Parallelism in Four Numerical Computational Algorithms

Damian Franco
Department of Computer Science
CS-575 - Numerical Linear Algebra

Intro

Non-parallel and parallel numerical algorithms are very important to every aspect of computer science in today’s world.

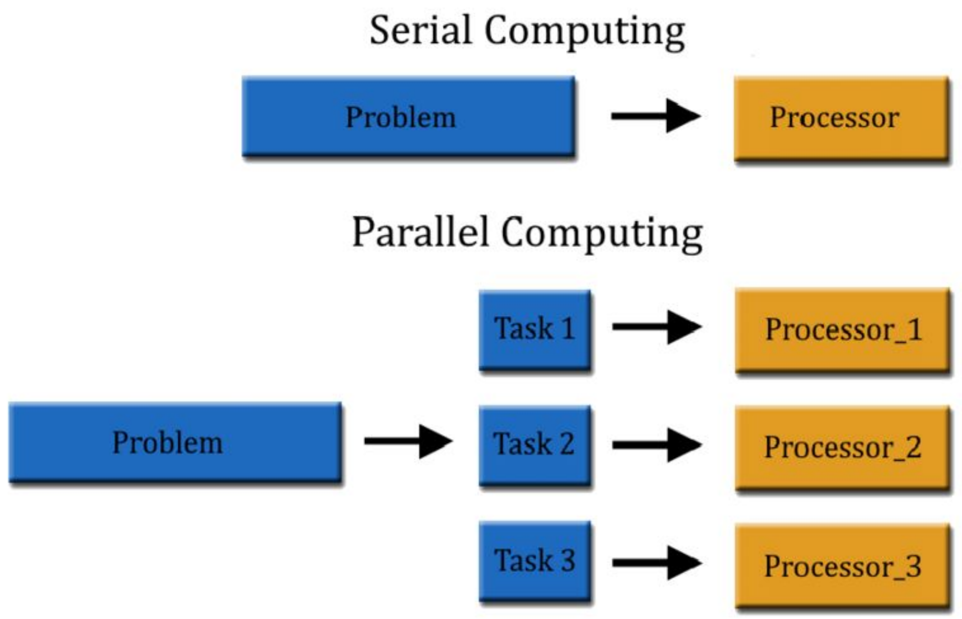
There are countless applications that require parallel computation to efficiently complete a goal.

My Plan

- The study focuses on implementing four different algorithms:
  - *Matrix-matrix multiplication with a simple algorithm*
  - *Solving a linear system with Gaussian Elimination*
  - *LU decomposition with Doolittle's Algorithm*
  - *Approximating the largest eigenvalue with the Power Method*
- Compare each algorithms performance and accuracy between parallel and non-parallel implementations.

What does “parallel” mean

Parallel processing refers to the use of multiple processors or cores within a computer system to simultaneously execute multiple tasks or parts of a single task.



MPI & C/C++

- *MPI (Message Passing Interface)* is a standard library for parallel computing that enables communication between processes running on different nodes in a distributed system.
- *C/C++* are programming languages commonly used for system programming, software development, and numerical computing.
- Can use *MPI* in *C/C++* to make parallel numerical algorithms, you can divide the tasks into smaller subtasks that can be solved concurrently on different nodes using MPI calls

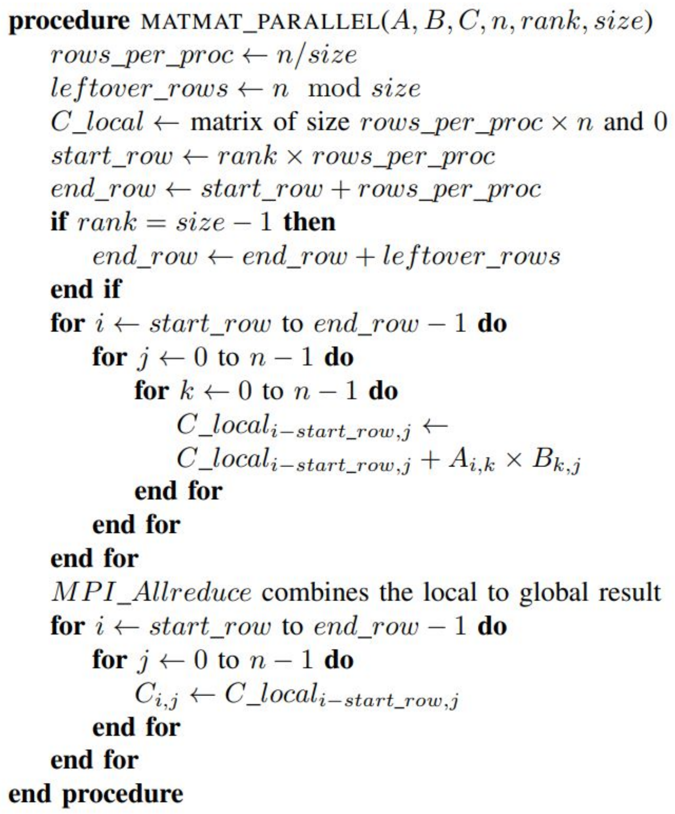
Matrix-Matrix Multiplication

Matrix-matrix multiplication is the process of multiplying two matrices together to produce a third matrix by taking the dot product of each row of the first matrix with each column of the second matrix.

```
procedure MATMAT_SERIAL($A, B, C, n$)
    for $i \leftarrow 0$ to $n-1$ do
        for $j \leftarrow 0$ to $n-1$ do
            $C_{i,j} \leftarrow 0$
            for $k \leftarrow 0$ to $n-1$ do
                $C_{i,j} \leftarrow C_{i,j} + A_{i,k} \times B_{k,j}$
            end for
        end for
    end for
end procedure
```

