigenvalue of a input matrix using the Power Method. I will discuss both theoretical and numerical approaches of each algorithm in the following sections as well as results from my implementations.

II. Methods

This section will discuss the methodology of the project. First, some insight on Message Passing Interface or MPI will be presented to help the reader develop a full understanding of what MPI is and how it was used to implement parallel algorithms. There will then be more theoretical explanations of each algorithm that was implemented, as well as presenting the serial (non-parallel) and parallel version of the algorithms that was used. Testing will be done at different scales of larger and smaller matrices and vectors and numerical results will be viewed but not presented while performance results will be the focus. Each algorithm was implemented in the programming language C/C++.

C/C++ is a low-level programming language that is used in numerical computing and scientific research. C is a procedural language that is known for its simplicity, efficiency, and low-level access to computer memory while C++ is an extension of C that adds object-oriented programming features such as classes, templates, and inheritance. Numerical algorithms require high computational efficiency, and C and C++ are optimized for this purpose. This project only uses the low-level features of C, but have the file extension of C++ for use of the MPI library functions.

A. Message Passing Interface (MPI)

Message Passing Interface (MPI) is a communication protocol for implementing parallel algorithms in distributed memory systems. MPI allows multiple processes or threads to communicate with each other and exchange data across different compute nodes in a parallel computing environment. It provides a standardized set of functions and commands for processes to communicate with each other and coordinate their activities. MPI enables parallel programs to run on a variety of different hardware platforms, including distributed memory systems such as clusters and supercomputers.

To use MPI efficiently, a programmer typically divides the problem into smaller sub-problems that can be executed in parallel on different processors or compute nodes. Each process is assigned a unique process ID (rank) and a subset of the data to work on. When I reference rank and size in the algorithms presented in the paper below, rank refers to the unique identifier of the calling process within a particular communicator, while size refers to the total number of processes that are part of the same communicator. The processes communicate with each other using MPI communication functions such as:

- *MPI_Send* - Send a message from a source process to a destination process in a parallel computing environment.
- *MPI_Recv* - Receive a message from a source process in a parallel computing environment.
- *MPI_Bcast* - Broadcast data from one process to all other processes in a parallel computing environment.
- *MPI_Reduce* - Reduce data from multiple processes to a single value on the root process in a parallel computing environment.
- *MPI_Allreduce* - Reduce data from multiple processes to a single value and distribute the result to all processes in a parallel computing environment.
- *MPI_Scatter* - Distribute data from a single process to multiple processes in a parallel computing environment.
- *MPI_Gather* - Gather data from multiple processes onto a single root process in a parallel computing environment.
- *MPI_Allgather* - Gather data from multiple processes onto all process in a parallel computing environment.
- *MPI_Barrier* - Synchronize all processes in a parallel computing environment. It forces each process to wait until all processes in the communicator have called MPI_Barrier before continuing execution.

There are many important communication functions that MPI offers, and many more than the examples listed above. These are just the more important functions that were used in the implementation of each numerical algorithm that was the focus in this paper.

To convert a non-parallel numerical algorithm to a parallel numerical algorithm using MPI, a programmer needs to first identify the parts of the algorithm that can be executed in parallel. These parts can then be distributed among the different processes using MPI collective communication functions and sending and receiving specific messages can also be used for point-to-point communication between specific processes. Once the parallelization scheme has been determined, the algorithm can was implemented in C or C++ using MPI libraries. It is important to ensure that the parallel algorithm maintains the accuracy and stability of the non-parallel algorithm while improving the performance through efficient use of parallel processing. Additionally, the algorithm should be tested and benchmarked (in this paper it will be a 10x10 matrix) to evaluate its scalability and performance under different problem sizes and numbers of processes.

When it comes to parallel computing, there are several different approaches that can be used to partition the data and distribute it across multiple processors. Three common approaches are chunk, block, and row. Each of these approaches will be investigated with in this paper with some of them being tied directly to algorithms. In the chunk approach, the data is partitioned into equal-sized chunks, and each chunk is assigned to a different processor. This approach is particularly useful when the computation involves independent sub-tasks that can be performed in parallel. The block approach is very similar to the chunk approach. The data is partitioned into equal-sized blocks, and each block is assigned to a different processor. This approach is often used in matrix operations, such as matrix multiplication or LU Factorization. The main difference between the block and the chunk approach is that the chunk approach partitions the data or problem into non-overlapping pieces of arbitrary size, which may or may not be of equal size while the block approach have equally sized blocks for subtasks. The row approach is where each processor is assigned a different row(s) of the data, and each processor performs the same computation on its assigned row. This approach is often used in iterative algorithms.

B. Matrix-Matrix Multiplication

Matrix-matrix multiplication is a fundamental operation in linear algebra and is used extensively in scientific computing and data analysis. It involves multiplying two matrices together to produce a third matrix. To perform matrix-matrix multiplication, the number of columns in the first matrix must match the number of rows in the second matrix. The resulting matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix. The elements of the resulting matrix are calculated by taking the dot product of each row of the first matrix with each column of the second matrix. This operation can be performed using nested loops in C. Matrix-matrix multiplication is a computationally intensive operation. In this non-parallel approach, shown in Algorithm 1, it is the most simple approach for matrix-matrix multiplication with little to no optimization.

The theoretical aspect of matrix-matrix multiplication involves understanding the properties of matrices and the rules of matrix algebra that govern the multiplication operation. Matrix multiplication is not commutative, meaning that the order in which matrices are multiplied can affect the result. The numerical stability of matrix-matrix multiplication can also be an important consideration in some applications. The presence of round off errors in floating-point arithmetic can

Algorithm 1 Pseudocode of matrix-matrix multiplication without parallel techniques. This is a Naive approach where A and B are nxn dense matrices that are being multiplied and C is the resulting matrix.

```
procedure MATMAT_SERIAL(A, B, C, n)
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C_{i,j} ← 0
            for k ← 0 to n − 1 do
                C_{i,j} ← C_{i,j} + A_{i,k} × B_{k,j}
            end for
        end for
    end for
end procedure
```

accumulate during matrix multiplication, leading to loss of accuracy and stability in the result.

