Fredrik Johannessen

Norwegian University of
Science and Technology

Accelerated Smoothing and Construction of Prolongation Operators for the Multiscale Restricted-Smoothed Basis Method on Distributed Memory Systems

Master of Science in Physics and Mathematics

Submission date: October 2016

Supervisor: Knut Andreas Lie, MATH

Co-supervisor: Olav Møyner, SINTEF anvendt matematikk

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

During the last two decades, several multiscale solvers have been developed in an attempt to reduce the computational cost of reservoir simulations. One such method is the recently proposed and promising multiscale restricted-smoothed basis method. As with other multiscale methods, it relies on capturing local variations in form of basis functions, which are represented by a prolongation operator. The prolongation operator is used to develop a coarse system, and after this system has been solved, the operator is used once more to construct a fine-scale pressure approximation from the coarse-scale solution. The basis functions are solutions to local flow problems, and are formed by an iterative algorithm that gradually makes them algebraically smooth while restricting them to remain local and preserving partition of unity for the union of basis functions.

The work presented in this thesis has been made to advance the computational efficiency in the construction of the prolongation operator. A modified version of the preexisting construction algorithm is presented, which has shown to be more numerically stable. Further, two Gauss-Seidel type smoothers are proposed as alternatives to the currently used relaxed Jacobi smoother. Numerical evidence is presented which suggests that the new smoothers have improved convergence rate. The second contribution in this thesis is a program able to compute the prolongation operator on distributed memory systems. Results show that a high speedup of the iteration algorithm can be achieved, but it greatly depends on the number of connections in the reservoir model.

Sammendrag

Flere multiskalametoder har blitt utviklet gjennom de siste par tiårene i et forsøk på å redusere beregningskostnadene innenfor reservoarsimulering. En av disse er den nylig foreslåtte og lovende multiskala-begrenset-glattede basis metoden. I likhet med andre multiskalametoder bygger den på å fange lokale variasjoner i mediets egenskaper i form av basisfunksjoner, som er representert med en prolongeringsoperator. Prolongeringsopperatoren brukes til å mappe ukjente definert på et grovt grid til ukjente definert på et fint grid. Ved å sette operatoren inn det fine likningssystemet og summere disse for hver grov blokk, får man et redusert sett av likninger. Etter at det reduserte problemet er løst, brukes prolongeringsopperatoren igjen til å konstruere en finskala trykkapproximasjon fra den reduserte løsningen. Basisfunksjonene er løsninger av lokale problemer, og er konstruert gjennom en iterativ algoritme som gradvis gjør dem algebraisk glatte, samtidig som de begrenses til å forbli lokale og bevare partisjon av enheten for unionen av basisfunksjoner.

Arbeidet som presenteres i denne avhandlingen har blitt gjort for å utvikle beregningseffektiviteten i konstruksjonen av prolongeringsoperatoren. En modifisert versjon av den eksisterende konstruksjonsalgoritmen presenteres, som har vist seg å være mer numerisk stabil. Det introduseres også to modifiserte Gauss-Seidel glattere som foreslåtte alternativer til den nåværende Jacobiglatteren. Numeriske resultater blir presentert, som tyder på at de nye glatterene konvergerer raskere enn Jacobi. Det andre bidraget i avhandlingen er et program som kan konstruere prolongeringsoperatoren på distribuert minne systemer. Vi presenterer tester som viser at programmet oppnår god effektivitet for iterasjonsalgoritmen på et høyt antall prosessorer, men effektiviteten avhenger i stor grad av antall forbindelser i reservoarmodellen.

Preface

The following thesis is written for the of master of science degree in numerical mathematics, at the Norwegian University of Sciences and Technology (NTNU), Trondheim, Norway.

Acknowledgment

First of all, I would like to express my gratitude to my advisor Prof. Knut-Andreas Lie for his help and support, for guiding my work in the right direction, and for providing such an exiting and enjoyable assignment. I want to thank my advisor PhD candidate Olav Møyner for his valuable feedback, and for providing the starting point of my programing project. I also want to thank Prof. Jo Eidsvik for his help with the structure of my thesis, and Per Kristian Hove for his technical support. A special thanks goes to Kenneth Aase, Christer Hølestøl and Endre Jacobsen for allowing me to sleep in their dining room during the last weeks of my work.

