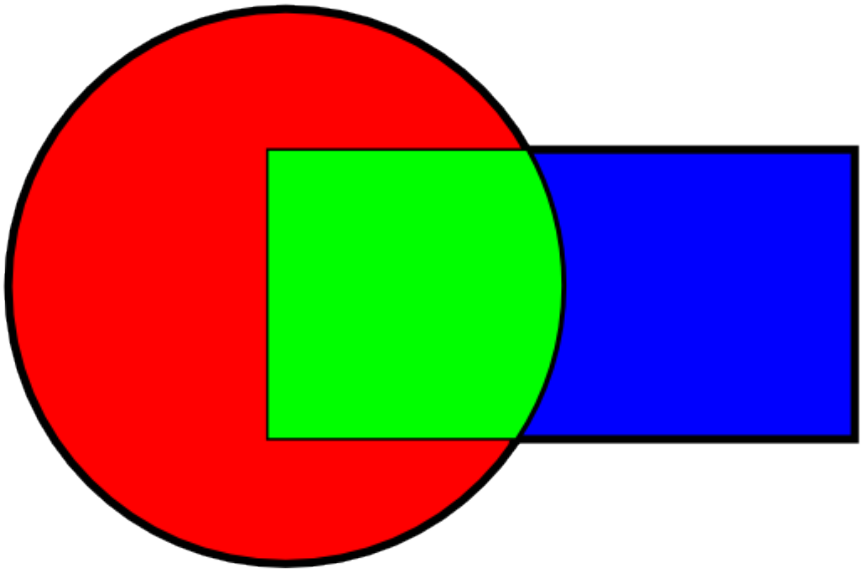
Leveraging Parallel Schwarz Domain Decomposition

Using node level parallelism for the implementation of the parallel Schwarz method



Karel Jan Gimbergh

Supervisor:
Alexander Heinlein

TU Delft

Second corrector:
Bart van den Dries

Abstract

This thesis concerns the implementation of parallel Schwarz domain decomposition using node-level parallelism, focusing on the parallel Schwarz method in comparison with the Jacobi iterative method. The study goes into the complexities of domain decomposition methods for solving partial differential equations, which are essential in fields such as fluid dynamics, solid mechanics, quantum mechanics, and financial mathematics. The research examines the convergence and performance of these methods within a parallel computing framework. A large portion of the work involves the comparison of varying configurations of block sizes and overlaps within the use of the block Jacobi iterative method, used for the implementation of the parallel Schwarz method. Numerical experiments are conducted for a stationary heat problem on a 2-dimensional grid with 256 points in each direction. The results show optimal performance for small block sizes, attributed to the use of a dense solver for the subdomains. Larger blocks and larger overlaps show superior convergence properties, up to the limit of an overlap of half the block size. The efficiency of the parallel Schwarz method remains high for an increasing number of threads unlike the standard Jacobi iteration, showing it is better suitable to a parallel environment.

Contents

Abstract ii

1 Introduction 1

2 Fundamental Concepts and Problem Formulation 2

2.1 Problem Formulation . . . 2

2.2 Schwarz Methods . . . 4

2.2.1 Alternating Schwarz Method . . . 4

2.2.2 Parallel Schwarz Method . . . 5

2.3 Node Level Parallelization . . . 5

2.3.1 Nodes and Cores . . . 5

2.3.2 Types of Parallelism . . . 6

2.3.3 Evaluating Parallel Performance . . . 7

3 Methodology 8

3.1 Iterative Methods . . . 8

3.1.1 Jacobi Iteration . . . 9

3.1.2 Block Jacobi Iteration . . . 9

3.1.3 Block Jacobi Iteration With Overlap . . . 11

3.2 Convergence of Iterative Methods . . . 12

4 Implementation and Software 15

4.1 Functions and Data Types . . . 15

4.1.1 Functions . . . 15

4.1.2 Data Types . . . 16

4.2 Parallel Implementation of Iterative Methods . . . 17

4.2.1 Jacobi Iteration . . . 17

4.2.2 Block Jacobi Iteration (with Overlap) . . . 18

5 Numerical Results 21

5.1 Convergence Results . . . 21

5.1.1 Theoretical vs Experimental Convergence . . . 21

5.1.2 Influence of Block Size and Overlap . . . 22

5.2 Parallel Performance . . . 23

6 Discussion 25

7 Conclusion 26

A Additional Numeric Results 29

A.1 Convergence results. . . . 29

A.2 Parallel performance results . . . 30

1 Introduction

Within the field of high-performance computing, domain decomposition methods (DDMs) are a widely adopted approach for solving partial differential equations (PDEs). These methods involve partitioning the computational domain into subdomains, allowing for parallel execution. DDMs employ diverse linear algebra techniques to solve PDEs on parallel architectures, tackling problems derived from various scientific domains, including the Navier-Stokes equations in fluid dynamics, elasticity systems in solid mechanics, Schrödinger equations in quantum mechanics or the Black and Scholes equations in financial mathematics [7]. The complexity of these equations often calls for large-scale computations, driven by the increasing demand for precision and the expanding availability of computational resources.

This thesis focuses on Schwarz methods, the earliest form of DDMs. The alternating Schwarz method, introduced by mathematician Hermann Amandus Schwarz in 1869 [21], utilizes the decomposition of a domain into overlapping subdomains, iteratively solving the differential equations within these subdomains. The uprise of parallel computing sparked renewed interest in this method, resulting in the development of the parallel Schwarz method by Lions in 1988, allowing for large-scale parallelization [9][11]. This research provides a comprehensive study on the convergence and performance of the parallel Schwarz method in comparison to the Jacobi iterative method, within the Kokkos parallel computing framework [23], focusing on node-level parallelism. The aim is to align theory with numerical experiments, and test to which extent the parallel Schwarz method allows for better use of computational resources.

The structure of this thesis is as follows: Chapter 2 offers preliminary information, starting with a formulation of the problem used for convergence and performance analysis, followed by a formulation of both the alternating- and parallel Schwarz method, and concluding with an introduction into node-level parallelization. Chapter 3 showcases the methodology of applying the parallel Schwarz method by relating it to the block Jacobi iterative method, after introducing the regular Jacobi iterative method which will be the point of comparison. This chapter concludes with an analysis of the convergence properties of these methods. Chapter 4 discusses several software routines used for the (parallel) implementation of the previously presented methods. The numerical results of this implementation are presented in Chapter 5, starting with experimental convergence results, followed by the parallel performance. Finally, Chapter 7 presents the conclusions drawn from this study, complemented by a reflection and suggestions for further research in Chapter 6.

2

Fundamental Concepts and Problem Formulation

Before going into the analysis of this thesis, several foundational concepts must be introduced. We begin by defining the differential problem to be solved in Section 2.1. Following this, Section 2.2 addresses domain decomposition, discussing two Schwarz methods: the alternating and the parallel Schwarz method. This chapter concludes with a brief introduction to node-level parallelization in Section 2.3.

2.1. Problem Formulation

The differential problem used in this thesis is a stationary heat equation in two dimensions with constant thermal conductivity, defined on the square $\Omega = [0,1]^2 \subset \mathbb{R}^2$ with Dirichlet boundary conditions:

$$\begin{cases} \Delta u(\mathbf{x}) = -\dfrac{Q(\mathbf{x})}{k} & \text{on } \Omega, \\ u = 0 & \text{on } \partial\Omega, \end{cases}$$

where $u(\mathbf{x})$ is the temperature at position $\mathbf{x} = (x, y) \in \Omega$, $Q(\mathbf{x})$ is the generated heat inside the domain at position $\mathbf{x} = (x, y) \in \Omega$, k is the thermal conductivity inside the domain, taken to be constant, and $\partial\Omega$ is the boundary of Ω.

As k is taken to be constant, we can simplify the expression $-\frac{Q(\mathbf{x})}{k} = b(\mathbf{x})$, resulting in the following form:

$$\begin{cases} \Delta u(\mathbf{x}) = b(\mathbf{x}) & \text{on } \Omega, \\ u = 0 & \text{on } \partial\Omega. \end{cases} \tag{2.1}$$

For simplicity, we choose a forcing function b that vanishes at the boundaries of the domain and for which the solution of problem (2.1) is easily derived:

$$b(\mathbf{x}) = -2\pi^2 \cdot \sin(\pi x) \cdot \sin(\pi y). \tag{2.2}$$

which implicates $u(\mathbf{x}) = \sin(\pi x)\sin(\pi y)$ as the solution, depicted in Figure 2.1.

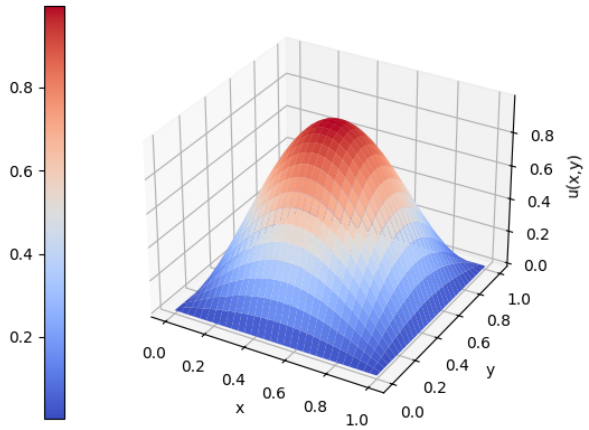


Figure 2.1: Exact solution of stationary heat equation (2.1) with forcing function (2.2).

Next, we discretize the domain, and with that, the differential problem. We use an equidistant discretization grid of the unit square $[0,1]^2 \subset \mathbb{R}$ using $n+2$ discretization points in each direction, resulting in the nodes:

$$(x_i, y_j) = (h \cdot i, h \cdot j), \qquad h = \frac{1}{n+1}, \qquad \text{for } i, j \in \{0, 1, \ldots, n, n+1\}.$$

Recalling problem (2.1) and noting that the boundary values are known, there are n unknowns in both directions for a total of n^2 unknown values. We denote these unknowns by $u_{i,j} = u(x_i, y_j)$. To approximate the Laplacian of problem (2.1), we use a 5-point central difference scheme which has a discretization error of $\mathcal{O}(h^2)$, see [2]:

$$(\Delta u)_{i,j} = \frac{u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1}}{h^2} + \mathcal{O}(h^2).$$

This discretization allows us to formulate the differential problem (2.1), at each internal node (x_i, y_j) as:

$$\frac{u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1}}{h^2} = b_{i,j} \qquad \text{for } i, j \in \{1, \ldots, n\}.$$

Considering that $u_{i,j} = 0$ for $i, j \in \{0, n+1\}$, and writing the entries of $\mathbf{u}$ and b in lexicographic order, we derive the linear system:

$$A\mathbf{u} = \mathbf{b}, \tag{2.3}$$

where $A \in \mathbb{R}^{n^2 \times n^2}$, $\mathbf{u}, \mathbf{b} \in \mathbb{R}^{n^2}$, and the entries of the sparse matrix A are given by

$$A = \frac{1}{h^2} \begin{bmatrix} B & I & & & \\ I & B & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & B & I \\ & & & I & B \end{bmatrix}, \qquad B = \begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 1 \\ & & & 1 & -4 \end{bmatrix}. \tag{2.4}$$

where $I, B \in \mathbb{R}^{n\times n}$, and I is the identity matrix. This matrix A will be referred to as the 2D Laplacian. We can see that with $n+2$ discretization points in both directions, we get a linear system (2.3) of n^2 equations, resulting in an $n^2 \times n^2$ matrix. When the solution is required at many grid points, this system quickly becomes very large, making direct solutions for (2.3) infeasible. In the following sections, an iterative method will be discussed to decompose this problem into smaller sub-problems.

2.2. Schwarz Methods

As mentioned in the introduction, Schwarz methods, named after Hermann Amandus Schwarz, are the earliest domain decomposition methods. These methods form the foundation for many of the domain decomposition techniques used in numerical analysis and computational science. The alternating Schwarz method, one of the earliest of these methods, was initially developed to rigorously validate results established by Bernhard Riemann. Riemann had proven that a harmonic function, which is a solution of the Laplace equation $\Delta u = 0$ on a bounded domain Ω with Dirichlet boundary conditions, is the infimum of the Dirichlet integral:

$$\int_\Omega |\nabla v|^2 dx$$

over all functions v satisfying the boundary conditions. While the Dirichlet principle had readily been proven for simple domains on which Fourier analysis was possible, Schwarz developed a domain decomposition method to extend this to more complicated domains. [9]

2.2.1. Alternating Schwarz Method

We describe the alternating Schwarz method for the domain presented in Figure 2.2. This domain Ω consists of two overlapping subdomains: a circle Ω_1' and a rectangle Ω_2'. Here $\Omega_1 = \Omega \setminus \Omega_2'$, $\Omega_2 = \Omega \setminus \Omega_1'$, and the intersection of these domains is $\Omega_3 = \Omega_1' \cap \Omega_2'$. The boundaries of Ω_1 and Ω_2 that lie in the interior of Ω are denoted as $\Gamma_1 = \partial\Omega_1' \cap \Omega_2'$ and $\Gamma_2 = \partial\Omega_2' \cap \Omega_1'$, respectively, where ∂ denotes the boundary of a domain.