The asymptotic run time of the simple matrix matrix multiplication algorithm you provided is $O(n^3)$, where n is the size of the matrices being multiplied. This is because for each element in the resulting matrix, the algorithm needs to perform n multiplications and n-1 additions. Since it is needed to do this for each element in the resulting matrix, the overall run time of the algorithm is proportional to n^3.

The non-parallel version of a simple matrix-matrix multiplication algorithm to a parallel version shown in Algorithm 2 divides the input matrices into smaller sub-matrices and distributes them across multiple processes based on the row approach of partitioning data. Each process would then perform the multiplication of its assigned sub-matrices and communicate the partial results to other processes to compute the final result. The run time complexity of the parallel matrix-matrix multiplication algorithm would be $O(n^3/p)$, where p is the number of processes used for parallelization.

C. Solve Linear System with Gaussian Elimination

Gaussian elimination is a widely used method for solving systems of linear equations, and it is based on the principle of row operations. The theoretical goal of Gaussian elimination is to transform the original system of equations into an equivalent system that is much easier to solve, typically a row echelon form or a reduced row echelon form which is called Forward Elimination.

The basic idea behind Gaussian elimination is to perform a series of row operations on the augmented matrix that represents the system of linear equations. The two main steps for Gaussian elimination are Forward Elimination and Backward Substitution. Forward elimination start with row operations that include adding a multiple of one row to another row, multiplying a row by a nonzero scalar, and swapping two rows. The objective of these row operations is to transform the matrix into a triangular form, where all the entries below the main diagonal are zero. Once the augmented matrix is in triangular form, the solution can be easily obtained through

Algorithm 2 Pseudocode of matrix-matrix multiplication with parallel techniques from MPI. This is a much similar to the serial version approach where A and B are nxn dense matrices that are being multiplied and C is the resulting matrix, but the current rank and size of the properties the programmer decides to run.

```
procedure MATMAT_PARALLEL(A, B, C, n, rank, size)
    rows_per_proc ← n/size
    leftover_rows ← n mod size
    C_local ← matrix of size rows_per_proc × n and 0
    start_row ← rank × rows_per_proc
    end_row ← start_row + rows_per_proc
    if rank = size − 1 then
        end_row ← end_row + leftover_rows
    end if
    for i ← start_row to end_row − 1 do
        for j ← 0 to n − 1 do
            for k ← 0 to n − 1 do
                C_local_{i−start_row,j} ←
                C_local_{i−start_row,j} + A_{i,k} × B_{k,j}
            end for
        end for
    end for
    MPI_Allreduce combines the local to global result
    for i ← start_row to end_row − 1 do
        for j ← 0 to n − 1 do
            C_{i,j} ← C_local_{i−start_row,j}
        end for
    end for
end procedure
```

a process Backward Substitution which involves starting with the last equation in the system and solving for the last variable.

Then, the solution can be substituted back into the previous equation to solve for the second-to-last variable, and so on, until all the variables have been solved for. This approach was used for the implementation of this algorithm and is shown in Algorithm 3.

To make a parallel version of the Gaussian elimination function using MPI, you need to split the computation among different processes. One possible approach shown in Algorithm 4 is to split the rows of the matrix among different processes and perform the elimination and substitution steps in parallel. In this code, each process receives its own start and end indices to determine which rows of the matrix to work on. The forward elimination step is performed in parallel using MPI_Bcast to broadcast the pivot row to all processes and MPI_Barrier to synchronize the processes after each elimination step. The backward substitution step is also performed in parallel, and the solutions are gathered from all processes using MPI_Allgather. Note that this code assumes that the matrix is evenly divisible by the number of processes. If this is not the case, you may need to handle the remainder rows separately.

The asymptotic run time of Gaussian elimination is $O(n^3)$,

Algorithm 3 Pseudocode of solving a linear system without parallel techniques from MPI. This version is a serial version approach where A is a nxn dense matrices, b is the right-hand vector of the system and that x is the resulting solution vector.

```
procedure GE_SERIAL(A, b, x, n)
    for k ← 0 to n − 1 do
        for i ← k + 1 to n do
            xmult ← A_{i,k}/A_{k,k}
            for j ← k to n do
                A_{i,j} ← A_{i,j} − xmult × A_{k,j}
            end for
            b_i ← b_i − xmult × b_k
        end for
    end for
    x_{n−1} ← b_{n−1}/A_{n−1,n−1}
    for i ← n − 2 down to 0 do
        s ← b_i
        for j ← i + 1 to n do
            s ← s − A_{i,j} × x_j
        end for
        x_i ← s/A_{i,i}
    end for
end procedure
```

where n is the dimension of the matrix. This is because the algorithm requires $O(n^2)$ operations to eliminate each element below the diagonal in each column, and there are n columns. Therefore, the total number of operations required is proportional to n^3.

For the parallel implementation, since the inputs of a matrix has size nxn and p processors are used, the time complexity of the algorithm can be expressed as $O(n^3/p + n^2 log(p))$, where the first term represents the computation time and the second term represents the communication time. The communication time is dominated by the cost of performing p-1 global broadcasts, which is proportional to log p. The computation time is reduced by a factor of p compared to the serial Gaussian elimination algorithm, as each processor works on a local portion of the matrix. This parallel algorithm in particular may suffer from load imbalance if the matrix is not evenly distributed among the processors or if some processors have to perform more computations than others. This can result in idle processors and longer execution time which is the reason why I attempted to use the chunk approach to parallel computing here.

D. LU Factorization with Doolittle's Algorithm

Doolittle's algorithm is a method for computing the LU Factorization of a square matrix. The LU Factorization is a factorization of a matrix into the product of a lower triangular matrix (L) and an upper triangular matrix (U). The theoretical aspect of how this algorithm works is based on the fact that any invertible square matrix can be factored into the product of a lower triangular matrix and an upper triangular matrix.

