Racing to Convergence

Comparing Parallel Speedup of Damped Jacobi and Gauss-Seidel Methods

Dennys Huber

March 27, 2024

Contents

1 Introduction **3**

2 Iterative Methods for Solving Linear Systems **4**

2.1 Damped Jacobi 4

2.2 Gauss-Seidel 5

2.3 Convergence 5

3 Model Problem **6**

3.1 Five-Point Formula 6

3.2 L2-Projection 9

4 Implementation Details **10**

4.1 Load Balancing 10

4.2 Hardware and Operating System limitations 10

4.3 Iterative Methods 11

4.4 Experiment remarks 12

5 Speedup **13**

5.1 Five-Point 13

5.2 L2-Projection 13

6 Timing **16**

6.1 Five-Point 16

6.2 L2-Projection 16

7 Efficiency **21**

7.1 Five-Point 21

7.2 L2-Projection 21

8 Conclusion **24**

References **25**

1 Introduction

Parallel computing is essential in solving mathematical problems which demand high computational resources, such as optimization problems, large-scale linear systems, or partial differential equations. Putting in the extra effort to parallelize your algorithm can cut down execution time by a large factor and increase an algorithm's efficiency. Even though parallelization should increase the performance of every algorithm, not every algorithm scales the same. This can lead to situations where a faster and more efficient serial implementation, can suddenly become slower than another usually slower algorithm when implemented in parallel.
This exact situation appears with the Gauss-Seidel and Damped Jacobi methods, which are used to approximate solutions for linear systems iteratively. Gauss-Seidel only needs roughly half as many iterations to converge compared to Damped-Jacobi, but due to a calculation step, which cannot be efficiently implemented in parallel, Damped Jacobi becomes the preferred choice for large-scale linear systems.
This study aims to analyze and compare the speedup, timing, and efficiency of the two iterative methods with varying numbers of unknowns and parallel processes.

2 Iterative Methods for Solving Linear Systems

Let $A \in \mathbb{R}^{n\times n}$ be a symmetric positive definite matrix with elements (a_{ij}), $f \in \mathbb{R}^n$ a column vector and $x^{(0)} \in \mathbb{R}^n$ an initial guess for the solution of the linear system $Ax = f$.
An iterative solution of the system is obtained by computing a sequence of iterates $x^{(m)}$ for $m = 1, 2, \ldots$.

Both Damped Jacobi and Gauss-Seidel are linear and consistent iterative methods, therefore they can be expressed by the Second Normal Form

$$x^{(m+1)} = x^{(m)} - Mr^{(m)}, \tag{1}$$

where $r \in \mathbb{R}^n$ is the *residual* and corresponce to the difference between the optimal solution $Ax^* = f$ and the current approximated solution. For solving linear systems, the residual is given by

$$r^{(m)} = \left(Ax^{(m)} - f\right). \tag{2}$$

The regular matrix $M \in \mathbb{R}^{n\times n}$ updates the current approximated solution and is constructed by the coefficient of A, which is often split up into the following parts

$$A = D + L + U. \tag{3}$$

Here D is a diagonal matrix containing the main diagonal entries of A, L is the strictly lower triangular, and U is the strictly upper triangular matrix of A. Gauss-Seidel and Damped Jacobi only differ in the construction of matrix M.

To determine when the iteration process should halt, for a stopping threshold $\epsilon > 0$, we introduce the stopping criteria

$$\frac{\|r^{(m)}\|}{\|x^{(m)}\|} < \epsilon. \tag{4}$$

If the approximated solution approaches the optimal solution x^*, $r^{(m)}$ will approach 0 and so will the fraction in (4).

2.1 Damped Jacobi

For the Damped Jacobi, M_{DJ} is constructed as follows

$$M_{DJ} = \omega D^{-1} \tag{5}$$

the damping parameter ω helps control the system's convergence rate and stability. Inserting equation (5) and (2) into (1) results in the matrix-based formula

$$x^{(m+1)} = x^{(m)} - \omega D^{-1}\left(Ax^{(m)} - f\right), \tag{6}$$

which can also be rewritten in an element-based formula

$$x_i^{(m+1)} = x_i^{(m)} - \frac{\omega}{a_{ii}} \left[\sum_{j=1}^{n} a_{ij} x_j^{(m)} - f_i \right]. \tag{7}$$

2.2 Gauss-Seidel

The matrix M_{GS} for Gauss-Seidel is defined by

$$M_{GS} = (D + L)^{-1} \tag{8}$$

and results in the matrix-based formula

$$x^{(m+1)} = x^{(m)} - (D + L)^{-1} \left(Ax^{(m)} - f \right). \tag{9}$$

Inverting matrices is computationally expensive, one can therefore define $y^{(m)} = M_{GS} r^{(m)}$ and can be rewritten as a linear system such that

$$r^{(m)} = M_{GS}^{-1} y^{(m)}. \tag{10}$$

Due to M_{GS}^{-1} being a lower triangular matrix, the linear system can be solved for $y^{(m)}$ by forward substitution. Hence $x^{(m+1)}$ can then be approximated by

$$x^{(m+1)} = x^{(m)} - y^{(m)}. \tag{11}$$

We can write Gauss-Seidel in an element-based formula similar to the Damped Jacobi

$$x_i^{(m+1)} := x_i^{(m)} - \frac{1}{a_{ii}} \left[\sum_{j=1}^{i-1} a_{ij} x_j^{(m+1)} + \sum_{j=i}^{n} a_{ij} x_j^{(m)} - f_i \right]. \tag{12}$$

In the Gauss-Seidel method unlike the Jacobi Method, the previous estimations are instantly replaced by the new values. Thus the Gauss-Seidel Method needs less iteration to converge, more precisely it can be shown that Damped Jacobi with an optimal ω needs approximately twice as many iterations.

2.3 Convergence

Understanding convergence for iterative methods is important to determine if, how quickly, and how accurately a method will converge to a desired solution. Having a good understanding of how a problem converges can save time and important resources.

For a given linear system $Ax^{(m)} = f$ with the optimal solution x^*, the iteration error $e^{(m+1)}$ of $x^{(m+1)}$ is given by

$$e^{(m+1)} := x^{(m+1)} - x^* \tag{13}$$

This is equivalent to

$$\begin{aligned} e^{(m+1)} &= x^{(m)} - x^* - M^{-1}\left(Ax^{(m)} - Ax^*\right) \\ &= (I - M^{-1}A)(x^{(m)} - x^*) \end{aligned} \tag{14}$$

Define $C = I - M^{-1}A$, hence

$$e^{(m+1)} = Ce^{(m)} \tag{15}$$

A linear iteration converges when the convergence rate ρ, which is defined by the spectral radius is

$$\rho(C) < 1. \tag{16}$$

In order to describe linear convergence the following inequality is used

$$\|e^{(m+1)}\| \leq \|C\|\|e^{(m)}\|, \quad \|C\| < 1. \tag{17}$$

where $\|C\|$ is called the contraction number and it describes how much the error is reduced after each iteration. If the contraction number is small this indicates faster convergence, which means the error decreases more rapidly. To predict the rate of convergence, the condition number κ is used, which is given by

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}} \tag{18}$$

where $\lambda_{\max}$ refers to the biggest eigenvalue of A and $\lambda_{\min}$ to the smallest. The condition number measures how sensitive the problem reacts to small changes to the input. The convergence rate for the linear iterative solvers used in this study is given by

$$\|C\| \leq \frac{\kappa(A) - 1}{\kappa(A) + 1} = \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} \tag{19}$$

3 Model Problem

In order to measure and compare the performance of Gauss-Seidel and Damped Jacobi an appropriate model problem needs to be selected, which is easily scalable and guarantees convergences. The decision fell on approximating the partial differential equation with the finite difference method.