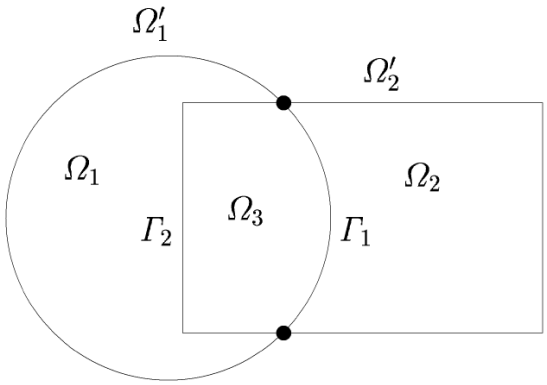


Figure 2.2: Overlapping partition for the Schwarz alternating method with two subdomains [22]

When solving problem (2.1) using the alternating Schwarz method, the differential problem is iteratively solved on subdomains Ω_1' and Ω_2', where on each iteration, the solution is updated within one of the two subdomains, followed by the other. The method starts with an initial guess u^0 on the whole domain that vanishes on the boundary $\partial\Omega$. On each iteration of the method, the values of the interior of one domain, say Ω_1' without loss of generality, are used as boundary values of Γ_2, which is subsequently used to solve the differential problem on Ω_2'. We use problem (2.1) together with the formulation of the method in [22] to denote the alternating Schwarz method as follows:

$$\begin{cases} \Delta u^{k+\frac{1}{2}} = b & \text{in } \Omega_1' \\ u^{k+\frac{1}{2}} = 0 & \text{on } \partial\Omega_1' \cap \partial\Omega \\ u^{k+\frac{1}{2}} = u^k & \text{on } \Gamma_1 \end{cases} \quad \begin{cases} \Delta u^{k+1} = b & \text{in } \Omega_2' \\ u^{k+1} = 0 & \text{on } \partial\Omega_2' \cap \partial\Omega \\ u^{k+1} = u^{k+\frac{1}{2}} & \text{on } \Gamma_2 \end{cases}. \tag{2.5}$$

The iterations of this method proceed until a certain convergence criterion is met, either based on a known exact solution or by an approximation of the error. The specific metric for determining convergence in this study will be discussed in Section 3.2. The method presented above is inherently sequential as the values of $u^{k+\frac{1}{2}}$ are used in the process of updating u^{k+1}. It thus does not allow for parallelization on the level of the two subdomains. To enable this, we consider a variation on the alternating Schwarz method, the parallel Schwarz method.

2.2.2. Parallel Schwarz Method

To be able to use parallel computing when using a Schwarz method, Lions introduced the parallel Schwarz method [11]. This is a modification to the alternating Schwarz method where the differential problem is solved on each subdomain simultaneously. Both internal boundaries Γ_1 and Γ_2 are initialized an initial guess u_0 on Ω. The problem is solved on both subdomains, with the values of Γ_1 and Γ_2 being exchanged between the subdomains after each iteration. This is concretely formulated by Jovilet [15] as:

$$\begin{cases} Du_1^{k+1} = b & \text{in } \Omega_1' \\ u_1^{k+1} = g & \text{on } \partial\Omega_1' \cap \partial\Omega \\ u_1^{k+1} = u_2^k & \text{on } \Gamma_1 \end{cases} \quad \begin{cases} Du_2^{k+1} = b & \text{in } \Omega_2' \\ u_2^{k+1} = g & \text{on } \partial\Omega_2' \cap \partial\Omega \\ u_2^{k+1} = u_1^k & \text{on } \Gamma_2. \end{cases} \tag{2.6}$$

This method allows for full parallelization on the level of the subdomains as the boundary updates per domain are independent on each iteration. This is especially beneficial when using multiple overlapping subdomains, which allows information to be shared between multiple subdomains simultaneously. The multiple overlapping regions allow the information of the known boundary to spread more quickly through the domain, improving the convergence of the method.

2.3. Node Level Parallelization

The basis of node-level parallelization is to divide up a large computational problem into smaller parts, called tasks. Each of these tasks is a portion of the total problem that can be separately executed on a computational core. In this section, we further explain the concepts of cores, nodes and tasks at the hand of the work of [13], and highlight critical aspects that have to be addressed when using node-level parallelization.

2.3.1. Nodes and Cores

A node typically refers to one single physical or virtual machine within a larger system of computational machines, see Figure 2.3. Key components of a node which are essential for understanding node-level parallelization include random access memory (RAM) to temporarily store data, and a central processing unit (CPU) composed of multiple computational cores. Each of these cores can perform one or more tasks, which are scheduled as threads. A thread is the smallest unit of processing that can be scheduled by a computer. They are lightweight processes that are executed independently but can communicate through the use of their shared memory.

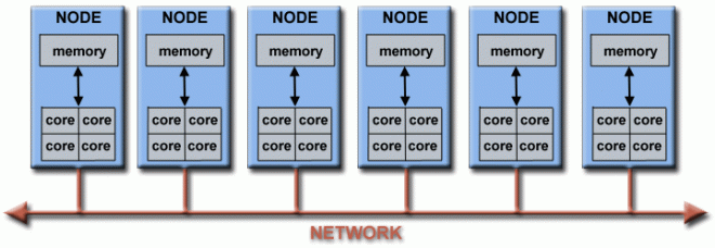


Figure 2.3: A schematic of several nodes in a larger parallel computer cluster [18]

2.3.2. Types of Parallelism

According to [13], there are two distinct types of parallelism when using multiple cores core computer nodes:

1. **Data parallelism:** The computational problem involves processing a large amount of data. This data can be split up into different parts, which are then processed by separate cores, or distributed over multiple nodes.

2. **Functional parallelism:** The computational problem can be subdivided into smaller tasks which are independent or slightly overlapping. Each core is assigned a specific task that is performed on a separate portion of the data. Due to the possible varying computational complexities of these tasks, the challenge arises of efficiently scheduling and distributing these tasks over the available cores to optimize performance.

In this thesis, only data parallelism will be used. Both of these types of parallelism can be employed using only one node, using node-level parallelism, or using a network of multiple nodes, using distributed parallelism. When using distributed parallelism, communication is needed between the different nodes, as they typically do not share memory. This communication induces added execution time, or overhead, which is avoided when using node-level parallelism. Communication between the different cores in a node happens through the shared memory to which all of the cores have access. However, the restriction to one node implies fewer computational resources in comparison to distributed parallelism. The choice between node-level and distributed parallelism is highly dependent on the problem at hand.

The goal of node-level parallelism is to launch threads and distribute the workload over these threads as efficiently as possible. A program usually starts with a master thread, which runs immediately from the start of a program. When the program gets to a parallel region, a team of threads is launched to perform a computation, which is called forking, see 2.4. This forking process takes time per thread, making it so that using more threads does not necessarily mean a better performance of the parallel program. The distribution of work over the threads can either explicitly be done or is done automatically using a scheduler, depending on the parallel framework that is used.

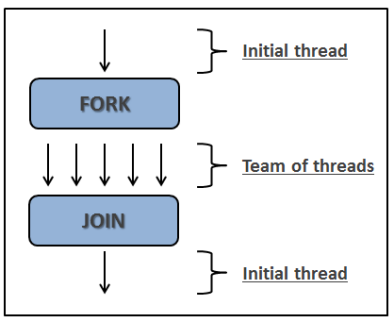


Figure 2.4: The fork-join model showing the creation of threads.

2.3.3. Evaluating Parallel Performance

Using node-level parallelism can significantly reduce the execution time of a program. To evaluate parallel performance, one must use a metric to compare it to its sequential counterpart. In this thesis, we will use parallel efficiency, as discussed in [13]. Consider a task that takes T seconds to complete sequentially. When executed using N threads, this same task ideally takes T/N seconds, suggesting a speedup of N. However, this perfect speedup is often not obtained in applications due to several factors:

1. **Load Imbalance:** Threads might not execute their tasks simultaneously because the task is not divided into pieces of equal complexity. This imbalance can result in threads having to wait for others to get to the end of the parallel region. This can significantly harm performance when the load imbalance is large.
2. **Resource Contention:** Some resources might be shared and can only be accessed by one thread at a time, making parts of the task sequential.
3. **Communication Overhead:** Parallel tasks can require communication between threads, causing overhead.

The scalability of a task is measured using a metric based on its serial and parallelizable parts. The overall problem size is $s+p$, where s is the serial part and p is the perfectly parallelizable part of the problem. Serial time using one worker is given by:

$$T_s = s + p.$$

Using N workers, the execution time is

$$T_p = s + \frac{p}{N}.$$

This is called strong scaling. Application speedup is defined as the ratio of serial time to the parallel time for a fixed problem size:

$$S(N) = \frac{T_s}{T_p} = \frac{1}{s + \frac{1-s}{N}}$$

which is known as Amdahl's Law, first introduced by Gene Amdahl in 1967 [3]. This law implies that the speedup is limited by $\frac{1}{s}$ as $N \to \infty$. The efficiency of a parallel performance is the ratio of the performance using N threads to N times the performance using one thread, expressed as:

$$\epsilon = \frac{S(N)}{N} = \frac{1}{s(N-1)+1}.$$

This metric assesses the efficiency of using N threads for executing a parallel program. Now as $s(N-1) \geq 0$, we get that the efficiency of a parallel program should be bounded by 1. However, superlinear speedup, where the speedup exceeds the number of cores used can occur. This can happen when the fractions of the data distributed to each core fit better into the local caches of the cores [19]. As the data is split into smaller parts when using more threads, it can happen that these threads can load the data into their local cache more efficiently, resulting in superlinear speedup.

3

Methodology

As discussed in the previous chapter, the parallel Schwarz method involves simultaneously solving a differential problem on multiple subdomains. Using a finite difference discretization to approximate the solution on these subdomains gives that, in each iteration, multiple linear systems of the form (2.3) must be solved. This chapter is concerned with the methodology for solving these systems. To this end, the parallel Schwarz method will be directly related to the block Jacobi iterative method.

First, we discuss the details of the block Jacobi method along with two other iterative methods in Section 3.1. We begin with the standard Jacobi iterative method, which is a simpler method to solve (2.3) on Ω without the use of subdomains. Next, the block Jacobi iterative method will be discussed, which uses a disjoint partition of the unknowns, allowing these partitioned sets to be solved for independently. This approach, as we will show, is directly related to the parallel Schwarz method with a small overlap. Finally, we will discuss a variation of this block Jacobi method that allows for overlap in the partitioning of unknowns. Section 3.2 hereafter discusses the general convergence of iterative methods, providing a convergence criterion and indicating the rate of convergence for these three methods.

3.1. Iterative Methods

This section will discuss three iterative methods: the Jacobi iterative method, the block Jacobi iterative method, and the block Jacobi iterative method with overlap. These three methods are all variations of the first, named after mathematician Carl Gustav Jacob Jacobi. All these iterative methods are designed to converge to a solution of Equation (2.3). To assess convergence, we define the residual vector on the k'th iteration as

$$\mathbf{r}^k = \mathbf{b} - A\mathbf{u}^k. \tag{3.1}$$

To make the convergence criterion independent of the problem size, we choose that the iterative method has converged when the l^2 norm of this vector, divided by the number of grid points n^2, falls below a specified threshold τ. This gives convergence criterion (3.2), where we r^k will be referred to as the residual at iteration k.

$$r^k = \frac{\|\mathbf{r}^k\|_2}{n^2} < \tau. \tag{3.2}$$

3.1.1. Jacobi Iteration

The Jacobi iteration involves extracting the diagonal elements from A in equation (2.3) . For the given system, this means that a diagonal matrix D containing the diagonal of A is extracted as follows:

$$\begin{aligned} & (A-D+D)\mathbf{u} = \mathbf{b} \\ \iff \quad & D\mathbf{u} = \mathbf{b} - (A-D)\mathbf{u}. \end{aligned} \tag{3.3}$$

This results in the Jacobi iteration for system (2.3):

$$D\mathbf{u}^{k+1} = \mathbf{b} - (A-D)\mathbf{u}^k = D\mathbf{u}^k + \mathbf{r}^k \tag{3.4}$$

where $\mathbf{r}^k$ is as in (3.1). This system is solved on each iteration using $D^{-1} = \text{diag}(\frac{1}{(D)_{ii}})$. For a 4×4 2D Laplacian matrix A (see (2.4)):

$$A = \frac{1}{h^2}\begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix},$$

the Jacobi iteration becomes

$$\begin{aligned} & \frac{-4}{h^2}\cdot\mathbf{u}^{k+1} = \mathbf{b} - \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}\mathbf{u}^k = \frac{-4}{h^2}\cdot\mathbf{u}^k + \mathbf{r}^k \\ \implies & \mathbf{u}^{k+1} = \mathbf{u}^k - \frac{h^2}{4}\mathbf{r}^k. \end{aligned}$$

3.1.2. Block Jacobi Iteration

The block Jacobi iterative method is a variation of the standard Jacobi iterative method, wherein the set of indices of the solution vector are partitioned into non-overlapping subsets. In Section 3.1.3, subsets with overlap will be discussed. Consider a partitioning of a 4×4 grid of internal nodes into 4 blocks of size 2×2, as shown in Figure 3.1. To solve for these blocks independently, each of the nodes in the domain is assigned a pair of indices (L, G), a local index L, and a global index G.