Flops ~ $O(n^3)$

Matrix-Matrix Multiplication

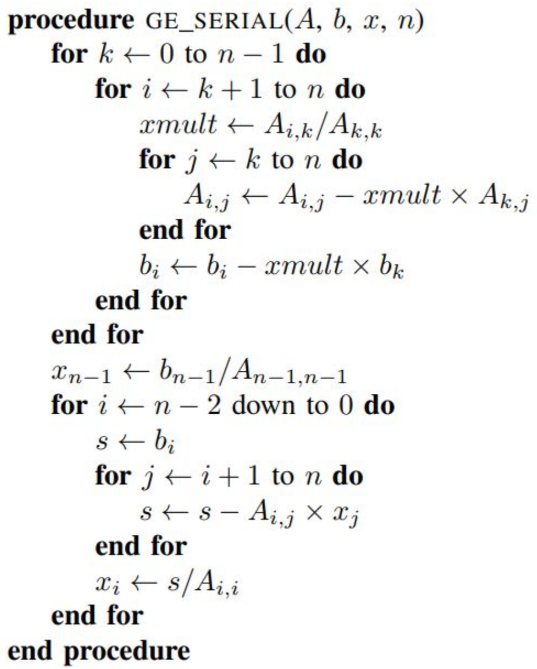


```
procedure MATMAT_PARALLEL($A, B, C, n, rank, size$)
    $rows\_per\_proc \leftarrow n/size$
    $leftover\_rows \leftarrow n \mod size$
    $C\_local \leftarrow$ matrix of size $rows\_per\_proc \times n$ and 0
    $start\_row \leftarrow rank \times rows\_per\_proc$
    $end\_row \leftarrow start\_row + rows\_per\_proc$
    if $rank = size - 1$ then
        $end\_row \leftarrow end\_row + leftover\_rows$
    end if
    for $i \leftarrow start\_row$ to $end\_row - 1$ do
        for $j \leftarrow 0$ to $n - 1$ do
            for $k \leftarrow 0$ to $n - 1$ do
                $C\_local_{i-start\_row,j} \leftarrow$
                $C\_local_{i-start\_row,j} + A_{i,k} \times B_{k,j}$
            end for
        end for
    end for
    $MPI\_Allreduce$ combines the local to global result
    for $i \leftarrow start\_row$ to $end\_row - 1$ do
        for $j \leftarrow 0$ to $n - 1$ do
            $C_{i,j} \leftarrow C\_local_{i-start\_row,j}$
        end for
    end for
end procedure
```

Flops ~ $O(n^3 / p)$
Where p is the the total number of processes.

Solve Linear System with Gaussian Elimination

Gaussian elimination is an algorithm used to solve a system of linear equations by transforming the augmented matrix into an upper triangular matrix through a series of row operations. Once the matrix is in upper triangular form, back substitution can be used to solve for the unknown variables.



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

Flops ~ O(n^3)

Solve Linear System with Gaussian Elimination

```
procedure GE_PARALLEL($A, b, x, n, rank, size$)
    $chunk\_size \leftarrow n/size$
    $start \leftarrow rank \cdot chunk\_size$
    $end \leftarrow start + chunk\_size$
    if $rank = size - 1$ then
        $end \leftarrow n$
    end if
    for $k \leftarrow 0$ to $n - 1$ do
        $pivot\_row \leftarrow$ array of $n$ zeros
        if $rank = k/chunk\_size$ then
            for $j \leftarrow 0$ to $n - 1$ do
                $pivot\_row_j \leftarrow A_{k,j}$
            end for
        end if
        $MPI\_Bcast$ to all processes
        for $i \leftarrow start$ to $end - 1$ do
            if $i \leq k$ then
                continue
            end if
            $xmult \leftarrow A_{i,k}/pivot\_row_k$
            for $j \leftarrow k$ to $n - 1$ do
                $A_{i,j} \leftarrow A_{i,j} - xmult \cdot pivot\_row_j$
            end for
            $b_i \leftarrow b_i - xmult \cdot b_k$
        end for
        $MPI\_Barrier$ to halt processes
    end for
    for $i \leftarrow end - 1$ down to $start$ do
        $s \leftarrow b_i$
        for $j \leftarrow i + 1$ to $n - 1$ do
            $s \leftarrow s - A_{i,j} \cdot x_j$
        end for
        $x_i \leftarrow s/A_{i,i}$
    end for
    $MPI\_Allgather$ to gather solutions
end procedure
```

Flops ~ O(n^3 / p)

Where p is the the total number of processes.