Algorithm 4 Pseudocode of solving a linear system with parallel techniques from MPI. This version is a parallel version approach where A is a nxn dense matrices, b is the right-hand vector of the system and that x is the resulting solution vector and where the current rank and size of the properties the programmer decides to run.

```
procedure GE_PARALLEL(A, b, x, n, rank, size)
    chunk_size ← n/size
    start ← rank · chunk_size
    end ← start + chunk_size
    if rank = size − 1 then
        end ← n
    end if
    for k ← 0 to n − 1 do
        pivot_row ← array of n zeros
        if rank = k/chunk_size then
            for j ← 0 to n − 1 do
                pivot_row_j ← A_{k,j}
            end for
        end if
        MPI_Bcast to all processes
        for i ← start to end − 1 do
            if i ≤ k then
                continue
            end if
            xmult ← A_{i,k}/pivot_row_k
            for j ← k to n − 1 do
                A_{i,j} ← A_{i,j} − xmult · pivot_row_j
            end for
            b_i ← b_i − xmult · b_k
        end for
        MPI_Barrier to halt processes
    end for
    for i ← end − 1 down to start do
        s ← b_i
        for j ← i + 1 to n − 1 do
            s ← s − A_{i,j} · x_j
        end for
        x_i ← s/A_{i,i}
    end for
    MPI_Allgather to gather solutions
end procedure
```

The Doolittle Algorithm iteratively solves for the entries of U and L row-by-row, using the entries of the previous rows to determine the new entries. The algorithm begins by setting the diagonal entries of L to 1, since these entries will always be equal to 1 in a lower triangular matrix. It then proceeds to compute the entries of U and L row-by-row, using the entries of the previous rows to determine the new entries. This process continues until all of the entries of U and L have been computed. Its computational time complexity is $O(n^3)$, where n is the size of the matrix, which makes it relatively efficient for small and medium-sized matrices.

Algorithm 5 Pseudocode of approximating the LU Factorization of matrix A without parallel techniques from MPI. This version is a parallel version approach where A is a nxn dense matrices and an empty lower L and upper U nxn empty matrices.

```
procedure DOOLITTLE_SERIAL(A, L, U, n)
    for k ← 0 to n − 1 do
        for j ← k to n − 1 do
            sum ← 0
            for p ← 0 to k − 1 do
                sum ← sum + L_{k,p} × U_{p,j}
            end for
            U_{k,j} ← A_{k,j} − sum
        end for
        for i ← k + 1 to n − 1 do
            sum ← 0
            for p ← 0 to k − 1 do
                sum ← sum + L_{i,p} × U_{p,k}
            end for
            L_{i,k} ← (L_{i,k} − sum)/U_{k,k}
        end for
    end for
end procedure
```

To make a non-parallel version of Doolittle's algorithm into a parallel version, parts of the algorithm that can be parallelized need to be split in sub-tasks. This approach specifically divides the matrix A into smaller blocks and distribute them among multiple processes. Each process performs the LU Factorization on its local block and communicates the necessary information to update the global L and U matrices. In Algorithm 6, you can see this process occurring and local blocks computing and communicating the updated local L and U matrices to the root process to update the global matrices. This parallel version of Doolittle's algorithm should be faster than the non-parallel version if the matrix is large enough and the number of processes is chosen appropriately. Overhead may exist due to communication between processes.

Doolittle's algorithm for LU Factorization has a computational complexity of O(n^3), where n is the size of the input matrix. This means that the algorithm's run time grows as a cube of the size of the input matrix. This complexity arises from the fact that the algorithm requires performing $\frac{2}{3}n^3$ floating-point operations to compute the LU Factorization of an $n \times n$ matrix.

For the parallel version of Doolittle's algorithm, there are two main contributions to the run time. The first is the LU Factorization of local blocks and the second is the communication between processes. In this section, like other section speaking about asymptotic run time, p refers to the total number of MPI processes used. The computational complexity of LU Factorization is $O(n^3/p)$ since each process is responsible for a block of size nxn/p, and the time complexity of LU Factorization for each block is $O(n^3)$. The communication between processes has a cost of $O(n^2/p)$, since each process needs to send and receive a block of size n/size x n. So the overall asymptotic run time of in parallel is $O(n^3/p + n^2/p)$.

Algorithm 6 Pseudocode of approximating the LU Factorization of matrix A with parallel techniques from MPI. This version is a parallel version approach where A is a nxn dense matrices and an empty lower L and upper U nxn empty matrices and where the current rank and size of the properties the programmer decides to run.

```
procedure DOOLITTLE_PARALLEL(A, L, U, n, rank, size)
    block_size ← n/size
    L_local, U_local ← create block_size × n matrices
    for i ← 0 to block_size − 1 do
        L_local_{i,i} ← 1
        for j ← 0 to n − 1 do
            U_local_{i,j} ← 0
        end for
    end for
    A_block ← new double array of size block_size × n
    MPI_Scatter to distribute data
    for k ← 0 to block_size − 1 do
        for j ← k to n − 1 do
            sum ← 0
            for p ← 0 to k − 1 do
                sum ← sum + L_local_{k,p} × U_local_{p,j}
            end for
            U_local_{k,j} ← A_block_{k*n+j} − sum
        end for
        for i ← k + 1 to block_size − 1 do
            sum ← 0
            for p ← 0 to k − 1 do
                sum ← sum + L_local_{i,p} × U_local_{p,k}
            end for
            L_local_{i,k} ← (A_bloc_{i*n+k} − sum)/U_local_{k,k}
        end for
    end for
    MPI_Barrier to halt processes
    for i ← 0 to size − 1 do
        if i = rank then
            MPI_Send local U block
            MPI_Send local L block
        else if rank = 0 then
            MPI_Recv local U block
            MPI_Recv local L block
        end if
    end for
end procedure
```