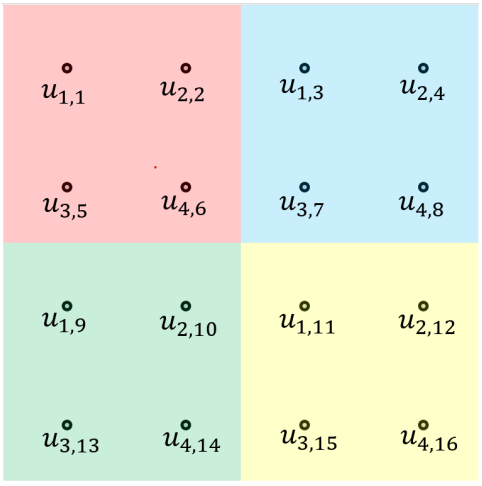


Figure 3.1: Partitioning of a 4×4 grid of internal nodes into 4 blocks of size 2×2.

We recall from Section (2.1) that to solve for the entire domain, system (2.3) must be solved. To restrict solving (2.3) to solving on one of the domains, say domain i, we use certain restriction matrices R_i and extension

matrices R_i^T, to restrict the problem to the individual subdomains. Consider $u_{L,G} = u_{2,10}$, which is located in the third domain, counted row-wise. Multiplying a residual vector $\mathbf{r}^k$ by R_3, should restrict it to the third subdomain, with its resulting second element equal to the tenth element of the initial residual vector. This implies $(R_3\mathbf{r}^k)_L = (\mathbf{r}^k)_G$, and more generally

$$\begin{aligned} &(R_i\mathbf{r}^k)_L = (\mathbf{r}^k)_G \\ \implies &(R_i)_{L,G} = 1. \end{aligned}$$

Let $\mathscr{A}_i = \{(L,G) : G \text{ in block } i\}$ denote the sets of index pairs, and p denote the size of the blocks in one direction. This leads to the following definition for the restriction matrices $R_i \in \mathbb{R}^{b^2 \times n^2}$:

$$(R_i)_{j,k} = \begin{cases} 1 & \text{if } (j,k) \in \mathscr{A}_i \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } j \in \{1,\dots,b^2\},\ k \in \{1,\dots n^2\}. \tag{3.5}$$

To relate this to the block Jacobi iteration, we recall (3.4) and note that any M could be extracted from A in the same way as 3.3, without M explicitly being contained in A, to get the iteration:

$$\mathbf{u}^{k+1} = \mathbf{u}^k + M^{-1}\mathbf{r}^k$$

under the assumption that this M matrix is non-singular. Our goal is to create the iteration matrix for the block Jacobi iteration M_{BJ} from A, such that the iteration can be performed for the blocks independently. Extracting N^2 blocks from A (provided that $N|n$ for equal block size) using restriction matrices, we get

$$\begin{aligned} M_{BJ} = \sum_{i=1}^{N^2} R_i^T A_i R_i &= \begin{bmatrix} A_1 & & & & \\ & 0 & & & \\ & & 0 & & \\ & & & \ddots & \\ & & & & 0 \end{bmatrix} + \begin{bmatrix} 0 & & & & \\ & A_2 & & & \\ & & 0 & & \\ & & & \ddots & \\ & & & & 0 \end{bmatrix} + \cdots + \begin{bmatrix} 0 & & & & \\ & 0 & & & \\ & & \ddots & & \\ & & & 0 & \\ & & & & A_{N^2} \end{bmatrix} \\ &= \begin{bmatrix} A_1 & & & & \\ & A_2 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & A_{N^2} \end{bmatrix}, \end{aligned}$$

where

$$A_i = R_i A R_i^T.$$

For each A_i, specific columns are extracted from A by multiplying with R_i^T, and the corresponding rows are extracted by multiplying with R_i. Each A_i is placed correctly on the diagonal using multiplications by R_i and R_i^T. Taking the inverse of M_{BJ} gives:

$$M_{BJ}^{-1} = \begin{bmatrix} A_1 & & & & \\ & A_2 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & A_{N^2} \end{bmatrix}^{-1} = \begin{bmatrix} A_1^{-1} & & & & \\ & A_2^{-1} & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & A_{N^2}^{-1} \end{bmatrix} = \sum_{i=1}^{N^2} R_i^T A_i^{-1} R_i.$$

This gives the block Jacobi iteration (3.6) where on each iteration, the steps 1-3 in (3.7) can be performed for each block i independently due to the disjoint partition of indices:

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \left(\sum_{i=1}^{N^2} R_i^T A_i^{-1} R_i \right) \mathbf{r}^k \tag{3.6}$$

$$\begin{aligned} &1. && \hat{\mathbf{r}}_i^k = R_i \mathbf{r}^k, \\ &2. && \text{solve } A_i \mathbf{x}_i = \hat{\mathbf{r}}_i^k \text{ for } \mathbf{x}_i, \\ &3. && \mathbf{y}_i = R_i^T \mathbf{x}_i, \\ &4. && \mathbf{u}^{k+1} = \mathbf{u}^k + \sum_{i=1}^{N^2} \mathbf{y}_i, \end{aligned} \tag{3.7}$$

where $\hat{\mathbf{r}}_i^k, \mathbf{x}_i \in \mathbb{R}^{b^2}$, and $\mathbf{y}_i \in \mathbb{R}^{n^2}$. Taking a closer look at the partition of the internal grid points and the solution process on these subdomains, it becomes clear that there is some overlap in the subdomains. In reality, the subdomains extend into their neighbouring domains with one node due to the use of their boundary, as shown in 3.2 for the bottom left domain. This shows that the block Jacobi iteration is analogous to the parallel Schwarz method when the overlap is exactly one discretization point. Moreover, it can be noted that the standard Jacobi iteration is a special case of the parallel Schwarz method, where the block size is one one discretization point.

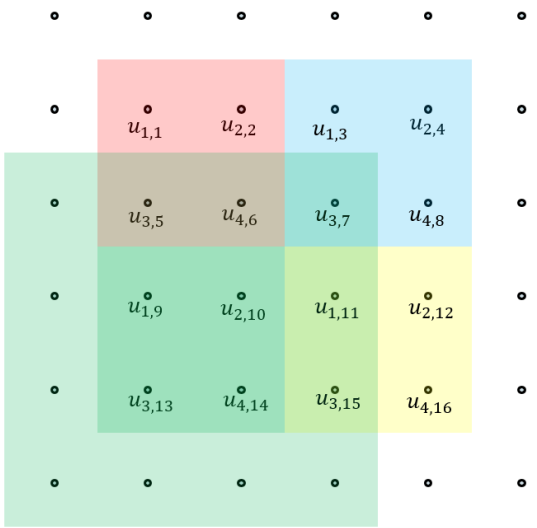


Figure 3.2: Illustration of subdomain size for partitioning in Figure 3.1.

3.1.3. Block Jacobi Iteration With Overlap

In the previous section, we saw that a non-overlapping partition of the inner points in a discretized domain is directly related to the parallel Schwarz method with an overlap of one discretization point. In this section, we discuss how the block Jacobi iteration can be applied when there is overlap in the partitioning of inner points. Consider a 3×3 grid of inner points partitioned into 4 overlapping blocks of size 2×2, as shown in Figure 3.3.

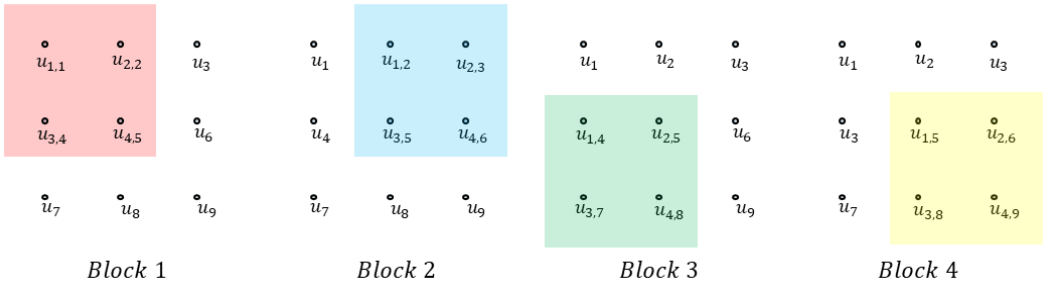


Figure 3.3: Partitioning of a 3×3 grid of internal nodes into 4 overlapping blocks of size 2×2.

It should be noted that the nodes in the overlap do not have a unique local index, as they are contained in multiple blocks. Despite these non-unique local indices, the index sets $\mathscr{A}_i = \{(L,G) : G \text{ in block } i\}$ defined in the previous section can still be used to construct the restriction matrices (3.5). The process for constructing these restriction matrices (3.5) remains valid, the only difference being the handling of the overlap. Consider the restriction matrices for the partitioning in 3.3:

$$R_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \textcircled{1} & 0 & 0 & 0 & 0 \end{bmatrix}, \quad R_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \textcircled{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$R_3 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \textcircled{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \quad R_4 = \begin{bmatrix} 0 & 0 & 0 & 0 & \textcircled{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Were we to continue with iteration (3.6), then the result vectors $\mathbf{y}_i$ in step 3 of (3.7) would all have overlapping non-zero entries. For example, as all restriction matrices have a non-zero entry in the fifth column (see encirclement), the resulting vectors $\mathbf{y}_i$ will also (likely) have non-zero fifth entries. These would be added together in step 4 of (3.7), potentially leading to divergence. To avoid this issue, we average the solutions in the overlap. Define a matrix $R \in \mathbb{R}^{b^2 \cdot N^2 \times n^2}$ as follows:

$$R = \begin{bmatrix} R_1 \\ \vdots \\ R_{N^2} \end{bmatrix}.$$

Each column of R corresponds to a global index of the discretization. The number of non-zero entries in a column indicates how many blocks contain this global entry. Dividing each entry of R by the amount of non-zero entries in its column, we obtain our averaging. The new matrix that is created in this manner is denoted as $\tilde{R}$, resulting in the following block Jacobi iteration with overlap:

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \left(\sum_{i=1}^{N^2} \tilde{R}_i^T A_i^{-1} R_i \right) \mathbf{r}^k = \mathbf{u}^k + M_{OBJ}^{-1} \mathbf{r}^k \tag{3.8}$$

3.2. Convergence of Iterative Methods

To demonstrate the convergence of the iterative methods discussed in the previous section, we will first present a general convergence condition, and relate this to the rate of convergence. For the standard Jacobi method, the convergence and convergence rate will be discussed explicitly, concluding with a note on

the convergence of the block Jacobi iteration. In [20], the convergence of iterative methods is presented as follows. Consider a general iterative method of the form

$$\mathbf{x}^{k+1} = G\mathbf{x}^k + \mathbf{f} \tag{3.9}$$

in which G is a square iteration matrix with spectral radius $\rho(G) = \max\{|\lambda| : \lambda \text{ is an eigenvalue of G}\}$. Note that the convergence of this method to a fixed point $\mathbf{x}$ implies the system $(I-G)\mathbf{x} = \mathbf{f}$. Non-singularity of $I-G$ ensures a unique solution of this system and thus a unique convergence point. As presented in [20], the following theorem holds for the convergence of the method with respect to an initial guess x_0:

Theorem 3.2.1 *Let G be a square matrix with $\rho(G) < 1$. Then $I-G$ is nonsingular and the iteration* (3.9) *converges for any* $\mathbf{f}$ *and* $\mathbf{x_0}$. *Conversely, if the iteration* (3.9) *converges for every* $\mathbf{f}$ *and* $\mathbf{x_0}$, *then* $\rho(G) < 1$.