Fredrik Johannessen
Trondheim, October 2016

Table of Contents

Abstract i

Sammendrag iii

Preface v

Acknowledgment v

Table of Contents ix

List of Tables xi

List of Figures xvi

1 Introduction 1

1.1 Background 1

1.1.1 Applications and Challenges for Reservoir Simulation . . 2

1.1.2 Multiscale Methods 3

1.1.3 Restricted-Smoothed Basis Functions 6

1.1.4 Parallel Computing 7

1.1.5 Matlab Reservoir Simulation Toolbox 9

1.2 Structure of the Thesis 10

2 Problem Derivation and Method Presentation 13

2.1 Flow Model and Discretization 13

2.1.1 Flow Model 14

2.1.2 Two-Point Flux-Approximation 15

2.1.3 SPE 10 Dataset 18

2.2 Jacobi and Gauss-Seidel Methods 18

2.2.1 Red-black Gauss-Seidel 22

2.3 Coarse Grid 23

2.4 Multiscale Restricted-Smoothed Basis Formulation 26

2.5 Constructing the Prolongation Operator 29

2.6 Multiscale Method Application 33

3 Improving Construction of the Prolongation Operator 35

3.1 A Change to the Original Construction Algorithm 36

3.2 Analyzing the Construction Algorithm 37

3.3 Gauss-Seidel as Smoother 40

3.4 Red-black Gauss-Seidel Smoother 42

3.5 Partially Red-black Gauss-Seidel Algorithm 44

3.6 Boundary Last Gauss-Seidel Algorithm 45

3.7 Numerical Results 47

3.7.1 Comparison of Construction Algorithms 47

3.7.2 Changing the Smoother 50

4 Constructing Prolongation Operator on Distributed Memory Systems 55

4.1 The Union Boundary 56

4.2 Sparse Data Formats 57

4.3 Sequential Program 60

4.4 Message-Passing Program 61

4.4.1 Graph Representing Computational Work 62
4.4.2 Graph Partitioning 63
4.4.3 Message-Passing 65
4.5 Testing the Program 70
4.5.1 Single-Processor Runtimes 71
4.5.2 Iteration Speedup 74
4.5.3 Setup Cost 82
4.5.4 Modification to Reduce Communication Cost 84
4.6 Results Summary and Improvement Ideas 87

5 Concluding Remarks **89**

Bibliography **90**

List of Tables

3.1 *Discrepancy in the prolongation operator resulting from implementations of Algorithm 1 and 2, with respect to the Matlab program, measured in the scaled L^∞ norm.* 49

3.2 *Number of iterations required to reach tolerance for three test-cases, and the converged quality measures τ^*.* 51

4.1 *Information about Vilje.* 70

4.2 *Time to read problem data and η for the five test cases.* 71

4.3 *The fraction of cells belonging to the union boundary, η, for the test-grids.* 74

4.4 *Number of iterations to reach tolerance for $s = 0$, with respective computation times, and number of iterations to reach tolerance for $s = 1$, with respective speedups relative to times for $s = 0$. The problem is Test Case 4, and the times are results of applying the hybrid approach with 8 OpenMP threads in each MPI process on 640 cores.* 86

List of Figures

1.1 *Visualization of the permeability field for a geological rock model of the Johansen formation, located offshore the south-west coast of Norway. The formation has been considered a candidate location for CO_2 storage.* . . . 3

1.2 *A basis function for a two-dimensional rock model with homogeneous and heterogeneous permeability fields, as computed by the MsRSB method.* . . . 7

1.3 *Perfect speedup and realistic speedup as functions of the number of processing units.* . . . 8

2.1 *Example of two connected cells resulting from a grid-partitioning.* 15

2.2 *Horizontal permeability field of the SPE 10 rock model. A logarithmic color-scale is used.* . . . 19

2.3 ***(a)** Red-black colored Cartesian grid, and **(b)** the regular five-point stencil.* . . . 23

2.4 *A triangular grid partitioned into blocks.* . . . 23

2.5 ***(a)** A grid with two support regions highlighted, and **(b)** a grid with one support region highlighted together with union boundary cells.* 25

2.6 *Interpretation of $(\boldsymbol{A}_c)_{jk}$. The red cells are the intersection of block j and the support region to basis function k. The blue arrows illustrate flux out of the area.* . . . 28

2.7 *A one-dimensional characteristic function, here denoted $f_{a,b}(x)$. It is zero everywhere except in the interval $[a, b]$, where it equals unity.* . . . 29

2.8 ***(a)** One-dimensional permeability field over 120 file cells, **(b)** development of a single basis function, and **(c)** resulting residuals $|\boldsymbol{AP}_j|$ in each cell.* . . . 31

2.9 ***(a)** Permeability field and **(b)** fine-scale pressure solution of the model-problem.* . . . 33

2.10 *Error measures of the multiscale pressure solution as a function of smoothing iterations applied to the prolongation operator.* . . . 34

3.1 ***(a)** Example-grid with union boundary and block centers highlighted, and **(b-i)** development of a basis function for a homogeneous permeability field. The grey line highlights the frontier of the basis function.* . . . 40

3.2 *Example-grid with **(a)** support regions and boundaries for the bottom-left and top-right basis functions highlighted, and **(b,c)** resulting basis functions after one sweep of Gauss-Seidel.* . . . 42

