Solving Poisson equation using Successive Over Relaxation

By Yashashvini Rachamallu

1. Introduction

The aim of this project is to solve the 2D poisson equation using the iterative Jacobi method Successive Over Relaxation (SOR) [2] , a popular iterative technique for solving partial differential equations. The Poisson equation [1] is a fundamental partial differential equation (PDE) that arises in a wide range of scientific and engineering applications, including fluid dynamics, electrostatics, and heat conduction. Given the importance of the Poisson equation, it is essential to develop efficient numerical methods to solve it in various contexts.

Our thesis is to implement the SOR method, which helps in solving the poisson equation both in serial and hybrid model [OPENMP+MPI] and to prove hybrid model has lower run time compared to the serial. The structure of the report is organized as Introduction, Methods, Results, Conclusions and References.

1.1. Background on Poisson Equation and Methods

The Poisson equation is a second-order partial differential equation(PDE) that arises in various scientific and engineering applications, such as heat conduction, fluid flow, and electrostatics [1]. It is given by the following expression:

$$\boxed{\nabla^2 \Phi = \sigma(\mathbf{x})}$$

where Δ denotes the Laplace operator, Here σ(x) is the "source term", and is often zero, either everywhere or everywhere bar some specific region. Solving the Poisson equation is crucial for understanding and predicting the behavior of many physical systems [1].

Successive Over-Relaxation (SOR) [2] is an iterative method used to solve linear systems of equations, particularly those arising from the discretization of partial differential equations, such as the Poisson and Laplace equations. The SOR method is an improvement upon the Gauss-Seidel method, which is itself an enhancement of the Jacobi method. The key idea behind SOR is to introduce a relaxation factor (ω) to the Gauss-Seidel method to accelerate the convergence of the solution. The relaxation factor is a scalar value, usually chosen between 1 and 2, that controls the amount of "over-relaxation" applied to each iteration. By tuning the relaxation factor, the SOR [2] method can achieve faster convergence compared to the Gauss-Seidel method, potentially reducing the number of iterations required to reach an acceptable solution.

One of the challenges in applying the Jacobi method to solve the Poisson equation is the computational complexity, particularly when dealing with large-scale problems or high-resolution simulations. To overcome this limitation, various strategies for parallelization and halo exchange have been proposed in the literature [3]. parallelization enables the distribution of computation across multiple processors, while halo exchange facilitates the communication of data between neighboring processors. These techniques can significantly improve the computational efficiency and scalability of the SOR method [4].

A key aspect of implementing parallelization and halo exchange is the use of efficient communication schemes and topologies [5]. Message Passing Interface (MPI) is a widely used standard for parallel programming, providing a set of functions for exchanging messages between processors in a parallel computing environment [5]. In particular, MPI Cartesian topology and communication functions allow for the creation of a structured grid of processors, finding the neighbors, simplifying the communication patterns and improving performance [5].

Another important aspect of parallel programming with MPI is ensuring synchronization between processors. MPI_Barrier is a function that provides a synchronization mechanism, making sure that all processes have reached the same point in their execution before proceeding [6]. This can be crucial for maintaining consistency in the Jacobi method's iterative process.

In addition to parallelization and communication strategies, efficient memory management is also essential for the successful implementation of the Jacobi method. Dynamic memory allocation in C, such as malloc, calloc, free, and realloc, allows for the allocation and deallocation of memory during runtime [7]. By using these functions, it is possible to optimize memory usage and prevent memory leaks, leading to a more efficient and robust implementation of the Jacobi method.

In summary, the background study for the project "Solving Poisson Equation using SOR" involves understanding the Poisson equation, the Jacobi method, and various computational techniques for improving the efficiency and accuracy of the method. The literature review encompasses sources related to the mathematical foundations of the Poisson equation and Jacobi method [1,2], parallelization and halo exchange strategies [3], communication schemes and topologies [4], synchronization mechanisms [5, 6], and dynamic memory allocation [7].

2. Methods

In this section, I will briefly describe the method that was employed to achieve the task of solving poisson equations with SOR. Before going through the actual methods, we will first look at the concept of successive over relaxation (SOR).

2.1. SOR

Successive Over-Relaxation (SOR) is an iterative method for solving linear systems of equations of the form:

$$\mathbf{Ax = b}$$

where A is an n × n matrix, x is an n × 1 column vector of unknowns, and b is an n × 1 column vector of constants. The SOR method is an improvement of the Gauss-Seidel method, which in turn is an enhancement of the Jacobi method. The main idea behind SOR is to use a relaxation factor (ω) to accelerate convergence. Let's break down the SOR method with mathematical explanations.

First, we rewrite the matrix A as the sum of its lower triangular matrix (L), diagonal matrix (D), and upper triangular matrix (U):

A = L + D + U

The Gauss-Seidel method can be expressed as:

x_new = (L + D)^(-1) * (b - U * x_old)

Now, to incorporate the relaxation factor (ω) into the Gauss-Seidel method, we introduce the SOR method as follows:

x_new = (1 - ω) * x_old + ω * (L + D)^(-1) * (b - U * x_old)

where ω is the relaxation factor, which is usually chosen between 1 and 2. In terms of individual components of the x vector, the SOR update formula can be expressed as:

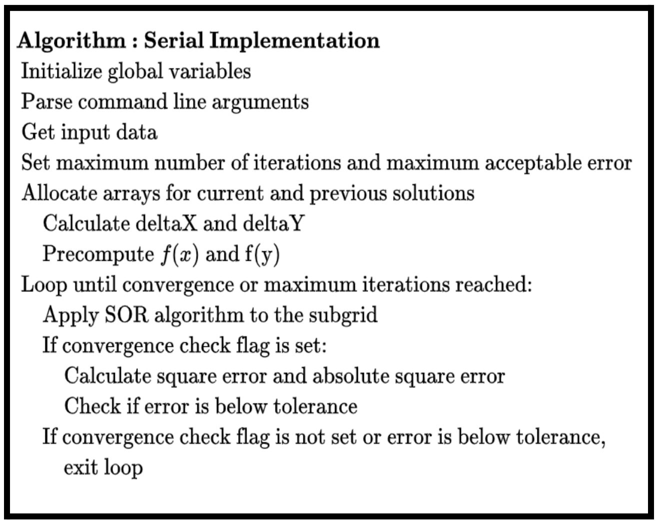
x_new(i) = (1 - ω) * x_old(i) + ω * (1 / A(i, i)) * (b(i) - Σ(A(i, j) * x_new(j) for j = 1 to i-1) - Σ(A(i, j) * x_old(j) for j = i+1 to n))

The SOR method is applied iteratively, starting with an initial guess for the solution x(0), and updating the solution using the formula above until a specified convergence criterion is met. The convergence criterion is typically based on the norm of the difference between consecutive iterates or the residual (b - Ax), which should be smaller than a given tolerance. The choice of the relaxation factor ω is crucial for the convergence properties of the SOR method. An optimal value of ω can lead to faster convergence compared to the Gauss-Seidel method (ω = 1). However, the choice of ω can be problem-dependent, and finding the optimal value may require experimentation.