3.1 Five-Point Formula

A differential equation is defined on the unit square $\Omega = (0,1) \times (0,1)$ as

$$-\Delta u(x,y) = f(x,y) \quad \text{for } (x,y) \in \Omega, \tag{20}$$

$$u(x,y) = \varphi(x,y) \quad \text{on } \Gamma = \partial\Omega \tag{21}$$

where the source function f, is known as well as the boundary values φ. The function u on the other hand is unknown. In equation (20) the operator Δ is the two-dimensional Laplace operator given by $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. In order to discretize the differential equation, we create a grid Ω_h of size $N \times N$ with step size $h = 1/N$.

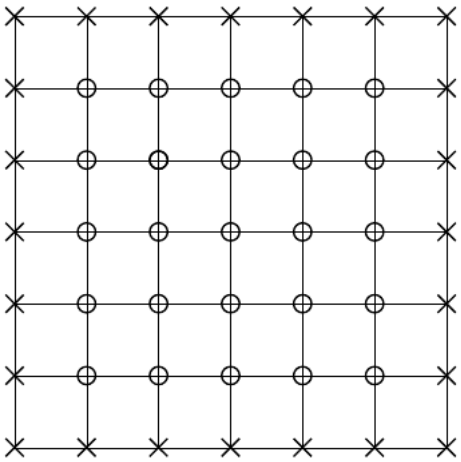


Figure 1: Grid Ω_h with inner grid points ($\circ$) and boundary points ($\times$) [2]

The inner grid points are represented by

$$\Omega_h := \{(x, y) = (ih, jh) : 0 < i, j < N\}, \text{ where } h = 1/N, \tag{22}$$

outer grid points also called boundary points are for this experiment set to 0. In order to simplify the notation we can also refer to a specific point in the grid by $u(x, y) = u(ih, jh)$ with $u_{i,j}$ and similarly for the function $f_{i,j} := f(ih, jh)$
The negative two-dimensional Laplace operator can now be approximated by the *five-point formula* as follows in equation (20)

$$\begin{aligned} -\Delta u(ih, jh) &= \frac{-u_{i-1,j} + 2u_{i,j} - u_{i+1,j}}{h^2} + \frac{-u_{i,j-1} + 2u_{i,j} - u_{i,j+1}}{h^2} \\ &= \frac{1}{h^2}[4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}] \\ &= f_{ij} \end{aligned} \tag{23}$$

The number of unknowns is given by the number of inner grid points, which is $n := (N - 1)^2$
To rewrite the equation in the desired matrix form $Ax = f$, one needs to transform the unknown grid points into a one-dimensional vector of size n. There are several different ordering that can achieve this. The lexicographical ordering was used in this study to enumerate the grid points as illustrated in figure 2.

	21	22	23	24	25
	16	17	18	19	20
	11	12	13	14	15
	6	7	8	9	10
	1	2	3	4	5

Figure 2: Lexicographical ordering [2]

x has to be interpreted as

$$\begin{aligned} x_k &= u_{ij} = u(ij, jh) \\ k &= i + (j-1)(N-1) \end{aligned} \tag{24}$$

where $0 < i, j < N$ and x can be decomposed into blocks of $N-1$, where j is corresponding to the row in the grid. x^j is ordered as follows

$$x^j := \begin{bmatrix} x_{k+1} \\ \vdots \\ x_{k+N-1} \end{bmatrix} = \begin{bmatrix} u_{1,j} \\ \vdots \\ u_{N-1,j} \end{bmatrix} \quad \begin{aligned} &\text{with } k := (j-1)(N-1) \\ &\text{for } j = 1, \ldots, N-1, \end{aligned} \tag{25}$$

The Matrix A then becomes a block-tridiagonal matrix with each matrix T and the identity matrix being of size $(N-1) \times (N-1)$

$$A = h^{-2} \begin{bmatrix} T & -I & & & \\ -I & T & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & T & -I \\ & & & -I & T \end{bmatrix}, \quad T = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix} \tag{26}$$

Concretely the following functions were chosen for this study

$$\begin{aligned} u(x, y) &= x(1-x)y(1-y) \\ f(x, y) &= 2(x(1-x) + y(1-y)) \end{aligned} \tag{27}$$

with damping parameter $\omega = 1$, condition number

$$\kappa(A) = Ch^{-2} \tag{28}$$

where C is a constant. Therefore the iteration needed to converge increases with the step size h and thus also with the grid size N.

3.2 L2-Projection

The L2 Projection is a method used to approximate a given function $u \in L_2(\Omega)$ onto a finite subspace $V_h \subset L_2(\Omega)$. This is done by the following optimization function

$$\Pi(v_h) := \frac{1}{2} \|v_h - u\|^2_{L^2(\Omega)} \quad \rightarrow \quad \min \tag{29}$$

which tries to minimize the L2-Norm of the error between the function u and the approximation $v_h \in V_h$.
While solving this problem a linear system $Mx = f$ is derived, which can be solved with the finite difference method identically to the Five-Point, but the derivation of the actual matrix M and how the L2 Projection optimization problem is solved was not part of this study, and Professor Sauter and his team kindly provided the solution for the mass matrix M_{L2}. This mass matrix $M_{L2} \in \mathbb{R}^{n \times n}$ posses the same dimensionality as Matrix A (23) and looks as follows

$$M_{L2} = \begin{bmatrix} S & L & & & \\ L & S & L & & \\ & \ddots & \ddots & \ddots & \\ & & L & S & L \\ & & & L & S \end{bmatrix} \tag{30}$$

with L, S being of size $(N-1) \times (N-1)$

$$S = \begin{bmatrix} \frac{4}{9} & \frac{1}{9} & & & \\ \frac{1}{9} & \frac{4}{9} & \frac{1}{9} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{1}{9} & \frac{4}{9} & \frac{1}{9} \\ & & & \frac{1}{9} & \frac{4}{9} \end{bmatrix}, \quad L = \begin{bmatrix} \frac{1}{9} & \frac{1}{36} & & & \\ \frac{1}{36} & \frac{1}{9} & \frac{1}{36} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{1}{36} & \frac{1}{9} & \frac{1}{36} \\ & & & \frac{1}{36} & \frac{1}{9} \end{bmatrix} \tag{31}$$

For this experiment, the following functions were used

$$\begin{aligned} u(x,y) &= (1-x)^2(1-y)^2 \\ \Delta u(x,y) &= f(x,y) = 2((1-x)^2 + (1-y)^2) \end{aligned} \tag{32}$$

with the boundaries set to 0 and the right-hand side of the linear system being $4h^2 f(ih, jh)$.Important to add here is that the positive Laplace Operator was used. For the Damped Jacobi the damping parameter $\omega = \frac{2}{3}$ was used and the condition number is

$$\kappa(M_{L2}) = C, \quad C > 1 \tag{33}$$

This leads to a constant number of iterations per iterative method regardless of the grid size N or step size h.

