Parallelization of stationary linear iterative methods for solving Poisson type PDE's

Henrik Chrintz

February 25, 2022

Abstract

In this paper a parallel implementation of three methods, Jacobi, Gauss-Seidel and Successive over relaxation (SOR), for solving large and sparse linear systems derived from Poisson type PDEs are investigated in order to find the faster solver method. Workload which is decoupled can be done faster in parallel than sequentially. We use a grid to discretize the PDE and the order of the grid cells yields the order of the equations in the system, called the lexicographical- or natural order. The natural order of the equations in the linear systems introduces coupling when Gauss-Seidel and SOR are applied. This coupling can be reduced by using a different order, namely the Red-Black order. Another motivation to investigate the three methods is to use them as preconditioners to other iterative algorithms such as GMRES, CG and Multi-grid. By solving problems of varying sizes with varying number of parallel processes the performance can be measured and compared. Using more processes is faster for larger problems, but not as beneficial for small problems. For the problems considered in this work SOR is faster than Gauss-Seidel which is in turn faster than Jacobi.

Table of Contents

1 Introduction 4

2 Background 4
- 2.1 From PDE to linear system 4
- 2.2 Symmetric difference approximation 5

3 Preliminaries 6
- 3.1 Notations 6
- 3.2 Existence and uniqueness properties 6
- 3.3 Definitions 7
- 3.4 Banach fixed-point theorem 8

4 Algorithm options 8
- 4.1 Direct method 8
- 4.2 Classical stationary linear iterative methods 8
  - 4.2.1 Jacobi 9
  - 4.2.2 Gauss-Seidel 10
  - 4.2.3 Successive over relaxation (SOR) 10
  - 4.2.4 Convergence rate 11
  - 4.2.5 Relation of spectral radii 11
- 4.3 Krylov subspace methods 11
- 4.4 Multi-grid methods 12
- 4.5 A note on preconditioners 12

5 Algorithm and test problem 12
- 5.1 The test problem 12
- 5.2 Eigenvalues and symmetric positive definitness proof 12
- 5.3 Convergence of the test problem 13
- 5.4 Red-Black ordering 13
- 5.5 Red-Black SOR on natural order matrices 15

6 Implementation details and numerics 15
- 6.1 Process partitioning 15
  - 6.1.1 By rows decomposition 15
  - 6.1.2 Domain decomposition 15
- 6.2 Partitioned matrix 16
- 6.3 Ghost rows 16
- 6.4 Implementation of Red-Black Gauss-Seidel and SOR 16
- 6.5 Solution to the waiting problem 17
- 6.6 Point to point communication 17
- 6.7 Sparse matrix implementation 18
- 6.8 Termination criterion 18
- 6.9 Initial guess 18
- 6.10 Relaxation factor 18

7 Performance analysis by experiments **18**
7.1 Main questions of the thesis 18
7.2 Method 19
7.3 POWER8 19
7.4 Experiment results, discussion and observations 19
7.4.1 Question 1 19
7.4.2 Question 2 21
7.4.3 Question 3 21
7.4.4 Question 4 21
7.4.5 Question 5 22

8 Conclusion and continued work **22**

9 Appendix A: Result tables **23**
9.1 Reduced version on POWER8 23
9.2 Full version on POWER8 24
9.3 Iteration timings on POWER8 24

10 Appendix B: source code **24**

1 Introduction

A big branch of modern mathematics is the field of partial differential equations, or PDE's for short. These equations are often hard to solve exactly, and need to be approximated instead. Some of these equations, like gravity or fluid dynamics (Navier-Stokes), has a Poisson type component, which is the target of investigation for this paper.
How this approximation is done will be presented and the resulting linear equation systems discussed. The size of these systems puts a constraint on our options for solver methods, due to memory consumption, and those options are discussed.
Solving these linear systems is a major bottleneck. Therefore, solving them faster is desired. Since decoupled workload can be done at the same time in parallel, it will be faster than to do it one at a time in sequence. To this end, a parallel implementation for all three methods, Jacobi, Gauss-Seidel and successive over relaxation were investigated.
Due to the natural order of the equations strong coupling is introduced when Gauss-Seidel and SOR are applied. This causes a waiting problem for the parallel implementations. A remedy is found in using a different order called the Red-Black order.

2 Background

The simplest PDE of Poisson type is Laplace's equation, which is a special case of Poisson's equation. Poisson's equation is formulated as:

$$\begin{aligned} -\Delta u(x) &= f(x), & x &\in \Omega \subset \mathbb{R}^n, \\ u(x) &= g(x), & x &\in \partial\Omega. \end{aligned}$$

Where $u(x)$ is our solution, $f(x)$ is the right-hand-side function describing the problem, $g(x)$ is the boundary condition, x is our vector-valued independent variable in $\mathbb{R}^n$, Ω is a compact subset of $\mathbb{R}^n$, the domain of f and u, where we want to find the solution and $\partial\Omega$ is its boundary. Finally $-\Delta$ is the Laplace operator.

2.1 From PDE to linear system

We want to approximate our PDE problem by a linear system of equations on the form:

$$A\mathbf{u} = \mathbf{f},$$

where A is some matrix, $\mathbf{u}$ and $\mathbf{f}$ are some vectors. To this end, we need a discretization of Ω. The chosen discretization was to partition Ω into an equidistant grid of cells, where all cells have equal side lengths. Then $f(x)$ can be evaluated at the center of each cell to generate $\mathbf{f}$. The corresponding component of $\mathbf{u}$ then approximates $u(x)$ at that cell center.

The cells in the grid, and therefore also the unknowns and equations, can be ordered in different ways, one of which is illustrated in Figure 1 for an $n \times n$ grid, called the natural ordering.

n^2-n	n^2-n+1	n^2-n+2	...	n^2-1
...	...	...	...	...
2n	2n+1	2n+2	...	3n-1
n	n+1	n+2	...	2n-1
0	1	2	...	n-1

Figure 1: Natural ordering of the cells.

The derivation of A comes next.

2.2 Symmetric difference approximation

Henceforth we will restrict our consideration of Ω to be a square subset of $\mathbb{R}^2$, with the accompanying grid also being square. With this restriction the $-\Delta$ operator is also 2-dimensional:

$$-\Delta u = -\frac{\partial^2 u}{\partial v_1^2} - \frac{\partial^2 u}{\partial v_2^2}, \qquad v \in \Omega \subset \mathbb{R}^2 \tag{1}$$

which can be approximated by a matrix. This can be done in several different ways, one of which is the symmetric (-central, -finite) difference scheme.
From the definition of the derivative:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \qquad x, h \in \mathbb{R}.$$

We get the two discrete versions:

$$f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \quad \text{and} \quad f'(x) \approx \frac{f(x) - f(x-\Delta x)}{\Delta x}.$$

Combining these two yields the central difference approximation to the second order derivative:

$$f''(x) \approx \frac{\frac{f(x+\Delta x)-f(x)}{\Delta x} - \frac{f(x)-f(x-\Delta x)}{\Delta x}}{\Delta x} = \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2}. \tag{2}$$

Using the formula in (2) we can now approximate $-\Delta u$ from (1) by:

$$-\Delta u \approx -\frac{u(v_1+\Delta v_1, v_2) - 2u(v_1, v_2) + u(v_1 - \Delta v_1, v_2)}{\Delta v_1^2} - \frac{u(v_1, v_2+\Delta v_2) - 2u(v_1, v_2) + u(v_1, v_2 - \Delta v_2)}{\Delta v_2^2},$$

and since $\Delta v_1 = \Delta v_2$ by choice since square grids have square cells, we get:

$$-\Delta u \approx \frac{-u(v_1, v_2+\Delta v_2) - u(v_1-\Delta v_1, v_2) + 4u(v_1, v_2) - u(v_1+\Delta v_1, v_2) - u(v_1, v_2-\Delta v_2)}{\Delta v_1^2}. \tag{3}$$

The formula in (3) can be represented by a stencil, which is illustrated in Figure 2.

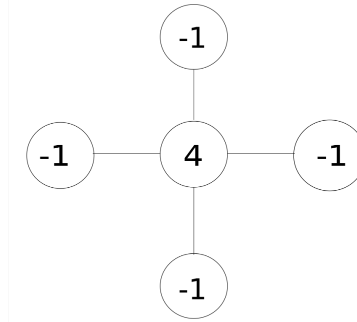

Figure 2: 5 point symmetric, central difference stencil.

To find out which unknowns enter which equation, one can use the stencil. Applying the stencil to grid-nodes on the boundary corresponds to an entry of the equation from the boundary conditions. Since the boundary conditions are known, we put them on the right-hand side. The same goes for the $\frac{1}{\Delta v_1^2}$ factor.