E. Largest Eigenvalue Approximation with Power Method

The Power Method is a iterative algorithm for approximating the largest eigenvalue of a matrix and its corresponding eigenvector. Given a square matrix A of size n x n, the Power Method starts with an initial nonzero vector x0 and iteratively computes the product of A and the previous vector approximation. At each iteration, the vector is normalized to

prevent it from getting too large or too small. After many iterations, the sequence of vectors converges to the eigenvector corresponding to the largest eigenvalue of A. The largest eigenvalue itself can be estimated by computing the ratio of the norm of the next approximation and the norm of the previous approximation. The Power Method is guaranteed to converge to the largest eigenvalue if A is diagonalizable and the initial vector x0 is not orthogonal to the eigenvector corresponding to the largest eigenvalue, but may not converge under various other circumstances. If the spectral radius of the input matrix is less than 1, then it will always converge to a true solution, which is why we always see convergence within the algorithm itself due to the matrix input having a spectral radius less than 1. This approach is shown in Algorithm 7.

Theoretical analysis of the Power Method reveals that the convergence rate depends on the ratio of the absolute values of the largest eigenvalue and the second largest eigenvalue. The closer this ratio is to one, the slower the convergence rate. In addition, the Power Method is sensitive to round-off errors and can produce inaccurate results if the matrix has a large condition number or if the initial vector is poorly chosen. The asymptotic run time of the power method algorithm for approximating the largest eigenvalue of a matrix is $O(n^2)$. This is because each iteration of the algorithm involves a matrix-vector multiplication, which takes $O(n^2)$ time.

To convert the serial implementation of the Power Method to a parallel implementation using MPI, a programmer can parallelize the computation of the matrix-vector product, which is the most computationally intensive part of the algorithm. In the serial implementation, the matrix-vector product is computed using a nested loop over the rows and columns of the matrix. In the parallel implementation, distributing the rows of the matrix across the MPI processes, and each process can compute the product of its assigned rows with the vector in parallel. To achieve load balancing and minimize communication overhead, the chunk method is used and assigns each chunk to a different process. Once the matrix-vector product is computed, the eigenvalue and eigenvector can be updated using MPI_Allreduce to perform a reduction operation across all processes. The convergence criteria and termination condition remain the same as in the serial implementation, and can be checked using MPI_Allreduce to determine whether any process has converged. This approach is shown in Algorithm 8.

The asymptotic run time of the Power Method with an input of nxn size is $O(k * n^2)$, where n is the dimensions of the matrix and k is the number of iterations required for convergence. This is because in each iteration, the power method involves a matrix-vector multiplication$O(n)$ and a vector normalization $O(n)$ operation. The number of iterations k that it takes for the algorithm to converge must also be taken into account when computing run time, which is why k is important to add additional complexity to the overall run time.

The asymptotic run time the parallel version of the Power Method with a nxn matrix can be analyzed by considering

Algorithm 7 Pseudocode using the Power Method of approximating the largest eigenvalue of matrix A without parallel techniques from MPI. This version is a parallel version approach where A is a nxn dense matrices, iters is the maximum number of iterations and tol is the tolerance level of the algorithm.

procedure POWER_SERIAL($A, n, iters, tol$)
  $\lambda \leftarrow 0.0$
  $\lambda_{old} \leftarrow 1.0$
  $x \leftarrow [1.0, 1.0, ..., 1.0]$
  for $iter \leftarrow 1$ to $iters$ **and** $|\lambda - \lambda_{old}| > tol$ **do**
    $\lambda_{old} \leftarrow \lambda$
    for $i \leftarrow 1$ to n **do**
      $y_i \leftarrow 0.0$
      for $j \leftarrow 1$ to n **do**
        $y_i \leftarrow y_i + A_{i,j} \cdot x_j$
      end for
    end for
    $\lambda \leftarrow 0.0$
    $norm_x \leftarrow 0.0$
    for $i \leftarrow 1$ to n **do**
      $\lambda \leftarrow \lambda + y_i \cdot x_i$
      $norm_x \leftarrow norm_x + x_i^2$
    end for
    $\lambda \leftarrow \lambda / norm_x$
    $norm\_x_{new} \leftarrow 0.0$
    for $i \leftarrow 1$ to n **do**
      $x_i \leftarrow y_i / \lambda$
      $norm\_x\_new \leftarrow norm\_x\_new + x_i^2$
    end for
    $norm\_x\_new \leftarrow \sqrt{norm\_x\_new}$
    for $i \leftarrow 1$ to n **do**
      $x_i \leftarrow x_i / norm\_x_new$
    end for
  end for
end procedure

the number of iterations required for convergence and the computational cost per iteration. The total number of iterations k adds additional complexity, similar to the serial version. Overall computational complexity of the Power Method in parallel is $O(k * n^2/p)$, assuming that the matrix size is much larger than the number of MPI processes. The number of total processes p affects the parallel efficiency of the algorithm, and the optimal value depends on the balance between the computational and communication costs and the available hardware resources.

III. Results

This section will discuss the results of the experimentation that took place with each algorithm which was tested for both numerical accuracy and performance. The results presented in this section will be mostly focused on the performance of each of the algorithms. Tests involving scaling of the size of the input matrices and the amount of total processes in parallel

Algorithm 8 Pseudocode using the Power Method of approximating the largest eigenvalue of matrix A with parallel techniques from MPI. This version is a parallel version approach where A is a nxn dense matrices, iters is the maximum number of iterations and tol is the tolerance level of the algorithm and where the current rank and size of the properties the programmer decides to run.