4 Implementation Details

The premise for this experiment was to implement the iterative methods in a low-level programming language to have as much control as possible over the parallel implementation. For this reason, the implementation was done in C/C++, and besides the standard data structures, only Chrono, a library for time measuring, and MPI for parallelization were used. The Messaging Protocol Interface (MPI) [4] library was chosen because it gives the developer the most control over the parallelization. With MPI, the programmer controls what each process, which in this project was one per core, is doing and what data each process has access to.

4.1 Load Balancing

This additional control came with the cost of additionally implementing how the data was split and distributed between processes. The goal is that every process receives the same number of rows of A, the vector b, and $x^{(m)}$. This turned out to be complicated with the distribution of submatrices of A to the processes if the number of rows of A was not divisible by the number of processes without rest. This was because MPI's implemented Scatter and Gather functions can only send chunks of data with the same size to different processes. Technically it would have been possible to implement such Scatter and Gather functions, but, as it turns out, these are slower than the already provided and optimized built-in functions of MPI. As a solution, the matrix size was increased with rows and columns containing only zeros up to the following number of rows, which was divisible without rest. Additionally, the number of rows per submatrix was increased by one. If a process encountered a zero row or column, the individual process skipped the calculations and waited until the other processes were done. This distribution of data is not optimal and leads to inefficiency, such as having unused cores and sending data to each process that is not used.

4.2 Hardware and Operating System limitations

The whole experiment was carried out on the *rambo* server [1], kindly provided by IMATH. To my knowledge, the server was not solely used for this study but also by other researchers. If they ran any programs, this could have potentially influenced the results of my experiment and could explain certain fluctuations in execution time.

Furthermore, the server has 64 CPU cores, allowing me to run up to 64 parallel processes. In MPI, each core has its own independent memory, which is limited in size. This led to segmentation faults when having a low number of parallel processes and a big matrix A, reflected in the missing data points. On the contrary, the serial implementation without MPI had more memory available.

4.3 Iterative Methods

A closer look at the implementation of the two iterative methods in figure 3 and 4 reveal the issue of why Gauss-Seidel is not suited for parallelization compared to Damped Jacobi. The bottleneck occurs when the forward substitution of equation (10) is performed. The problem is that to determine the $y_i^{(m)}$ in (10), every row of the linear system is dependent on the information of the previous one. Consequently, every process must wait until the previous one finishes his calculations. Therefore, performing the forward substitution on a singular process is more efficient for minimizing overhead communication.