For example, the first equation will be :

$$4\mathbf{u}_0 - \mathbf{u}_1 - \mathbf{u}_n = \mathbf{f}_0.$$

The second:

$$-\mathbf{u}_0 + 4\mathbf{u}_1 - \mathbf{u}_2 - \mathbf{u}_{n+1} = \mathbf{f}_1.$$

For $n+1$:

$$-\mathbf{u}_1 - \mathbf{u}_n + 4\mathbf{u}_{n+1} - \mathbf{u}_{n+2} - \mathbf{u}_{2n+1} = \mathbf{f}_{n+1}.$$

And so on.
The resulting equations gives the following block structure to the matrix A, see (4):

$$
\begin{bmatrix}
B & -I & & & & & & & \\
-I & B & -I & & & & & & \\
 & -I & B & -I & & \cdots & & & \\
 & & -I & B & -I & & & & \\
 & & & -I & B & & & & \\
 & & \vdots & & & \ddots & & & \\
 & & & & & & B & -I & \\
 & & & & & & -I & B & -I \\
 & & & & & & & -I & B
\end{bmatrix} \tag{4}
$$

where I in (4) is a square identity matrix and B in (4) is given by (5):

$$
\begin{bmatrix}
4 & -1 & & & & & & \\
-1 & 4 & -1 & & & \cdots & & \\
 & -1 & 4 & -1 & & & & \\
 & & -1 & 4 & -1 & & & \\
 & & & -1 & 4 & & & \\
 & & \vdots & & & \ddots & & \\
 & & & & & & 4 & -1 \\
 & & & & & & -1 & 4
\end{bmatrix} \tag{5}
$$

For a grid of size $n \times n$, $A \in \mathbb{R}^{n^2 \times n^2}$, and has at most 5 non-zero entries per row.

3 Preliminaries

Now that we reduced our original problem to a linear system of equations, we must establish a handful of properties before we continue. We need to know what properties of the matrix and target vector are necessary and sufficient for the existence and uniqueness of solutions.

Here is where some preliminary knowledge of linear algebra is assumed. Some properties will be presented but the proofs will be omitted, see any textbook on linear algebra for the proofs, for example [10].

3.1 Notations

Some commonly used notations are presented here.

x_i : the i'th element of a vector x.
$A_{i,j}$: the element of a matrix A, of row i and column j.
A^{-1} : the inverse of a matrix A.
A^T : the transpose of a matrix A.
x^T : the transpose of a vector x. Note: vectors don't have transposes. However, vectors are commonly thought of as "column vectors" or more accurately, 1-column matrices. In this context the transpose makes sense. The transpose of a "column vector" is a "row vector" or 1-row matrix, and vice versa.
$u^T v$: the inner product of two vectors u and v.
$x^{(k)}$: the k'th iterate of some iterative process.
$\lambda(A)$: the set of eigenvalues of A.
$\lambda \in \lambda(A)$: λ is an eigenvalue of A.

3.2 Existence and uniqueness properties

Theorem 1 (Existence and uniqueness properties)**.** *Given a matrix $A \in \mathbb{R}^{m \times m}$ and a vector $b \in \mathbb{R}^m$, a solution $x \in \mathbb{R}^m$ to:*

$$Ax = b, \tag{6}$$

is given by:

$$x = A^{-1}b.$$

A^{-1} and x, exists and are unique if and only if all of the following equivalent properties of A hold:

1. *A is non-singular.*

2. A has full rank.

3. $b \in Range(A)$.

4. $\lambda \neq 0 \quad \forall \lambda \in \lambda(A)$.

For a proof, see any textbook on linear algebra, for example [10](ch 1).

3.3 Definitions

Definition 1 (LU factorization)**.** The LU-factorization is given by:

$$LU = A, \tag{7}$$

where L in (7) is lower triangular with 1's on the diagonal and U in (7) is upper triangular.

Definition 2 (Symmetric Positive Definite, SPD)**.** Given a matrix $A \in \mathbb{R}^{m \times m}$, it is said to be symmetric if $A^T = A$ and has only real eigenvalues. If all eigenvalues are positive, $\lambda > 0 \;\; \forall \lambda \in \lambda(A)$, then it is also positive definite.
In addition, for any vector $x \in \mathbb{R}^m$ we have:

$$x^T A x \geq 0, \tag{8}$$

with equality in (8) if and only if $x = 0$.

Remark. *An SPD matrix has full rank.*

Definition 3 (Diagonal dominance)**.** Given a matrix $A \in \mathbb{R}^{m \times m}$, it is said to be diagonally dominant if and only if:

$$\sum_{j=0, j\neq i}^{m-1} |A_{i,j}| < |A_{i,i}| \qquad i = 0..m-1,$$

$$\sum_{i=0, i\neq j}^{m-1} |A_{i,j}| < |A_{j,j}| \qquad j = 0..m-1,$$

both hold, and diagonally semi-dominant if and only if they hold with $\leq$ instead.

Definition 4 (Spectral radius)**.** The spectral radius of a matrix $A \in \mathbb{R}^{m \times m}$, often denoted $\rho(A)$, is the largest modulus among the eigenvalues of A,

$$\rho(A) = \max_{\lambda \in \lambda(A)} |\lambda|.$$

Definition 5 (Residual)**.** The residual, $r \in \mathbb{R}^m$, for $A \in \mathbb{R}^{m \times m}$ and $x, b \in \mathbb{R}^m$ is defined as: $r = Ax - b$ and measures how much x fails to satisfy the equations. Sometimes also as: $r = b - Ax$, which only differ by sign.

Definition 6 (Condition number)**.** Given an SPD matrix $A \in \mathbb{R}^{m \times m}$ the 2-norm condition number of A, denoted $\kappa(A)$ is given by (9):

$$\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}}. \tag{9}$$

Where λ_{max} and λ_{min} are the largest and smallest eigenvalues, respectively.

Definition 7 (Lipschitz continuous)**.** A mapping $\phi : \mathbb{R}^m \to \mathbb{R}^m$ is said to be globally Lipschitz continuous if and only if:

$$\|\phi(x) - \phi(y)\| \leq L\|x - y\| \quad \forall x, y \in \mathbb{R}^m, \tag{10}$$

holds for some $L \in \mathbb{R}^+$. The smallest such L is called the Lipschitz constant of ϕ. If $L < 1$ then ϕ is said to be a contraction.

Remark (Affine mapping)**.** *An affine mapping* $\psi : \mathbb{R}^m \to \mathbb{R}^m$ *of the form:* $\psi(x) = Mx + c$ *with* $M \in \mathbb{R}^{m \times m}$ *and* $x, c \in \mathbb{R}^m$ *is Lipschitz continuous with Lipschitz constant* $L = \rho(M)$.

Remark (Sparse matrix)**.** *A matrix is said to be sparse if more than half of its entries are zero. In this paper sparse matrices are much more sparse than that, often exceeding 99.9% zero entries. The opposite of sparse is dense.*

3.4 Banach fixed-point theorem

Theorem 2 (Banach fixed-point theorem)**.** *Let ϕ be a Lipschitz continuous mapping (see Definition 7) $\phi : \mathbb{R}^m \to \mathbb{R}^m$ and consider the fixed-point problem in* (11)*:*

$$x^* = \phi(x^*), \qquad for\ some\ x^* \in \mathbb{R}^m. \tag{11}$$

A solution to (11) *exists and is unique if and only if ϕ is a contraction (see Definition 7).*
Equation (11) *can be solved by a process called fixed-point iteration, given by* (12)*.*

$$x^{(k+1)} = \phi(x^{(k)}), \tag{12}$$

where $x^{(k)} \in \mathbb{R}^m$ denotes the k'th iterate, and is started by an initial guess $x^{(0)} \in \mathbb{R}^m$. The fixed-point iteration converges globally to a unique x^* independent of initial guess $x^{(0)}$ [1]:

$$x^* = \lim_{k\to\infty} x^{(k)}.$$

4 Algorithm options

Assuming existence and uniqueness of solutions to the linear systems, we need to discuss how to solve the equations. The naive approach is to find A^{-1} and then multiply by b, but that won't work in practice, due to stability issues, ill-conditioness and memory constraints among others. Therefore, other alternatives are considered instead.
There are 4 different categories of solver algorithms that will be considered.

4.1 Direct method

The first solver method we will look at is Gaussian elimination, otherwise known as LU factorization. We solve systems by the following steps:

1. Factorize A: $A = LU$.
2. Substitute A for LU: $Ax = b \to LUx = b$.
3. Substitute Ux=y $\to Ly = b$.
4. Solve $Ly = b$ for y by forward substitution.
5. Solve $Ux = y$ for x by backward substitution.

This algorithm is fast for small problem sizes where memory consumption is not an issue, since it finishes in a finite number of steps. The drawback is that two new matrices L and U need to be constructed. Unfortunately, both L and U are dense even if A is sparse. Since we investigate big problems, easily $m > 10^5$, those matrices wont fit into memory. The same is true for A^{-1}, it is dense even if A is sparse.

4.2 Classical stationary linear iterative methods

The general idea of iterative methods is to approximate a solution by a sequence of iterates, produced by some matrix-vector product. This avoids the memory storage problem of direct methods, since A is sparse and no dense factorization is needed. Also note, since A is sparse, matrix-vector multiplication is fast, only $\mathcal{O}(m)$, compared to the dense matrix-vector multiplication which is $\mathcal{O}(m^2)$.