In summary, the Successive Over-Relaxation (SOR) method is an iterative technique for solving linear systems of equations by introducing a relaxation factor (ω) to improve the convergence rate over the Gauss-Seidel method. The SOR method is particularly useful for solving large-scale problems arising from the discretization of partial differential equations, such as the Poisson and Laplace equations, and can be further enhanced with parallel computing techniques.

2.1.1. Serial Implementation

The pseudo code for the serial implementation is shown below.

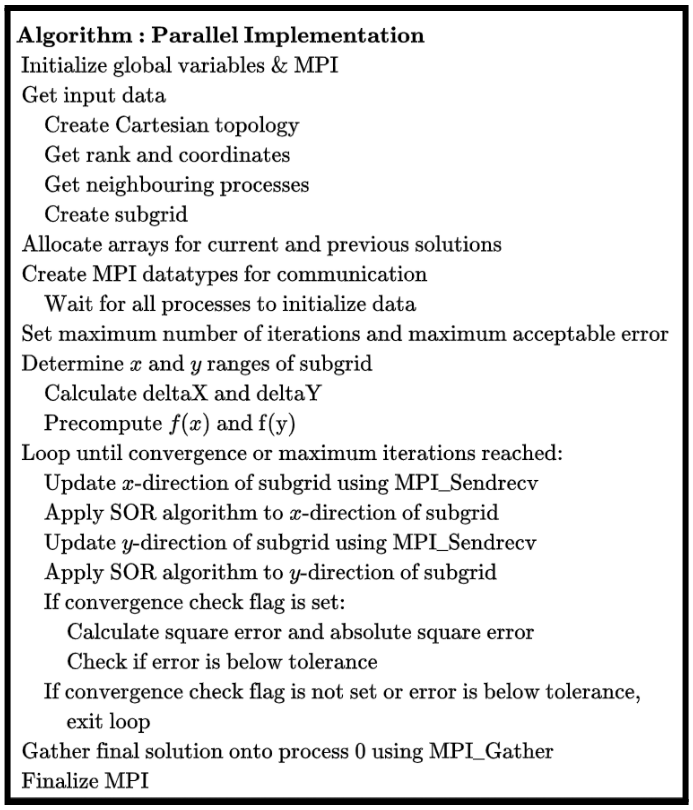


```
Algorithm : Serial Implementation
Initialize global variables
Parse command line arguments
Get input data
Set maximum number of iterations and maximum acceptable error
Allocate arrays for current and previous solutions
  Calculate deltaX and deltaY
  Precompute $f(x)$ and f(y)
Loop until convergence or maximum iterations reached:
  Apply SOR algorithm to the subgrid
  If convergence check flag is set:
    Calculate square error and absolute square error
    Check if error is below tolerance
  If convergence check flag is not set or error is below tolerance,
    exit loop
```

We have produced the runtime and error for different matrices sizes varying from 10*10 , 100*100, 1000*1000 for fixed constraints of Tolerance with value 1e-13 and number of iterations = 50, relaxation factor as 1.

2.1.2. Parallel Implementation [MPI+OpenMPI]

The pseudo code for the parallel implementation is shown below.

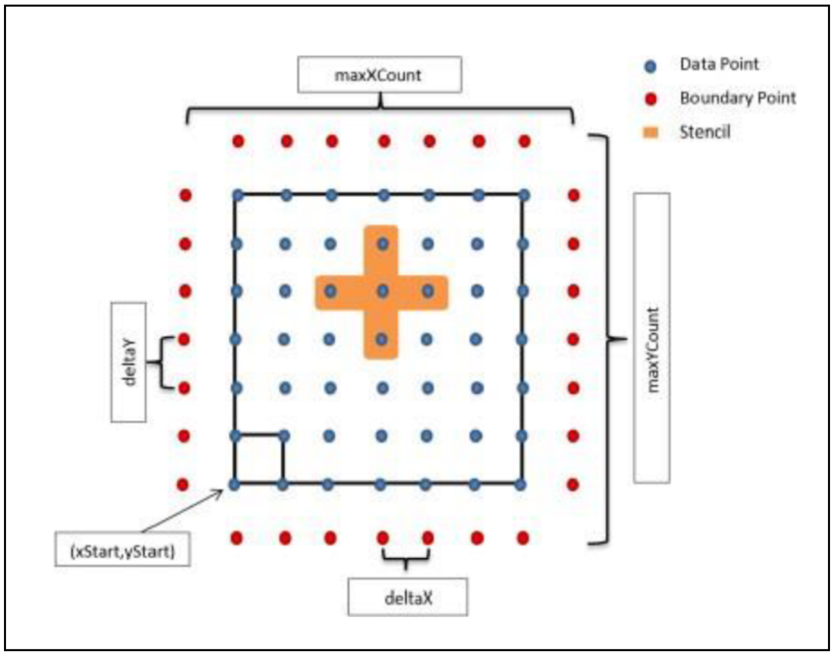


```
Algorithm : Parallel Implementation
Initialize global variables & MPI
Get input data
    Create Cartesian topology
    Get rank and coordinates
    Get neighbouring processes
    Create subgrid
Allocate arrays for current and previous solutions
Create MPI datatypes for communication
    Wait for all processes to initialize data
Set maximum number of iterations and maximum acceptable error
Determine $x$ and $y$ ranges of subgrid
    Calculate deltaX and deltaY
    Precompute $f(x)$ and f(y)
Loop until convergence or maximum iterations reached:
    Update $x$-direction of subgrid using MPI_Sendrecv
    Apply SOR algorithm to $x$-direction of subgrid
    Update $y$-direction of subgrid using MPI_Sendrecv
    Apply SOR algorithm to $y$-direction of subgrid
    If convergence check flag is set:
        Calculate square error and absolute square error
        Check if error is below tolerance
    If convergence check flag is not set or error is below tolerance,
        exit loop
Gather final solution onto process 0 using MPI_Gather
Finalize MPI
```

I produced the runtime and error for different matrices sizes varying from 10*10 , 100*100, 1000*1000, 10000*10000, different thread counts varying from 1,2,4,8,16,32,64 and different nodes from 1,2,4,8,16,32 for fixed constraints of Tolerance with value 1e-13 and number of iterations = 50, relaxation factor as 1.