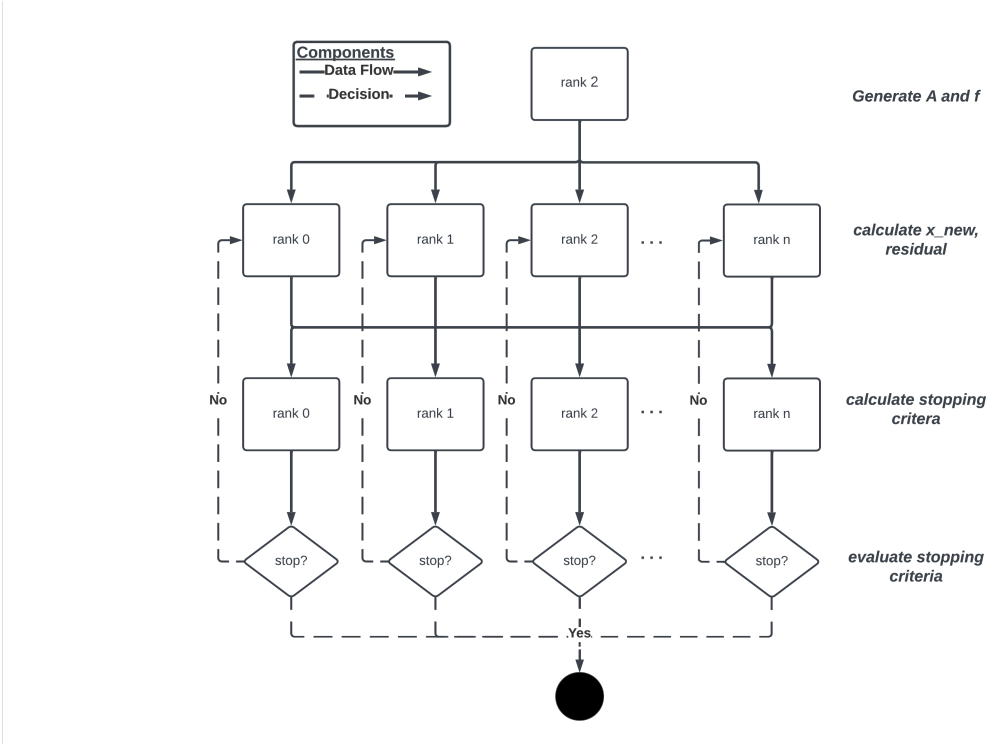


Figure 3: Flow Diagram Damped Jacobi, with n processes

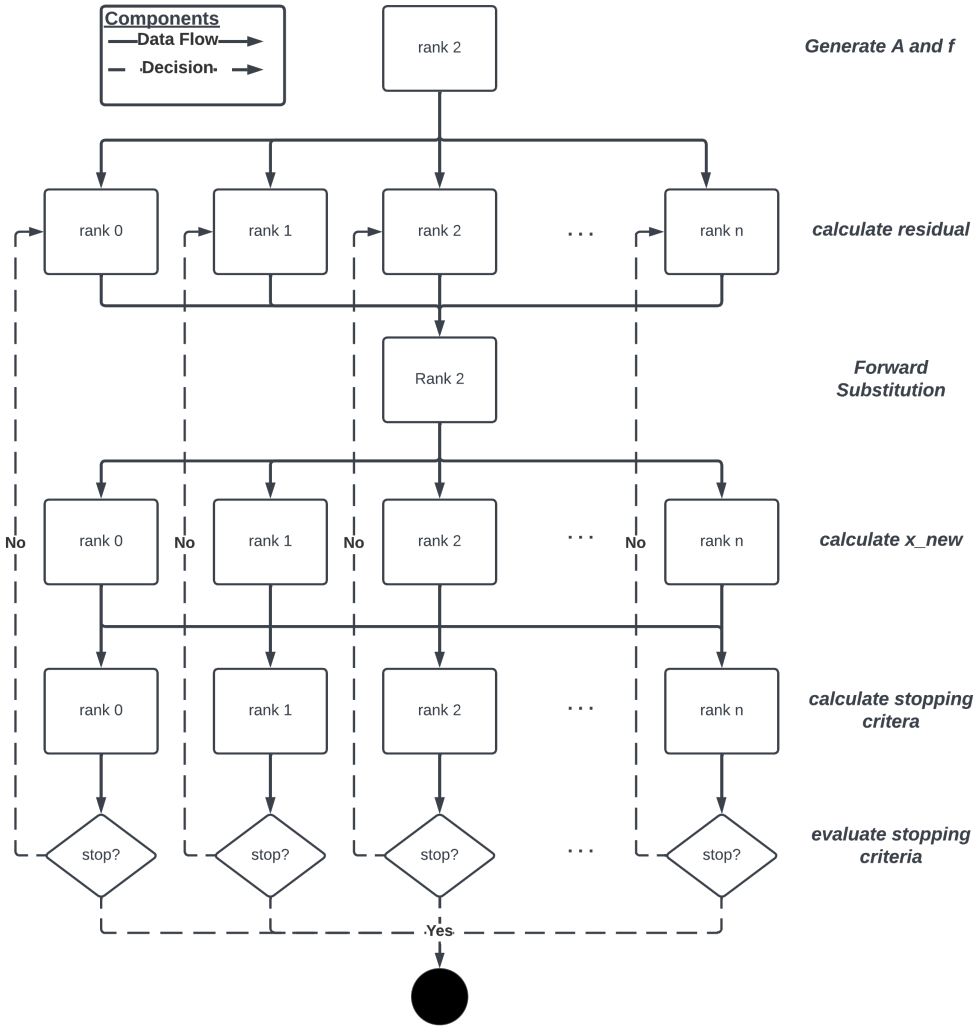


Figure 4: Flow Diagram Gauss-Seidel with n processes

4.4 Experiment remarks

The serial version of the methods are separate implementations and contain not a single line of code from the parallel version to improve performance.

Both Matrices (26) and (31) are sparse matrices, and the performance could be improved if this aspect had been taken into consideration. These improvements, though, would have been tailored to this specific matrix, which we opted not to do and focus on a solution that could take any matrix A as an input,

A single process does not include the creation of A and f in the execution time. The timing starts before the scattering of A and f and ends as soon as the stopping criteria are met.

The stopping threshold of $\epsilon = 1.0 \cdot 10^{-11}$ was used for all the experiments.

The code for this study can be found on a public repository on GitHub [3].

5 Speedup

The Speedup S_p for a number of processes p is one if not the most important measurement in parallel computing by capturing the performance improvement achieved by using multiple processes. This tells us how scalable a program is and can help to optimize resources, time, or costs.

$$S_p = \frac{t_1}{t_p} \tag{34}$$

t_1 is the execution time by the serial implementation and t_p the execution time with p processes.
The optimal speedup is represented by a green line in the graphs and shows the speedup for a program with perfected parallelization, no overhead for communication, load imbalance, or serial parts of the program.

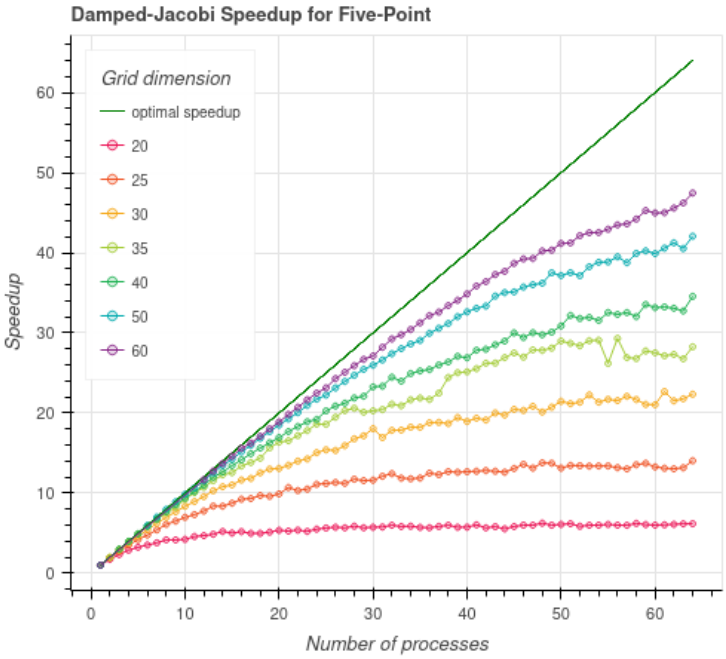
5.1 Five-Point

On one hand, Gauss-Seidel shows no speedup in 5b, which results in an almost flat and constant line. It appears that even though the bottleneck caused by the forward substitution, which should cost more time if the grid dimension is increased, cancels out with the performance gained by the parallelization of the rest of method.
On the other hand, the Damped Jacobi method 5a has a significant speed-up, when increasing the number of processes. Another observation is that with increasing grid size speedup increases. This is caused by the fact that the proportion between the process not performing an actual calculation, such as receiving or sending data, and all the processes performing the calculations in parallel, shifts to the latter. This also results in the processes being used more efficiently as described in section 7.

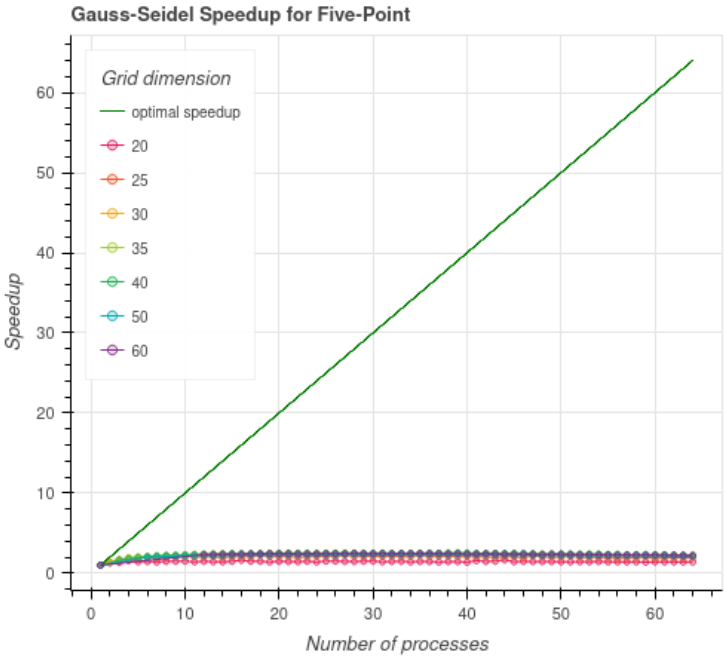
5.2 L2-Projection

We can observe that the behavior is almost identical for both methods in 6a and 6b compared to the Five-Point speedup. The big difference appears in the Damped Jacobi speedup curve in this version of the experiment, the speedup curve appears to be more linear in shape and the increase in grid size does not have the same increase in speedup. Due to the L2 Projection needing fewer iterations to converge, the speedup experiences more fluctuations. Another reason that could influence this behavior is the fact that perhaps somebody else was using the server at the time or

the chosen $\omega = \frac{2}{3}$ was not the optimal damping parameter and has a different effect on different sizes of the grid.