The simplest iterative process is called fixed-point iteration. Therefore, the idea of formulating (6) as a fixed-point problem (11) seems natural.
This formulation can be done in different ways, the simplest one being the following:

$$\begin{aligned} &Ax = b \\ \iff &Ax + Ix = Ix + b \\ \iff &Ix = Ix - Ax + b \\ \iff &x = (I - A)x + b. \end{aligned} \tag{13}$$

Where the right hand side of (13) is an affine mapping, which when applied to (12) gives rise to the Richardson iteration [8](ch 1.2) in (14).

$$x^{(k+1)} = (I - A)x^{(k)} + b. \tag{14}$$

The iterations in (14) converges globally if and only if $\rho(I - A) < 1$, by Theorem 2.

Another way to formulate the fixed-point problem (11) is to use splittings on the form:

$$A = A_1 + A_2. \tag{15}$$

One important requirement is that A_1 from (15) must be invertable, and A_1^{-1} must be easy to apply, for example with backward or forward substitution.
Then we get:

$$\begin{aligned} &Ax = b \\ \iff &(A_1 + A_2)x = b \\ \iff &A_1x + A_2x = b \\ \iff &A_1x = -A_2x + b \\ \iff &x = -A_1^{-1}A_2x + A_1^{-1}b, \end{aligned} \tag{16}$$

where the right hand side of (16) is an affine mapping, which when applied to (12) yields (17) which takes the general form (18):

$$x^{(k+1)} = -A_1^{-1}A_2x^{(k)} + A_1^{-1}b \tag{17}$$

$$\iff x^{(k+1)} = Mx^{(k)} + c. \tag{18}$$

Where $M = -A_1^{-1}A_2$ in (18) is called the iteration matrix. Convergence of (18) holds if and only if $\rho(M) < 1$ from Theorem 2.

Depending on the splitting (15) we get different iteration schemes.

4.2.1 Jacobi

From (15) we can choose to split in the following fashion:

$$A_1 = D, \quad A_2 = L + U. \tag{19}$$

Where D from (19) is the diagonal of A, L and U from (19) are the strict lower and upper triangular parts of A. This yields:

$$\begin{aligned} &Ax = b \\ \iff &(D + L + U)x = b \\ \iff &Dx + Lx + Ux = b \\ \iff &Dx = -Lx - Ux + b \\ \iff &x = -D^{-1}(L + U)x + D^{-1}b. \end{aligned}$$

$$x^{(k+1)} = -D^{-1}(L + U)x^{(k)} + D^{-1}b. \tag{20}$$

Where (20) is called the Jacobi iteration with iteration matrix in (21).

$$M_{JAC} = -D^{-1}(L + U). \tag{21}$$

4.2.2 Gauss-Seidel

Similarly to (19) we can use a different splitting:

$$A_1 = D + L, \qquad A_2 = U. \tag{22}$$

Then we get:

$$\begin{aligned} Ax &= b \\ \iff (D+L+U)x &= b \\ \iff Dx + Lx + Ux &= b \\ \iff (D+L)x &= -Ux + b \\ \iff x &= -(D+L)^{-1}Ux + (D+L)^{-1}b. \\ x^{(k+1)} &= -(D+L)^{-1}Ux^{(k)} + (D+L)^{-1}b. \end{aligned} \tag{23}$$

Where (23) is called the forward Gauss-Seidel iterations, from forward substitution, with iteration matrix in (24).

$$M_{fGS} = -(D+L)^{-1}U. \tag{24}$$

If we instead of (22) do the following splitting in (25):

$$A_1 = D + U, \qquad A_2 = L. \tag{25}$$

We get:

$$\begin{aligned} Ax &= b \\ \iff (D+U+L)x &= b \\ \iff (D+U)x &= -Lx + b \\ \iff x &= -(D+U)^{-1}Lx + (D+U)^{-1}b. \\ x^{(k+1)} &= -(D+U)^{-1}Lx^{(k)} + (D+U)^{-1}b. \end{aligned} \tag{26}$$

Where (26) is called the backwards Gauss-Seidel iterations, from backward substitution, and the iteration matrix is given in (27).

$$M_{bGS} = -(D+U)^{-1}L. \tag{27}$$

There are more flavours of Gauss-Seidel iterations but they fall outside of the scope of this report, with one exception which will be treated later.

4.2.3 Successive over relaxation (SOR)

SOR is very similar to Gauss-Seidel, but has an extra degree of freedom in the choice of a relaxation factor ω. This extra flexibility allows for extra weight to be put on the off-diagonal elements. The benefits to this will be apparent later in section 4.2.5. SOR with $\omega = 1$ is equivalent to Gauss-Seidel.
This yields:

$$\begin{aligned} Ax &= b \\ \iff \omega Ax &= \omega b \\ \iff \omega(D+L+U)x &= \omega b \\ \iff \omega Dx + \omega Lx + \omega Ux &= \omega b \\ \iff Dx + \omega Dx - Dx + \omega Lx &= -\omega Ux + \omega b \\ \iff Dx + (\omega-1)Dx + \omega Lx &= -\omega Ux + \omega b \\ \iff Dx + \omega Lx &= -\omega Ux - (\omega-1)Dx + \omega b \\ \iff (D+\omega L)x &= -(\omega U + (\omega-1)D)x + \omega b \\ \iff x &= -(D+\omega L)^{-1}(\omega U + (\omega-1)D)x + (D+\omega L)^{-1}\omega b. \\ x^{(k+1)} &= -(D+\omega L)^{-1}(\omega U + (\omega-1)D)x^{(k)} + (D+\omega L)^{-1}\omega b. \end{aligned} \tag{28}$$

$$M_{SOR}(\omega) = -(D+\omega L)^{-1}(\omega U + (\omega-1)D). \tag{29}$$

Where (28) is the SOR iterations with iteration matrix in (29) for some ω. More about the optimal choice of ω can be found in section 4.2.5.

4.2.4 Convergence rate

Lemma 1. *The iteration schemes considered above converges with a rate estimated by* (30)*.*

$$\|x^{(k+1)} - x^{(k)}\| \le \rho(M)^k \|x^{(1)} - x^{(0)}\|. \tag{30}$$

Proof. Let $\psi : \mathbb{R}^m \to \mathbb{R}^m$ be an affine mapping on the form $\psi(x) = Mx + c$ with $M \in \mathbb{R}^{m\times m}$ and $x, y, c \in \mathbb{R}^m$. ψ is Lipschitz continuous by Definition 7 and from (10) we get (31):

$$\|\psi(x) - \psi(y)\| = \|Mx - My\| = \|M(x-y)\| \le \|M\|\|x-y\| = \rho(M)\|x-y\|. \tag{31}$$

Assuming $\rho(M) < 1$ we can use the iterations from (12) and $x^{(0)} \in \mathbb{R}^m$ to obtain (32)(see also [1]):

$$\|x^{(2)} - x^{(1)}\| \le \rho(M)\|x^{(1)} - x^{(0)}\|. \tag{32}$$

Repeating this process using induction we get (30) [1]: □

From (30) we get an estimate on the number of iterations needed to converge depending on $\rho(M)$, the initial guess $x^{(0)}$ and the vector $c = A_1^{-1}b$. The left hand side of (30) is a Cauchy sequence termination criteria.

4.2.5 Relation of spectral radii

Lemma 2. *Comparing the three methods Jacobi, Gauss-Seidel and SOR we get* (33)*:*

$$0 < \rho(M_{SOR}(\omega_{opt})) < \rho(M_{SOR}(1)) = \rho(M_{GS}) < \rho(M_{JAC}) < 1. \tag{33}$$

Proof. Given $0 < \mu = \rho(M_{JAC}) < 1$ the formula for the spectral radius of $M_{SOR}(\omega)$ is given by (34), [6](ch 4.6.2):

$$\rho(M_{SOR}(\omega)) = \begin{cases} \frac{1}{4}(\omega\mu + \sqrt{\omega^2\mu^2 - 4(\omega-1)})^2 & 0 < \omega \le \omega_{opt}, \\ \omega - 1 & \omega_{opt} \le \omega < 2. \end{cases} \tag{34}$$

With ω_{opt} from (35).

$$\omega_{opt} = 1 + \left(\frac{\mu}{1+\sqrt{1-\mu^2}}\right)^2. \tag{35}$$

By (35) we have $1 < \omega_{opt} < 2$.

We know from before that $M_{SOR}(1) = M_{GS}$ so this proves the equality in (33). Plugging in $\omega = 1$ into (34) gives $\rho(M_{GS}) = \mu^2$ and since $\mu < 1 \implies \mu^2 < \mu$ the last inequality of (33) hold. Plugging in ω_{opt} into (34) we get $\rho(M_{SOR}(\omega_{opt})) = (\frac{\mu}{1+\sqrt{1-\mu^2}})^2 < \mu^2$, thus the first inequality of (33) hold. □

A good choice of ω lowers the spectral radius and therefore gives better convergence rate than the corresponding Gauss-Seidel method, which is in turn better then the Jacobi method.