We have used the **two parallel paradigms** which are MPI and OPENMP. I used a hybrid parallel programming model, MPI + OPENMP to implement the SOR method, which helped in decreasing the runtime compared to serial.

2.3. Boundary Conditions & Neighboring Processes



The boundary conditions are set to solve the Poisson equation in the SOR method, where the domain is discretized into a grid of points, and each point is assigned an initial guess value for the solution. At each iteration, the value of each point is updated based on the average of its four neighboring points.

To apply the boundary conditions, the values of the points on the edges of the domain are set to the desired boundary conditions values, and they are excluded from the iteration. In the code, the boundary condition values are read from the input by process 0 and broadcast to all other processes using MPI_Bcast.

The Cartesian topology created using MPI_Cart_create helps to determine the neighboring processes to which process I should communicate to exchange boundary values. This is done using the MPI_Sendrecv function. The code treats the four edges of the domain separately, and for each edge, the neighboring points are determined using MPI_Cart_shift.

2.4. Different Parallelization techniques:

The Parallelization strategies implemented in the our algorithm was as follows:

1. **Task Parallelization:** This refers to dividing the computation into smaller, independent tasks that can be executed concurrently. The code does not explicitly use any task parallelization constructs, but the MPI implementation allows for natural task parallelization. Each process works on a separate subset of the problem, and the computation is divided into smaller tasks that can be executed independently by each process.

2. **Data Parallelization:** This refers to dividing the data across multiple processes. The code uses MPI to split the data across different processes. The input data is read on process 0 and broadcast to all other processes using MPI_Bcast. This allows each process to work on a portion of the input data. Additionally, the solution is divided across different processes, with each process computing a portion of the solution.
3. **Domain Parallelization:** This refers to dividing the computation across a two-dimensional grid or domain, with each process responsible for computing a subset of the grid. The code uses MPI to create a Cartesian topology, with each process assigned a unique coordinate in the grid. This allows each process to compute a portion of the solution corresponding to its portion of the grid. The domain parallelization technique is particularly useful when the computation involves solving partial differential equations, where the solution depends on multiple variables. Here the solution is divided into a two-dimensional grid, with each process responsible for computing a subset of the grid. The MPI_Cart_create function is used to create a Cartesian topology, with each process assigned a unique coordinate in the grid. This allows each process to compute a portion of the solution corresponding to its portion of the grid.
4. **Hybrid Parallelization:** This refers to using a combination of different parallelization techniques, such as MPI and OpenMP. The code uses a combination of MPI and OpenMP to parallelize the computation. MPI is used for parallelization across multiple processes, with each process utilizing OpenMP to parallelize computation within itself. Specifically, OpenMP is used in the SolveJacobi function to parallelize the computation of the SOR method for solving the system of equations. The hybrid parallelization technique can be useful when the computation involves different types of parallelism, and when using a single parallelization technique is not sufficient to achieve the desired performance.

2.5. Load Balancing

The load balancing in this code is done using MPI, specifically by distributing the work across multiple processes. The function Get_input() reads input from the standard input and broadcasts it to all the processes using MPI_Bcast(). This function is called by all processes, but only the process with rank 0 reads the input. Then, a cartesian topology is created using MPI_Cart_create() with the number of processes as input. This is done to make communication between processes easier since each process is now identified by a set of Cartesian coordinates.
The Grid is then distributed among processes, and each process is assigned to work on a sub-grid, also known as a local_grid. The local_grid size is calculated based on the global grid size and the number of processes. Each process is then assigned an equal number of rows and columns of the Grid to work on, and a local_grid is created for each process. This step ensures that the work is distributed evenly among the processes. The SOR method is then applied to each local_grid, and the solution is computed. The computation is done in a loop, where each process sends its boundary values to its neighboring processes using MPI_Send() and MPI_Recv(). This is done in a non-blocking manner using MPI_Isend() and MPI_Irecv(), and the computation is continued while the messages are being sent and received. Once the desired tolerance level is reached, the final solution is gathered to the process with rank 0 using MPI_Gatherv() and printed to the standard output.

The load balancing in this code is done by distributing the work among multiple processes, where each process works on a sub-grid. The work is distributed evenly among processes by assigning each process an equal number of rows and columns of the Grid to work on. This ensures that the work is distributed evenly among processes. The communication overhead is minimized by using non-blocking message passing and overlapping communication with computation.

2.6. Memory Usage

Code reads some input parameters from the standard input and then creates a Cartesian topology of MPI processes. Then, it distributes the work to each process and applies the numerical method to compute the solution of the PDE. Finally, it gathers the results from all the processes and computes some statistics about the solution. The input parameters are read from the standard input, which is typically a small amount of data, so it does not have a significant impact on the memory usage.

The numerical method requires an array of size (n+2) x (m+2) to store the solution of the PDE. Each MPI process stores a portion of this array, which has size (n_p+2) x (m_p+2), where n_p and m_p are the local sizes of the array. Therefore, the total memory used to store the solution is proportional to the product of the number of MPI processes and the size of the local array. This means that the memory usage scales linearly with the number of MPI processes, but it depends on the local size of the array, which may or may not be proportional to the global size of the array. The MPI functions MPI_Scatterv and MPI_Gatherv are used to distribute and gather the data, respectively. These functions require additional memory to store the send and receive buffers, which may increase with the number of MPI processes and the size of the local arrays.