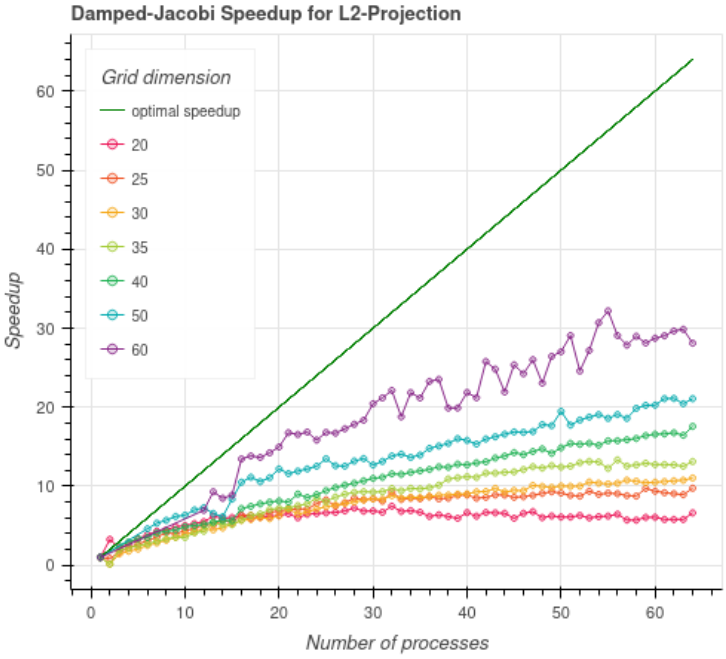


(a) Damped Jacobi

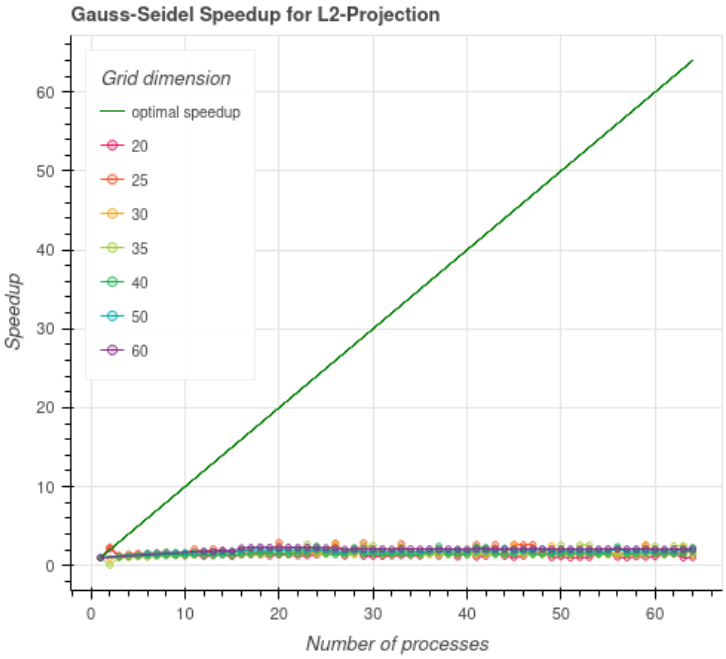


(b) Gauss-Seidel

Figure 5: Speedup for the Five-Point



(a) Damped Jacobi



(b) Gauss-Seidel

Figure 6: Speedup for the L2-Projection

6 Timing

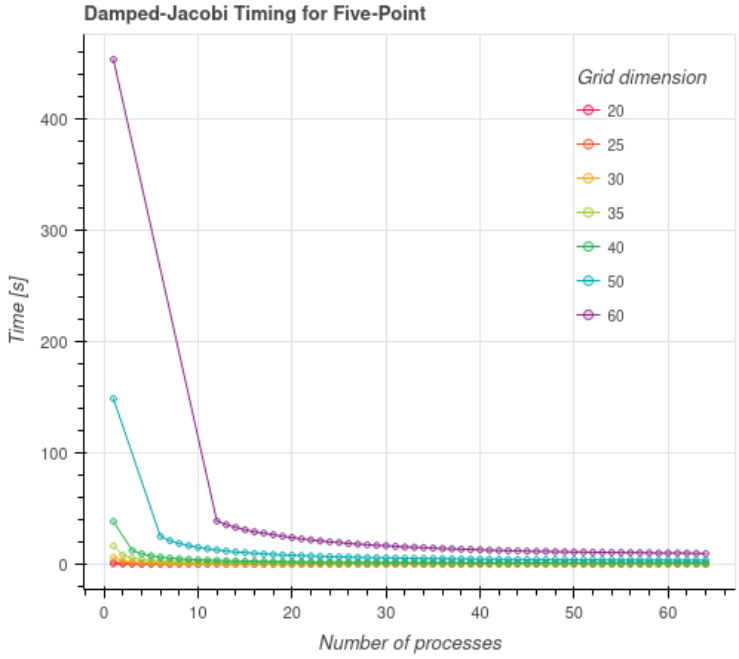
Timing plots provide insights into the raw execution time of a parallel program and can give valuable information about the performance characteristics.

6.1 Five-Point

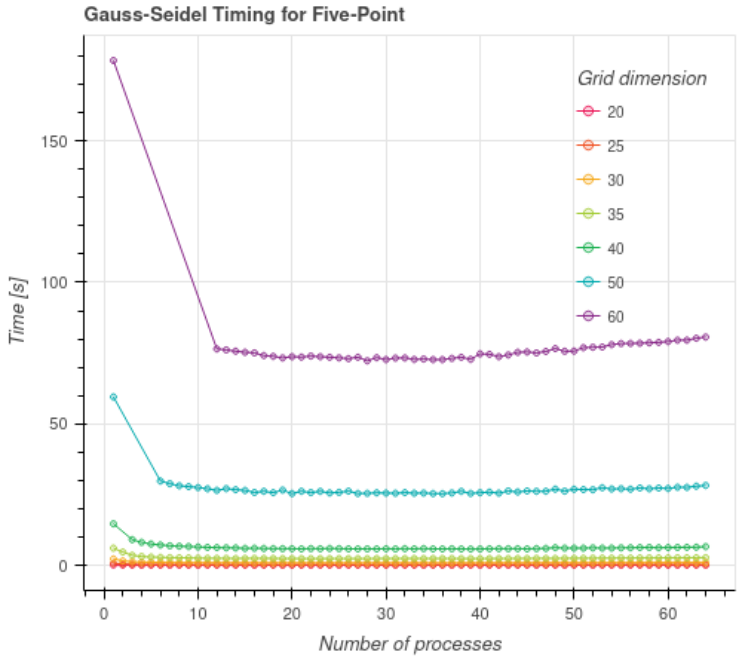
The timing plots 7a and 7b show that increasing the number of processes is especially beneficial for lower numbers of processes and then increasingly flattens out with higher numbers of processes.
The Gauss-Seidel method also has a significantly stronger increase in execution time when increasing the grid size due to the worse scalability of the method.
For a closer comparison between the two methods in figures 9a and 9b we can clearly see that the Gauss-Seidel starts off faster than gets overtaken by the Damped Jacobi method rather quickly with increasing number of processes. With increasing grid dimensions both curves retain the same shape, only the difference between the execution timings increased for a higher number of processes.