From (30) we get a relative estimate of the number of iterations needed for convergence. Since $\rho(M_{GS}) = \mu^2$ we can estimate that Jacobi needs twice as many iterations to converge compared to Gauss-Seidel. The estimate for SOR compared to Gauss-Seidel is a bit harder to derive, since $\rho(M_{SOR})$ depends non-linearly on μ, but it should be smaller for a well chosen ω.

4.3 Krylov subspace methods

The two most famous Krylov subspace methods are GMRES(Generalized Minimum RESidual, works for general matrices of full rank) and CG(Conjugate Gradients, works for SPD systems). They are similar to the stationary methods in that they are iterative, but they differ in most other aspects. For example, CG and GMRES depend on the past history of the iterations where Jacobi, Gauss-Seidel and SOR does not. GMRES and CG can be viewed as optimization algorithms, finding the linear combination of the current Krylov subspace basis which minimizes the residual measured in the weighted A-norm, $\|\cdot\|_A$ given by (36):

$$\|x\|_A = \frac{x^T A x}{x^T x}. \tag{36}$$

If the termination criterion is not met, the iteration continues by extending the Krylov subspace basis by another basis vector, then proceed to find the minimum residual and repeat.
Overall the convergence rate of GMRES and CG is much superior to that of SOR, but not always. GMRES and CG have poor convergence if the eigenvalues of A are scattered, so we need to apply a remedy: preconditioners. As we shall see in section 4.5 Jacobi, Gauss-Seidel and SOR can be used as preconditioners.

4.4 Multi-grid methods

Multi-grid methods are the state of the art solver algorithm for linear problems in scientific computing, see for eample [4] for an overview. They can be regarded as a type of stationary method, but this connection is non-trivial.
The core concept of multi-grid methods is to make use of the geometry where the linear system arises. By solving the systems of smaller size resulting from a coarser refinement(smaller grid) you get information about the solution to the larger system(finer, larger grid). This process is called coarse-grid correction and its purpose is to eliminate the high frequency error components. As a compliment to the coarse-grid correction we also need to do something called smoothing in order to eliminate low frequency error components.

There is now one more problem, how do we solve the smaller system? One approach is to use the same multi-grid method recursively, until we reach a point where we can solve the systems with another method. Usually only a few recursive steps are taken, so the smallest system we would encounter is still far too big for a direct method. This leaves for example GMRES or -CG with preconditioner or SOR, both of which are viable choices. Another choice is to mimic FASMG, the non-linear multi-grid method, and not do a full solve of the system at the coarsest level, but instead do a few iterations of a preconditioner there.

4.5 A note on preconditioners

A preconditioner is a perturbation of a system such that the solution remains the same and the new system is easier to solve.
The worst preconditioner is the identity matrix; this choice improves nothing, but is also not harmful. The best one is A^{-1}, which would solve the systems in a single iteration, granted you could find the solution simply by matrix-vector multiplication instead. This gives us a general idea of what a preconditioner should be: it should be a good approximation of A^{-1} and fast to apply.
Given an iterate $x^{(k)}$ of CG or GMRES we can apply a preconditioner such that the preconditioned iterate, call it $\hat{x}^{(k)}$, is closer to $A^{-1}b$ than $x^{(k)}$ is. This intermediate step improves the convergence rate of CG and GMRES and can be performed, for example, by SOR.

5 Algorithm and test problem

Jacobi, Gauss-Seidel and SOR were chosen because they can be used to both, solve systems fully and be used as preconditioners.

5.1 The test problem

As the test problem the following choices were made:

$$f(x,y) = \sin(\pi x)\cdot\sin(\pi y), \quad x,y\in\mathbb{R},$$

$$\Omega = [0,1]\times[0,1]\subset\mathbb{R}^2,$$

$$g(x,y)\equiv 0.$$

The boundary conditions hold since $\sin(0)=\sin(\pi)=0$. The region of interest, Ω, is the unit square.
For a given grid size n, the width of grid cells are $\Delta x = \frac{1}{n}$. From the symmetric difference scheme we have a factor $\frac{1}{\Delta x^2}$ on the left hand side, but this can be canceled out by multiplying both sides by Δx^2. Evaluating $f(x,y)$ over each interior grid cell center yields the right hand side vector $\mathbf{f}$. Therefore we get the following system:

$$A\mathbf{u} = \Delta x^2\mathbf{f}, \tag{37}$$

where $\mathbf{u}$ in (37) is our solution on sampled form and A is our symmetric difference operator matrix as in (4).

5.2 Eigenvalues and symmetric positive definitness proof

Theorem 3 (The symmetric difference approximation is SPD)**.** *Let $A\in\mathbb{R}^{m\times m}$ be as in* (4) *with $m=n^2$, then A is SPD (see also [4](ch 1), [7](ch 4.4)).*

Proof. The eigenvalues $\lambda_k\in\lambda(A)$ are given by the following formula (38) [7](ch 4.4):

$$\lambda_k = 4\sin^2\left(\frac{\pi i}{2(n+1)}\right) + 4\sin^2\left(\frac{\pi j}{2(n+1)}\right), \quad k = i+(j-1)n, \quad i,j=1..n. \tag{38}$$

Clearly $\lambda_k \geq 0$ since its a sum of squares. It remains to show that $\lambda_k \neq 0$. Notice:

$$\sin^2(a) = 0 \implies a = \pi z, \quad z \in \mathbb{Z}.$$

But from (38) we have:

$$0 < z = \frac{i}{2(n+1)} < \frac{1}{2} \notin \mathbb{Z} \implies \sin^2(\pi z) > 0.$$

Likewise for j. This proves that A is positive definite by Definition 2. Since A is clearly symmetric, it is SPD and non-singular. □

Corollary 3.1. *Let $A \in \mathbb{R}^{m\times m}$ be as in (4) and let λ_{min} and λ_{max} be the smallest and largest eigenvalues of A respectively, then the following identity holds (39):*

$$1 - \frac{\lambda_{min}}{4} = \rho(M_{JAC}) = \frac{\lambda_{max}}{4} - 1 < 1. \tag{39}$$

Proof. From (21) we get:

$$M_{JAC} = -D^{-1}(L+U) = -D^{-1}(A-D) = -(D^{-1}A - I). \tag{40}$$

Here (40) describes the affine transform acting on the eigenvalues of A. In (40) $D^{-1} = \frac{1}{4}I$ meaning $D^{-1}A$ divides all eigenvalues by 4. Now subtracting I means all eigenvalues get subtracted by 1. Finally multiplying by -1 changes the sign of all eigenvalues, a reflection over the origin.
From (38) we obtain λ_{max} when $i, j = n$, $\lambda_{max} \lessapprox 8$ and λ_{min} when $i, j = 1$, $\lambda_{min} \gtrapprox 0$. Investigating the first equality of (39) we find that λ_{min} is mapped to $-(\frac{\lambda_{min}}{4} - 1) = 1 - \frac{\lambda_{min}}{4} > 0$ which is the largest eigenvalue of M_{JAC}. Similarly λ_{max} is mapped to $-(\frac{\lambda_{max}}{4} - 1) = 1 - \frac{\lambda_{max}}{4} < 0$ which is the smallest eigenvalue of M_{JAC}. Since the spectral radius takes the absolute values of the eigenvalues we establish the second equality of (39). Both equalities hold due to symmetry.
The inequality hold from the mentioned estimates of λ_{min} and λ_{max}.
No other eigenvalue can become the spectral radius of M_{JAC}. This is because the largest and smallest eigenvalues remain so after scaling and translations. Finally a multiplication by -1 reverses the smallest and largest eigenvalues. □

Theorem 4 (Condition number of A)**.** *Let $A \in \mathbb{R}^{m\times m}$ be as in (4) with $m = n^2$, then $\kappa(A) = \mathcal{O}(n^2)$.*

Proof. From Definition 6 we get:

$$\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}} \approx \frac{8}{8\sin^2(\frac{\pi}{2(n+1)})} \approx \frac{1}{\frac{\pi^2}{4(n+1)^2}} = \frac{4(n+1)^2}{\pi^2} = \mathcal{O}(n^2).$$

Here, we used the small angle approximation of sin since n is large. □

5.3 Convergence of the test problem

The convergence properties for Jacobi, Gauss-Seidel and SOR are problem dependent. There are two properties of the matrix that guarantee global convergence:

1. The problem matrix is strictly diagonal dominant or,
2. The problem matrix is SPD.

Our test problem has the matrix from (4) which is SPD! Therefore Jacobi, Gauss-Seidel and SOR will converge globally independent of initial guess [5](ch 10.1).