LU Factorization with Doolittle's Algorithm

Doolittle's algorithm for LU factorization is a numerical method that decomposes a matrix into a lower triangular matrix and an upper triangular matrix such that the product of these two matrices is equal to the original matrix. The algorithm is based on the Gaussian elimination method and uses partial pivoting to avoid division by small numbers.

```
procedure DOOLITTLE_SERIAL($A, L, U, n$)
    for $k \leftarrow 0$ to $n-1$ do
        for $j \leftarrow k$ to $n-1$ do
            $sum \leftarrow 0$
            for $p \leftarrow 0$ to $k-1$ do
                $sum \leftarrow sum + L_{k,p} \times U_{p,j}$
            end for
            $U_{k,j} \leftarrow A_{k,j} - sum$
        end for
        for $i \leftarrow k+1$ to $n-1$ do
            $sum \leftarrow 0$
            for $p \leftarrow 0$ to $k-1$ do
                $sum \leftarrow sum + L_{i,p} \times U_{p,k}$
            end for
            $L_{i,k} \leftarrow (L_{i,k} - sum)/U_{k,k}$
        end for
    end for
end procedure
```

Flops ~ $O(n^3)$

LU Factorization with Doolittle's Algorithm

```
procedure DOOLITTLE_PARALLEL($A, L, U, n, rank, size$)
    $block\_size \leftarrow n/size$
    $L\_local, U\_local \leftarrow$ create $block\_size \times n$ matrices
    for $i \leftarrow 0$ to $block\_size - 1$ do
        $L\_local_{i,i} \leftarrow 1$
        for $j \leftarrow 0$ to $n - 1$ do
            $U\_local_{i,j} \leftarrow 0$
        end for
    end for
    $A\_block \leftarrow$ new double array of size $block\_size \times n$
    $MPI\_Scatter$ to distribute data
    for $k \leftarrow 0$ to $block\_size - 1$ do
        for $j \leftarrow k$ to $n - 1$ do
            $sum \leftarrow 0$
            for $p \leftarrow 0$ to $k - 1$ do
                $sum \leftarrow sum + L\_local_{k,p} \times U\_local_{p,j}$
            end for
            $U\_local_{k,j} \leftarrow A\_block_{k*n+j} - sum$
        end for
        for $i \leftarrow k + 1$ to $block\_size - 1$ do
            $sum \leftarrow 0$
            for $p \leftarrow 0$ to $k - 1$ do
                $sum \leftarrow sum + L\_local_{i,p} \times U\_local_{p,k}$
            end for
            $L\_local_{i,k} \leftarrow (A\_bloc_{i*n+k} - sum)/U\_local_{k,k}$
        end for
    end for
    $MPI\_Barrier$ to halt processes
    for $i \leftarrow 0$ 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
```

Flops ~ O(n^3 / p)

Where p is the the total number of processes.

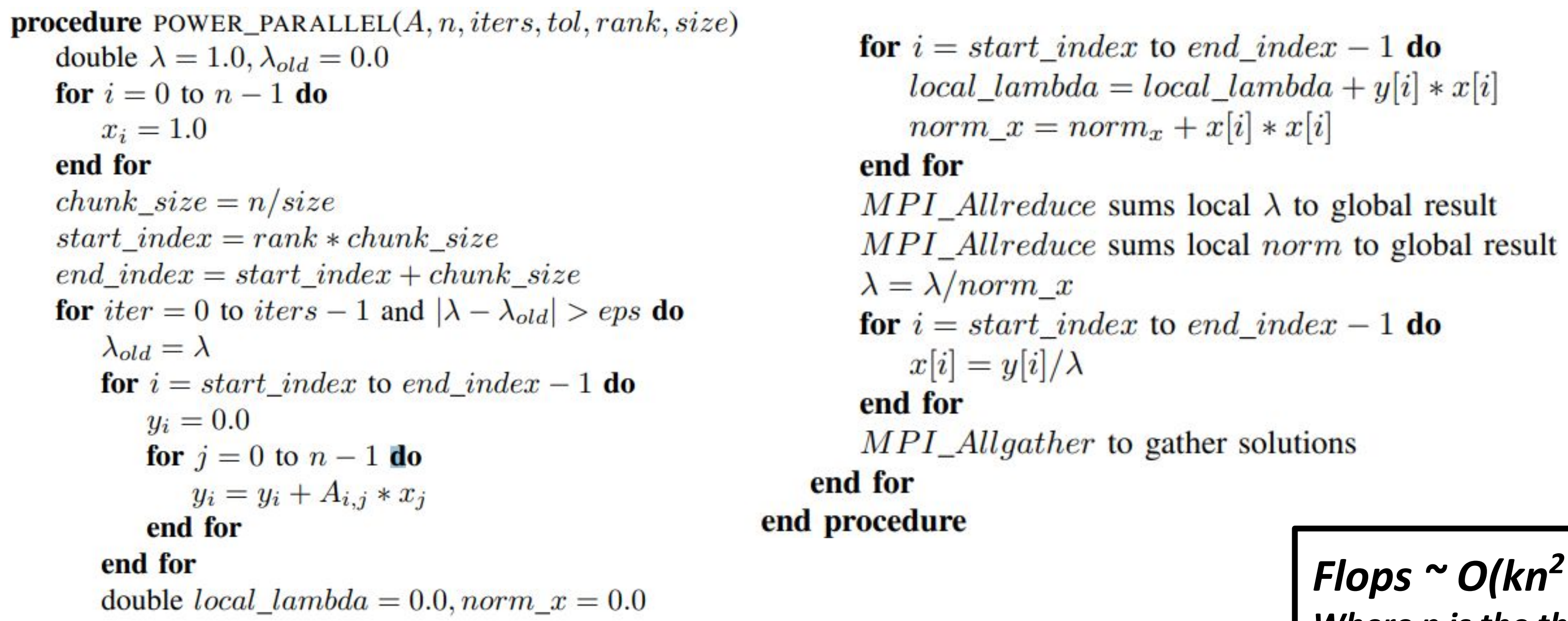
Largest Eigenvalue Approximation with Power Method