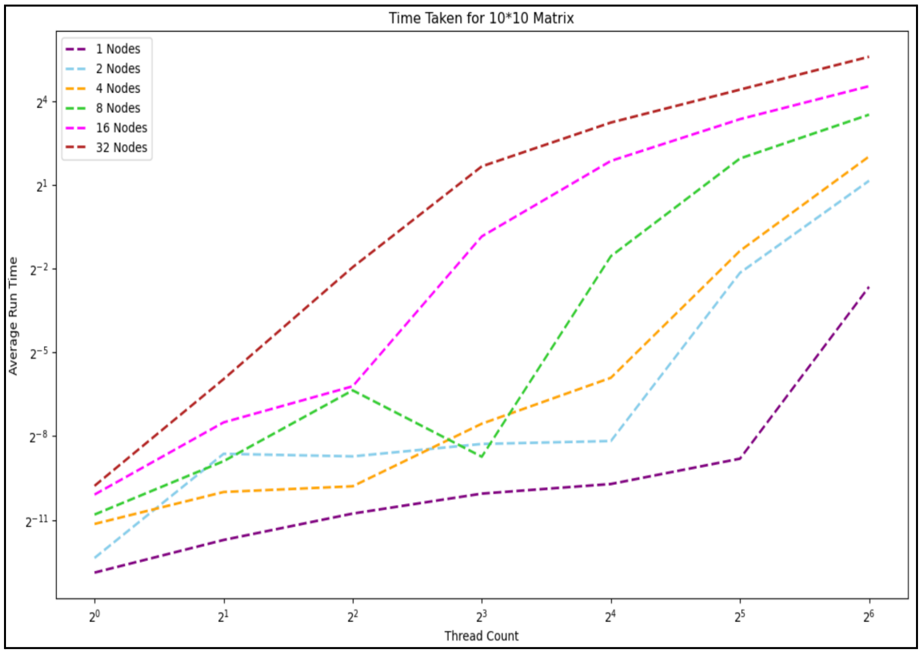
The OpenMP parallel region in the inner loop of the numerical method creates a team of threads that execute the loop in parallel. Each thread has its own stack and may require additional memory to store temporary variables. The memory usage of the OpenMP parallel region depends on the number of threads and the size of the local array, but it does not depend on the number of MPI processes.

To handle large sized matrices, we allocate memory dynamically using malloc() function. This allows you to allocate memory as needed during program execution rather than reserving memory in advance. This way you can allocate and deallocate memory on the fly based on the data size.

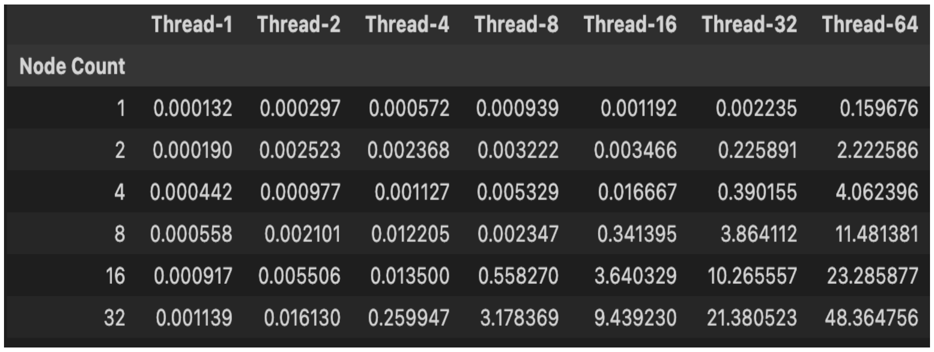
3. Results

The main idea/aim for our project is to prove that the hybrid model has lower run time compared to the serial in the case where we solve poisson equation using SOR.
By running the hybrid programming model code I produced the runtime and residual value for different matrices sizes varying from 10*10 , 100*100, 1000*1000, 10000*10000 with different thread counts varying from 1,2,4,8,16,32,64 and different nodes from 1,2,4,8,16,32 for fixed constraints of Tolerance with value 1e-13 and number of iterations as 50, relaxation factor as 1. The below are the results that were obtained from my implementations in graphical representation.

For Matrix-Size : 10X10



We plotted the log-log scale line plots between the run time and thread count for the matrix size 10*10 and each line in the graph has a fixed number of nodes used. Fixing the matrix size at 10x10, for all nodes count except 8 the runtime continues to increase with increasing thread count.We expect this is happening because of the overhead associated with thread creation, synchronization, and communication. As the number of threads increases, the overhead may start to outweigh the benefits of parallelization, leading to diminishing returns or even a slowdown in performance. Only in the case of 8 nodes, we see a sudden drop in runtime followed by gradual increase this may be because of the reduced communication overhead.This again suggests that as we increase the number of nodes, the communication cost between ranks begins to overwhelm the gain from increasing the threads per rank. For a better understanding of the plot you can refer to the below table which shows you the run time values at different node counts and thread counts.