5.4 Red-Black ordering

One important fact about Jacobi, which is not easy to see from section 4.2, is that Jacobi only uses the previous iterate for its updating scheme, whereas Gauss-Seidel and SOR use both, previous and new iterates as soon as they are available. A consequence of this is that the order in which one updates the values matters for Gauss-Seidel and SOR, but not for Jacobi.
When using forward SOR to solve (37) in parallel, coupling is introduced. This means that each process beyond the first needs to wait for the neighbouring process to finish before it can finish its own work, since it needs the newly updated values. This means only one process is doing work at a time, and all the others are waiting. See section 6.5 for further details.
The remedy is to reorder the equations and unknowns in the so called Red-Black order, based on the description

found in [9](ch 12.4). Using this order every process has work to do at all times. This order is illustrated in Figure 3 for a 6x4 grid.

21	9	22	10	23	11
6	18	7	19	8	20
15	3	16	4	17	5
0	12	1	13	2	14

Figure 3: Red-Black ordering for a 6x4 grid.

Note how two consecutive indices have the same color when wrapping around the edge of the grid. This is true only for even sized grids.
This reordering yields a different matrix, right hand side and solution, where the solution has been permuted as compared to the natural order solution.

$$A_{RB}\hat{\mathbf{u}} = \Delta x^2 \hat{\mathbf{f}}. \tag{41}$$

The new A_{RB} in (41) has the following block structure in (42) [9](ch 12.4):

$$\begin{bmatrix} D & \hat{B} \\ \hat{B}^T & D \end{bmatrix} \tag{42}$$

where $D = 4I$ in (42) and $\hat{B}$ in (42) is in turn given by (43) if the grid size is even and (44) if it is odd.

$$\begin{bmatrix} W & -I & & & & & \\ -I & E & -I & & \cdots & & \\ & -I & W & -I & & & \\ & & -I & E & & & \\ & \vdots & & & \ddots & & \\ & & & & & W & -I \\ & & & & & -I & E \end{bmatrix} \tag{43}$$

$$\begin{bmatrix} W & -I & & & & & \\ -I & W & -I & & \cdots & & \\ & -I & W & -I & & & \\ & & -I & W & & & \\ & \vdots & & & \ddots & & \\ & & & & & W & -I \\ & & & & & -I & W \end{bmatrix} \tag{44}$$

W in (43) and (44) is given by (45).

$$\begin{bmatrix} -1 & & & & & \\ -1 & -1 & & \cdots & & \\ & -1 & -1 & & & \\ & \vdots & & \ddots & & \\ & & & & -1 & \\ & & & & -1 & -1 \end{bmatrix} \tag{45}$$

And E in (43) by (46).

$$\begin{bmatrix} -1 & -1 & & & & \\ & -1 & -1 & \cdots & & \\ & & -1 & & & \\ & \vdots & & \ddots & & \\ & & & & -1 & -1 \\ & & & & & -1 \end{bmatrix} \tag{46}$$

A_{RB} is clearly symmetric from (42) and can be written on the form:

$$A_{RB} = PAP^T,$$

for some permutation matrix P. Since permutation matrices are orthogonal, eigenvalues are preserved under multiplication. In addition, this equation is a similarity transform. Thus A and A_{RB} have the same eigenvalues. Since A is SPD so is A_{RB}.

5.5 Red-Black SOR on natural order matrices

Solving (41) with forward Gauss-Seidel/SOR is optimal since it removes the waiting entirely. The drawback of this approach is that one has to reorder not only the matrix itself but also the target vector $\hat{f}$ and the solution vector $\hat{u}$ afterwards. Ideally, we want to keep the matrix and target vector as in (37).

As mentioned in section 5.4 the order in which one updates the values matters for Gauss-Seidel and SOR but not for Jacobi. However, we are not limited to just forward or backward Gauss-Seidel. There is one more option: the Red-Black Gauss-Seidel/SOR presented in Algorithm 1.

Algorithm 1 Red-Black SOR

```
1: while not converged do
2:     update all values corresponding to red grid-cells
3:     update all values corresponding to black grid-cells
4:     check for convergence
5: end while
```

This works because all red-cells only depend on black-cells and vise versa. Solving (37) with Red-Black SOR is entirely equivalent to solving (41) with forward SOR!

6 Implementation details and numerics

There are two flavours of parallel CPU computing architecture, shared memory and multi-processing/message passing. The former uses many logical "threads" to manipulate shared memory. To combat race conditions synchronization mechanisms are needed. For message passing the only synchronization needed is for the sharing of data, hence message passing. The most dominant standard of message passing is called MPI.
The implementation in this work achieves parallelism by using the MPI standard. MPI stands for Message Passing Interface and works by invoking the MPI run time called mpirun. The MPI run time starts several processes with the same executable, where each process have a unique process id within the MPI run time environment.

6.1 Process partitioning

One big benefit of running multiple processes is that one can split the original problem into smaller chunks and let each process solve one chunk each. To this end we must do a partition of the unknowns and equations. This partition can be done in many different ways, out of which two were considered: partition by rows and by domain decomposition.

6.1.1 By rows decomposition

The simplest way to partition a grid is by assigning unknowns and equations by rows, see Figure 4(left).
Doing this has one constraint: the number of processes must divide the grid size evenly, so that each process is assigned the same number of rows. This is important for load balancing.

6.1.2 Domain decomposition

The other way to to assign equations and unknowns is the domain decomposition, which is to assign by blocks, or tiles, instead of rows. This is illustrated in Figure 4(right).

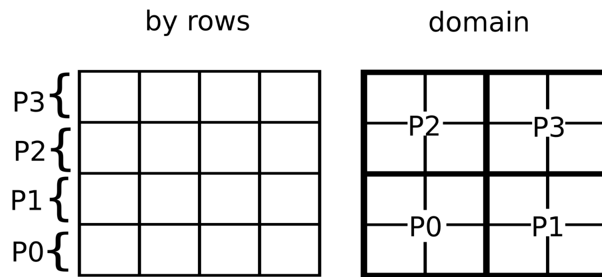

Figure 4: Partitions for a 4x4 grid.

6.2 Partitioned matrix

Since each process only has a part of the unknowns assigned to them, naturally only those equations need to be allocated in the matrix for that process. But there is a caveat: some equations involve some unknowns of a neighbouring process. The solution to this problem is to use ghost rows and ghost values (sometimes called halo rows or halo values).

6.3 Ghost rows

The number of unknowns that are involved in our assigned equations happen to be more than the number of equations. This means that the matrix would be rectangular. We want square matrices to avoid headache, therefore extra rows are added for the unknowns in the neighbouring processes. These rows are identity rows, they have only one non-zero entry at the main diagonal, which is 1.

6.4 Implementation of Red-Black Gauss-Seidel and SOR

Compared to algorithm 1 in section 5.5 there is one small adjustment needed when run in parallel: the communication step. This is because the freshly updated values are needed by the neighbouring processes. The algorithm in parallel, which will be referred to as the full version, is presented in Algorithm 2:

Algorithm 2 Full version of Red-Black SOR

```
1: while not converged do
2:     Communicate, send and receive values from the previous iteration.
3:     Update red values using only old black values.
4:     Communicate, exchange the freshly updated values with neighbouring processes.
5:     Update black values using only new red ones.
6:     Check for convergence.
7: end while
```

But, there is now a drawback: there are now two communication steps per iteration! Since communication is suspected to be slow, we want to avoid any extra communication. The solution that was found is to use the old values instead. Thus the reduced version is presented in Algorithm 3:

Algorithm 3 Reduced version of Red-Black SOR

```
1: while not converged do
2:     Communicate, send and receive values from the previous iteration.
3:     Update red values using only old black values.
4:     Update black values using mostly new red ones and a few old.
5:     Check for convergence.
6: end while
```

The reduced version is no longer a proper Gauss-Seidel or SOR since using the old values instead of the new ones from the neighbouring processes corresponds to Jacobi iterations. In a proper Gauss-Seidel or SOR half of the values are new and half are old. In the reduced version however slightly more old values are used than new values. Therefore the convergence rate is expected to be slightly lower for the reduced version compared to the full version, but in turn the reduced version is expected to be faster per iteration.

6.5 Solution to the waiting problem

When forward Gauss-Seidel or SOR is used to solve (37) in parallel, each process beyond the first need the fresh values from their neighbours before they can finish updating their own. During this time they are simply waiting. For example:

$$P_n \to P_{n-1} \to \ldots \to P_1 \to P_0,$$

means that P_1 needs updated values from P_0 and has to wait for it to finish in order to complete its own update. P_2 waits for P_1 in turn and so on. The workflow is illustrated in Figure 5. In the figures below, white indicates idle time, light grey indicates workload and dark grey indicates communication.

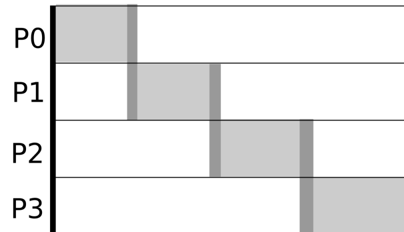

Figure 5: Workflow with waiting.

If each process does partial updates using all the old values while waiting for the new ones the workflow is improved but not satisfying. See Figure 6.

