Project 3 Report

Justin Burzachiello

1 Project Description

This project presents a parallel implementation of the Jacobi and Gauss-Seidel iteration algorithms for solving system of linear equations obtained through discretization of the domain involved in solving the elliptic PDE of the Poisson equation. In particular, the $\nabla^2\phi = f$ is discretized to $Ax = b$, where $A \in \mathbb{R}^{128\times128}$, $b \in \mathbb{R}^{128\times1}$, and $x \in \mathbb{R}^{128\times1}$. This matrix A is circulant since discretizing the Laplacian operator amounts to solving a convolution problem on a discretized mesh. Thus, this problem lends itself well to parallelization since convolution amounts to many local, independent subproblems that can be solved effectively through domain decomposition via row partitioning of the system of linear equations among P different processors. The exact structure of matrix A and vector b are shown in the following image. The implementation, written in C++, leverages MPI functions Allgatherv, Allreduce, and Broadcast to efficiently manage data distribution and collection across multiple processors. Additionally, the validity of the solvers is confirmed by testing them on a secondary diagonally dominant system, characterized by a 128×128 matrix with values of 1000 along the diagonal and values of 1 elsewhere, where the vector b of size 128×1 has values of 1 everywhere. This unit test verifies the results of the numerical solver. Finally, speedup curves obtained from using 1, 2, 4, and 16 processors are used to analyze performance gains through parallel Jacobi iteration and Gauss-Seidel iteration. These results highlight the performance improvements gained from parallelized domain decomposition, demonstrating the utility of parallel computing for efficiently solving convolution problems involving circulant matrices.

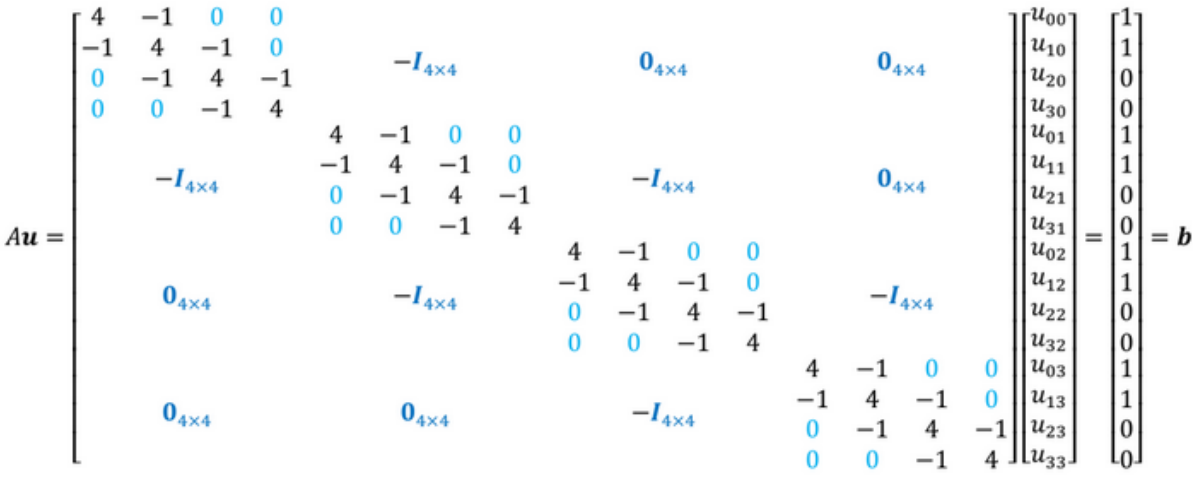

Figure 1: Au=b structure. In the rest of the text, u is called x.

2 Description of My Solution

Please access to following link to be able to view the exact details of the codebase used to complete this assignment: `https://github.com/jburz2001/ams_530/tree/main/project_3`.

2.1 Parallel Jacobi Algorithm for Solving $Ax = b$ with MPI

The following section describes the implementation of a parallel Jacobi algorithm using the Message Passing Interface (MPI) to solve a diagonally dominant system of linear equations $Ax = b$. The Jacobi method, known for its simplicity and suitability for parallelization, iteratively refines the solution vector until it converges within a specified error tolerance.

2.1.1 Problem Setup and Initialization

In this implementation, a matrix A of size 128×128 and a corresponding vector b are defined. The matrix A is set up as a Laplacian-like matrix, representing a discretized 2D grid with each cell connected to its adjacent cells. Specifically, each diagonal entry A_{ii} is set to 4, while entries corresponding to adjacent cells (left, right, top, and bottom) are set to -1, creating a sparse structure favorable to iterative methods. This structure makes A diagonally dominant, which improves the stability and convergence of the Jacobi method.