The general convergence factor ϕ, the factor with which the error of an iterative method is reduced on each iteration, is also determined by $\rho(G)$, as presented in [20]. More precisely, $\phi = \rho(G)$, making it sufficient to know the spectral radius, to determine whether an iterative method converges, and how fast it converges. A distinction should be made, however, between error and residual regarding convergence. The convergence criterion of the iterative methods was based on the residual in Section 3.1. However, one might be more interested in the error of the iteration compared to an exact solution $\mathbf{u}^*$. We show that the error $\mathbf{e}^k = \mathbf{u}^* - \mathbf{u}^k$ is related to the residual $\mathbf{r}^k$ through matrix A of (2.3). Note that since $A\mathbf{u}^* = \mathbf{b}$:

$$A\mathbf{e}^k = A(\mathbf{u}^* - \mathbf{u}^k) = A\mathbf{u}^* - A\mathbf{u}^k = \mathbf{b} - A\mathbf{u}^k = \mathbf{r}^k.$$

Now using the non-singularity of A (the system (2.3) has a unique solution due to the fixed boundary conditions), $\|A\mathbf{v}\|_2 \leq \|A\|_2\|\mathbf{v}\|_2$ for any $\mathbf{v}$, and $\|A^{-1}\|_2 = \frac{1}{\sigma_{\min}}$, where $\sigma_{\min}$ is the smallest singular value of A; see for example [17], we get

$$\|\mathbf{e}^k\|_2 = \|A^{-1}\mathbf{r}^k\|_2 \leq \|A^{-1}\|_2\|\mathbf{r}^k\|_2 = \frac{1}{\sigma_{\min}}\|\mathbf{r}^k\|_2. \tag{3.10}$$

Now this $\sigma_{\min}$ can be determined using that A is symmetric and that the singular values of a matrix are the square root of the eigenvalues of A^TA. If λ is an eigenvalue of A with eigenvector $\mathbf{v}$, then $A^TA\mathbf{v} = A^T\lambda\mathbf{v} = \lambda^2\mathbf{v}$ since $A^T = A$. This gives that the singular values of A are equal to the absolute values of its eigenvalues and with that $\sigma_{\min} = |\lambda_{\min}|$. This shows that it suffices to investigate residual (3.2) to enforce convergence to the exact solution, as $\lambda_{\min}$ is constant. This bound might, however, not be very tight when the smallest absolute eigenvalue of A is very small. We proceed by giving the exact eigenvalues, and thus spectral radius of the iteration matrix G for the standard Jacobi method applied to the model problem. This will prove convergence and give the convergence rate of the method. For the eigenvalues of the iteration matrices of the block Jacobi iteration, with and without overlap, no exact eigenvalues will be derived, but the calculation of these eigenvalues will be discussed at the end of this section.

As presented above, the convergence condition and the convergence rate can be determined using the spectral radius ρ of the iteration matrix G. We recall the Jacobi iteration from Section 3.1.1 and write it in the form of (3.9):

$$\mathbf{u}^{k+1} = \mathbf{u}^k - \frac{h^2}{4}\left(\mathbf{b} - A\mathbf{u}^k\right) = (I + \frac{h^2}{4}A)\mathbf{u}^k - \frac{h^2}{4}\mathbf{b}$$

So, to determine convergence we, must determine the eigenvalues λ of $G_J = I + \frac{h^2}{4}A$, which satisfy

$$\begin{aligned}
& \det\left(I + \frac{h^2}{4}A - \lambda I\right) = 0 \\
\iff & \left(\frac{h^2}{4}\right)^n \det\left(\frac{4}{h^2}I + A - \frac{4}{h^2}\lambda I\right) = 0 \\
\iff & \det\left(A - (\frac{4}{h^2}\lambda - \frac{4}{h^2})I\right) = 0 \\
\iff & \det\left(A - \mu I\right) = 0
\end{aligned} \tag{3.11}$$

where $A, I \in \mathbb{R}^{n\times n}$, and $\mu = \frac{4}{h^2}\lambda - \frac{4}{h^2}$. The eigenvalues of A satisfy (3.11), so knowing the eigenvalues of A is enough to determine the eigenvalues of the iteration matrix G_J by the relation $\lambda = \frac{h^2}{4}\mu + 1$. We note that the 2D Laplacian can be constructed as the Kronecker sum of 2 one-dimensional Laplacian matrices with Dirichlet boundary conditions $A = A_{1D} \oplus A_{1D}$; see for example [14] for more details. Moreover, [14] gives that A_{1D} has eigenvalues $\tilde{\mu_1}, \ldots, \tilde{\mu_n}$, where

$$\tilde{\mu_k} = -\frac{4}{h^2}\sin^2\left(\frac{\pi \cdot k}{2(n+1)}\right) \qquad \text{for } k \in \{1, \ldots, n\}.$$

This allows us to compute the eigenvalues of A, as [16] shows that the eigenvalues of a Kronecker sum are the sum of the eigenvalues of its arguments. Therefore we get the eigenvalues of A, and subsequently the eigenvalues of G_J:

$$\begin{aligned} \mu_{i,j} &= -\frac{4}{h^2}\sin^2\left(\frac{\pi \cdot i}{2(n+1)}\right) - \frac{4}{h^2}\sin^2\left(\frac{\pi \cdot j}{2(n+1)}\right) \qquad \text{for } i, j \in \{1, \ldots, n\}, \\ \implies \lambda_{i,j} &= 1 - \sin^2\left(\frac{\pi \cdot i}{2(n+1)}\right) - \sin^2\left(\frac{\pi \cdot j}{2(n+1)}\right) \qquad \text{for } i, j \in \{1, \ldots, n\}. \end{aligned} \tag{3.12}$$

A short analysis of these eigenvalues shows that $-1 < \lambda_{i,j} < 1$ as the sine function is between 0 and 1. From this we can conclude that $\rho(G_J) = \max\{|\lambda_{i,j}| : i, j = 1, \ldots, n\} < 1$ and by Theorem 3.2.1, the Jacobi method is convergent for every f and x_0. Noting that the sine function is a strictly increasing positive function for the arguments in (3.12) (as $0 < \frac{\pi \cdot i}{2(n+1)} < \frac{1}{2}$ for $i \in \{1, \ldots, n\}$), we obtain the smallest and largest possible eigenvalues:

$$\begin{aligned} \lambda_{1,1} &= 1 - 2\sin^2\left(\frac{\pi}{2(n+1)}\right) = \cos\left(\frac{\pi}{n+1}\right) \text{ or, equivalently} \\ \lambda_{n,n} &= \cos\left(\pi - \frac{\pi}{n+1}\right) = \cos\left(\frac{\pi}{n+1}\right), \end{aligned}$$

giving that $\rho(G_J) = \cos(\frac{\pi}{n+1})$, which is the convergence factor for fixed n. As mentioned in [8], there is little known about the general convergence properties of the block Jacobi method. This paper does, however, provide a convergence proof for the block Jacobi iteration without the overlap. For the convergence of our specific case including overlap and the rate of convergence, we rely on computation of the eigenvalues of the iteration matrices

$$\begin{aligned} G_{BJ} &= I - M_{BJ}^{-1}A, \text{ and} \\ G_{OBJ} &= I - M_{OBJ}^{-1}A \end{aligned}$$

of the block Jacobi method and block Jacobi method with overlap, respectively, where M_{BJ}^{-1} and M_{BJ}^{-1} are as in 3.1. These eigenvalues are computed using the `scipy.linalg.eigvals` function from the Scipy linear algebra library [24] in Python. The approximate spectral radius of these matrices will be presented, along with some experimental convergence results in Section 5.1.

From literature, it is known that the size of the subdomains, and the size of the overlap positively influence the convergence rate; see [7], [22]. Specifically, the convergence becomes faster when the ratio of the size of the overlap over the size of the subdomains gets bigger.

4

Implementation and Software

In this chapter, we discuss the parallel implementation of the iterative methods presented in Chapter 3. First, a description of the functions and data types used for this implementation will be given in Section 4.1. For each of the iterative methods, an overview of the parallelized parts will then be given of the algorithm using pseudocode in Section 4.2. The parallelization is done using Kokkos, a 'Performance Portability Ecosystem' in C++. Section 4.1 gives a small introduction to Kokkos. For the full documentation, we refer to the Kokkos repository [23].

In this chapter, we discuss the parallel implementation of the iterative methods presented in Chapter 3. First, a description of the functions and data types used for this implementation will be given in Section 4.1. For each of the iterative methods, an overview of the parallelized parts will then be given of the algorithm using pseudocode in Section 4.2. The parallelization is done using Kokkos, a 'Performance Portability Ecosystem' in C++. Section 4.1 gives a small introduction to Kokkos. For the full documentation, we refer to the Kokkos repository [23]. The complete source code for this project is accessible on Github under the username 'kgimbergh' [10].

4.1. Functions and Data Types

There are several functions and data types which are crucial to understanding the implementation of the iterative methods. These are functions provided by the Kokkos ecosystem and are also common in other parallel frameworks. These functions are designed to make a code portable, making it efficient regardless of the computer architecture used, including CPU clusters and GPUs. Kokkos makes an abstraction of thread employment and memory access patterns, which removes the need for explicitly tailoring these to the hardware. This means that when a program originally written to be executed on a CPU cluster is run using a GPU accelerator, the programmer does not need to revise the parallel execution strategy, or memory layout and management, all of which would typically require significant changes without the use of Kokkos.

4.1.1. Functions

- **Parallel for loop:**
  Distributes the elements of 'range' over the available threads, and executes the instructions within the brackets in parallel on these threads. Each thread must get a unique segment of the data to work on in the instruction. If not, race conditions may occur where multiple threads are simultaneously altering part of the data, yielding incorrect results.

```
1    parallel_for(i in <range>) {
2        <instruction>
```

```
    }
```

- **LU factorization functor:**
  A functor contained in the Kokkos Kernels library that performs an $A = LU$ factorization using a parallel for loop. The factorization is performed on the subblocks of the matrix A in parallel, where the number of blocks must be specified in the range of the for loop. The factorization is performed in-place, meaning that the result is stored in A, removing the need for extra memory usage.

```
    parallel_for(i in <range>) {
        BatchedSerialLU(A)
    }
```

- **LU solve functor:**
  A functor contained in the Kokkos Kernels library that solves the system $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ in batches. For each block i, the system $A_i\mathbf{x}_i = \mathbf{b}_i$ is solved in parallel, where the number of blocks must be specified in the range of the for loop. Each of these systems is solved densely using forward-backward substitution. The result of this functor is stored in A.

```
    parallel_for(i in <range>) {
        BatchedSerialSolveLU(A, b)
    }
```

- **BLAS l^2 norm:**
  A function contained in the Kokkos Kernels library that used a BLAS routine, a simple but optimized linear algebra operation [5], to calculate the l^2 norm of a vector $\mathbf{v}$.

```
    l2_norm = nrm2(v);
```

4.1.2. Data Types

The following are two ways in which sparse matrices are stored in the implementation of the iterative methods, to reduce memory usage. A small, and not sparse matrix is used as an illustration. As this matrix gets bigger, it becomes more sparse, making these data types much more efficient than a standard one or two-dimensional array in which all elements are stored.

- **Crs matrix:**
  A way to store sparse matrices where only the non-zero entries are stored. The entries are stored using three one-dimensional arrays: the `Values` array containing the non-zero values, the `Entries` array containing the column indices of the values in `Values`, and the `Row_map` array specifying at which index of `Values`, a new row starts. All of these values are in row-major format. For example, a 6×6 B matrix as in (2.4) with blocks of size 2×2 is stored as follows:

$$\begin{bmatrix} -4 & 1 & 1 & 0 & 0 & 0 \\ 1 & -4 & 0 & 1 & 0 & 0 \\ 1 & 0 & -4 & 1 & 1 & 0 \\ 0 & 1 & 1 & -4 & 0 & 1 \\ 0 & 0 & 1 & 0 & -4 & 1 \\ 0 & 0 & 0 & 1 & 1 & -4 \end{bmatrix}$$

$$\begin{aligned} &\texttt{Values:} \quad [-4,1,1,1,-4,1,1-4,1,1,1,1,-4,1,1,-4,1,1,1,-4] \\ &\texttt{Entries:} \quad [1,2,3,1,2,4,1,3,4,5,2,3,4,6,3,5,6,4,5,6] \\ &\texttt{Row\_map:} \quad [1,4,7,11,15,18,21] \end{aligned}$$

- **Bsr matrix:**
  A way to store sparse matrices where only blocks that contain non-zero entries are stored. The entries of these blocks, as well as their location, are stored densely in row-major format. The `Values` array contains the values of the blocks in row-major format, the `Entries` array contains the column indices of the blocks, and the `Row_map` array contains the start of each block row. For the above matrix, where each block is 2×2, this is:

$$\begin{aligned} &\texttt{Values:} \quad [-4,1,1,-4,1,0,0,1,1,0,0,1,-4,1,1,-4,1,0,0,1,1,0,0,1,-4,1,1,-4] \\ &\texttt{Entries:} \quad [1,2,2,3,4,3,4] \\ &\texttt{Row\_map:} \quad [1,3,6,8] \end{aligned}$$

4.2. Parallel Implementation of Iterative Methods

4.2.1. Jacobi Iteration

It is not immediately clear from the formulation of the Jacobi iteration how it should be parallelized. Before going into the parallelization, we look at the different sequential parts of the algorithm within each iteration. Recalling the description in Section 3.1.1, we note that the method consists of 4 computational steps with the following complexities:

1. **Sparse matrix-vector product $A\mathbf{u}^k$:**
   From 2.4, one can deduce that A has $2 \cdot 2 \cdot 3 + (n-2) \cdot 2 \cdot 4 + 2 \cdot (n-2) \cdot 4 + (n-2) \cdot (n-2) \cdot 5 = 5n^2 - 4n$ non-zero entries[1]. Therefore, the matrix-vector product involves $5n^2 - 4n$ computations.
2. **Vector sum $\mathbf{r}^k = \mathbf{b} - A\mathbf{u}^k$:**
   This step consists of adding two vectors of size $n^2 \times 1$, resulting in n^2 computations.
3. **Scaling $D^{-1}\mathbf{r}^k$:**
   Given that D is a diagonal matrix, this operation consists of n^2 computations, one for each element of the $\mathbf{r}^k$ vector.
4. **Vector sum $\mathbf{r}^{k+1} = \mathbf{u}^k + D^{-1}\mathbf{r}^k$:**
   Similarly to the second step, this operation requires n^2 computations.