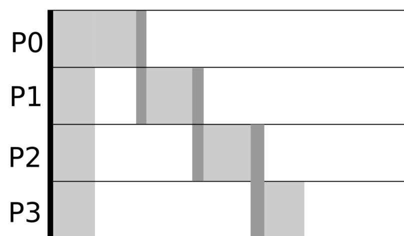

Figure 6: Improved workflow with less waiting.

With the red-black ordering we only use old values for updating the first half of the values. Therefore, each process has work to do without waiting for its neighbours to finish. In addition, each process will communicate roughly at the same time. The workflow is portrayed in Figure 7.

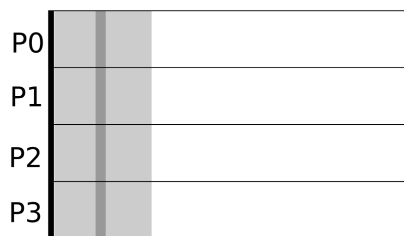

Figure 7: Workflow without waiting.

6.6 Point to point communication

Using the C implementation of the MPI standard is a hassle since the calling convention is cumbersome. In addition, keeping track of the data and the information about which process to send to and receive from is a nightmare. Therefore, a more modern object oriented implementation, which does all of the things mentioned, was provided and used.

6.7 Sparse matrix implementation

A dense matrix of the sizes encountered in this work simply wont fit into RAM of a modern system. For example, a matrix $A \in \mathbb{R}^{10^6 \times 10^6}$ needs 8TB of RAM which far exceeds the capacity of modern computers. A sparse representation only stores the non-zero entries of the matrix, reducing the memory requirement by several orders of magnitude. The storage for the sparse representation of A goes down from 8TB to 40MB.
There are a few different formats for sparse matrices, three of which are by-row storage, by-column storage and by-diagonal storage. In this work, a by-row storage implementation was provided and used, see [3] and [2] for more details. This is the ideal for cache coalescence.

6.8 Termination criterion

Every iterative process needs a stopping criterion, otherwise they would go on forever. In scientific computing the most common choice is measuring the error or residual in each step with a suitable norm and compare it with a relative tolerance, called the residual termination criterion. For example, if the chosen norm is the 2-norm and relative tolerance is 10^{-3} we get (47):

$$\frac{\|r^{(k)}\|_2}{\|r^{(0)}\|_2} = \frac{\|Ax^{(k)} - b\|_2}{\|Ax^{(0)} - b\|_2} \leq 10^{-3}. \tag{47}$$

In this work however, i chose a different one, namely the Cauchy criterion, measured in the 2-norm, with an absolute tolerance of 10^{-6} which is given by (48).

$$\|x^{(k+1)} - x^{(k)}\|_2 \leq 10^{-6}. \tag{48}$$

When two consecutive iterates differ by less than the tolerance measured in the 2-norm, the progress has halted and we stop.
The residual termination criterion has a downside as it is subject to the condition number of the matrix. From Theorem 4 we know that the condition number scales with the size of the grid making the problem more and more ill-conditioned. The Cauchy criterion does not suffer from the condition number, which is a good argument in its favour.

6.9 Initial guess

All iterative algorithms in scientific computing needs an initial guess to start the iteration process. If the initial guess is close to the correct solution, then convergence is faster, meaning fewer iterations are needed to converge, than if the initial guess was far from the solution. A good choice that is at least not harmful is to use the zero vector, 0, as initial guess. For some algorithms, for example GMRES or CG, the very first iterate generated from the 0 initial guess is the right hand side b. In such cases the initial guess should be b instead of 0 to save one iteration.
Having a better initial guess then 0 is more common in time dependent or time evolving problems where the previous state can be used as an initial guess. Our problem in this report is not such a problem. Therefore the choice of initial guess was made to be 0.

6.10 Relaxation factor

From (35) we get a formula for ω_{opt} in terms of $\rho(M_{JAC})$ for which we have an explicit expression from (39) and (38).
The relaxation factor used in all experiments was estimated by experiments and binary search for a grid size $n = 10$. The value which was found to be best was $\omega_{chosen} = 1.33$.
Sadly, the formula above was discovered after the experiments were done but the impact of this error is not so great that it is necessary to redo them.

7 Performance analysis by experiments

7.1 Main questions of the thesis

In order to measure and compare performance we need to know what effects on the execution time certain parameters have. This leads us to the main questions we want to answer in this report, which are:

1. What is the effect of bigger grid size?
2. What is the effect of more processes?

3. What is the effect of full compared to reduced versions?
4. What is the effect of domain decomposition compared to row partitioning?
5. How fast are these to apply when used as preconditioners?

7.2 Method

To answer the main questions several experiments were designed and carried out. There are two versions of the program, one using the domain decomposition and the other using row partitioning.
To answer question 1: run the two programs with varying grid size inputs.
To answer question 2: run the two programs with varying process count.
To answer question 3: run the two programs with both full and reduced versions of the implementation.
To answer question 5: run the two programs with fixed grid size, vary the process count and implementation version. Stop after a fixed number of iterations.

To automate this ordeal, several shell scripts were designed.

7.3 POWER8

The experiments were performed on the POWER8 server at Lunds Tekniska Högskola, LTH. The CPU of this machine has 10 cores with 8 hardware threads each, and a clock frequency of 3.5 GHz. The system has 15GB of RAM and runs Ubuntu 18.

7.4 Experiment results, discussion and observations

In the plots to come some naming declarations needs to be made: the first letter indicates which method it is: J for Jacobi, G for Gauss-Seidel and S for SOR. The second letter indicates which partitioning scheme was used: R means row partitioning and D means domain decomposition.

7.4.1 Question 1

Bigger grids yields bigger matrices and therefore more workload per iteration, taking more time. Compare the 10 process charts in Figure 8 and 9, we see an increase from less then 1 second to about 220 seconds for Jacobi and from 0.3 seconds to about 100 seconds for SOR.

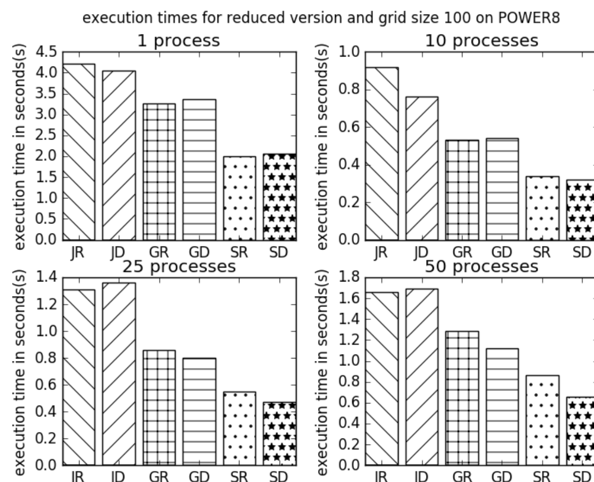

Figure 8: Execution times for reduced version, grid size 100. See section 7.4 for an explanation of the labels.

But the workload per iteration only increased by 25 times, while the time increase was 250 times for Jacobi and 300 times for SOR. The discrepancy comes from the increase in iteration count needed to converge, as can be seen in Figures 10 and 11, growing by a factor 20 for Jacobi and SOR.

From section 4.2.5 we got a theoretical estimate of the relative number of iterations needed for convergence between Jacobi, Gauss-Seidel and SOR. There, it was found that Jacobi should need twice as many iterations as Gauss-Seidel. Looking at Figure 11 we can see that the experimental results are close to the theoretical

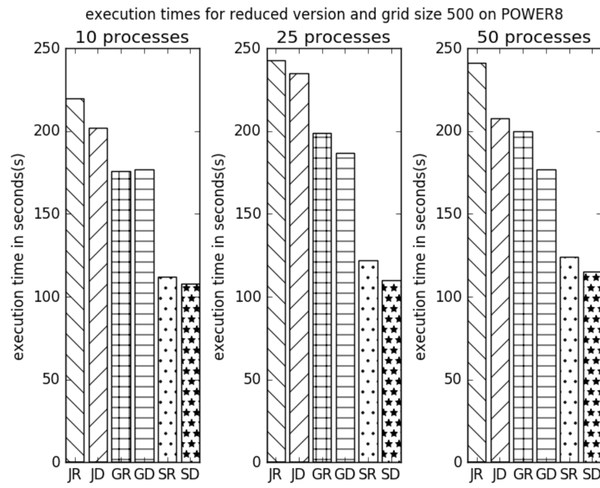

Figure 9: Execution times for reduced version, grid size 500. See section 7.4 for an explanation of the labels.