procedure POWER_PARALLEL($A, n, iters, tol, rank, size$)
  double $\lambda = 1.0, \lambda_{old} = 0.0$
  for $i = 0$ to $n - 1$ **do**
    $x_i = 1.0$
  end for
  $chunk\_size = n/size$
  $start\_index = rank * chunk\_size$
  $end\_index = start\_index + chunk\_size$
  for $iter = 0$ to $iters - 1$ and $|\lambda - \lambda_{old}| > eps$ **do**
    $\lambda_{old} = \lambda$
    for $i = start\_index$ to $end\_index - 1$ **do**
      $y_i = 0.0$
      for $j = 0$ to $n - 1$ **do**
        $y_i = y_i + A_{i,j} * x_j$
      end for
    end for
    double $local\_lambda = 0.0, norm\_x = 0.0$
    for $i = start\_index$ to $end\_index - 1$ **do**
      $local\_lambda = local\_lambda + y[i] * x[i]$
      $norm\_x = norm_x + x[i] * x[i]$
    end for
    $MPI\_Allreduce$ sums local λ to global result
    $MPI\_Allreduce$ sums local $norm$ to global result
    $\lambda = \lambda / norm\_x$
    for $i = start\_index$ to $end\_index - 1$ **do**
      $x[i] = y[i]/\lambda$
    end for
    $MPI\_Allgather$ to gather solutions
  end for
end procedure

runs will be shown as well as showing a comparison between serial and parallel versions of the code.

In every algorithm, a matrix (or matrices) and/or vector(s) are generated for testing. The dense matrices are of the type double and are random numbers initialized. Each

All tests were experimented on different machines with different architectures. The first is my local machine which is not optimized to run parallel algorithms. There was no previous MPI environment set up on this machine. The next environment that was tested was the Department of Computer Science's basement machines which have a pre-existing MPI environment. The last, and a huge contribution to this project was UNM's Center for Advanced Research Computing (CARC) and more specifically the Wheeler supercomputer. This supercomputer was used to test most runs that experienced significant performance improvements and most results shown below are results from CARC experiments. I will be describing the differences I found between each test on the various machines in the respective sections of each algorithm.

A. Matrix-Matrix Multiplication

Experimentation of matrix-matrix multiplication first started with running multiple tests on the each machine. Let's first talk about the numerical accuracy of both the serial and parallel algorithms. Smaller matrices of dimensions such as 10x10 to 25x25 matrices seemed to have very small numerical errors ($\sim$1e-13), but as the matrix increased, numerical errors also increased. Numerical errors seem to increase with at most of an error at the 2048x2048 matrices that had numerical errors or $\sim$1e-10 and no larger or smaller. As matrices grown, numerical errors increase. This is expected because of round-off errors or truncation errors in floating-point arithmetic, so it is not all surprising that the errors were much larger than the smaller matrices. Something interesting to note here and in Table I, is that there is a bit of a drop off in numerical accuracy within the parallel approach to this algorithm. It is not vastly something that is troubling, but it is very noteworthy considering that numerical algorithms have a very important goal in being the most accurate version it can be.

n	Serial Error	Parallel Error
32	4.379e-13	2.309e-13
64	7.081e-13	6.714e-13
128	4.945e-12	8.226e-12
256	3.162e-11	1.492e-11
512	5.813e-11	5.824e-10
1024	1.247e-10	3.997e-10
2048	8.039e-10	1.784e-9

TABLE I: Numerical errors for serial and parallel matrix-matrix multiplication algorithms at increasing dimensions of matrix sizes.

Moving on to the performance test, the experimentation that was done was the serial and parallel tests on the same matrices. The sizes tested were matrices A and B were of the following n sizes: [32, 64, 128, 256, 512, 1024, 2048].

You can see the outcome to the experiment runs on the Wheeler supercomputer with 64 processes used for the parallel version of matrix-matrix multiplication in Fig. 1 where each of these matrices were tested. The parallel algorithm performed significantly better. This was expected since more often than not, the parallel version of the algorithm should cause significant speed up in the algorithm. Fig. 2 demonstrates the exact speed up that is represented with the parallel version of the algorithm.

The parallel version of a simple implementation of matrix-matrix multiplication seemed to drastically improve the performance of the serial version of the algorithm. This speedup of around 20x is very impressive but it did have a trade off with some smaller computational errors with larger matrix dimensions. The trade-off will be at the programmers discretion to whether they would like to be more numerically accurate or perform computations faster.

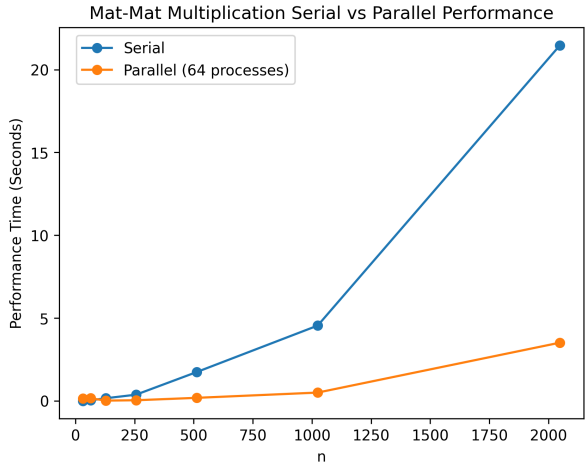

Fig. 1: This figure compares the performance times of a serial implementation and a parallel implementation of matrix-matrix multiplication. The x-axis shows the matrix size, ranging from small to large [32-2048], while the y-axis shows the execution time in seconds. You can see that the parallel implementation outperforms the serial implementation for larger matrix sizes, as expected. The difference in performance between the two implementations becomes more significant as the matrix size increases.

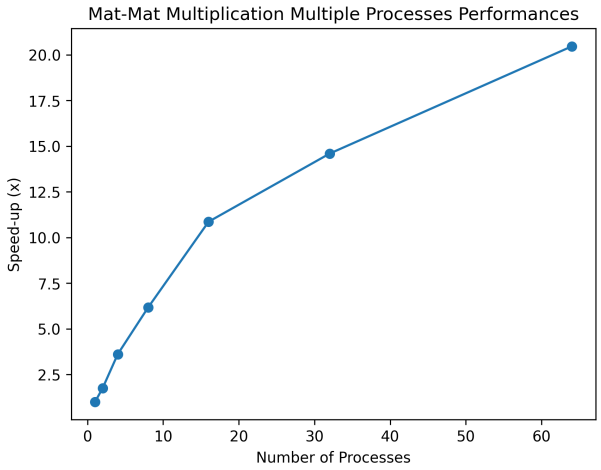

Fig. 2: This figure shows the speed up in performance times for the parallel version of matrix-matrix multiplication as the number of processes increase [1-64]. The x-axis represents the number of processes used for the computation, while the y-axis shows the speed up ratio, which is the ratio of the computation time for the serial version of the algorithm to the computation time for the parallel version of the algorithm. As the number of processes increases, the speed up ratio increases, indicating that the parallel version of the algorithm is performing better than the serial version. You can see that with a small number of processes, the speed up ratio is low, but as the number of processes increases, the speed up ratio becomes more significant with a 20x speed up.