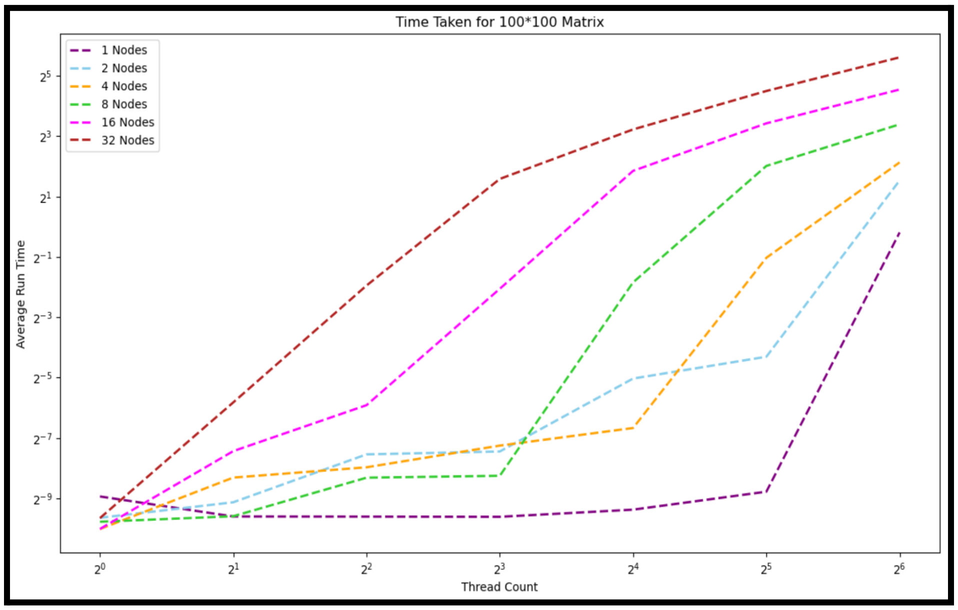


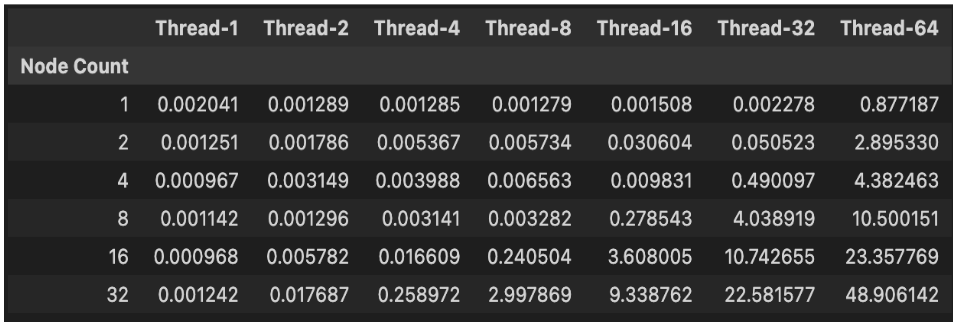
	Thread-1	Thread-2	Thread-4	Thread-8	Thread-16	Thread-32	Thread-64
Node Count							
1	0.000132	0.000297	0.000572	0.000939	0.001192	0.002235	0.159676
2	0.000190	0.002523	0.002368	0.003222	0.003466	0.225891	2.222586
4	0.000442	0.000977	0.001127	0.005329	0.016667	0.390155	4.062396
8	0.000558	0.002101	0.012205	0.002347	0.341395	3.864112	11.481381
16	0.000917	0.005506	0.013500	0.558270	3.640329	10.265557	23.285877
32	0.001139	0.016130	0.259947	3.178369	9.439230	21.380523	48.364756

For Matrix-Size : 100X100

We analyzed the plots displaying the run time for a 100x100 matrix, considering various combinations of node and thread counts. For most of the node counts, we observed that the runtime tends to increase as the thread count increases. This trend can be attributed to the overhead associated with thread creation, synchronization, and communication due to the smaller matrix size. As the number of threads increases, the overhead may outweigh the benefits of parallelization, leading to diminishing returns or even a slowdown in performance. The only exception is the case with 1 node, where we observe a minimum runtime at 16 threads, followed by an increase in runtime as the thread count increases.

When we examine the graph , we notice a few interesting trends. For 2, 4, and 8 nodes, the run time increases slowly. It's like till the thread count 8 then after the run time increases with high step count, this may be because of the increase in communication head as increase in threads and nodes count. In the case of 16,32 nodes, the run time was gradually increased since the thread count -1 to 64. For a better understanding of the plot you can refer to the below table which shows you the run time values at different node counts and thread counts.





	Thread-1	Thread-2	Thread-4	Thread-8	Thread-16	Thread-32	Thread-64
Node Count							
1	0.002041	0.001289	0.001285	0.001279	0.001508	0.002278	0.877187
2	0.001251	0.001786	0.005367	0.005734	0.030604	0.050523	2.895330
4	0.000967	0.003149	0.003988	0.006563	0.009831	0.490097	4.382463
8	0.001142	0.001296	0.003141	0.003282	0.278543	4.038919	10.500151
16	0.000968	0.005782	0.016609	0.240504	3.608005	10.742655	23.357769
32	0.001242	0.017687	0.258972	2.997869	9.338762	22.581577	48.906142

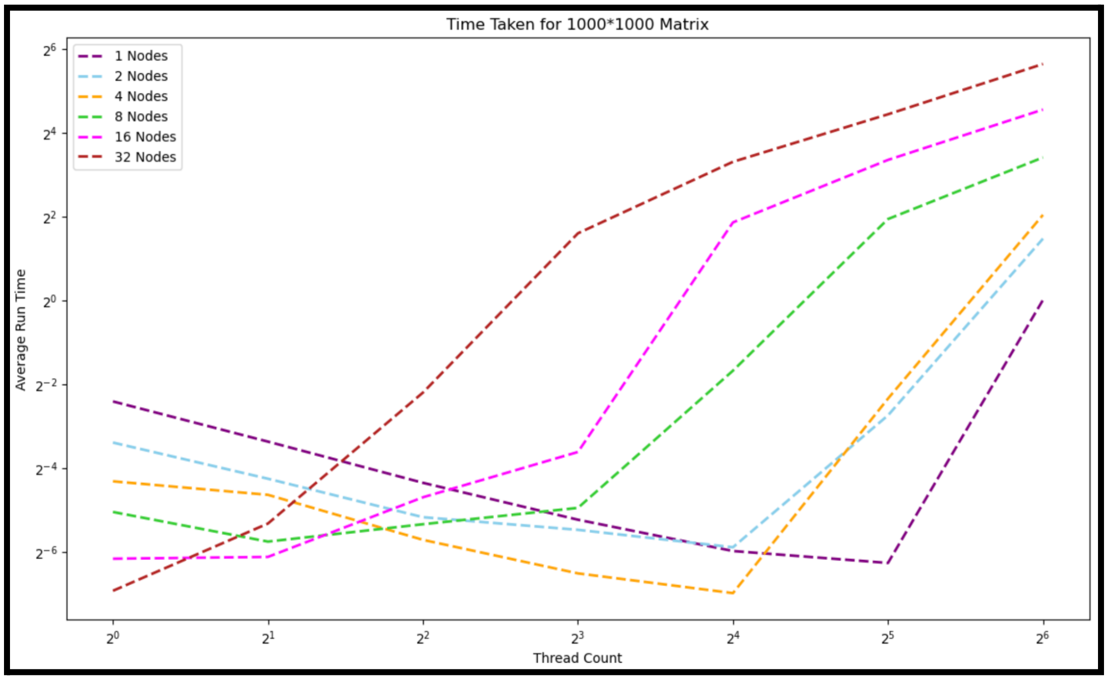
For Matrix-Size : 1000X1000

For a single node, the runtime decreases with an increase in the number of threads up to 32 threads. This is expected, as more threads allow for better parallelization and efficient use of computational resources. However, beyond 32 threads, there is a significant increase in runtime, possibly due to the overhead associated with thread creation, synchronization, and communication outweighing the benefits of parallelization.

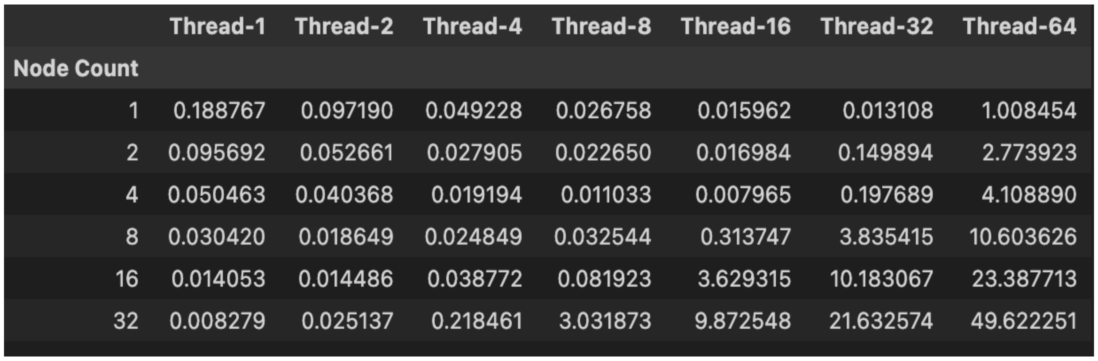
For 2 and 4 nodes, the runtime consistently decreases as the number of threads increases up to 8 and 16 threads, respectively. This suggests that the benefits of parallelization are still being realized at these levels. However, beyond these points, the runtime increases again, likely due to the increasing overhead and communication costs between threads.