The Power method is an iterative numerical algorithm used to find the largest/dominant eigenvalue and corresponding eigenvector of a square matrix by repeatedly multiplying the matrix by a vector and normalizing the result. The algorithm is based on the fact that if the matrix is diagonalizable, then the dominant eigenvalue will be the one with the largest absolute value, and the corresponding eigenvector will be in the direction of the dominant eigenvector.

```
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
            $x_i \leftarrow x_i / norm\_x_{n}ew$
        end for
    end for
end procedure
```

Flops ~ O(kn^2)
Where k is the the total number of iteration.

Largest Eigenvalue Approximation with Power Method

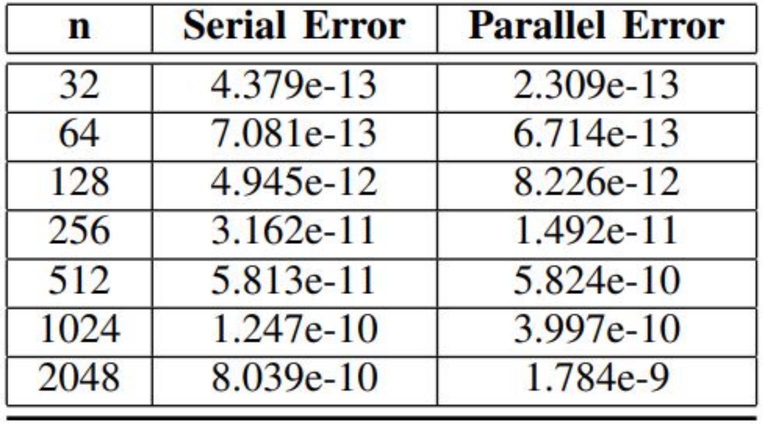


```
procedure POWER_PARALLEL($A, n, iters, tol, rank, size$)
    double $\lambda = 1.0, \lambda_{old} = 0.0$
    for $i = 0$ to $n - 1$ do
        $x_i = 1.0$
    end for
    $chunk\_size = n/size$
    $start\_index = rank * chunk\_size$
    $end\_index = start\_index + chunk\_size$
    for $iter = 0$ 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 $\lambda$ 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
```

Flops ~ $O(kn^2 / p)$
Where p is the the total number of processes.

Results

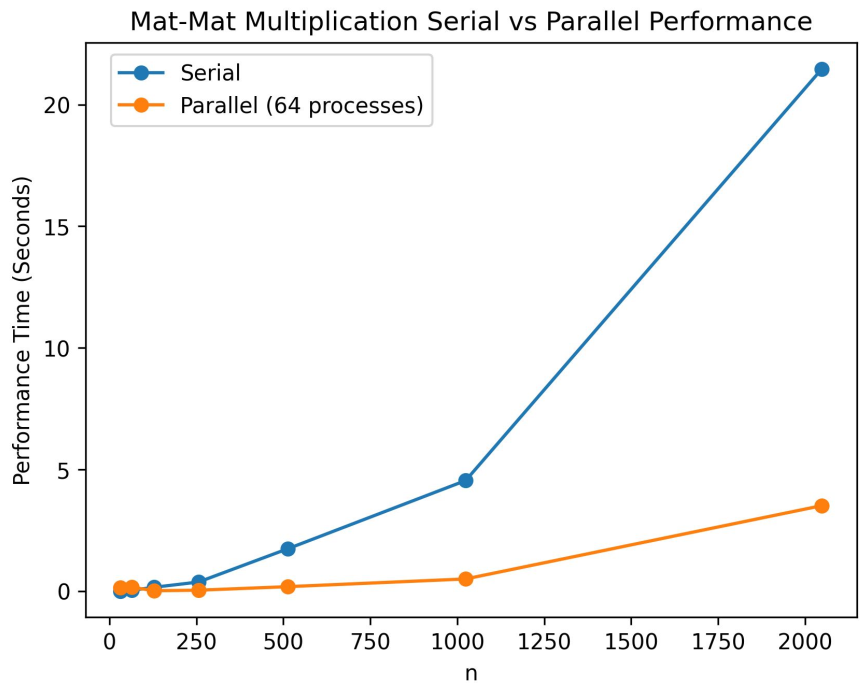
Findings (Matrix-matrix multiplication)