The vector b is initialized in a repeating pattern ‘1, 1, 0, 0‘, simulating a structured right-hand side for the system.

2.1.2 Parallel Distribution with MPI

The code utilizes MPI to distribute the computation across multiple processors. Each processor:

- Initializes MPI and obtains its unique rank and the total number of processors.
- Verifies that the number of processors evenly divides the matrix size, ensuring equal distribution of rows across processors.
- Assigns each processor a block of rows of matrix A (determined by `rows_per_proc`) based on its rank. Each processor is responsible for updating the solution vector x for its specific rows.

Matrix A and vector b are created by the root processor (rank 0) and distributed to all other processors using `MPI_Bcast`. This ensures that each processor has a complete copy of A and b, allowing them to compute independently.

2.1.3 Jacobi Iteration

The Jacobi iteration updates each entry of the solution vector x based on the formula:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} A_{ij} x_j^{(k)}}{A_{ii}}$$

where $x_i^{(k+1)}$ is the new value of x at the current iteration. Each processor performs the following steps during each iteration:

1. **Local Calculation**: Each processor computes the new values of x for its assigned rows. For each row, it calculates the sum of $A[i, j] \times x[j]$ (excluding the diagonal entry A_{ii}) using values from the previous iteration. The new value for x is computed based on this sum and the diagonal element, ensuring no division by zero occurs.
2. **Global Synchronization**: After calculating the new values, the updated solution vector parts from each processor are gathered into a global solution vector using `MPI_Allgather`. This process synchronizes the x values across all processors before the next iteration.

3. **Error Checking for Convergence**: Each processor calculates its local error, defined as the maximum absolute change in x for the rows it owns. A global error is determined by taking the maximum of local errors across all processors using `MPI_Allreduce`. This global error serves as the convergence criterion, and the iteration continues until the error drops below the predefined tolerance or the maximum number of iterations is reached. Additionally, the infinity norm of the residual $r = Ax - b$ is calculated for analysis and stored during each iteration.

2.1.4 Timing and Output

The algorithm tracks the elapsed time on each processor to evaluate performance:

- **Elapsed Time Measurement**: The time taken by each processor is measured, and both total and maximum times are computed using `MPI_Allreduce` for reporting.
- **Output**: After convergence or reaching the maximum iteration count, the root processor (rank 0) outputs:
  - Total elapsed time across all processors and maximum time for any single processor.
  - Approximate solution vector.
  - Residual infinity norm values for each iteration, allowing analysis of convergence behavior.

2.1.5 Summary

This parallel Jacobi implementation demonstrates how MPI enables efficient distribution of work for large linear systems by:

- Distributing the workload evenly across processors to reduce individual computational load.
- Synchronizing updates to ensure consistency across iterations.
- Using the Jacobi method's suitability for parallelization, where each row of the solution vector can be computed independently based on values from the previous iteration.

The result is a scalable and efficient solution to solving large, sparse systems, which are common in scientific and engineering applications.

2.2 Parallel Gauss-Seidel Algorithm for Solving $Ax = b$ with MPI

This section describes the implementation of a parallel Gauss-Seidel algorithm using MPI for solving a diagonally dominant system of linear equations $Ax = b$. The Gauss-Seidel method, which improves upon Jacobi iteration by updating solution values in-place as they are computed, is inherently more challenging to parallelize. This implementation achieves parallelization by using a red-black ordering scheme, where updates alternate between "red" and "black" points, allowing synchronization points in between.

2.2.1 Problem Setup and Initialization

In this implementation, the matrix A has dimensions 128×128, corresponding to a discretized 2D grid, with each entry representing a grid point and connected to its adjacent points. The diagonal elements A_{ii} are set to 4, while elements corresponding to neighboring points (left, right, top, and bottom) are set to -1, creating a sparse and diagonally dominant matrix structure. This setup ensures stability in the Gauss-Seidel iterations.

The vector b is initialized in a repeating pattern '1, 1, 0, 0', which provides a structured right-hand side for the system.

2.2.2 Parallel Distribution with MPI

MPI is used to distribute the matrix A and vector b across multiple processors. Each processor:

- Initializes MPI and retrieves its unique rank and the total number of processors.
- Verifies that the matrix size n is evenly divisible by the number of processors, ensuring an equal number of rows per processor.
- Determines the specific rows assigned to it, based on rank, which each processor will update during iterations.