B. Solve Linear System with Gaussian Elimination

Gaussian Elimination tests were the exact same tests as the previous experimentation's done for the matrix-matrix multiplication. With an expectation of larger numerical errors in the parallel algorithm, it was a pleasant surprise that there very little numerical errors in comparison to the serial version of the algorithm. In Table II, there is a significant drop in accuracy as the dimensions of the input matrix and vector increase, but both algorithms are very comparable in numerical accuracy.

n	Serial Error	Parallel Error
32	8.401e-11	7.135e-11
64	8.909e-10	2.673e-10
128	3.769e-10	7.008e-10
256	7.535e-10	4.921e-10
512	6.219e-10	2.755e-10
1024	8.814e-10	4.279e-10
2048	5.223e-9	3.439e-9

TABLE II: Numerical errors for serial and parallel Gaussian elimination algorithms at increasing dimensions of matrix sizes.

When measuring the performance time, there was another significant increase in computational time for Gaussian Elimination done in parallel. This improvement was significant, but not as significant as matrix-matrix multiplication. As you can see in Fig 3, there is a very significant increase in performance when using 64 processes to compute the approximation of a the solution to a linear system. An interesting thing to note in this and the rest of the run time performance analysis plots is that when testing lower dimensional matrices and vectors, the serial algorithm will actually perform better than the parallel algorithm. This is interesting and I believe this is due to the sizes of the data to properly fit into the cache memory system on the CPU. If a program can have their data fit into the cache, that can cause significant speed up because cache is the fastest way to read and write data when computing on a CPU. This will be prevalent throughout all of the tests done on each algorithm.

The speedup time was at an average of about 14x when comparing serial to parallel computational times as seen in Fig 4 and using 64 processes. As the number of processes increase, you can see that the program will rapidly increase in performance time. An interesting note to make about these findings is that jumping from 8 to 16 processes has a smaller speed up increase than expecting. It is an interesting not and I believe that this may be due to the way the parallel operation is implemented and the overhead when sending messages in MPI. Message passing involves copying data from one process to another, which requires communication and synchronization between processes. This communication and synchronization overhead can become significant, especially for applications that require frequent message passing. The algorithm uses a blocking message technique with the MPI_Barrier operation which could contribute to the lack of speed up that that

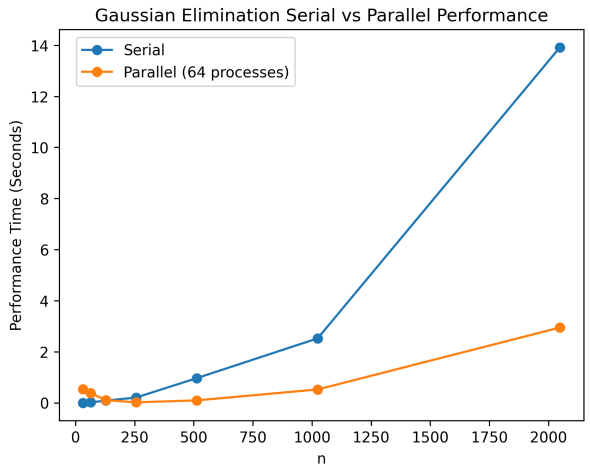

Fig. 3: This figure compares the performance times of a serial implementation and a parallel implementation of Gaussian Elimination. The x-axis shows the matrix size, ranging from small to large [32-2048], while the y-axis shows the execution time in seconds. You can see that the parallel implementation outperforms the serial implementation for larger matrix sizes, just like the matrix-matrix findings. The serial version actually outperforms the parallel version at smaller matrix sizes due to the cache memory usage.

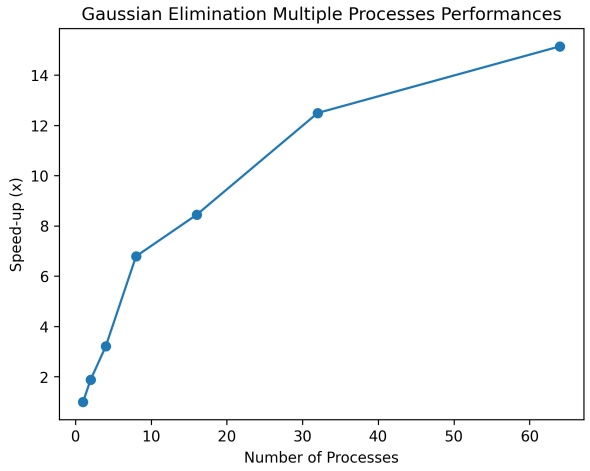

Fig. 4: This figure shows the speed up in performance times for the parallel version of Gaussian Elimination as the number of processes increase [1-64]. The x-axis represents the number of processes used for the computation, while the y-axis shows the speed up ratio. You can see that the maximum speed up with 64 processes was about 14x for Gaussian Elimination.

is seen in this algorithm in comparison to matrix-matrix multiplication.