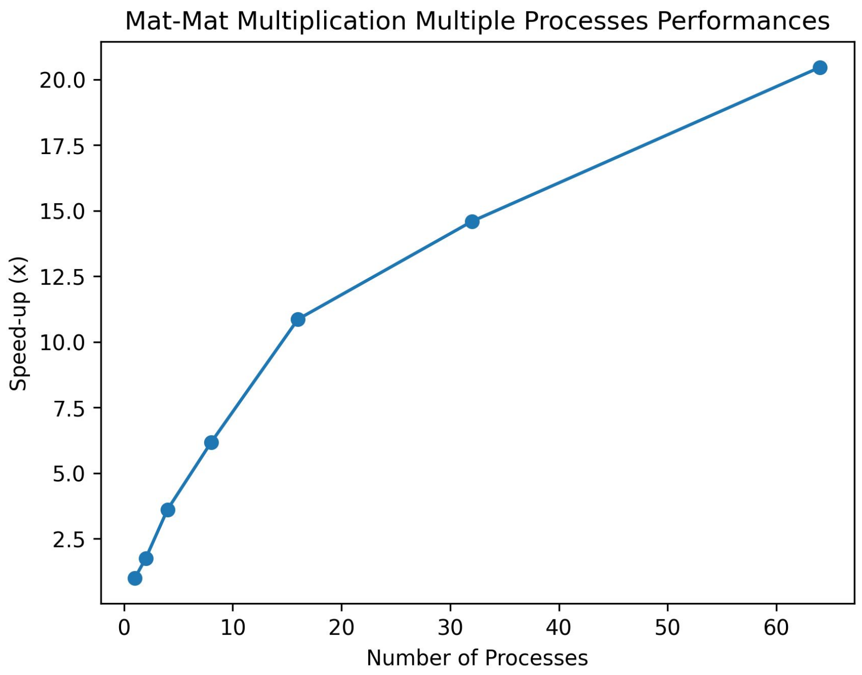


n	Serial Error	Parallel Error
32	4.379e-13	2.309e-13
64	7.081e-13	6.714e-13
128	4.945e-12	8.226e-12
256	3.162e-11	1.492e-11
512	5.813e-11	5.824e-10
1024	1.247e-10	3.997e-10
2048	8.039e-10	1.784e-9

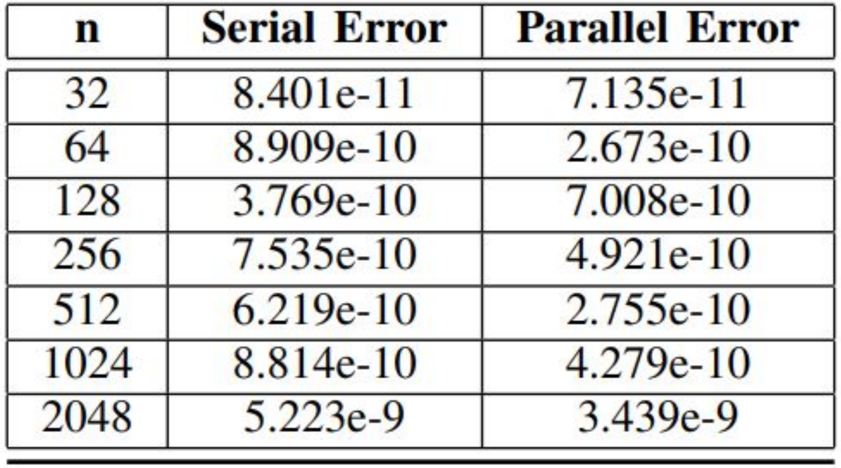
Smaller matrices of dimensions such as 10x10 to 25x25 matrices seemed to have very small numerical errors (~1e-13), but as the matrix increased, numerical errors also increased. Numerical errors seem to increase with at most of an error at the 2048x2048 matrices that had numerical errors or !1e-10 and no larger or smaller. As matrices grown, numerical errors increase.

Findings (Matrix-matrix multiplication)





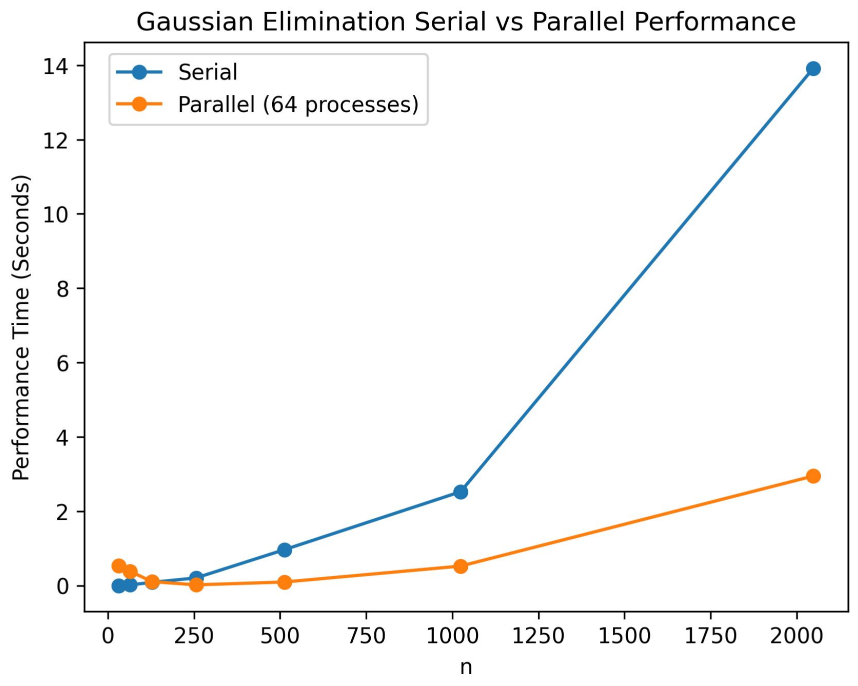
Findings (Gaussian Elimination)