Matrix A and vector b are created by the root processor (rank 0) and distributed to all other processors using `MPI_Bcast`. This ensures that all processors have access to a complete copy of A and b, allowing local computations.

2.2.3 Red-Black Gauss-Seidel Iteration

In this Gauss-Seidel implementation, a red-black ordering scheme is used to alternate updates between two sets of grid points:

- **Red Points**: Each point in the matrix A is classified as "red" or "black" based on a checkerboard pattern. The red points are updated first in each iteration, using values from the previous iteration for black points.
- **Black Points**: After updating all red points, the black points are updated, using the newly computed red values.

Each iteration proceeds as follows:

1. **Local Calculation for Red Points**: Each processor computes the new values of x for its assigned rows that correspond to red points. For each row, the sum of the off-diagonal elements is calculated, and the new value is computed as:

$$x_i = \frac{b_i - \sum_{j \neq i} A_{ij} x_j}{A_{ii}}$$

   The solution is updated in-place for each red point, with a check to track the maximum change (error) locally.

2. **Synchronization for Red Points**: After updating the red points, `MPI_Allgather` is used to synchronize the updated values across all processors, ensuring that the global solution vector x reflects the red updates before moving to black points.

3. **Local Calculation for Black Points**: Following synchronization, each processor computes and updates the solution for its black points in a similar manner to red points, using the latest values from the red update.

4. **Synchronization for Black Points**: Another `MPI_Allgather` operation synchronizes the black point updates across all processors, ensuring consistent values in x for the next iteration.

2.2.4 Error Calculation and Convergence Check

Each processor calculates its maximum local error, defined as the maximum absolute difference between values in consecutive iterations for its rows. A global error is then determined by taking the maximum of the local errors across all processors using `MPI_Allreduce`. This global error is used to check convergence, with iterations continuing until either the error is below a predefined tolerance (`tolerance`) or the maximum number of iterations is reached.

Additionally, the infinity norm of the residual $r = Ax - b$ is computed at each iteration, storing the values in a vector for analysis. This norm provides a measure of accuracy over time.

2.2.5 Timing and Output

The algorithm tracks the elapsed time on each processor:

- **Elapsed Time Measurement**: Each processor measures its elapsed time for performance analysis, with both the total time and the maximum time recorded using `MPI_Allreduce`.
- **Output**: After convergence or reaching the maximum iteration count, the root processor (rank 0) outputs:
  - The total and maximum elapsed time across all processors.
  - The approximate solution vector.
  - The infinity norm of the residual for each iteration, giving insights into convergence behavior.

2.2.6 Summary

This parallel Gauss-Seidel implementation demonstrates how MPI can enable effective parallelization of iterative methods:

- The red-black ordering allows synchronization points between updates, facilitating parallel updates in the Gauss-Seidel method.
- MPI-based communication ensures consistent data across processors, enabling iterative refinement of the solution.
- The structured nature of the Laplacian matrix A makes it suitable for this approach, and the algorithm demonstrates scalability for larger systems.

The result is a robust and efficient solution for solving large, sparse systems of linear equations, with broad applications in scientific computing.

3 Results

After implementing my custom parallel Jacobi and Gauss-Seidel algorithms, I performed experiments to test their ability to solve the discretized Poisson equation and determine how long it takes. The following figure shows the speedup plot for solving a 128x128 Laplacian with a forcing vector described in the Project Description section. The later subsections of this report show the numerical solution obtained, along with some statistics regarding convergence.

The following figure shows the speedup from parallelization of the respective Jacobi and Gauss-Seidel algorithms. This shows that both can be sped up via parallelization but that using more processors eventually has diminishing returns due to communication costs incurred through allgather and allreduce. I anticipated Jacobi having superior parallelization, which leads me to suspect that there is a minor error in how I compute timing results.