From this, it is clear that the most computationally intensive part is the matrix-vector product. It is therefore this part which becomes the focus of our parallelization. By parallelizing over the rows of the matrix A, the rest of the computations, scaling and summing vectors, can be included in the parallelization. This does require us to use an additional vector to store the result, as the vector u^k must remain unchanged during these computations. Therefore, a distinction is made between a `u_old` and `u_new` vector.

It should be noted that the parallelization over the rows of A potentially causes improper load-balancing. The number of non-zero entries varies across different nodes from 3 to 5 entries, meaning that some rows require more computations than others. As discussed in 2.3, this can negatively impact the efficiency.

A flowchart of this implementation is shown in Figure 4.1. Before the iterative process, several vectors and the matrix A must be initialized. The matrix A, used in the computation of the residual, is initiated as a Crs matrix. As discussed above, the entire process of calculating the residual vector, and updating `u_new` is done entirely in one `parallel_for` loop. After this, `u_old` is updated with the values of `u_new`, and the l^2 norm of the residual is calculated using the `nrm2` function. The pseudocode of this implementation is shown in Listing 4.1. For the full implementation, including the initialization of all objects, we refer to the Github repository of this thesis [10].

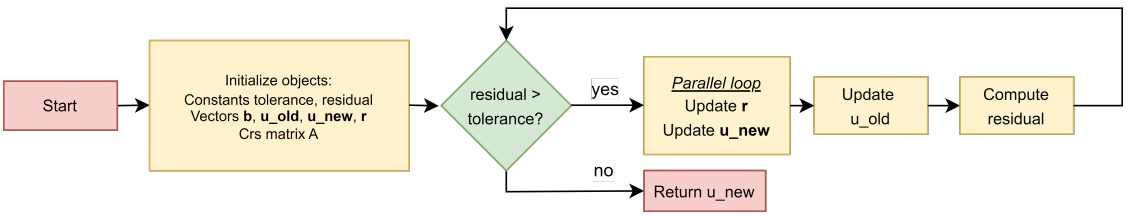


Figure 4.1: Flowchart of Jaocbi implementation

[1]The parts of this sum are: (first and last block row, first and last row) + (first and last block row, inner rows), (inner block rows, first and last row) + (inner block rows, inner rows)

```
// Initialize objects
tolerance = 0.0001
residual = 1
initiate b, u_old, u_new, and r vectors
initiate A matrix

while (residual > tolerance)
{
    // Update r and u_new
    parallel_for (i in [0,n*n]){
        r[i] = b[i] + (row [i] of A) * u_old
        u_new[i] = u_old[i] + (1/D) * r[i]
    }

    // Update u_old
    u_old = u_new

    // Compute residual
    residual = (l2 norm of r) / (n*n)
}

// Return the result
return u_new vector
```

Listing 4.1: Parallel implementation of the standard Jacobi iteration.

4.2.2. Block Jacobi Iteration (with Overlap)

The implementation of the block Jacobi iteration is almost identical to that of its variation with overlap. The difference lies in the construction of the matrices R and $\tilde{R}$. We again look at the sequential steps that must be performed on each iteration, and their complexities. The steps to consider are the calculation of the residual, which we take as one step due to the observations in the previous section, and steps 3.7. Step 2 is done with forward-backward substitution using the `BatchedSerialSolveLU` functor. This requires an LU factorization of the M matrix before the iterative process starts, using the `BatchedSerialLU` functor.

1. **Residual calculation $\mathbf{r}^k = \mathbf{b} - A\mathbf{u}^k$:**
   Combining the first two steps in the previous section, this consists of $6n^2 - 4n$ computaitons.
2. **Sparse matrix-vector product $\hat{\mathbf{r}}^k = \mathbf{r}^k R$:**
   Matrix R contains $b^2 \cdot N^2$ non-zero entries, as can be deduced from 3.5, resulting in $b^2 \cdot N^2$ computations. This step will be called the extension of $\mathbf{r}$.
3. **LU solve for subsystems $A_i\,\mathbf{x}_i = \hat{\mathbf{r}}_i^k$ for $\mathbf{x}_i$:**
   Before the iterations start, an LU factorization is performed on the M matrix, allowing for forward-backward substitution, where one should recall that the A_i are sub-matrices on the diagonal of M. A total of N^2 systems must be solved of size $b^2 \times b^2$. Both forward and backward substitution requires $(b^2)^2$ computations; see [12], resulting in $2 \cdot b^4 \cdot N^2$ computations. As these systems are independent, the result can be stored in $\hat{\mathbf{r}}^k$.
4. **Sparse matrix-vector product $\mathbf{y} = R^T\hat{\mathbf{r}}^k$:**
   Similar to step 2, this consists of $b^2 \cdot N^2$ computations. For the implementation with overlap, R^T should be $\tilde{R}^T$. This step will be called the reduction of $\hat{\mathbf{r}}$.
5. **Vector sum $\mathbf{u}^{k+1} = \mathbf{u}^k + \mathbf{y}$:**
   This step is an addition of 2 vectors of size n^2, resulting in $2n^2$ computations.

There are several comments to be made about these steps. The first is that the `BatchedSerialSolveLU` functor does not allow for any other computations to be merged into its `parallel_for` loop. Noting that step 2 is dependent on step 1, make it so that the first three steps cannot be merged for parallel execution. The last two steps, however, can be merged when we parallelize over the n^2 rows of the R^T (or $\tilde{R}^T$) matrix, similar to the implementation of the Jacobi iteration.

Figure 4.2 shows a flowchart of the implementation. The matrices A, R and $\tilde{R}^T$ are initiated as Crs matrices, while the M matrix is initiated as a Bsr matrix. This allows for the subsequent batched LU factorization of M using the `BatchedSerialLU` functor. The first `parallel_for` loop updates the residual vector, where the parallelization happens over the rows of A. This is followed by the calculation of the norm of the residual vector using the `nrm2` function. The extension of the residual vector $\mathbf{r}$ to $\hat{\mathbf{r}}$, as well as the solving of the systems, is parallelized over the number of blocks N. This improves memory efficiency, as the relevant parts of both R and M for each block are stored contiguously. The final parallel loop which reduces $\hat{\mathbf{r}}$ and updates u is parallelized over the rows of R^T (or $\tilde{R}^T$). The pseudocode of this implementation is shown in Listing 4.2. For the full implementation, including the initialization of all the objects, we refer to the Github repository of this thesis [10].

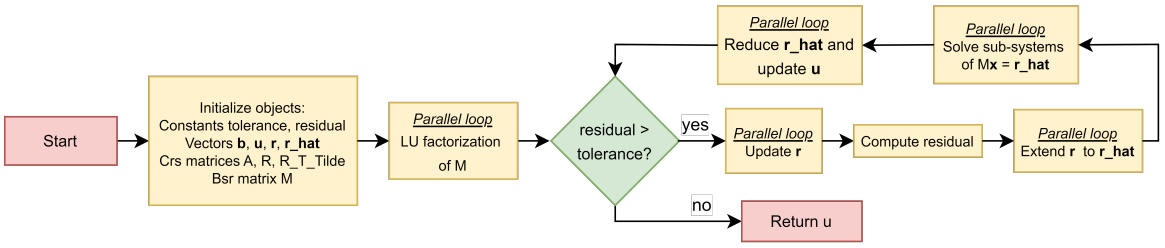


Figure 4.2: Flowchart of Jaocbi implementation

```
// Initialize pbjects
tolerance = 0.0001
residual = 1
initiate b, u, r, and r_hat vectors
initiate A, M, R, and R_T_Tilde matrices

// Perform LU factorization on M
parallel for (i in [0, nb*nb]){
    BatchedSerialLU(M)
}

while (residual > tolerance)
{
    // Update residual vector
    parallel for (i in [0,n*n]){
        r[i] = b[i] + (row i of A) * u
    }

    // Compute norm of residual vector
    residual = nrm2(r) / (n*n)

    // Extend residual vector
    parallel for (i in [0, nb*nb]){
        r_hat[i*b:(i+1)*b] = R_i * r
    }

    // Solve the sub-systems
    parallel for (i in [0, nb*nb]) {
        BatchedSerialSolveLU(M, r_hat)
    }

    // Reduce residual vector, and update u
    parallel for (i in [0, n*n]) {
        u[i] = u[i] + (row i of R_T_Tilde) * r_hat
    }
}

// Return the result
return u
```

Listing 4.2: Parallel implementation of the block Jacobi iteration (with overlap)

It is important to note that the block Jacobi iteration is implemented using a dense solver for each block, which is expected to determine the overall computation time significantly. To investigate the use of this dense solve for the iteration, a comparison was made with a sparse solver using Python's SciPy library [24]. Specifically, the dense `scipy.linalg.lu_solve` function was compared to the sparse solve method of the `scipy.sparse.linalg.splu` object.

Both solvers were tested on block sizes ranging from 2 up to 64. The average solving time was measured over 10 iterations with randomly initialized right-hand sides. The results, shown in Figure 4.3 reveal that the dense solver performs the same as the sparse solver for block sizes of up to 16. However, as the block sizes become larger, the computation time for the dense solver grows much faster than that of the sparse solver. When using larger blocks, it should thus be considered to use a sparse solver instead of a dense one. Moreover, the steep increase in computation times will likely be reflected in the results of the block Jacobi iteration.

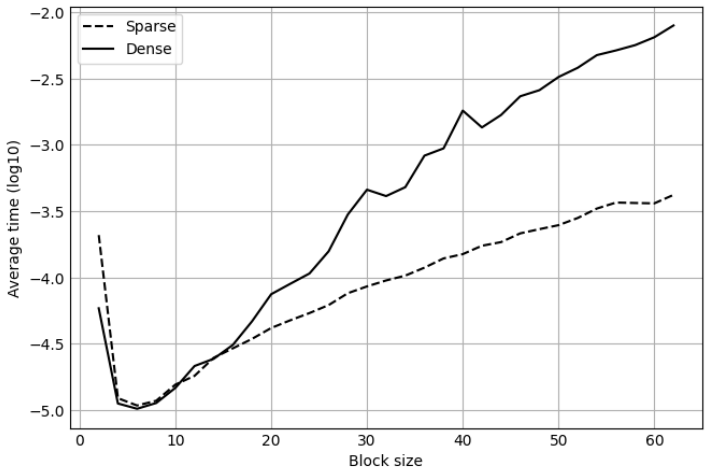


Figure 4.3: Computation times comparing a sparse solver to a dense solver, both using forward-backward substitution.

5

Numerical Results

There are two results of the implementation that will be discussed in this Chapter. First, the convergence of the block Jacobi iteration will be discussed in Section 5.1, including an analysis of the influence of the block size and the size of the overlap, after which a comparison will be made with the standard Jacobi iteration. In Section 5.2, the parallel performance of both the standard and block Jacobi iteration will be compared. Both of these methods will be assessed on parallel performance and speedup, concluding with some remarks on the scalability of both methods. The numerical experiments were conducted on the DelftBlue supercomputer of the Delft High-Performance Computing Centre [1]. The nodes used are Intel Xeon compute nodes with 64 CPU cores and 256GB RAM.

5.1. Convergence Results

As discussed in Section 3.2, the convergence of the iterative methods is governed by the spectral radius of their iteration matrices. In this section, we first show that the theoretical convergence and experimental convergence of our implementation agree. Furthermore, we show the influence of the block size and the overlap on the convergence rate.