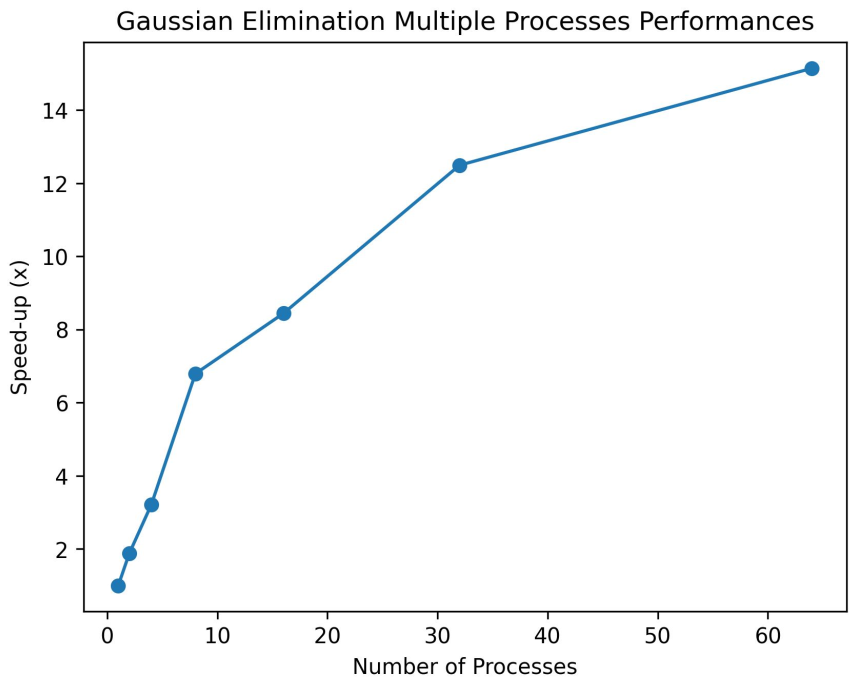


n	Serial Error	Parallel Error
32	8.401e-11	7.135e-11
64	8.909e-10	2.673e-10
128	3.769e-10	7.008e-10
256	7.535e-10	4.921e-10
512	6.219e-10	2.755e-10
1024	8.814e-10	4.279e-10
2048	5.223e-9	3.439e-9

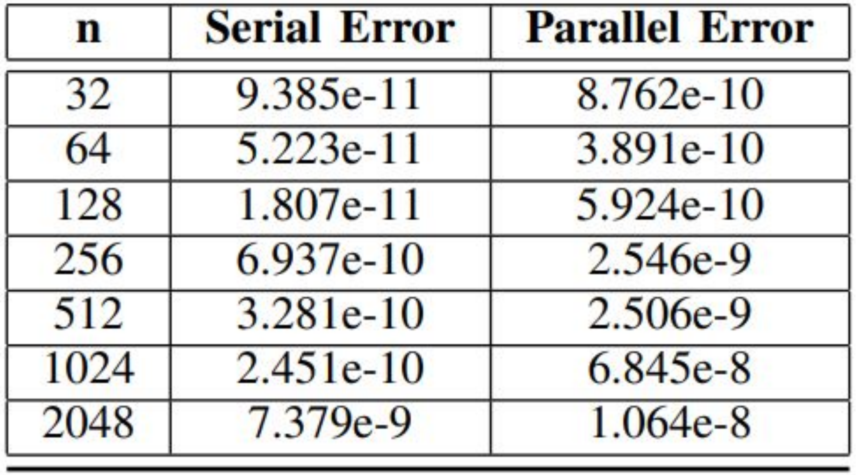
It was a pleasant surprise that there very little numerical errors in comparison to the serial version of the algorithm. In the table above, there is a significant drop in accuracy as the dimensions of the input matrix and vector increase, but both algorithms are very comparable in numerical accuracy.