Speedup for solving Poisson equation with Jacobi and Gauss-Seidel

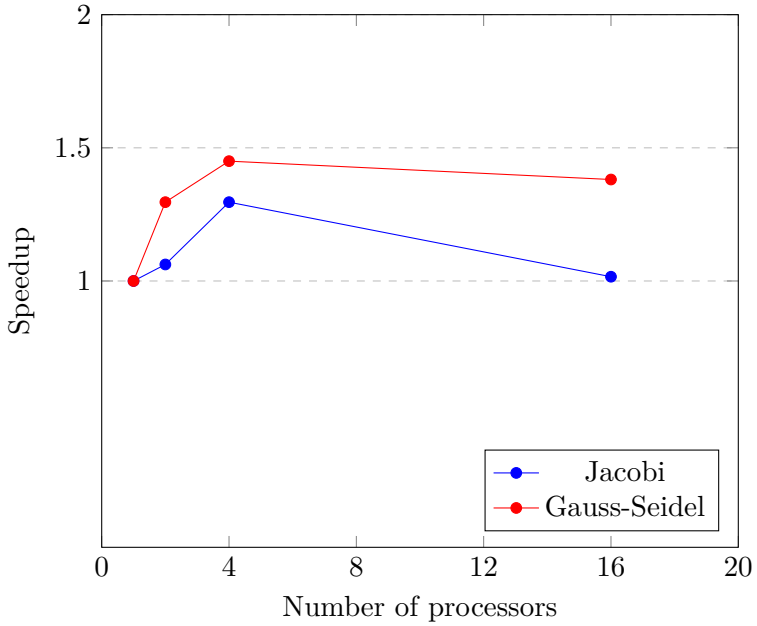

Figure 2: Speedup

3.1 Jacobi

This section shows the results obtained from my Jacobi algorithm. These figures convey timing results along with results of the numerical solution vector and error convergence.

```
[jburzachiell@login2 laplacian_128x128]$ cat p3_j_{1,2,4,16}.out
JACOBI
Elapsed time (root node) with 1 processors:
0.0228879

JACOBI
Elapsed time (root node) with 2 processors:
0.021549

JACOBI
Elapsed time (root node) with 4 processors:
0.0176547

JACOBI
Elapsed time (root node) with 16 processors:
0.0225182
```

Figure 3: Timing results.

```
[jburzachiell@login2 laplacian_128x128]$ cat p3_j_1.out
JACOBI
Elapsed time (root node) with 1 processors:
0.0229042
Approximate final solution vector:
0.8545 1.342 1.383 1.507 1.865 1.966 1.68 1.504 1.563 1.339 0.6698 1.076 2.
131 2.684 2.78 2.986 3.32 3.25 2.773 2.408 2.124 1.34 1.317 2.423 3.441 3.9
43 3.981 4.076 4.228 3.931 3.172 2.411 1.565 1.77 2.802 3.713 4.57 4.917 4.
777 4.656 4.549 3.938 2.781 1.509 1.96 3.303 4.04 4.708 5.34 5.457 5.07 4.6
73 4.249 3.268 1.69 1.766 3.409 4.437 4.882 5.277 5.643 5.492 4.823 4.119 3
.351 1.983 1.694 3.131 4.416 5.107 5.242 5.346 5.434 5.007 4.053 3.035 1.88
9 1.879 3.006 3.989 4.887 5.237 5.066 4.89 4.719 4.049 2.847 1.539 1.815 3.
026 3.648 4.214 4.754 4.791 4.342 3.928 3.579 2.765 1.42 1.354 2.633 3.362
3.568 3.775 4.002 3.758 3.073 2.574 2.212 1.377 0.9698 1.791 2.6 2.921 2.77
4 2.685 2.616 2.031 1.434 1.131 0.877 0.7331 0.9626 1.326 1.741 1.716 1.348
 0.9911

Residual infinity norm for iterations:
-0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.4985 -0.496 -0.4926 -0.4873 -0.4816 -0.474
1 -0.4663 -0.4572 -0.448 -0.4379 -0.4277 -0.4171 -0.4064 -0.3956 -0.3847 -0
.374 -0.3632 -0.3528 -0.3422 -0.3321 -0.3219 -0.3123 -0.3025 -0.2933 -0.284
 -0.2752 -0.2664 -0.2582 -0.2498 -0.242 -0.2341 -0.2268 -0.2194 -0.2125 -0.
2055 -0.199 -0.1925 -0.1864 -0.1802 -0.1745 -0.1688 -0.1634 -0.158 -0.153 -
0.148 -0.1433 -0.1385 -0.1341 -0.1297 -0.1256 -0.1214 -0.1175 -0.1136 -0.11
 -0.1064 -0.103 -0.09959 -0.09642 -0.09323 -0.09026 -0.08727 -0.08449 -0.08
169 -0.07909 -0.07647 -0.07404 -0.07159 -0.06931 -0.06701 -0.06488 -0.06273
```

Figure 4: Solution vector and error convergence.