5.1.1. Theoretical vs Experimental Convergence

First, we show that the numerical convergence rate is equal to the theoretical convergence rate as discussed in Section 3.2. There is a limitation to the usefulness of this theory which is the computationally intensive calculation of the spectral radius of a matrix. For large matrices, this becomes a numerical challenge in itself, which is why we demonstrate the convergence of the methods using a small example of $n = 32$. To show that the numerical convergence rate agrees with the theoretical convergence rate, the results of a representative for each of the iterative methods are shown in Figure 5.1. In each figure, the numerical convergence is plotted alongside the theoretical convergence, more specifically, the residual (3.2) is plotted against the iteration number. The theoretical convergence is an exponential decay function with growth factor $\rho(G)$ through the convergence threshold $\tau = 10^{-4}$. The results show a strong correlation between the theoretical and numerical convergence rate, validating the theory discussed in Section 3.2. It can also be observed that the residual decays more quickly than the rate indicated by the spectral radius of the iteration matrix for the first few matrices. As noted in [20], this can be explained by the initial error vector being more aligned with eigenvectors of smaller eigenvalues. This results in these errors decaying quickly first, after which the largest eigenvalue becomes dominant and the convergence becomes linear.

5.1.2. Influence of Block Size and Overlap

The convergence results presented in Figure 5.2 show a strong negative correlation between the block size and the number of iterations needed. The number of iterations needed seems to scale inverse linearly with the size of the blocks and the size of the overlap, except for the sizes of overlap which are (close to) half the block size. Close to the diagonal line (where the overlap is equal to the block size), a spike in the number of iterations can be observed. Figure 5.3 shows an example of this, for a block size of 16. Configurations with moderate overlap have significantly better convergence rates compared to those with no overlap. When the overlap gets close to half the block size, however, the convergence rate drops, possibly due to points in the domain being used in the calculation of many subdomains.

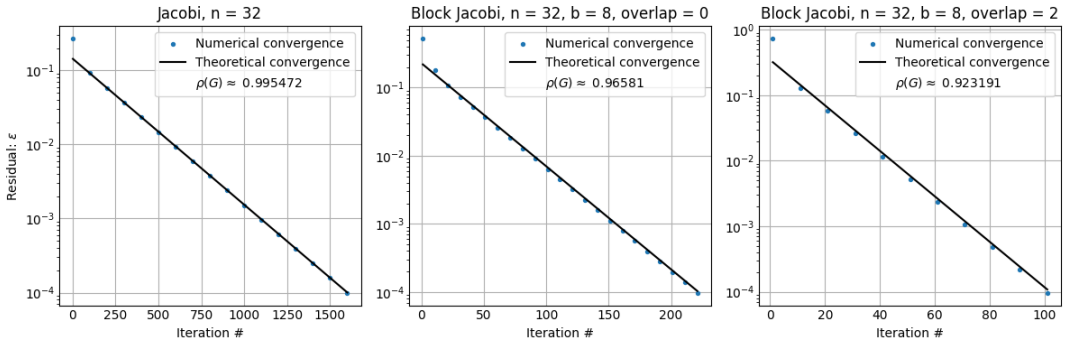


Figure 5.1: Numeric convergence and theoretical convergence of the three iterative methods

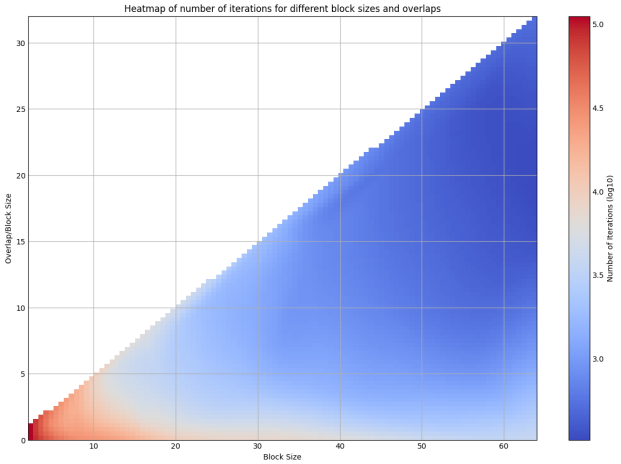


Figure 5.2: Convergence of the block Jacobi iteration against block size.

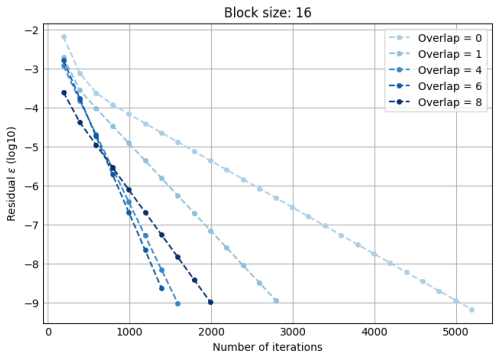


Figure 5.3: Convergence for block size 16 and different sizes of overlap

5.2. Parallel Performance

The main points of interest in this section are the parallel performance and the efficiency of the block Jacobi iteration compared to the standard Jacobi iteration, and the strong scalability of these methods. First, we make a remark on the type of solver that is used for the individual blocks of the block Jacobi method, which will later be reflected in the results of the total computation times of the method.

To investigate the performance of the block Jacobi iteration, all possible configurations with integer overlap were tested for $n = 256$ with a block size smaller or equal to 64. These block sizes were chosen due to empirical results, in which larger blocks did not seem feasible due to computation times over 1 hour. This may be explained by the results of Figure 4.3, which show a steep increase in the computation time of the dense LU solve for larger blocks. The size $n = 256$ was chosen so that there was a substantial amount of configurations possible, without excessive computation times. Figure 5.4 shows the logarithm of the computation times of all the possible configurations with integer overlap and even block sizes between 2 and 64. The horizontal black line in each figure represents the computation time of the standard Jacobi iteration. The results using 1 thread and 8 threads are shown here. For the results using other amounts of threads, we refer to Appendix A

The computation time of the dense solver in Figure 4.3 is reflected in these computation times, since it is clear that the smaller block sizes perform the best, and the computation times quickly increase for larger block sizes, regardless of the size of the overlap. The configurations with block sizes between 4 and 32 outperform the standard Jacobi iteration, where the configurations with an overlap larger than 0 and smaller than half the block size perform optimally. This clearly reflects the convergence results of Figure 5.2. The results for an overlap equal to half the block size are marked with an x, clearly showing the effect of the drop in convergence. The increase in computation times for larger blocks can be explained by the steep increase of the computation times for the dense LU solve, shown in Figure 4.3. The decrease of iterations required appears to be insufficient compared to these increasing computation times.

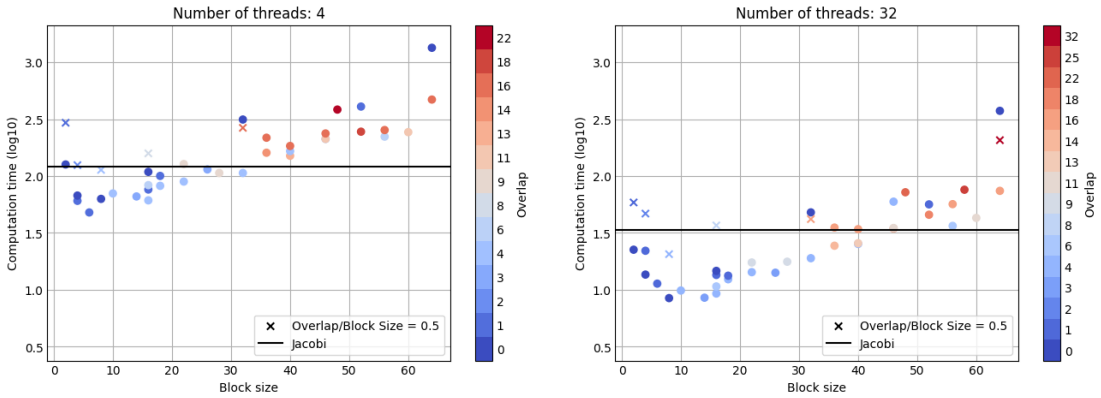


Figure 5.4: Computation times of the block Jacobi implementation for $n = 256$ using 1, and 8 threads

To compare the parallel speedup of these different configurations with the standard Jacobi iteration, we shift the attention to figure 5.5, in which the efficiency, as discussed in Section 2.3, is plotted against the number of threads employed. We are interested in the general speedup of the method since we do not know which configuration will work best for a specific amount of threads. Therefore the full range of speedups is shown for the block Jacobi iteration, as well as the middle 80% and 60% range, and the average speedup for each number of threads. The results are shown for 2, 4, 8, 16, 32, and 64 threads, where 64 is the maximum number of available threads on one node. Since we saw quite a significant difference in the performance of the configurations with a block size between 4 and 32, the results of these configurations are independently shown in the adjacent plot.

From the first plot, it can already be seen that on average the efficiency of the block Jacobi iteration stays better for a larger number of threads. This becomes even more apparent when only the block sizes between 4 and 32 are considered. There, the efficiency of the block Jacobi iterations stays quite stable between 0.7 and

1.3, while the efficiency of the standard Jacobi iteration drops quite rapidly for a larger number of threads, nearly falling below 0.4 for 64 threads. Figure 5.6 shows the efficiency of all configurations with a block size of 16. There does not seem to be a correlation between the size of the overlap, and with that the number of blocks, and the efficiency of the configuration. One aspect that could have an effect is how close the number of blocks to being divisible by the number of threads, since this makes for a more balanced load, as discussed in Section 2.3. Table 5.1 shows the remainder for each configuration when dividing the number of blocks N^2, divided by the number of threads. This does not explain the trend of the speedup. For example, the configuration with an overlap of 8 and 31 blocks has a constant remainder of 1, while there is a clear drop in efficiency when employing 32 threads.

$\downarrow N(overlap)$ \ Threads $\rightarrow$	2	4	8	16	32	64
16 (0)	0	0	0	0	0	0
17 (1)	1	1	1	1	1	33
21 (4)	1	1	1	9	25	57
25 (6)	1	1	1	1	17	49
31 (8)	1	1	1	1	1	1

Table 5.1: Remainder of dividing N^2 by the number of threads for each configuration with block size 16.

One additional comment must be made about these figures. At several points, we observe a speedup for some of the configurations larger than 1. As discussed in Section 2.3, this is likely due to better cache utilization when using more threads.

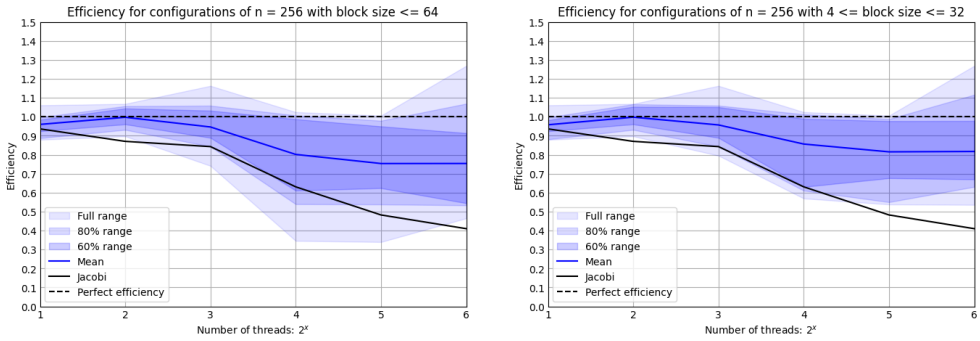


Figure 5.5: Efficiency range of all configurations for $n = 256$ with varying block sizes, and between 2 and 64 threads

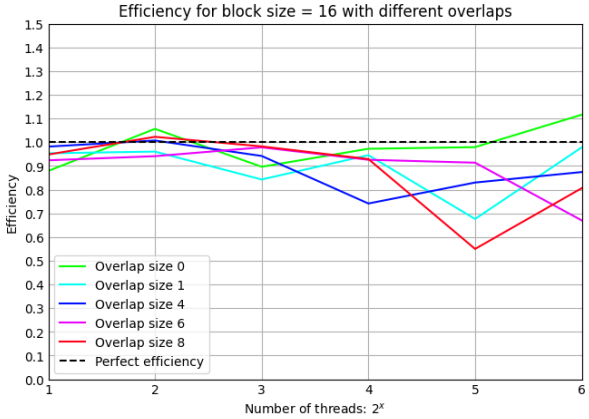


Figure 5.6: Efficiencies of all configurations for $n = 256$ with block size 16, and between 2 and 64 threads

6

Discussion

The results of this thesis give insight into the practical application of the parallel Schwarz domain decomposition in high-performance computing environments. While the findings are promising, there are several limitations that must be addressed and call for further research.

The numerical results of this thesis are based on relatively small problems. With 256 discretization points in each direction, there are 65.536 unknowns to be solved for, which is relatively small compared to other research done in high-performance computing. Larger scale problems often give rise to additional challenges such as memory usage. Further research could apply this implementation to larger-scale problems to investigate its scalability.