For 8 nodes, the runtime first decreases, then increases, and finally decreases again with an increasing number of threads. The minimum runtime is observed at 2 threads per processor. This unusual pattern might be attributed to specific characteristics of the underlying hardware or software environment, which could result in more efficient performance at particular combinations of nodes and threads.

For 16 and 32 nodes, the runtime exhibits a more irregular pattern, with the minimum runtime achieved at lower thread counts (4 threads for 16 nodes and 2 threads for 32 nodes). As the number of nodes increases, the communication costs between ranks seem to have a more significant impact on the runtime, resulting in a higher sensitivity to the number of threads employed.



For a better understanding of the plot you can refer to the below table which shows you the run time values at different node counts and thread counts.



Node Count	Thread-1	Thread-2	Thread-4	Thread-8	Thread-16	Thread-32	Thread-64
1	0.188767	0.097190	0.049228	0.026758	0.015962	0.013108	1.008454
2	0.095692	0.052661	0.027905	0.022650	0.016984	0.149894	2.773923
4	0.050463	0.040368	0.019194	0.011033	0.007965	0.197689	4.108890
8	0.030420	0.018649	0.024849	0.032544	0.313747	3.835415	10.603626
16	0.014053	0.014486	0.038772	0.081923	3.629315	10.183067	23.387713
32	0.008279	0.025137	0.218461	3.031873	9.872548	21.632574	49.622251

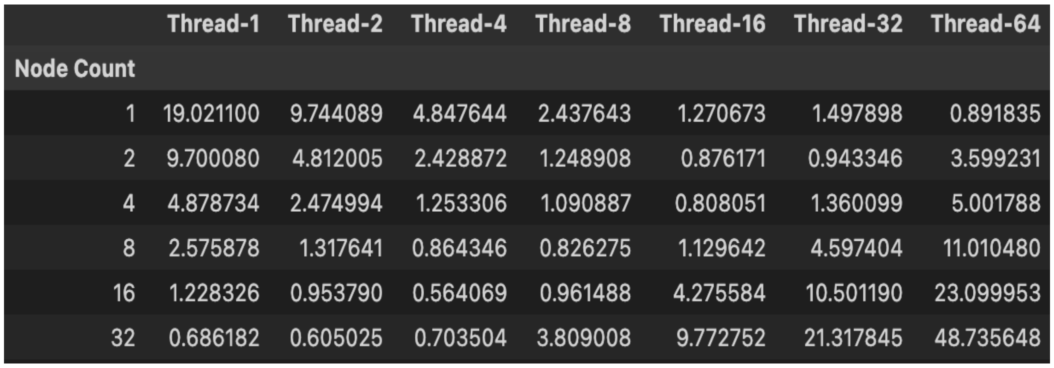
For Matrix-Size : 10000X10000

For a single node, the runtime consistently decreases with an increase in the number of threads up to 16 threads. This is expected, as more threads allow for better parallelization and efficient use of computational resources. However, beyond 16 threads, the runtime starts to increase again, possibly due to the overhead associated with thread creation, synchronization, and communication outweighing the benefits of parallelization.

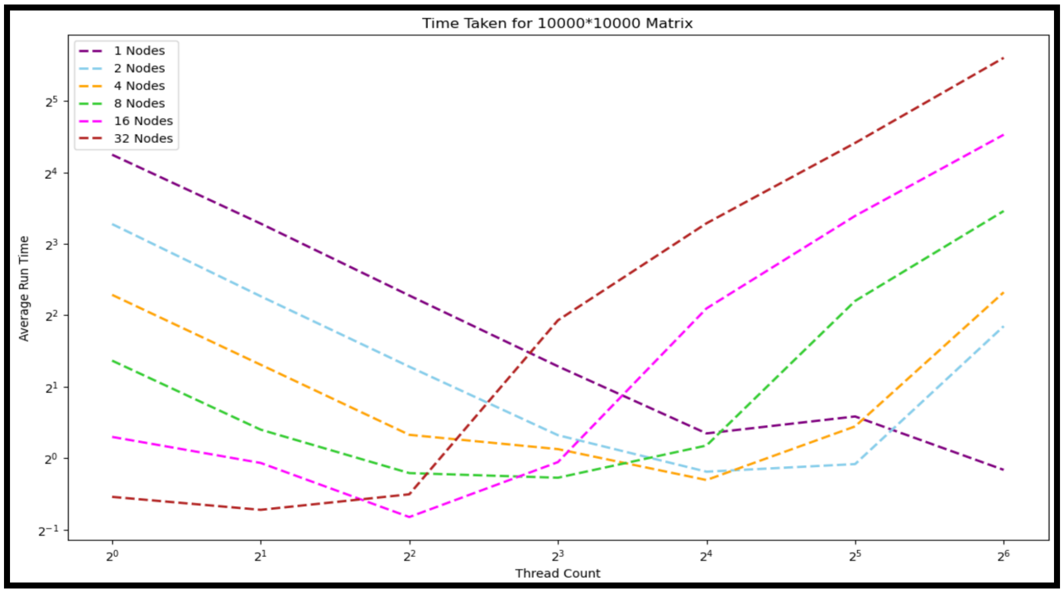
For 2 and 4 nodes, the runtime generally decreases as the number of threads increases up to 4 threads and 8 threads, respectively. This suggests that the benefits of parallelization are still being realized at these levels. However, beyond these points, the runtime increases again, likely due to the increasing overhead and communication costs between threads.

For 8 nodes, the runtime first decreases, then increases, and finally decreases again with an increasing number of threads. The minimum runtime is observed at 4 threads per processor. This unusual pattern might be attributed to specific characteristics of the underlying hardware or software environment, which could result in more efficient performance at particular combinations of nodes and threads.