3.2 Gauss-Seidel

This section shows the results obtained from my Gauss-Seidel algorithm. These figures convey timing results along with results of the numerical solution vector and error convergence. Note that the final iterate of the solution vector for Gauss-Seidel is approximately equal to the final iterate of the solution vector obtained through Jacobi iteration shown in the previous subsection. Additionally, my results indicate that my parallel Jacobi converges faster than my parallel Gauss-Seidel algorithm.

```
[jburzachiell@login2 laplacian_128x128]$ cat p3_gs_{1,2,4,16}.out
GAUSS-SEIDEL
Elapsed time (root node) with 1 processors:
0.0262904

GAUSS-SEIDEL
Elapsed time (root node) with 2 processors:
0.0202825

GAUSS-SEIDEL
Elapsed time (root node) with 4 processors:
0.0181198

GAUSS-SEIDEL
Elapsed time (root node) with 16 processors:
0.0190306
```

Figure 5: Timing results.

```
[jburzachiell@login2 laplacian_128x128]$ cat p3_gs_1.out
GAUSS-SEIDEL
Elapsed time (root node) with 1 processors:
0.0263193
Approximate final solution vector:
0.8545 1.342 1.383 1.507 1.865 1.966 1.68 1.504 1.563 1.339 0.6698 1.076 2.
131 2.684 2.78 2.986 3.32 3.25 2.773 2.408 2.124 1.34 1.317 2.423 3.441 3.9
43 3.981 4.076 4.228 3.931 3.172 2.411 1.565 1.77 2.802 3.713 4.57 4.917 4.
777 4.656 4.549 3.938 2.781 1.509 1.96 3.303 4.04 4.708 5.34 5.457 5.07 4.6
73 4.249 3.268 1.69 1.766 3.409 4.437 4.882 5.277 5.643 5.492 4.823 4.119 3
.351 1.983 1.694 3.131 4.416 5.107 5.242 5.346 5.434 5.007 4.053 3.035 1.88
9 1.879 3.006 3.989 4.887 5.237 5.066 4.89 4.719 4.049 2.847 1.539 1.815 3.
026 3.648 4.214 4.754 4.791 4.342 3.928 3.579 2.765 1.42 1.354 2.633 3.362
3.568 3.775 4.002 3.759 3.073 2.574 2.212 1.377 0.9698 1.791 2.6 2.921 2.77
4 2.685 2.616 2.031 1.434 1.131 0.877 0.7331 0.9626 1.326 1.741 1.716 1.348
 0.9911

Residual infinity norm for iterations:
-0.8125 -0.7695 -0.7394 -0.7296 -0.7173 -0.7039 -0.6892 -0.6737 -0.6552 -0.
6349 -0.6134 -0.5912 -0.5686 -0.546 -0.5236 -0.5016 -0.4801 -0.4592 -0.4389
 -0.4193 -0.4004 -0.3823 -0.3649 -0.3482 -0.3322 -0.3169 -0.3022 -0.2882 -0
.2748 -0.2621 -0.2499 -0.2382 -0.2271 -0.2166 -0.2065 -0.1968 -0.1876 -0.17
89 -0.1705 -0.1625 -0.1549 -0.1477 -0.1408 -0.1342 -0.1279 -0.122 -0.1163 -
0.1108 -0.1056 -0.1007 -0.09598 -0.09149 -0.08721 -0.08314 -0.07925 -0.0755
4 -0.07201 -0.06864 -0.06543 -0.06237 -0.05945 -0.05667 -0.05402 -0.05149 -
0.04908 -0.04679 -0.0446 -0.04251 -0.04052 -0.03863 -0.03682 -0.0351 -0.033
46 -0.03189 -0.0304 -0.02898 -0.02762 -0.02633 -0.0251 -0.02393 -0.02281 -0
.02174 -0.02072 -0.01975 -0.01883 -0.01795 -0.01711 -0.01631 -0.01555 -0.01
482 -0.01413 -0.01346 -0.01284 -0.01223 -0.01166 -0.01112 -0.0106 -0.0101 -
0.009629 -0.009178 -0.008749 -0.00834 -0.00795 -0.007578 -0.007223 -0.00688
```

Figure 6: Solution vector and error convergence.