Findings (Gaussian Elimination)





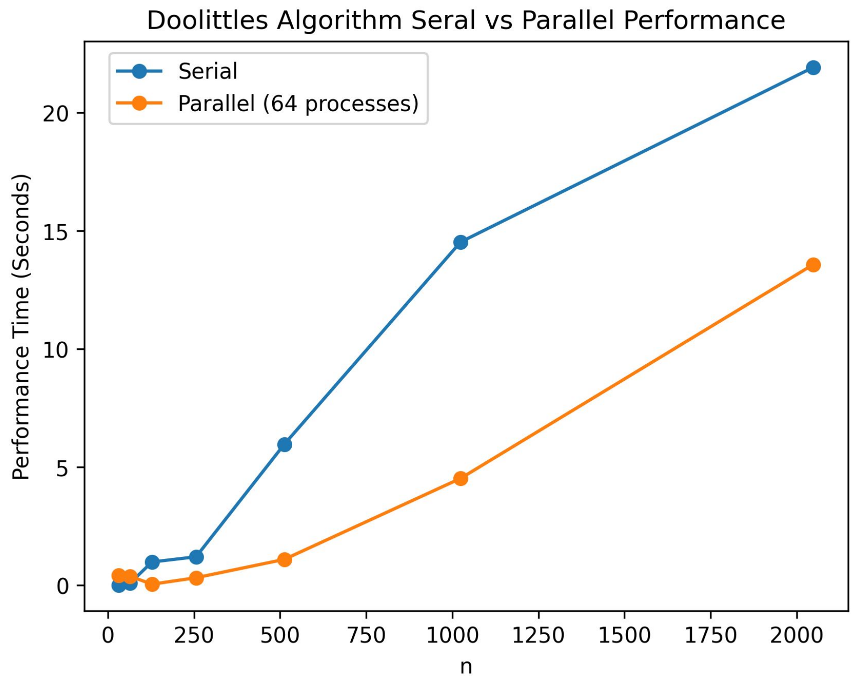
Findings (Doolittle’s Algorithm)

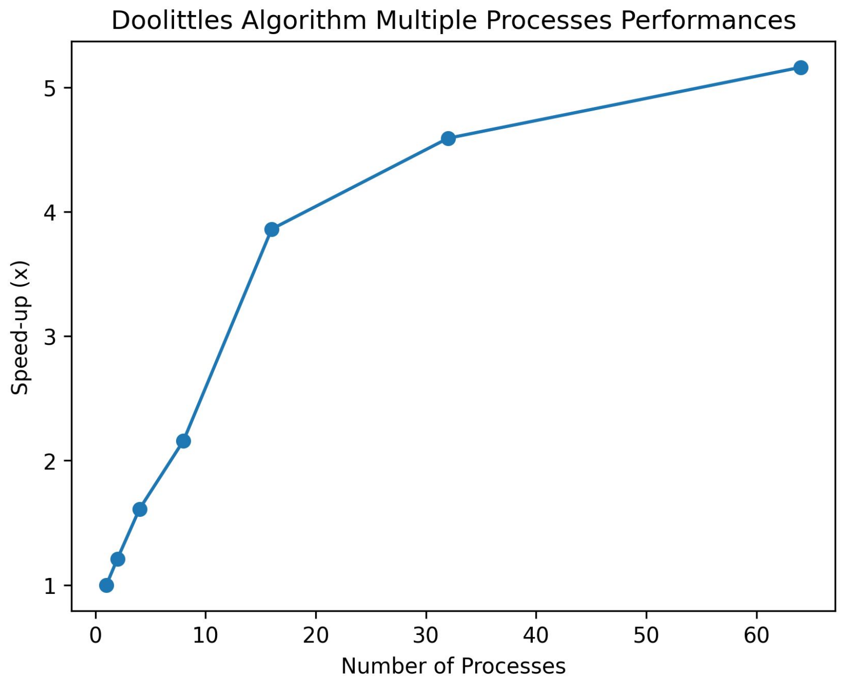


n	Serial Error	Parallel Error
32	9.385e-11	8.762e-10
64	5.223e-11	3.891e-10
128	1.807e-11	5.924e-10
256	6.937e-10	2.546e-9
512	3.281e-10	2.506e-9
1024	2.451e-10	6.845e-8
2048	7.379e-9	1.064e-8

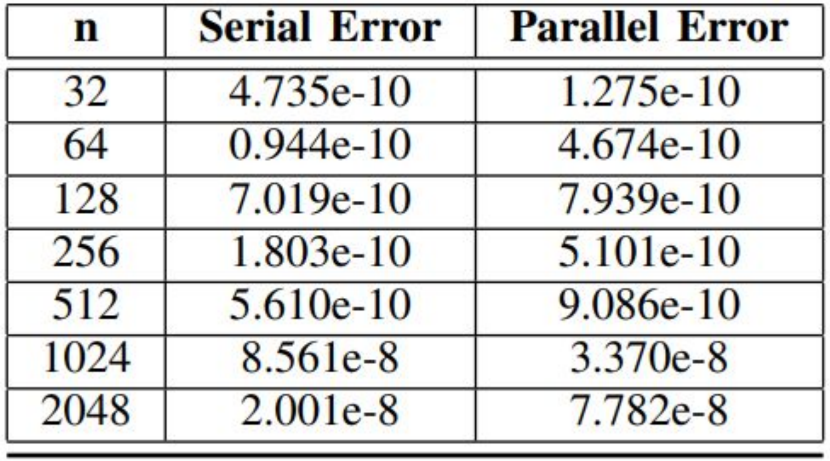
As you see in the table above, the numerical errors are notablably bigger for the parallel algorithm compared to the serial version of the algorithm. I believe that since I used point-to-point alongside the block method of distributing tasks, that there was some numerical errors that significantly impacted the algorithms accuracy itself.