Addressing these larger-scale problems will likely require more computation power than available when using node-level parallelism as used in this thesis. Altering the implementation to use distributed memory, possibly using Message Passing Interface (MPI) for cross-node communication could significantly improve the parallel performance.

The study in this thesis was limited to a relatively simple partial differential equation (PDE) with a smooth solution. Whether the implementation proposed here gives similar results for more complex PDEs involving non-linearities, higher dimensions, or more complex boundaries, is unknown. Although it is uncertain for this specific implementation, there is abundant research on the use of the parallel Schwarz method on more complex problems, suggesting wide ranging applicability. Further research could investigate the versatility and robustness of this specific implementation.

The use of the specific dense solver in this implementation for the individual subdomains appears to heavily impact the total computation time. There are numerous other solvers, including sparse ones like UMFPACK [6] or MUMPS [4], that could potentially increase performance, especially for larger-scale sparse problems. In addition, this dense solver made it infeasible to investigate large blocks, which limited the possible configurations with integer overlaps smaller than half the block size. A sparse solver would allow for larger blocks, allowing for more detailed results on the influence of the overlap size on the convergence rate.

It should also be noted that although the parallel Schwarz method outperformed the standard Jacobi iteration, there are more sophisticated Schwarz methods which have been proven to yield better results. These include, but are not limited to two-level Schwarz methods, Schwarz preconditioners combined with Kyrlov methods, and optimized Schwarz methods with modified transmission conditions [9]. These approaches are very abundant in research and show promising results.

7

Conclusion

In this thesis, we investigated the performance of the parallel Schwarz domain decomposition method in comparison to the standard Jacobi iterative method, which is a special case of the parallel Schwarz method where the subdomain size is one discretization point. The research shows the effectiveness of the parallel Schwarz method in leveraging node-level parallelism to solve partial differential equations.

The parallel Schwarz method, implemented as the block Jacobi iterative method, demonstrated superior convergence rates compared to the standard Jacobi iterative method. This was especially apparent for larger subdomains and overlaps (up to half the subdomain size), which significantly improved convergence speed. Theoretical convergence rates were validated by the numerical results, showing that the number of iterations decreased as the size of the overlap increased, up to the limit of half the block size.

In terms of parallel performance, the parallel Schwarz method performed best for small block sizes between 4 and 32. This performance was attributed to the combination of the decrease in the number of iterations and the steep increase of the dense LU solver used in the implementation when solving for larger blocks. These smaller block sizes outperformed the Jacobi iteration, with a larger number of configurations falling below this computation time when employing a larger number of threads.

This was also shown in the higher efficiency of the parallel Schwarz method, averaging at just below 0.7 for the maximum number of threads considered, compared to an efficiency of 0.4 for the Jacobi iteration. The superiority of this method when using parallel computing became even more apparent when only the best-performing configurations, with block sizes between 4 and 32 were considered. The average efficiency increased to just over 0.8. with a less wide spread.

The use of the Kokkos library made it possible to write portable and efficient parallel code. This makes it possible to efficiently run the source code of this project using GPUs without the need for large-scale alterations to its parallel execution strategy or memory layout and management. Overall, this research gives a valuable overview of some of the intricacies that go into the implementation of Schwarz domain decomposition methods.

Bibliography

[1] Delft High Performance Computing Centre (DHPC). *DelftBlue Supercomputer (Phase 2)*. https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2. 2024.

[2] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Ninth printing. Table 25.3.30. Dover, 1970.

[3] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1465482.1465560.

[4] P.R. Amestoy et al. "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41.

[5] L Susan Blackford et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[6] Timothy A. Davis. "Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method". In: *ACM Trans. Math. Softw.* 30.2 (June 2004), pp. 196–199. ISSN: 0098-3500. DOI: 10.1145/992200.992206. URL: https://doi.org/10.1145/992200.992206.

[7] Victorita Dolean, Pierre Jolivet, and Frédéric Nataf. "An Introduction to Domain Decomposition Methods: algorithms, theory and parallel implementation". Master. Lecture. France, Jan. 2015. URL: https://hal.science/cel-01100932.

[8] Ludwig Elsner and Volker Mehrmann. "Convergence of block iterative methods for linear systems arising in the numerical solution of Euler equations". In: *Numerische Mathematik* 59.1 (1991), pp. 541–559. DOI: 10.1007/bf01385795.

[9] Martin J. Gander. "Schwarz methods over the course of time." eng. In: *ETNA. Electronic Transactions on Numerical Analysis [electronic only]* 31 (2008), pp. 228–255. URL: http://eudml.org/doc/130616.

[10] Karel Gimbergh. *BEP*. 2024. URL: https://github.com/kgimbergh/BEP.

[11] R Glowinski, Groupe pour l'avancement des méthodes numériques dans les sciences de l'ingénieur, and Société de mathématiques appliquées et industrielles. "First International Symposium on Domain Decomposition Methods for Partial Differential Equations: proceedings of the First International Symposium on Domain Decomposition Methods for Partial Differential Equations, Ecole Nationale des Ponts et Chaussees, Paris, France, January 7-9, 1987". In: Society for Industrial and Applied Mathematics. Paris, France, 1988, pp. 1–42. URL: http://catalog.hathitrust.org/api/volumes/oclc/18250385.html.

[12] Gene H. Golub and Charles F. Van Loan. *Matrix Computations - 4th Edition*. Philadelphia, PA: Johns Hopkins University Press, 2013. DOI: 10.1137/1.9781421407944. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781421407944. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781421407944.

[13] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers Georg Hager; Gerhard Wellein*. CRC Press, 2011.

[14] Gennadij Heidel et al. "Tensor product method for fast solution of optimal control problems with fractional multidimensional Laplacian in constraints". In: *Journal of Computational Physics* 424 (2021), p. 109865. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2020.109865. URL: https://www.sciencedirect.com/science/article/pii/S0021999120306392.

[15] Pierre Jolivet. "Domain decomposition methods. Application to high-performance computing". In: 2014. URL: https://api.semanticscholar.org/CorpusID:125443560.

[16] Alan J. Laub. *Matrix Analysis for Scientists and Engineers*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2004. DOI: 10.1137/1.9780898717907. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898717907. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898717907.

[17] Tom Lyche. *Numerical Linear Algebra and Matrix Factorizations*. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-36468-7.

[18] *Network connections*. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial. Accessed: 05-08-2024.

[19] Thomas Rauber and Gudula Rünger. *Parallel Programming*. Cham: Springer International Publishing, 2023. DOI: 10.1007/978-3-031-28924-8.

[20] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003. DOI: 10.1137/1.9780898718003.

[21] H. A. Schwarz. *On passing to the limit by the alternating process*. German. Wolf J. XV. 272-286. 1870 (1870). 1870.

[22] Andrea Toselli and Olof Widlund. *Domain Decomposition Methods – Algorithms and Theory*. Vol. 34. Jan. 2005. ISBN: 978-3-540-20696-5. DOI: 10.1007/b137868.

[23] Christian R. Trott et al. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[24] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

A

Additional Numeric Results

A.1. Convergence results

This section contains additional convergence results for the block Jacobi iteration. These results are contained in Figure 5.2. The contents of Table A.1 are as follows:

- b: Block size in one direction,
- nb: Number of blocks in one direction,
- o: Size of the overlap.

b	nb	o	# iterations
2	128	0	36689
	255	1	36688
4	64	0	19024
	85	1	13003
	127	2	15181
6	51	1	8074
8	32	0	9942
	63	4	5910
10	42	4	3212
14	23	3	2117
16	16	0	5223
	17	1	2914
	21	4	1637
	25	6	1513
	31	8	2070
18	15	1	2595
	18	4	1413
22	14	4	1112
	19	9	891
26	11	3	1057
28	13	9	608

b	nb	o	# iterations
32	8	0	2756
	9	4	733
	15	16	657
36	11	14	382
	12	16	385
40	7	4	580
	9	13	328
	10	16	318
46	6	4	503
	7	11	289
	8	16	253
48	9	22	233
52	5	1	947
	7	18	204
56	5	6	323
	6	16	190
58	7	25	164
60	5	11	209
64	4	0	1471
	5	16	159
	7	32	202

Table A.1: Convergence results for $n = 256$ against block sizes

A.2. Parallel performance results

This section contains the parallel performance results for all numbers of threads employed. The plots in Figure A.1 are an addition to Figure 5.4. The tables that follow contain the exact values that are presented in these figures, as well as the values used to produce Figure 5.5.

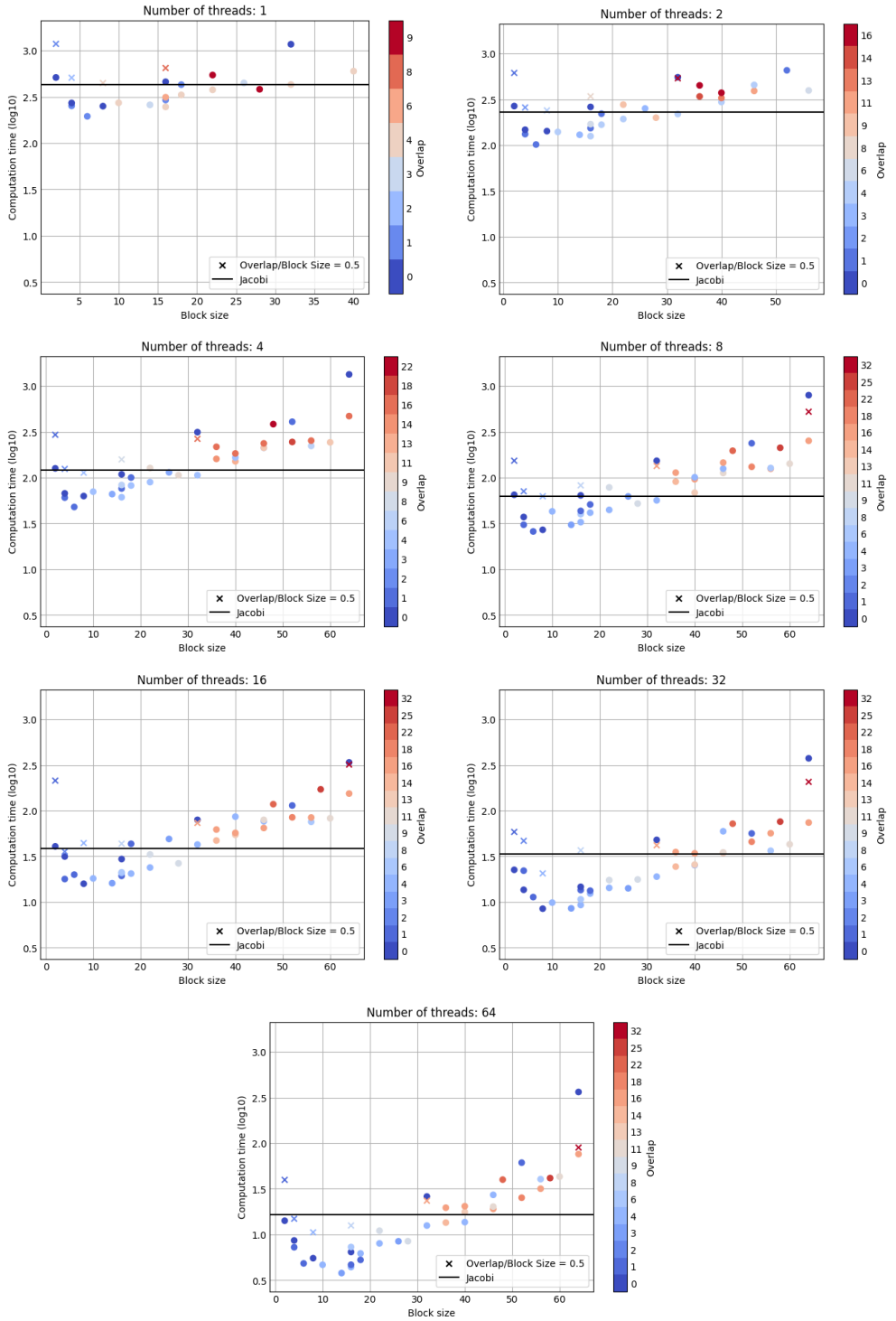


Figure A.1: Efficiency range of all configurations for $n = 256$ with varying block sizes, and between 2 and 64 threads

Table A.2: Numeric results using 1 thread

n_t	b	o	t_in	n_it	t_tot	e
1	6	1	0.065	8074	194.898	-
1	16	4	3.530	1637	246.310	-
1	8	0	0.130	9942	250.807	-
1	4	1	0.020	13003	252.785	-
1	14	3	1.946	2117	258.942	-
1	4	0	0.011	19024	271.557	-
1	10	4	0.885	3212	272.684	-
1	16	1	2.299	2914	292.445	-
1	16	6	4.539	1513	313.078	-
1	18	4	5.206	1413	331.853	-
1	22	4	10.494	1112	376.692	-
1	28	9	38.162	608	381.664	-
1	18	1	3.583	2595	428.351	-
1	32	4	41.833	733	429.533	-
1	8	4	0.548	5910	446.033	-
1	26	3	17.395	1057	446.613	-
1	16	0	2.084	5223	459.672	-
1	4	2	0.046	15181	508.104	-
1	2	0	0.003	36689	511.536	-
1	22	9	19.321	891	543.644	-
1	40	4	97.326	580	598.916	-
1	16	8	7.672	2070	647.419	-
1	32	0	32.758	2756	1169.035	-
1	2	1	0.010	36688	1179.857	-