C. LU Factorization with Doolittle's Algorithm

Computing the LU Factorization with Doolittle's Algorithm proved to have the most interesting accuracy and performance findings in my opinion. First, lets discuss the numerical accuracy of each of the algorithms. As you see in Table III, the numerical errors are not too different for the parallel algorithm compared to the serial version of the algorithm. I believe that since I used point-to-point alongside the block method of distributing tasks, that there was no significant impact on the algorithms accuracy itself.

n	Serial Error	Parallel Error
32	9.385e-11	8.762e-10
64	5.223e-11	3.891e-10
128	1.807e-11	5.924e-10
256	6.937e-10	2.546e-9
512	3.281e-10	2.506e-9
1024	2.451e-10	6.845e-8
2048	7.379e-9	1.064e-8

TABLE III: Numerical errors for serial and parallel Doolittle's algorithms at increasing dimensions of matrix sizes.

While did not see any large accuracy discrepancies between both algorithms, I did see are large difference in performance. In Fig. 5, you can see the performance times measured for both serial and parallel algorithms. At first, it seems like the other experimentation's that have been done with other parallel versus serial algorithms, but here it is seen that there is not a very significant increase in performance for the parallel algorithm. There is still an increase in performance time, but not nearly as impactful as other algorithms that were implemented in parallel. I believe that since I did take a different approach by using the block approach and point to point communication with MPI_Send and MPI_Recv that a large amount of overhead when sending messages and waiting on processes to finish their tasks. That being, the algorithm may decrease performance because the communication patterns involve a large number of messages and the overhead associated with initiating and completing each individual message transfer can become is significant. Very interesting and noteworthy, because I took a risk with implementing this a different way by using point-to-point communication rather than collective communication.

You can see in Fig. 6 that the speed up is at most 5x from the parallel version of Doolittle's algorithm when it is usually expected over a 10x speed up. It also seems to be somewhat plateauing and coming to a threshold of how much this parallel algorithm can actually speed up the serial version.

D. Largest Eigenvalue Approximation with Power Method

The Power Method was the final algorithm for experimentation. This algorithm was tested a bit different because it is an iterative method. With the tolerance level set to 1e-10 and maximum iteration number set to 10000, that allowed

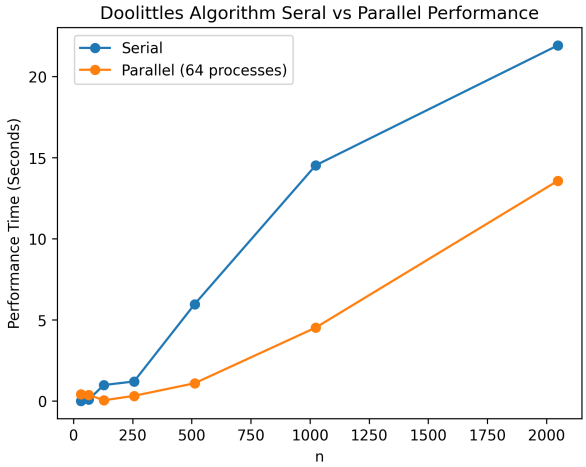

Fig. 5: This figure compares the performance times of a serial implementation and a parallel implementation of Dootlittle's Algorithm. You can see that the parallel algorithm does not have significant performance improvement compared to other parallel algorithms, but it still outperforms the serial version.

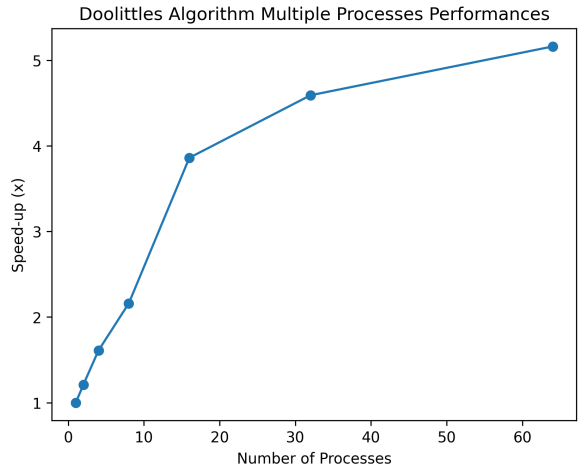

Fig. 6: This figure shows the speed up in performance times for the parallel version of Doolittle's Algorithm as the number of processes increase [1-64]. You can see that the maximum speed up with 64 processes was only about a 5x for Doolittle's algorithm, which is significantly worse than other parallel implementations. You can also see a plateauing effect occurring if and no significant increase if 64 or more processes are used.

experimentation for both numerical accuracy and performance testing. With these configurations, I was expecting over a 10x increase with little to no numerical errors for the parallel version and convergence at the tolerance level for smaller matrices and iteration limit to be reached on larger matrices with more numerical errors appearing for larger matrices. As far as the numerical errors, both algorithms are very comparable with some minor accuracy improvement in the serial algorithm, but nothing that will be concerning. You can

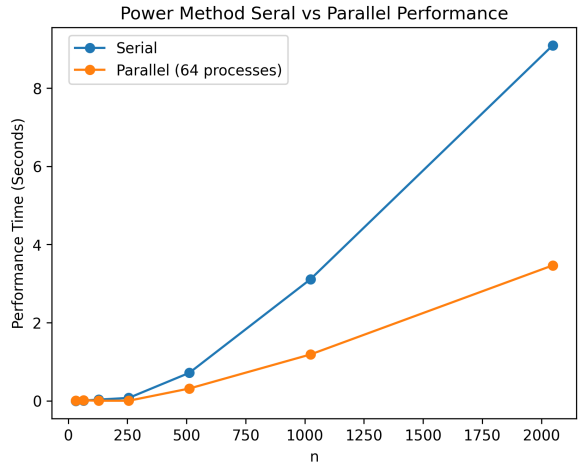

Fig. 7: This figure compares the performance times of a serial implementation and a parallel implementation of the Power Method. You can see that the parallel algorithm performs as expected by outperforming the serial version quite significantly. There is a large divergence when larger matrices are tested on compared to the smaller matrices.

view the numerical accuracy differences between the serial and parallel Power Method algorithms in Table IV.

n	Serial Error	Parallel Error
32	4.735e-10	1.275e-10
64	0.944e-10	4.674e-10
128	7.019e-10	7.939e-10
256	1.803e-10	5.101e-10
512	5.610e-10	9.086e-10
1024	8.561e-8	3.370e-8
2048	2.001e-8	7.782e-8

TABLE IV: Numerical errors for serial and parallel Power Method algorithms at increasing dimensions of matrix sizes.