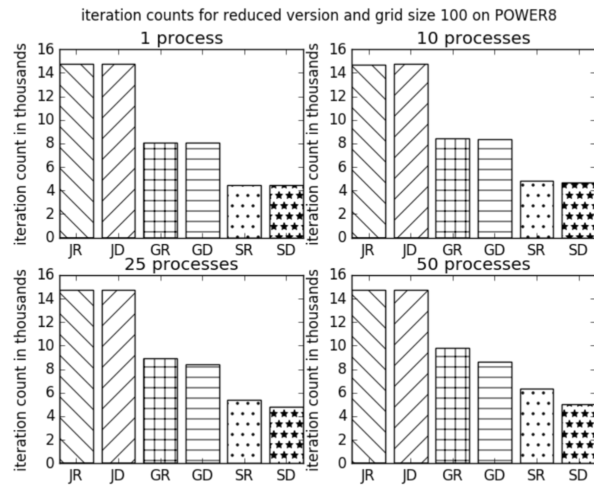

Figure 10: Iteration counts for the reduced version, grid size 100. See section 7.4 for an explanation of the labels.

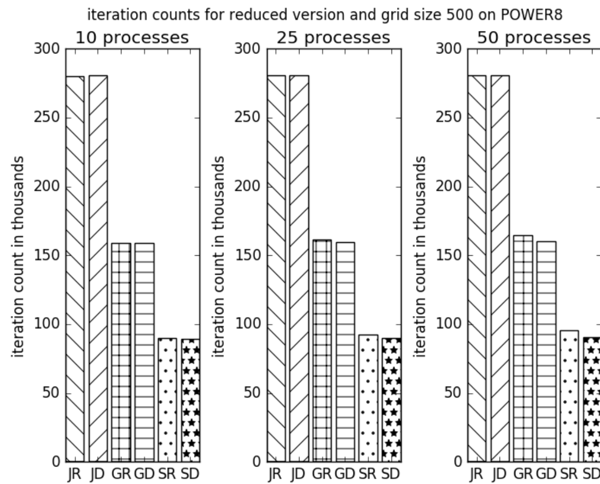

Figure 11: Iteration counts for the reduced version, grid size 500. See section 7.4 for an explanation of the labels.

estimate. The estimate for SOR was much harder to make, only concluding that it should be smaller. Looking at Figure 11 again, we can see that it is indeed smaller than Gauss-Seidel.

7.4.2 Question 2

For this experiment a grid size of 250 was used. In Figure 12 we see a speed up of around 10 times when 1 process is compared to 10 processes. This is what we expect when the problem is big enough. Further increasing the process count to 25 or 50 does not improve performance much since the problem is yet too small to benefit from them and the communication is now a much bigger part of the execution time.

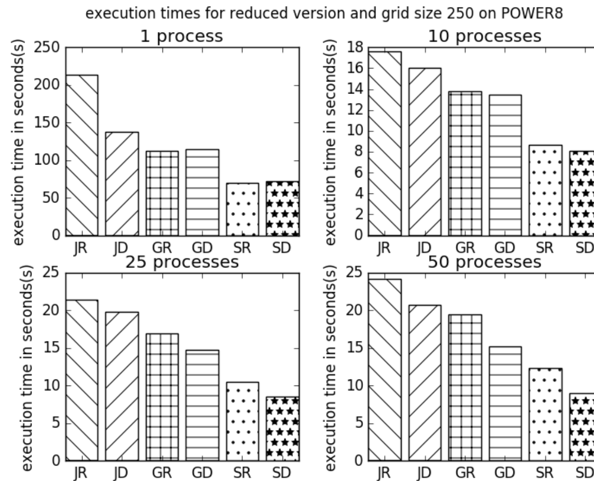

Figure 12: Execution times for reduced version, grid size 250. See section 7.4 for an explanation of the labels.

7.4.3 Question 3

Figures 13 and 9 contains execution times when a grid size of 500 was solved with the reduced and full versions respectively. Comparing Figure 13 and 9 we see no noticeable difference. This means that the extra communication is not a big part of the bottleneck for big enough problems.

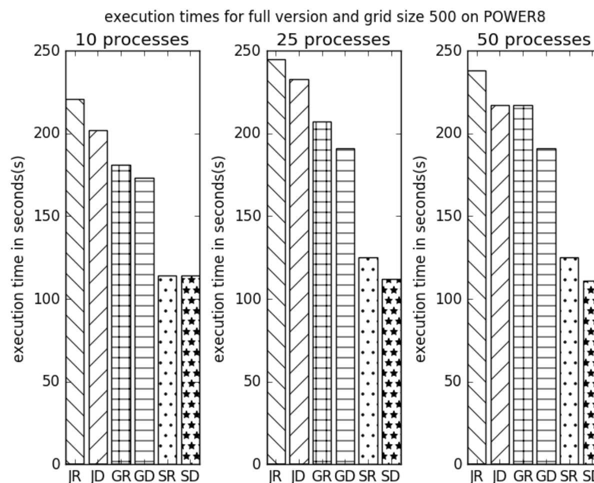

Figure 13: Execution times for full version, grid size 500. See section 7.4 for an explanation of the labels.

7.4.4 Question 4

looking at Figures 8, 9, 12 and 13 we see that the domain decomposed version is slightly faster across the board. The reason should be that the communication is faster that way. As for the reason why, a qualified guess would be that the communication is less congested.

7.4.5 Question 5

The number of iterations for this experiment was 100. In Figure 14 we see that Jacobi is much faster than Gauss-Seidel and SOR. In appendix A section 9.3 we see that the reduced version is faster than the full version, although not by much.
One more interesting observation was that the residual norm after the 100 iterations were all around 10^{-3}, which means that Jacobi would be the best preconditioner. This is not the full story though, since the convergence rate is fast early on, but Jacobi makes less progress the closer it gets to the solution. Maybe an adaptive scheme could be the winner here.

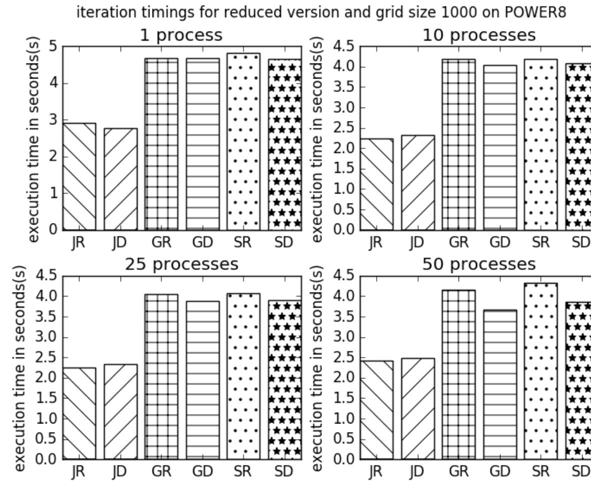

Figure 14: Iteration timings for reduced version. See section 7.4 for an explanation of the labels.

8 Conclusion and continued work

We started of with Poisson's equation and realised that solving it exactly was too difficult. Instead we needed to solve it approximately. A grid was introduced in order to approximate the problem with a linear system of equations. Some necessary properties for the existence and uniqueness of solutions to the linear system were stated.

Next, we investigate the different options we have of solving the systems. Direct methods are ruled out due to instability, ill-conditionedness and memory constraints. Iterative methods like Multi-grid, CG and GMRES are good options but need some help in the form of precoditioners. The stationary linear iterative methods are investigated thoroughly because they can both solve the systems alone and be used as preconditioners for the other iterative methods.

The aim is to solve the systems faster than sequential by doing a parallel implementation. To test the performance we need a reproducible test problem. The solvability of the test problem was established. A problem with coupling is introduced when Gauss-Seidel and SOR are applied to the test problem. The remedy that was found is to use a different order, namely the Red-Black order.

The choice of parallel architecture was MPI since the test problem was well suited for it. Two different process partition schemes were investigated. For the Red-Black SOR implementation we notice that two communication steps per iteration is required. Suspecting that communication is slow we find an option is to use old values instead of the second communication step. Both of these versions are investigated.

We see why the coupling is a problem and how the Red-Black order solves the problem.
To measure the performance and answer the questions that are stated, several experiments were described and the results are presented.
Looking at the results we see that larger problems take longer so solve, no surprises there. The reason is partly due to increased work load and partly reduced convergence rate. Increasing the process count is not always faster, especially for small problems. Doing the extra communication step takes longer per iteration but compensate with fewer iterations to converge, ending up about the same as using old values. using domain

decomposition is slightly faster than row partitioning.

Overall SOR is faster than Gauss-Seidel which is faster than Jacobi. SOR and Gauss-Seidel does almost exactly the same amount of work per iteration but SOR has better convergence rate. Jacobi is faster per iteration but the reduced convergence rate does not compensate fully.

For the purpose of preconditioning, an adaptive scheme might be the better choice. I propose:

1. Use Jacobi until the residual norm is smaller than some threshold.

2. Then switch to SOR.

This scheme takes advantage of the speed of Jacobi early when a lot of progress can be made and the extra precision of SOR in the final phase.