6.2 L2-Projection

The behavior seen in 8a and 8b is identical as described with the Five-Point experiment. The only significant difference is that the execution time for both methods is a lot lower due to the lower number of iterations. Also in 10a and 10b the plots appear to be almost identical to the corresponding five-point version, but the difference between execution time appears to be lower, which is again caused by a lower number of iterations because the slow down effect caused the bottleneck is not as severe.

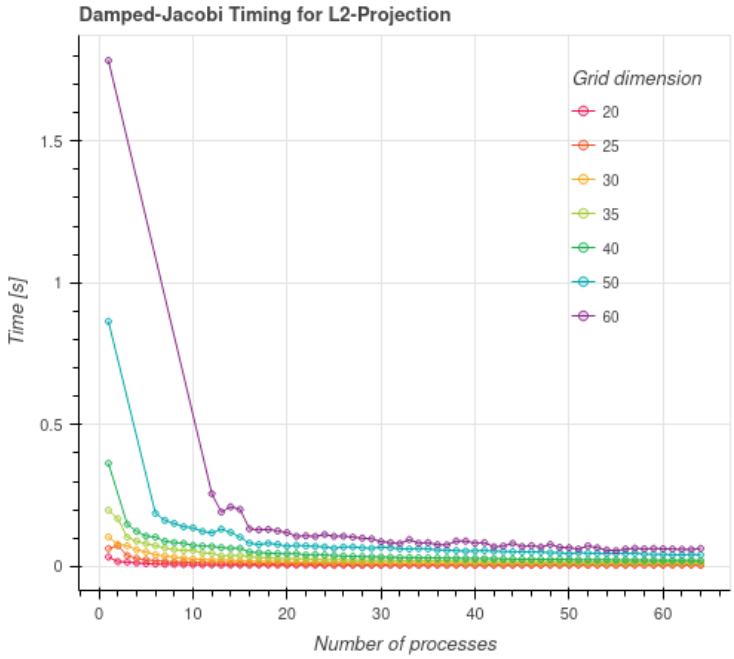


(a) Damped Jacobi

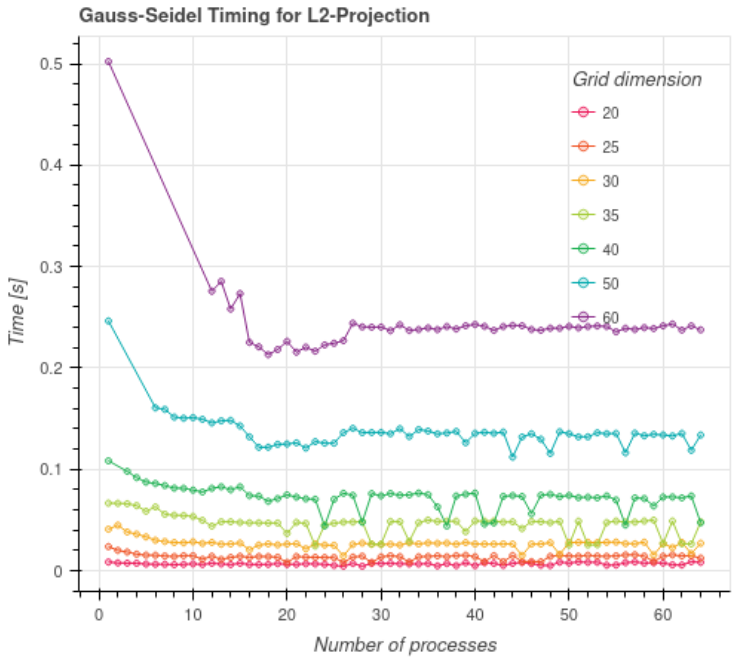


(b) Gauss-Seidel

Figure 7: Execution time of the Five-Point with different Grid dimensions

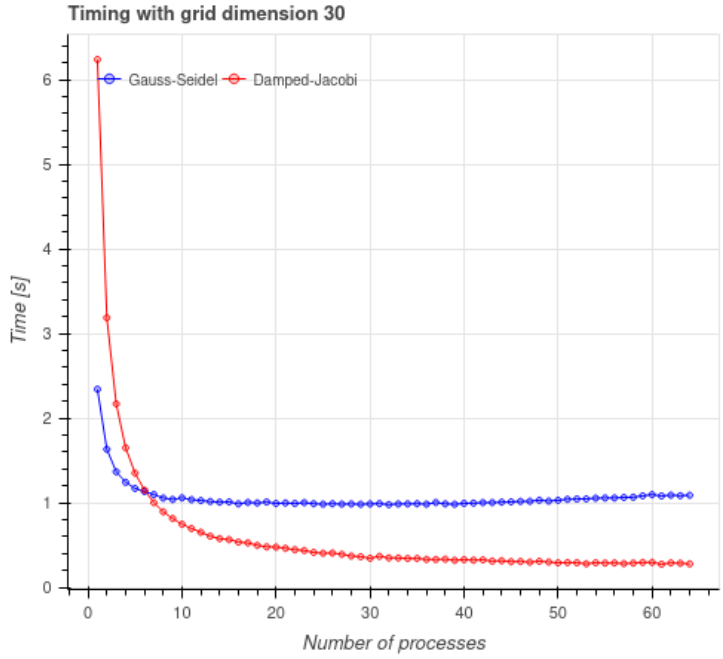


(a) Damped Jacobi

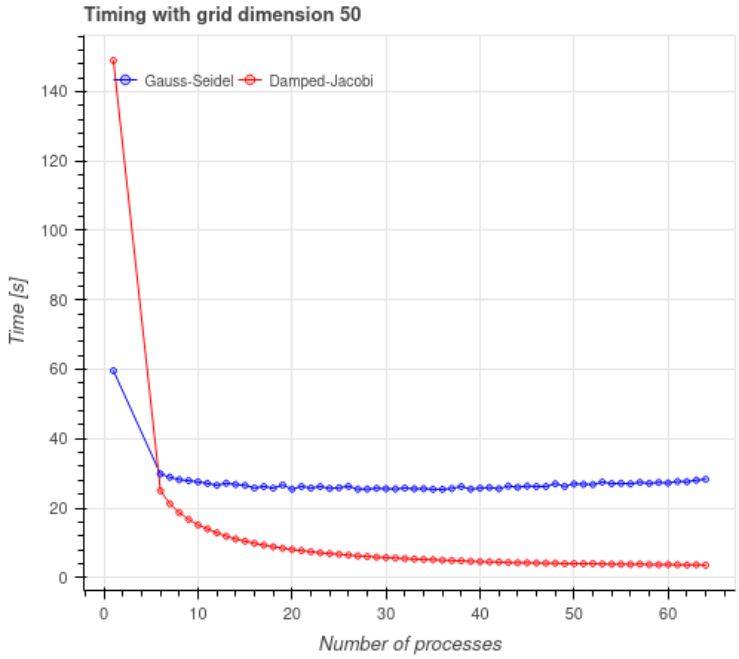


(b) Gauss-Seidel

Figure 8: Execution time of the L2-Projection with different Grid dimensions