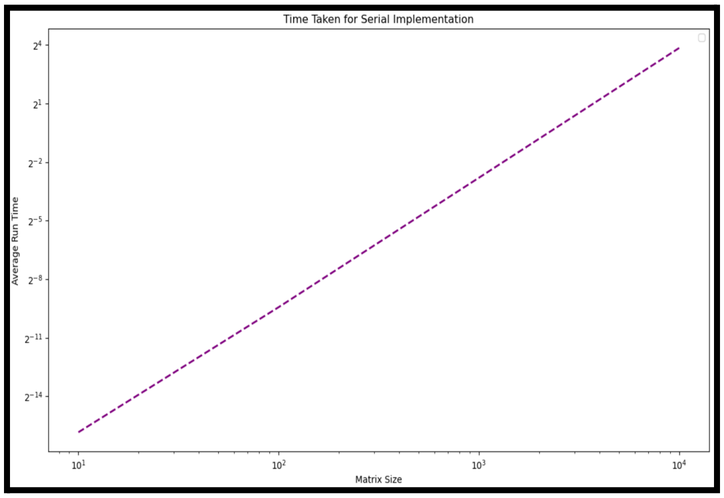
For 16 and 32 nodes, the runtime exhibits a more irregular pattern, with the minimum runtime achieved at lower thread counts (2 threads for 16 nodes and 1 thread for 32 nodes). As the number of nodes increases, the communication costs between ranks seem to have a more significant impact on the runtime, resulting in a higher sensitivity to the number of threads employed. For a better understanding of the plot you can refer to the below table which shows you the run time values at different node counts and thread counts.

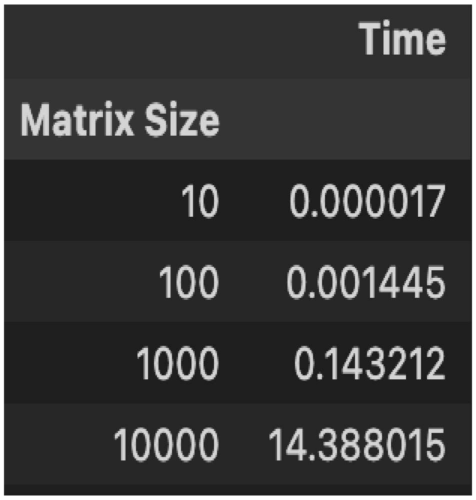


Node Count	Thread-1	Thread-2	Thread-4	Thread-8	Thread-16	Thread-32	Thread-64
1	19.021100	9.744089	4.847644	2.437643	1.270673	1.497898	0.891835
2	9.700080	4.812005	2.428872	1.248908	0.876171	0.943346	3.599231
4	4.878734	2.474994	1.253306	1.090887	0.808051	1.360099	5.001788
8	2.575878	1.317641	0.864346	0.826275	1.129642	4.597404	11.010480
16	1.228326	0.953790	0.564069	0.961488	4.275584	10.501190	23.099953
32	0.686182	0.605025	0.703504	3.809008	9.772752	21.317845	48.735648



For serial:





	Time
Matrix Size	
10	0.000017
100	0.001445
1000	0.143212
10000	14.388015

For a matrix size of 10x10, the run time is very short at 0.000017 seconds. This is expected since the problem size is small and can be solved quickly even without parallelization. As the matrix size increases to 100x100, the run time increases to 0.001445 seconds. The increase in run time is noticeable, but it is still relatively fast, as the problem size is still manageable for serial computation. When the matrix size grows to 1000x1000, the run time significantly increases to 0.143212 seconds. This indicates that as the problem size becomes larger, the computation time increases substantially. It is at this point where parallelization can become more beneficial in reducing the overall run time. For a matrix size of 10000x10000, the run time dramatically increases to 14.388015 seconds. The large problem size results in a significantly longer run time for serial computation, making parallelization an attractive option to enhance computational efficiency.

3.1. Scaling/performance studies

The performance of the parallel algorithm for different matrix sizes and varying combinations of node and thread counts. Based on the provided data, the following observations can be made:

For 10x10, As the number of threads increases, the runtime generally increases for a fixed number of nodes. This increase in runtime can be attributed to the overhead associated with thread creation, synchronization, and communication. For such a small problem size, the overhead of parallelization outweighs the potential performance benefits However, it is essential to note that even the minimum runtime for the parallelized algorithm is generally higher than the runtime of the serial algorithm for the 10x10 matrix size.

For smaller matrix sizes (e.g., 100x100), the run time generally decreases with an increase in the number of threads up to a certain point, after which the overhead associated with thread creation, synchronization, and communication outweighs the benefits of parallelization.

For larger matrix sizes (e.g., 1000x1000 and 10000x10000), the communication costs between ranks become more significant as the number of nodes increases. As a result, the optimal combination of nodes and threads becomes more sensitive to the matrix size, hardware, and software environment.

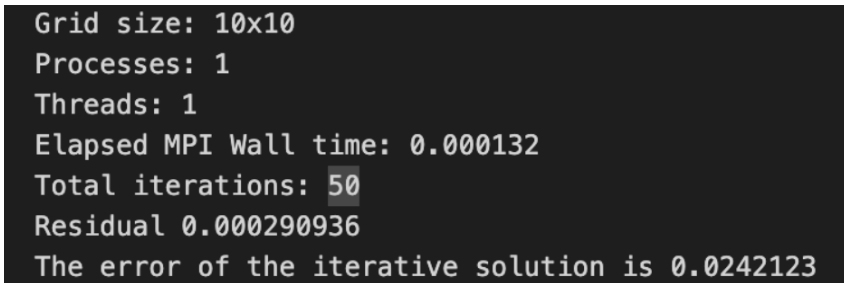
The optimal configuration of nodes and threads for a given matrix size depends on the problem size and the underlying hardware and software environment. The data suggests that as the number of nodes increases, the communication overhead between ranks can dominate the performance gains from parallelization.

As the matrix size increases, the run time for the serial algorithm grows significantly, suggesting that parallelization could offer substantial performance improvements for larger problems.