The next frontier in scientific computing is to use GPUs instead of CPUs for parallel workloads, which I propose for continued work. A GPU(graphics processing unit) is a hardware unit with far superior parallel processing capabilities compared with CPU's, nowadays over 10000 parallel compute units per chip for high end cards such as RTX 3090 compared to lower 100's for high end CPU's(counting several CPU's on the same motherboard).
Something else worth investigating are other stencils, say skewed 5-point or 9-point. In the case of the 9-point stencil, the resulting equations are even more coupled than the 5-point one. This leads to the study of graph coloring problems for the adjacency graph of the stencil to find another order to solve the equations, since the red-black order discussed in this report won't work for it.

9 Appendix A: Result tables

9.1 Reduced version on POWER8

execution time in seconds (s)

grid size \ processes	1	10	25	50
JAC rows				
100	4.2	0.9	1.3	1.7
250	213	17.6	21.3	24.2
500	-	220	243	241
JAC domain				
100	4.0	0.8	1.4	1.7
250	137	16.0	19.8	20.7
500	-	202	235	208
GS rows				
100	3.3	0.5	0.9	1.3
250	112	13.8	17.0	19.5
500	-	176	199	200
GS domain				
100	3.4	0.5	0.8	1.1
250	114	13.5	14.7	15.2
500	-	177	187	177
SOR rows				
100	2.0	0.3	0.5	0.9
250	69	8.6	10.5	12.3
500	-	112	122	124
SOR domain				
100	2.1	0.3	0.5	0.7
250	72	8.1	8.5	9.0
500	-	108	110	115

iteration count in thousands (k)

grid size \ processes	1	10	25	50
JAC rows				
100	15	15	15	15
250	79	79	79	79
500	-	280	281	281
JAC domain				
100	15	15	15	15
250	79	79	79	79
500	-	281	281	281
GS rows				
100	8.1	8.4	9.0	9.8
250	44	45	46	48
500	-	159	161	165
GS domain				
100	8.1	8.3	8.4	8.6
250	44	45	45	45
500	-	159	159	160
SOR rows				
100	4.4	4.8	5.4	6.3
250	24	25	26	29
500	-	90	92	96
SOR domain				
100	4.4	4.7	4.8	5.0
250	24	25	25	26
500	-	89	90	91

9.2 Full version on POWER8

execution time in seconds (s)

grid size \ processes	1	10	25	50
JAC rows				
100	4.2	0.9	1.2	1.7
250	213	17.8	21.7	24.4
500	-	221	245	238
JAC domain				
100	4.1	0.9	1.4	1.7
250	137	16.0	20.2	20.1
500	-	202	233	217
GS rows				
100	3.3	0.6	1.0	1.5
250	112	15.1	19.1	26.0
500	-	181	207	217
GS domain				
100	3.1	0.6	1.1	1.4
250	105	13.5	16.6	17.6
500	-	173	191	191
SOR rows				
100	2.0	0.4	0.5	0.8
250	70	9.7	11.1	16.9
500	-	114	125	125
SOR domain				
100	1.9	0.4	0.6	0.8
250	66	8.5	9.7	10.6
500	-	114	112	111

iteration count in thousands (k)

grid size \ processes	1	10	25	50
JAC rows				
100	15	15	15	15
250	79	79	79	79
500	-	280	281	281
JAC domain				
100	15	15	15	15
250	79	79	79	79
500	-	281	281	281
GS rows				
100	8	8	8	8
250	44	46	44	52
500	-	158	158	158
GS domain				
100	8	8	8	8
250	44	44	44	46
500	-	159	158	158
SOR rows				
100	4	4	4	4
250	24	26	24	33
500	-	88	88	88
SOR domain				
100	4	5	5	5
250	24	25	25	26
500	-	89	89	89

9.3 Iteration timings on POWER8

The time it took to do 100 iterations with grid size 1000.

Reduced. in seconds (s)

grid size \ processes	1	10	25	50
JAC rows	2.9	2.2	2.2	2.4
JAC domain	2.8	2.3	2.3	2.5
GS rows	4.7	4.2	4.0	4.2
GS domain	4.7	4.0	3.9	3.7
SOR rows	4.8	4.2	4.1	4.3
SOR domain	4.7	4.1	3.9	3.9

Full. in seconds (s)

grid size \ processes	1	10	25	50
JAC rows	2.9	2.4	2.3	2.6
JAC domain	2.8	2.3	2.3	2.3
GS rows	4.7	4.6	4.1	4.8
GS domain	4.6	4.1	4.0	4.3
SOR rows	5.0	4.5	4.1	4.8
SOR domain	4.7	4.1	4.0	4.4

10 Appendix B: source code

Below is the source code for the full version of Red-Black SOR.
The complete source code is publicly available at: https://gitlab.com/Drunte/public-sor.

C++ code

```
1  #include <iostream>
2  #include <vector>
3  #include "sor_mpi.h"
4  #include "dune_pch.h"
5  #include "bundle.h"
6  using std::cout;
7  using std::endl;
8  typedef Dune::BCRSMatrix< double > Matrix;
9  typedef Dune::BlockVector< double > Vector;
10 typedef Dune::SimpleMessageBuffer MsgBuffer;
11 typedef Dune::Point2PointCommunicator< MsgBuffer > ptpComm;
12
13 Vector sor_rb_mpi(const Matrix &A, const Vector &b, const Vector &x, Bundle &bundle, double
       relaxation_factor, double tol) {
14   Vector previous(x.size()); //zeros
```

```
15   Vector next(x);  //copy of initial guess. we need 2 buffers for the Cauchy termination
       criteria.
16
17   ptpComm& comm = bundle.comm; //communicator
18   auto& ghostbools = bundle.ghostbools; //vector<bool> with False for ghostrows
19
20   int iter = 0; //iteration counter
21   double termination_criteria;
22   do {
23     iter++; //increase iteration counter
24
25     //communication stage. send and recieve the most recently updated values.
26     bundle.sendAndRecieveUpdate(next);
27
28     //swap values such that previous has the values from the previous iteration.
29     next.swap(previous);
30
31     //algorithm
32     //first pass, update red values.
33     auto endrow = A.end();
34     for (auto row = A.begin(); row != endrow; ++row) {
35       const size_t rowindex = row.index();
36       if (bundle.is_black(rowindex)){  //skip updating black values.
37         continue;
38       }
39       double sum = b[rowindex]; //starting with b and subtracting later saves 1 flop.
40       auto endcol = (*row).end();
41       for (auto col = (*row).begin(); col != endcol; ++col) {
42         const size_t colindex = col.index();
43         if (rowindex != colindex) {
44           sum -= (*col) * previous[colindex];
45         }
46       }
47       //update scheme for SOR.
48       next[rowindex] = previous[rowindex] + relaxation_factor * (sum /
     A[rowindex][rowindex] - previous[rowindex]);
49     }
50
51     //extra communication step. send and recieve red values.
52     bundle.sendAndRecieveUpdate(next);
53     //second pass. update black values.
54     for (auto row = A.begin(); row != endrow; ++row) {
55       const size_t rowindex = row.index();
56       if (bundle.is_red(rowindex)) { //skip the red values this time.
57         continue;
58       }
59       double sum = b[rowindex];
60       auto endcol = (*row).end();
61       for (auto col = (*row).begin(); col != endcol; ++col) {
62         const size_t colindex = col.index();
63         if (rowindex != colindex) {
64           sum -= (*col) * next[colindex];
65         }
66       }
67       next[rowindex] = previous[rowindex] + relaxation_factor * (sum /
     A[rowindex][rowindex] - previous[rowindex]);
68     }
69
70     double partialsum = norm_of_diff_square(next, previous, ghostbools); //compute the
     2-norm squared of the difference, ignore ghost-values.
71     //compute termination criteria
72     termination_criteria = comm.sum(partialsum);  //sum all partial sums.
73   } while (termination_criteria > tol*tol); //2-norm < tol is equvalent to 2-norm squared <
       tol squared.
74
75   bundle.sendAndRecieveUpdate(next);  //do the final update
76
77   if (comm.rank() == 0) {
78     cout << "SOR RB converged in " << iter << " iterations" << endl;  //print info
79   }
80   return next;  //return the final iterate
81 }
```

References

[1] Praveen Agarwal, Mohamed Jleli, and Bessem Samet. *Banach Contraction Principle and Applications*, pages 1–23. Springer Singapore, Singapore, 2018.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. *Computing*, 82(2–3):103–119, 2008.

[3] M. Blatt and P. Bastian. The iterative solver template library. In B. Kagström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing – State of the Art in Scientific Computing*, pages 666–675, Berlin/Heidelberg, 2007. Springer.

[4] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial, Second Edition.* Society for Industrial and Applied Mathematics, second edition, 2000.

[5] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.).* Johns Hopkins University Press, USA, 1996.

[6] Wolfgang Hackbusch. *Iterative Solution of Large Sparse Systems of Equations (2nd ed.)*, volume 95. 06 2016.

[7] Wolfgang Hackbusch. *Elliptic Differential Equations: Theory and Numerical Treatment*, volume 78. 07 2017.

[8] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations.* Society for Industrial and Applied Mathematics, 1995.

[9] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, second edition, 2003.

[10] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra.* SIAM, 1997.