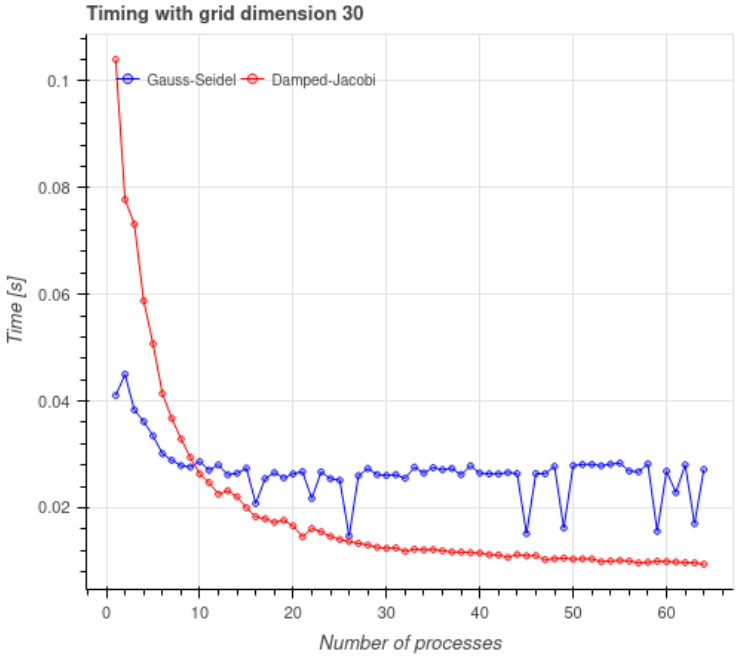


(a) Grid size 30×30

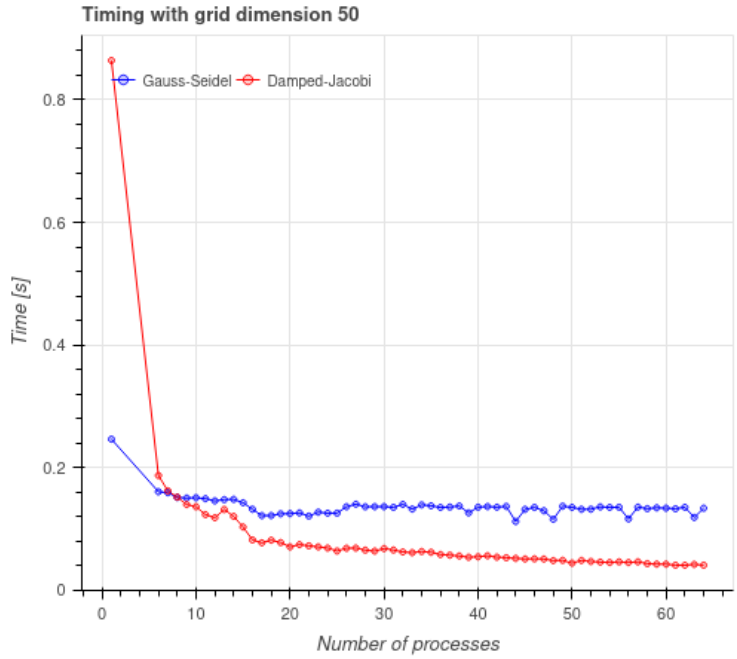


(b) Grid size 50×50

Figure 9: Head to Head Timing of the Five-Point with different grid sizes



(a) L2 Projection with grid dimension 30×30



(b) L2 Projection with grid dimension 50×50

Figure 10: Head to Head Timing of the L2-Projection with different grid sizes

7 Efficiency

Analyzing the efficiency can provide insight into how well a parallel program uses its resources. The efficiency is given by

$$E_p = \frac{S_p}{p} \tag{35}$$

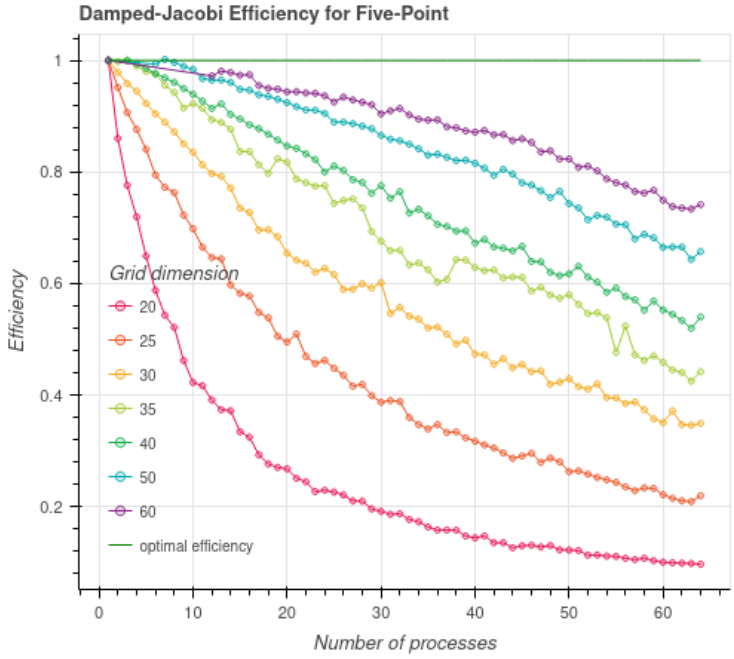
and ranges from 0 to 1, where 1 would mean perfect efficiency. Inefficiency can appear when processes are idle, this usually happens during communications, synchronization, waiting for serial code to finish, and other overhead generated by the parallelization.