Findings (Doolittle’s Algorithm)





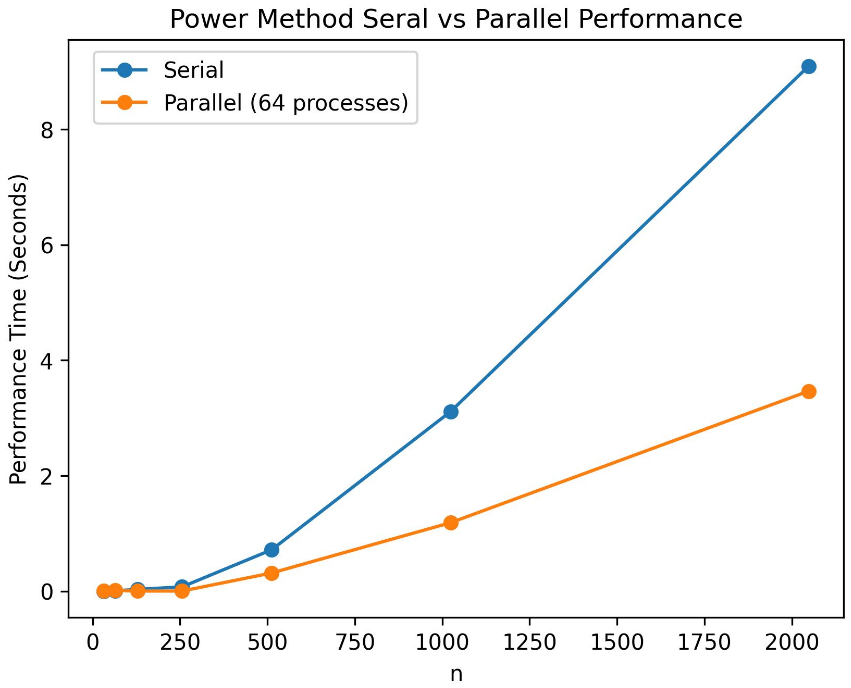
Findings (Power Method)

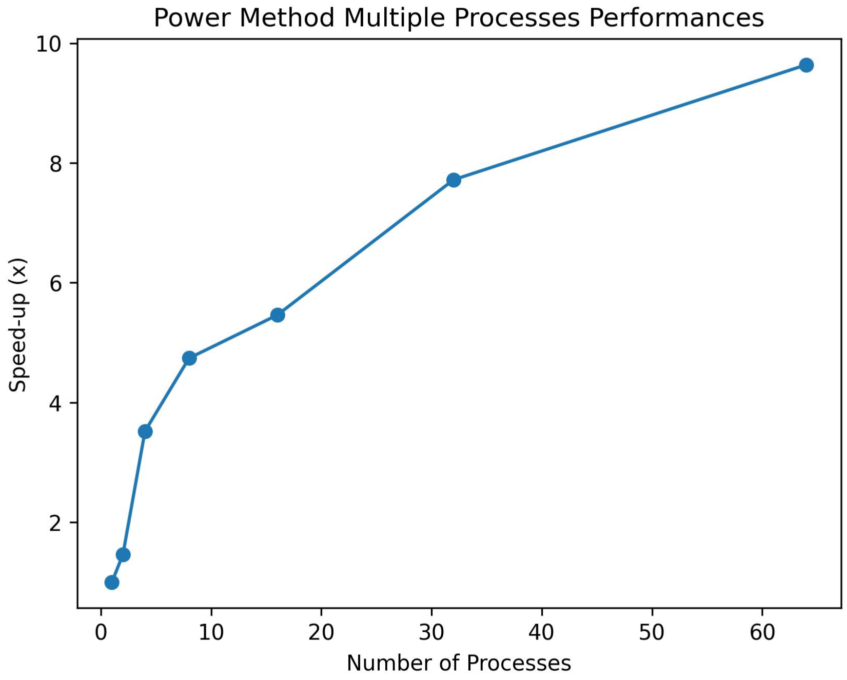


n	Serial Error	Parallel Error
32	4.735e-10	1.275e-10
64	0.944e-10	4.674e-10
128	7.019e-10	7.939e-10
256	1.803e-10	5.101e-10
512	5.610e-10	9.086e-10
1024	8.561e-8	3.370e-8
2048	2.001e-8	7.782e-8

With the tolerance level set to 1e-10 and maximum iteration number set to 10000, that allowed experimentation for both numerical accuracy and performance testing. Both algorithms are very comparable with some minor accuracy improvement in the serial algorithm, but nothing that will be concerning.

Findings (Power Method)





Conclusion & Takeaways

- MPI can be utilized in many ways to make serial numerical algorithms parallel.
- Speed up is guaranteed with a suitable parallel implementation.
- Performance of parallel numerical algorithms depends on many factors including hardware, communication overhead and many more.
- Parallel algorithms can potentially cause more numerical errors, but will never increase accuracy.
- Careful design and optimization practices must take place to ensure success.
- Parallel algorithms are not always the best fit for all situations.

Thank you!