4 Testing

To verify the results of my custom algorithms, I solved A x = b with non-Laplacian diagonally-dominant matrices. The first was a 4×4 matrix of with 10s along the diagonal and 1s elsewhere and the second was a 8192×8192 matrix with $10,000$s along the diagonal and 1s elsewhere. I did not perform this test with Gauss-Seidel since speedup is conveyed in the results section on Gauss-Seidel, along with proof of the validity of its solution.

The solution to the 4x4 system is shown in Wolfram Alpha.

Results of testing are shown in the following figures.

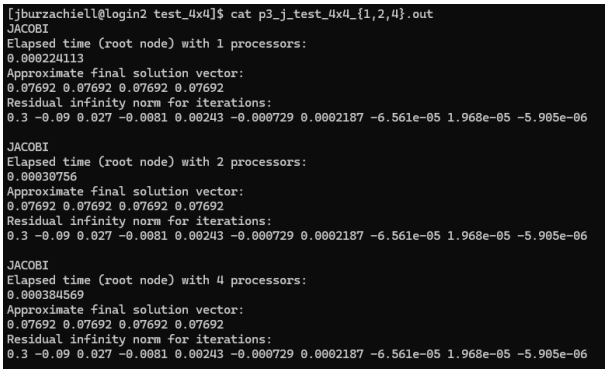

Figure 7: All results for 4x4.

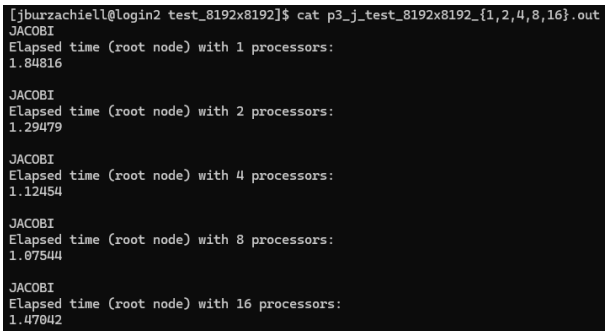

Figure 8: Timing results for 8192x8192.

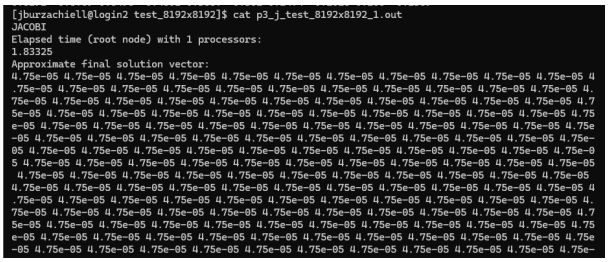

Figure 9: A snippet of the solution vector. This contains a subset of the 8192 elements of x, but they are all equal.

Additionally, here is a plot of the speedups. Importantly, this shows that parallelization slows down the computation when performed on a 4x4 system but can greatly speed it up on a 8192x8192 system. Thus, one must not naively believe that "parallelized" = "faster".

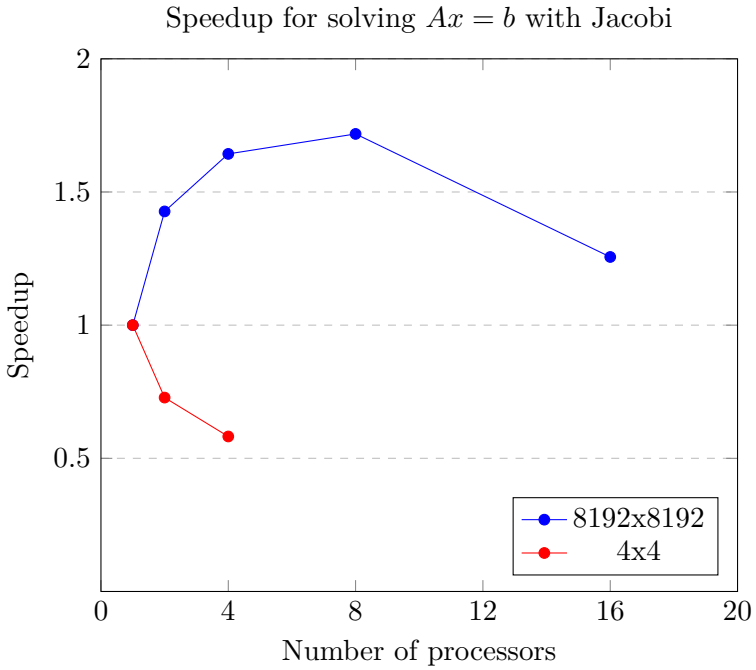

Figure 10: Speedup