7.1 Five-Point

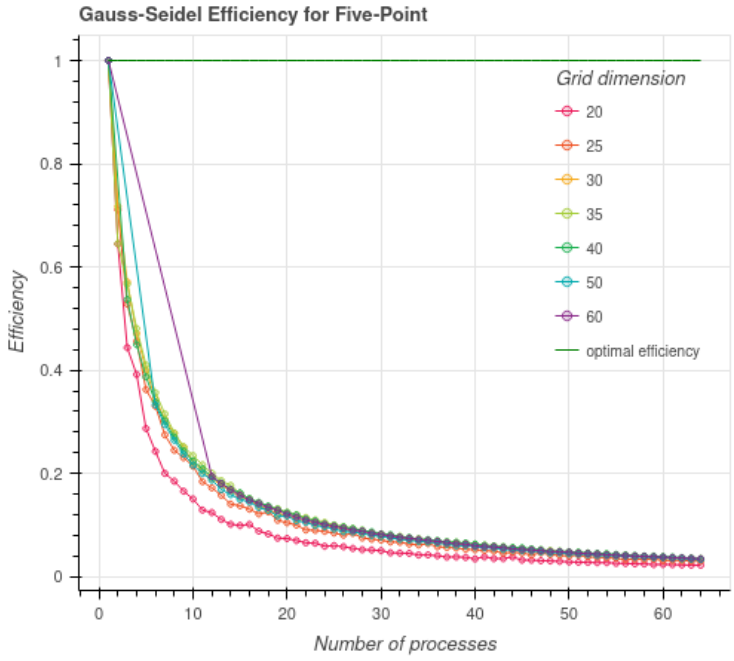
Gauss-Seidel in figure 11b clearly shows that cores are used more efficiently with a low number of processes. This inefficiency keeps increasing when using additional processes due to extra communication between the processes and the forward substitution bottleneck. For an increasing grid size, the efficiencies stay basically identical and do not improve at all. The Damped Jacobi on the other hand 11a efficiency has a similar curve to the Gauss-Seidel for a low number of processes, for when the number of processes is increased, the efficiency resembles more of a linear function. Additionally, the slope decreases when the number of processes is increased and therefore the same effect appears as already explained in the speedup section.

7.2 L2-Projection

The Efficiency plot for the Gauss-Seidel 12b behaves identically to the Five-Point, the only thing noticeable is that there are more fluctuations in the efficiencies. The Damped Jacobi 12a appears to behave similarly to his Five-Point counterpart, but the efficiency does not increase as strongly, due to the lower number of iterations.

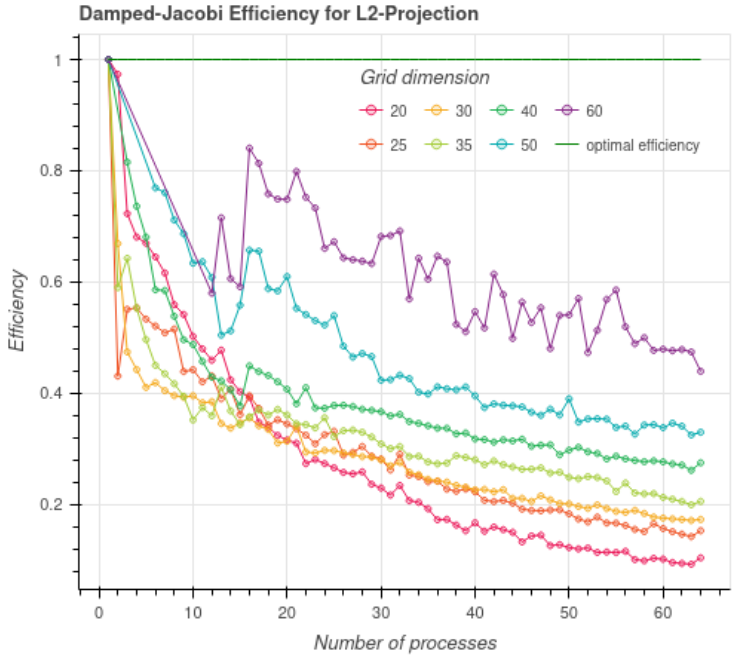


(a) Damped Jacobi

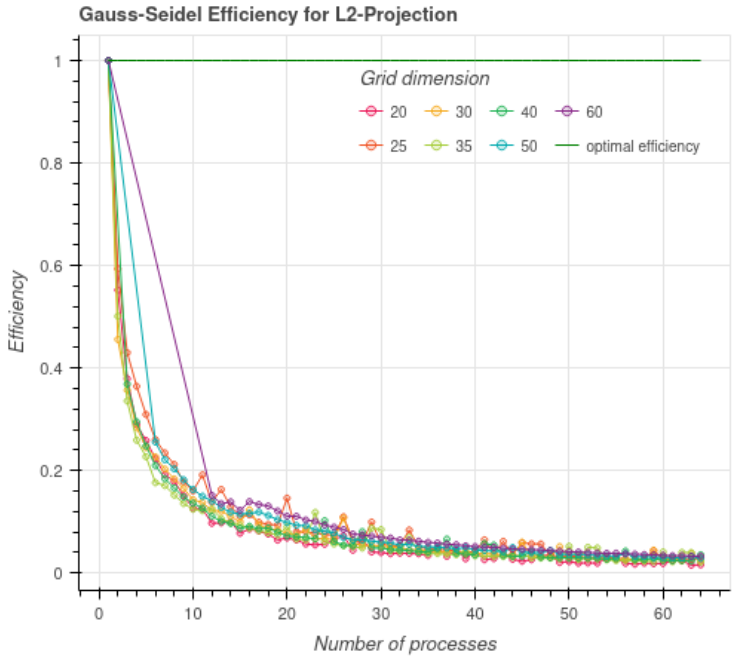


(b) Gauss-Seidel

Figure 11: Efficiency for the Five-Point



(a) Damped Jacobi



(b) Gauss-Seidel

Figure 12: Efficiency for the L2-Projection

8 Conclusion

In conclusion, this experiment was a success, the properties of Gauss-Seidel und Damped Jacobi that were established in theory, were represented in the output data of the experiment. Furthermore, there are still improvements that could be done to the implementation, especially in terms of load management, which would probably result in even better results. In this case, it would be interesting to see how other parallel libraries would perform against my implementation, do they have a similar execution time, do they experience similar fluctuations and how are they improving load balancing? It would be also interesting to compare more different iterative methods and see how they perform against Damped Jacobi and Gauss-Seidel.

References

[1] *compute_resources - PublicMathWiki*. URL: https://wiki.math.uzh.ch/public/compute_resources (visited on 05/16/2023).

[2] Wolfgang Hackbusch. *Iterative solution of large sparse systems of equations*. Second edition. Applied mathematical sciences volume 95, second edition. Cham: Springer, 2016. ISBN: 978-3-319-28481-1.

[3] Dennys Huber. *Numerical Training*. original-date: 2023-02-08T19:30:47Z. Feb. 17, 2023. URL: https://github.com/devnnys/numerical-training (visited on 05/09/2023).

[4] *Open MPI: Open Source High Performance Computing*. URL: https://www.open-mpi.org/ (visited on 05/15/2023).