As far as the performance and comparison to the serial version, I was not surprised to see that the Parallel Power Method performed significantly better. I was surprised because I did not notice a large increase in performance time until 32 and 64 processes were used, which you can see in Fig 8. This was interesting to me, but not surprising. Parallelism is known to increase performance time, but since there is a significant amount of communication overhead between smaller amount of processes, the serial version might be a better approach or there will not be significant improvement in performance. This is why you can see a large jump from 16 to 32 processes in Fig 8, because this run is using more processes to compute faster and although overhead exists, it is still significantly improving that overall computational time when compared to the serial version.

IV. Discussion and Conclusions

Parallel versions of numerical computation can often perform better than serial versions, but this is not always the case. The performance of parallel numerical algorithms depends

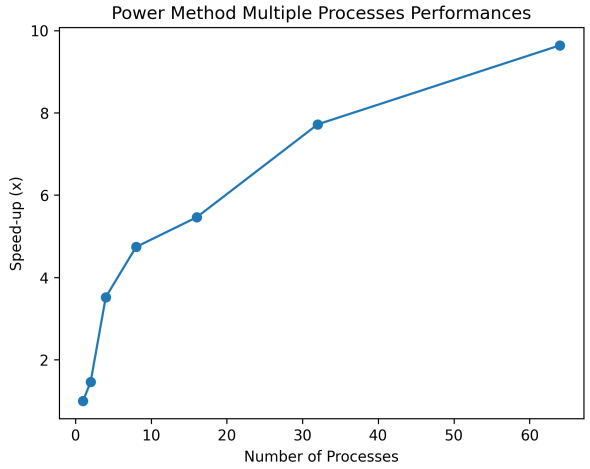

Fig. 8: This figure shows the speed up in performance times for the parallel version of the Power Method as the number of processes increase [1-64]. You can see that the maximum speed up with 64 processes was only about a 10x for the Power Method, which is as expected for convergence with a tolerance of 1e-10. There is also a large increase in speed up as the algorithm is experimented with 32 and 64 processes compared to lower number of total processes being used.

on a variety of factors, including the size of the problem, the hardware architecture, and the communication overhead incurred by distributing the computation across multiple processors. For small problems, the overhead of distributing the computation across multiple processors can outweigh the benefits of parallelism. In this case, a serial version of the algorithm may actually perform better than a parallel version. Furthermore, the hardware architecture can also play a role in determining the performance of parallel numerical algorithms. Some processors are better suited to parallel computation than others, and the performance of a parallel algorithm can be limited by the slowest processor in the system. In addition, the communication overhead incurred by distributing the computation across multiple processors can be significant, particularly if the problem involves a large amount of data. In such cases, the performance of the algorithm can be limited by the bandwidth of the interconnect between the processors.

Parallel algorithms can potentially cause more numerical errors in numerical algorithms, depending on various factors such as the specific algorithm being used, the parallelization approach being employed, the precision and accuracy requirements of the problem, and the hardware and software environment in which the algorithm is running. One reason why parallel algorithms can lead to more numerical errors is that they often involve more complex computations and communication between different processing units, which can introduce additional sources of numerical errors such as round-off errors, truncation errors, and communication errors. For example, in parallel matrix multiplication, the process of dividing the matrices into smaller sub-matrices for parallel processing can introduce rounding errors that accumulate and affect the final result.

In conclusion, parallel numerical algorithms offer the potential to greatly improve the performance of numerical computations, particularly for large-scale problems. However, the implementation of parallel algorithms requires careful consideration of a variety of factors, including load balancing, communication overhead, hardware utilization, and potential numerical errors. While the use of parallel algorithms can lead to significant performance improvements, the benefits are not always linear and must be carefully evaluated for each specific problem. It is essential to understand the trade-offs between serial and parallel implementations and to carefully design and optimize parallel algorithms for maximum performance. With careful consideration and optimization, parallel computing power can be utilized to accelerate numerical algorithms and enable the solution of previously intractable problems.

References

[1] Hoefler, T., Gropp, W., Thakur, R., Träff, J.L. (2010). Toward Performance Models of MPI Implementations for Understanding Application Scaling Issues. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds) Recent Advances in the Message Passing Interface. EuroMPI 2010. Lecture Notes in Computer Science, vol 6305. Springer, Berlin, Heidelberg. https://www.mcs.anl.gov/papers/P1758.pdf

[2] Zhang, J. & Maple, Carsten. (2002). Parallel solutions of large dense linear systems using MPI. 312 - 317. 10.1109/PCEE.2002.1115280. https://www.researchgate.net/publication/3981881-Parallel-solutions-of-large-dense-linear-systems-using-MPI

[3] Martin D. Schatz Robert A. Van De Geijn,J. Poulson, Parallel Matrix Multiplication, A Systematic Journey, Department of Computer Science, Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX https://www.cs.utexas.edu/ flame/pubs/SUMMA2d3dTOMS.pdf

[4] Buttari, A., Langou, J., Kurzak, J. and Dongarra, J. (2008), Parallel tiled QR factorization for multicore architectures. Concurrency Computat.: Pract. Exper., 20: 1573-1590. https://onlinelibrary.wiley.com/doi/10.1002/cpe.1301

[5] Karniadakis, G., & Kirby II, R. (2003). Roots and Integrals. In Parallel Scientific Computing in C and MPI: A Seamless Approach to Parallel Algorithms and their Implementation (pp. 188-254). Cambridge: Cambridge University Press. doi:10.1017/CBO9780511812583.005 https://www.cambridge.org/core/books/abs/parallel-scientific-computing-in-c-and-mpi/roots-and-integrals/17DFA62F5FC1DCE2035A3C6853209A1D