3.2. Verification test

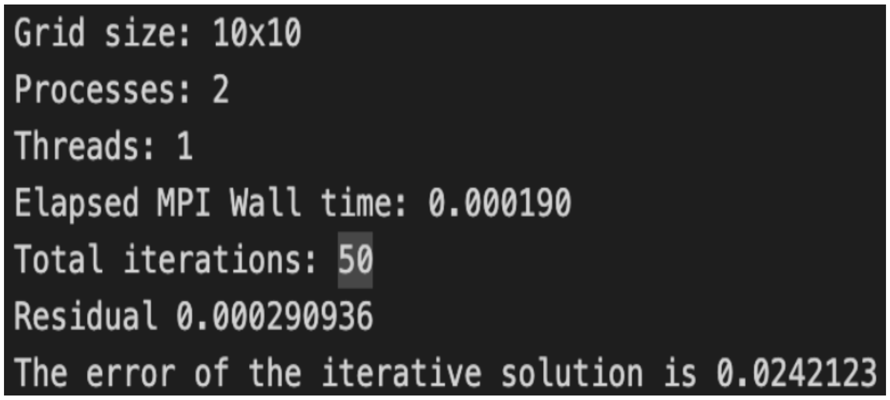
The verification test demonstrates that the method uses both serial and parallel programming for a small problem size (10x10 grid) using different Node-Thread Count configuration.

1. **For Nodes = 1 and Thread Count = 1**



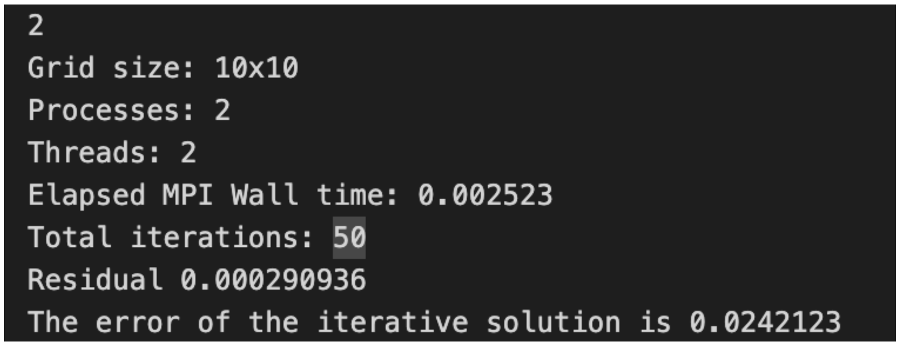
```
Grid size: 10x10
Processes: 1
Threads: 1
Elapsed MPI Wall time: 0.000132
Total iterations: 50
Residual 0.000290936
The error of the iterative solution is 0.0242123
```

2. **For Nodes = 2 and Thread count =1**



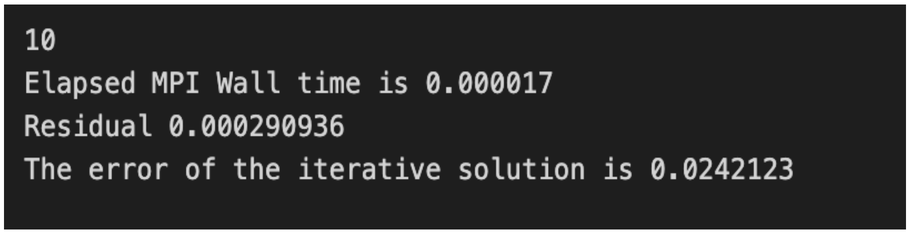
```
Grid size: 10x10
Processes: 2
Threads: 1
Elapsed MPI Wall time: 0.000190
Total iterations: 50
Residual 0.000290936
The error of the iterative solution is 0.0242123
```

3. **For Nodes = 2 and Threads = 2**



```
2
Grid size: 10x10
Processes: 2
Threads: 2
Elapsed MPI Wall time: 0.002523
Total iterations: 50
Residual 0.000290936
The error of the iterative solution is 0.0242123
```

4. **For serial setting**



```
10
Elapsed MPI Wall time is 0.000017
Residual 0.000290936
The error of the iterative solution is 0.0242123
```

Summary: From the above pictures, it is clear that for different combinations of Nodes and thread counts, the residual and error values are nearly the same. This verifies the fact that residual value is the same irrespective of the thread counts and node counts. We also compared our result with our serial implementation, which also had the same residual and error values, which made the verification testing concrete.

4. Conclusions

In conclusion, the primary goal of this project was to investigate the performance and scalability of the Poisson equation solver using the Successive Over-Relaxation (SOR) method, implemented with parallel processing. We employed various parallelization techniques, including MPI for inter-process communication and OpenMP for shared-memory parallelism. Our study aimed to understand the impact of these methods on performance and scalability across various problem sizes and system configurations.

Our analysis was conducted on four different problem sizes: 10x10, 100x100, 1000x1000, and 10000x10000 matrices. We observed that for small problem sizes, such as 10x10, the overhead associated with parallelization outweighed the potential performance benefits, leading to longer runtimes compared to the serial algorithm. However, as the problem size increased, parallelization became more attractive, with notable performance improvements for larger problem sizes.

The results demonstrated the importance of finding an optimal combination of nodes and threads for a given problem size to minimize runtime. As the number of nodes and threads increased, communication overhead and resource contention became significant factors, limiting the performance gains from parallelization. We observed a saturation point where increasing the number of threads further led to diminishing returns or even increased runtime.

In the broader context, our study highlights the need for careful consideration when employing parallelization techniques in numerical methods. It is crucial to understand the trade-offs between performance gains and overhead costs associated with parallelization, particularly for smaller problem sizes.

Future work in this area could include exploring other parallelization techniques, such as hybrid MPI and GPU-based parallelism, to further improve the performance of the Poisson equation solver. Additionally, research could be conducted on optimizing load balancing and communication patterns to minimize overhead and improve the scalability of the algorithm across various problem sizes and system configurations.

5. References

[1] Hinch, R. (n.d.). NST Part III Mathematical Tripos: Mathematical Methods II (MMII). Retrieved from https://www.damtp.cam.ac.uk/user/reh10/lectures/nst-mmii-chapter2.pdf
[2] Wikipedia contributors. (2021, September 9). Successive over-relaxation. In Wikipedia, The Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Successive_over-relaxation
[3] Gropp, W. (2016). Lecture 25: Domain Decomposition Methods. Retrieved from https://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture25.pdf
[4] Xu, Z. (2012). ACMS 40390/60390: Efficient Algorithms and Inexact Newton Methods. Retrieved from https://www3.nd.edu/~zxu2/acms40390F12/Lec-7.3.pdf
[5] Thomas, M. (2017). MPI Cartesian Communications and Topologies. Retrieved from https://edoras.sdsu.edu/~mthomas/sp17.605/lectures/MPI-Cart-Comms-and-Topos.pdf
[6] Open MPI. (n.d.). MPI_Barrier. Retrieved from https://www.open-mpi.org/doc/v3.0/man3/MPI_Barrier.3.php
[7] GeeksforGeeks. (2021). Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc(). Retrieved from https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/