Table A.3: Numeric results using 2 threads

n_t	b	o	t_in	n_it	t_tot	e
2	6	1	0.037	8074	101.612	0.959
2	16	4	1.823	1637	125.492	0.981
2	14	3	1.036	2117	129.873	0.997
2	4	1	0.012	13003	131.974	0.958
2	10	4	0.469	3212	139.514	0.977
2	8	0	0.079	9942	142.316	0.881
2	4	0	0.007	19024	147.416	0.921
2	16	1	1.126	2914	153.408	0.953
2	18	4	3.141	1413	167.302	0.992
2	16	6	2.963	1513	169.480	0.924
2	22	4	5.453	1112	192.953	0.976
2	28	9	20.192	608	199.398	0.957
2	32	4	21.734	733	218.704	0.982
2	18	1	1.878	2595	220.355	0.972
2	8	4	0.295	5910	239.568	0.931
2	26	3	8.927	1057	251.683	0.887
2	4	2	0.023	15181	258.060	0.984
2	16	0	1.179	5223	261.517	0.879
2	2	0	0.001	36689	267.248	0.957
2	22	9	11.084	891	277.047	0.981
2	40	4	51.045	580	295.857	1.012
2	40	13	88.159	328	327.816	-
2	36	14	65.890	382	340.461	-
2	16	8	4.132	2070	341.451	0.948
2	40	16	103.363	318	373.451	-
2	46	11	119.062	289	391.233	-
2	56	6	182.072	323	395.537	-
2	36	16	83.943	385	449.186	-
2	46	4	86.078	503	454.557	-
2	32	16	60.298	657	537.151	-
2	32	0	14.455	2756	551.157	1.061
2	2	1	0.006	36688	614.335	0.960
2	52	1	128.158	947	656.491	-

Table A.4: Numeric results using 4 threads

n_t	b	o	t_in	n_it	t_tot	e
4	6	1	0.017	8074	47.870	1.018
4	4	1	0.006	13003	60.550	1.044
4	16	4	0.861	1637	61.143	1.007
4	8	0	0.035	9942	62.985	0.996
4	14	3	0.533	2117	66.160	0.978
4	4	0	0.003	19024	67.381	1.008
4	10	4	0.239	3212	70.324	0.969
4	16	1	0.629	2914	76.135	0.960
4	18	4	1.429	1413	82.073	1.011
4	16	6	1.355	1513	83.181	0.941
4	22	4	2.634	1112	89.443	1.053
4	18	1	0.931	2595	100.290	1.068
4	28	9	11.624	608	106.371	0.897
4	32	4	10.488	733	106.398	1.009
4	16	0	0.505	5223	108.804	1.056
4	8	4	0.143	5910	113.259	0.985
4	26	3	4.843	1057	114.331	0.977
4	4	2	0.013	15181	124.731	1.018
4	2	0	0.001	36689	126.419	1.012
4	22	9	4.980	891	127.486	1.066
4	40	13	37.550	328	150.794	-
4	16	8	2.028	2070	158.272	1.023
4	36	14	32.632	382	160.332	-
4	40	4	29.701	580	165.772	0.903
4	40	16	48.765	318	183.832	-
4	46	4	39.887	503	211.923	-
4	46	11	61.740	289	212.111	-
4	36	16	43.637	385	217.263	-
4	56	6	102.229	323	222.047	-
4	46	16	73.219	253	236.801	-
4	60	11	147.372	209	243.280	-
4	52	18	121.440	204	245.531	-
4	56	16	154.656	190	253.760	-
4	32	16	29.126	657	265.401	-
4	2	1	0.004	36688	294.113	1.003
4	32	0	9.490	2756	313.878	0.931
4	48	22	143.938	233	384.059	-
4	52	1	77.959	947	408.056	-
4	64	16	236.124	159	470.221	-
4	64	0	135.073	1471	1341.286	-

Table A.5: Numeric results using 8 threads

n_t	b	o	t_in	n_it	t_tot	e
8	6	1	0.011	8074	25.880	0.941
8	8	0	0.017	9942	26.963	1.163
8	14	3	0.237	2117	30.571	1.059
8	4	1	0.003	13003	30.610	1.032
8	16	4	0.500	1637	32.688	0.942
8	4	0	0.002	19024	37.180	0.913
8	16	6	0.629	1513	40.009	0.978
8	18	4	0.752	1413	41.473	1.000
8	10	4	0.154	3212	42.874	0.795
8	16	1	0.412	2914	43.381	0.843
8	22	4	1.288	1112	44.487	1.058
8	18	1	0.542	2595	50.958	1.051
8	28	9	5.801	608	52.177	0.914
8	32	4	5.660	733	56.554	0.949
8	26	3	2.673	1057	62.377	0.895
8	8	4	0.082	5910	62.724	0.889
8	16	0	0.300	5223	64.104	0.896
8	2	0	0.000	36689	65.074	0.983
8	40	13	17.864	328	68.552	-
8	4	2	0.008	15181	70.985	0.895
8	22	9	3.170	891	78.274	0.868
8	16	8	0.958	2070	82.384	0.982
8	36	14	19.337	382	90.462	-
8	40	16	25.525	318	96.151	-
8	40	4	17.967	580	101.037	0.741
8	46	11	30.900	289	112.445	-
8	36	16	21.196	385	113.394	-
8	56	16	73.813	190	125.144	-
8	46	4	26.367	503	125.468	-
8	56	6	59.272	323	127.479	-
8	52	18	64.917	204	131.461	-
8	32	16	14.661	657	134.202	-
8	60	11	87.713	209	142.019	-
8	46	16	48.088	253	146.098	-
8	32	0	4.636	2756	152.915	0.956
8	2	1	0.002	36688	153.225	0.963
8	48	22	72.410	233	196.830	-
8	58	25	125.010	164	212.034	-
8	52	1	42.183	947	237.534	-
8	64	16	120.977	159	252.822	-
8	64	32	236.559	202	524.890	-
8	64	0	85.710	1471	795.921	-

Table A.6: Numeric results using 16 threads

n_t	b	o	t_in	n_it	t_tot	e
16	8	0	0.010	9942	15.847	0.989
16	14	3	0.143	2117	16.071	1.007
16	4	1	0.002	13003	17.853	0.885
16	10	4	0.065	3212	18.135	0.940
16	16	1	0.174	2914	19.336	0.945
16	6	1	0.005	8074	19.947	0.611
16	18	4	0.386	1413	20.477	1.013
16	16	4	0.261	1637	20.764	0.741
16	16	6	0.363	1513	21.135	0.926
16	22	4	0.786	1112	23.849	0.987
16	28	9	2.916	608	26.506	0.900
16	16	0	0.148	5223	29.551	0.972
16	4	0	0.001	19024	31.492	0.539
16	22	9	1.417	891	33.166	1.024
16	4	2	0.005	15181	35.656	0.891
16	2	0	0.000	36689	40.690	0.786
16	32	4	3.685	733	42.604	0.630
16	18	1	0.280	2595	43.399	0.617
16	16	8	0.572	2070	43.564	0.929
16	8	4	0.050	5910	44.243	0.630
16	36	14	9.683	382	47.137	-
16	26	3	1.492	1057	49.036	0.569
16	40	13	13.682	328	54.313	-
16	40	16	15.559	318	57.125	-
16	36	16	11.279	385	62.171	-
16	46	16	20.945	253	64.642	-
16	32	16	8.931	657	73.421	-
16	56	6	34.942	323	75.498	-
16	46	4	15.698	503	76.624	-
16	32	0	2.366	2756	79.353	0.921
16	46	11	21.345	289	79.420	-
16	60	11	50.503	209	82.568	-
16	56	16	51.035	190	84.179	-
16	52	18	43.583	204	84.539	-
16	40	4	11.090	580	85.997	0.435
16	52	1	21.465	947	114.032	-
16	48	22	41.469	233	117.819	-
16	64	16	79.142	159	154.029	-
16	58	25	90.280	164	171.350	-
16	2	1	0.003	36688	213.416	0.346
16	64	32	152.002	202	319.671	-
16	64	0	38.361	1471	337.883	-

Table A.7: Numeric results using 32 threads

n_t	b	o	t_in	n_it	t_tot	e
32	8	0	0.005	9942	8.455	0.927
32	14	3	0.072	2117	8.525	0.949
32	16	4	0.239	1637	9.276	0.830
32	10	4	0.033	3212	9.854	0.865
32	16	6	0.185	1513	10.709	0.914
32	6	1	0.006	8074	11.330	0.538
32	18	4	0.225	1413	12.376	0.838
32	18	1	0.149	2595	13.309	1.006
32	16	1	0.109	2914	13.516	0.676
32	4	0	0.002	19024	13.609	0.624
32	26	3	0.675	1057	14.130	0.988
32	22	4	0.535	1112	14.290	0.824
32	16	0	0.075	5223	14.671	0.979
32	22	9	0.741	891	17.385	0.977
32	28	9	1.598	608	17.658	0.675
32	32	4	1.816	733	18.966	0.708
32	8	4	0.032	5910	20.574	0.677
32	4	1	0.003	13003	22.045	0.358
32	2	0	0.000	36689	22.516	0.710
32	36	14	4.916	382	24.409	-
32	40	4	4.601	580	25.314	0.739
32	40	13	6.870	328	25.673	-
32	40	16	9.269	318	34.134	-
32	46	16	10.972	253	34.600	-
32	46	11	10.609	289	34.949	-
32	36	16	6.319	385	35.256	-
32	56	6	17.404	323	36.449	-
32	16	8	0.320	2070	36.801	0.550
32	32	16	4.876	657	41.815	-
32	60	11	25.914	209	42.851	-
32	52	18	22.312	204	45.706	-
32	4	2	0.005	15181	46.771	0.339
32	32	0	1.368	2756	47.925	0.762
32	52	1	10.994	947	56.371	-
32	56	16	34.107	190	56.642	-
32	2	1	0.002	36688	58.582	0.629
32	46	4	10.927	503	59.534	-
32	48	22	21.371	233	72.025	-
32	64	16	38.951	159	74.101	-
32	58	25	45.661	164	75.821	-
32	64	32	80.156	202	206.749	-
32	64	0	35.540	1471	374.332	-

Table A.8: Numeric results using 64 threads

n_t	b	o	t_in	n_it	t_tot	e
64	14	3	0.038	2117	3.785	1.069
64	16	4	0.081	1637	4.403	0.874
64	10	4	0.017	3212	4.662	0.914
64	16	1	0.092	2914	4.671	0.978
64	6	1	0.003	8074	4.830	0.630
64	18	1	0.075	2595	5.276	1.268
64	8	0	0.003	9942	5.509	0.711
64	18	4	0.112	1413	6.218	0.834
64	16	0	0.053	5223	6.436	1.116
64	4	1	0.001	13003	7.257	0.544
64	16	6	0.145	1513	7.301	0.670
64	22	4	0.249	1112	7.997	0.736
64	26	3	0.338	1057	8.434	0.827
64	28	9	0.799	608	8.451	0.706
64	4	0	0.001	19024	8.617	0.492
64	8	4	0.012	5910	10.566	0.660
64	22	9	0.741	891	11.017	0.771
64	32	4	1.202	733	12.538	0.535
64	16	8	0.149	2070	12.549	0.806
64	36	14	2.544	382	13.506	-
64	40	4	2.323	580	13.672	0.684
64	2	0	0.000	36689	14.160	0.564
64	4	2	0.003	15181	14.893	0.533
64	40	13	4.616	328	17.666	-
64	46	16	5.509	253	19.068	-
64	36	16	3.744	385	19.668	-
64	46	11	5.544	289	20.220	-
64	40	16	4.760	318	20.420	-
64	32	16	2.444	657	23.454	-
64	52	18	11.998	204	25.302	-
64	32	0	0.619	2756	26.103	0.700
64	46	4	5.316	503	27.188	-
64	56	16	17.459	190	31.763	-
64	2	1	0.002	36688	39.687	0.465
64	48	22	13.816	233	39.913	-
64	56	6	17.113	323	40.407	-
64	58	25	22.526	164	41.506	-
64	60	11	25.800	209	43.088	-
64	52	1	11.144	947	61.221	-
64	64	16	38.591	159	75.992	-
64	64	32	41.150	202	89.703	-
64	64	0	39.109	1471	363.548	-