3.3 *Example-grid **(a)** in red-black coloring, **(b)** with a support region and support boundaries B_j^0 and B_j^1 highlighted, and **(c)** nonzero updated values colored red and black.* . . . 43

3.4 *Example-grid with partially red-black coloring.* . . . 44

3.5 *Error measure between diverged prolongation operator and the prolongation operator produced by Algorithms 1 and 2 as a function of smoothing steps.* . . . 48

3.6 *The left figures show the quality measure of the prolongation operator, λ, as a function of the number of steps by the four smoothers. The right figures show the fraction of number of iterations required for the BLGS and partially red-black GS smoothers to reach equally high quality, s, as the Jacobi smoother, as a function of number of steps applied with the Jacobi smoother and $\omega = 0.95$* . . . 53

4.1 *Union boundary cells, U, for square fine and coarse grids, in two and three dimensions.* . . . 56

4.2 *Ratio of cells that belong to the union boundary, η, as a function of the upscaling factor for square fine and coarse grids, in two and three dimensions.* 57

4.3 *Example of a graph.* 59

4.4 ***(a,b)** A support region highlighted in grey before and after the center of exterior blocks are moved, and **(c)** the example grid.* . . 64

4.5 *Graph resulting from the grid in Figure 4.4c with vertex weights μ_j and edge weights γ_{jk}. Vertexes with the same weights have been highlighted with the same color.* 64

4.6 *Partitions of the example-grid considered in Section 4.4.1 resulting from the multilevel recursive bisection algorithm.* 66

4.7 *Partitions of the SPE 10 dataset with* $6 \times 11 \times 17$ *coarse blocks, resulting from the multilevel recursive bisection algorithm.* 66

4.8 ***(a)** Highlighted blocks distributed to a process, giving $X_c = \{a, b, c, d, e, f, g, h\}$. **(b)** Highlighted cells belonging to D_c, E_c and F_c. Here, cells in E_c are blue, cells in F_c are red, and D_c are cells in both E_c and F_c, and the grey ones. Neighboring block-centers are colored green.* 67

4.9 *Setup times used by the original (shared memory) program and the new (distributed memory) program when executed on a single core on Vilje. Numbers over the red bars are absolute times used by the original program, and numbers over the green bars are relative speedups of the new program, with respect to the original.* 72

4.10 *Times used to perform a single iteration by the original (shared memory) program and the new (distributed memory) program when executed on a single core on Vilje. Numbers over the red bars are absolute times used by the original program, and numbers over the green bars are relative speedups of the new program, with respect to the original.* 73

4.11 *Speedup of the iteration procedure using the pure MPI approach for Test Case 1.* 75

4.12 *Fraction of time used in different parts of the iteration procedure by the pure MPI approach for Test Case 1.* 75

4.13 *Speedup of the iteration procedure for the pure MPI approach and the hybrid approach with 8 OpenMP threads in each MPI processes, for Test Case 2. Figure **(a)** is a closeup for a small number of cores.* 76

4.14 *Fraction of time used in different parts of the iteration procedure by the pure MPI approach for Test Case 2.* 77

4.15 *Speedup of the iteration procedure for the pure MPI approach and the hybrid approach with 8 OpenMP threads in each MPI processes, for Test Case 3. Figure **(a)** is a closeup for a small number of cores.* 78

4.16 *Fraction of time used in different parts of the iteration procedure by the pure MPI approach for Test Case 3.* 79

4.17 *Speedup of the iteration procedure for the pure MPI approach and the hybrid approach with 8 OpenMP threads in each MPI processes, for Test Case 4. Figure **(a)** is a closeup for a small number of cores.* 79

4.18 *Fraction of time used in different parts of the iteration procedure by the pure MPI approach for Test Case 4.* 80

4.19 ***(a)** Highlighted coarse blocks for Test Case 5, and **(b)** a small 2.5 PEBI-grid.* 81

4.20 *Speedup of the iteration procedure for the pure MPI approach and the hybrid approach with 8 OpenMP threads in each MPI process, for Test Case 5.* 81

4.21 *Fraction of time used in different parts of the iteration procedure by the pure MPI and Hybrid approach for Test Case 5.* 81

4.22 *Setup times for the pure MPI approach and the pure OpenMP approach for Test Case 1, and setup times for the pure MPI approach and the hybrid approach for Test Case 2 and 3. The numbers over the bars are the speedups.* 83

4.23 *Setup times for the pure MPI approach and the hybrid approach for Test Case 4 and 5. The numbers over the bars are the speedups.* 84

Chapter 1

Introduction

1.1 Background

Modeling of fluid flow in the subsurface has several important applications. Among these are storing of greenhouse gases, acquiring knowledge of groundwater basins and exploiting geothermal energy. During the past decades and up until today, a prime driving force behind technological progress in modeling subsurface flow is the challenge of recovering oil and gas from petroleum reservoirs.

A petroleum reservoir is a subsurface formation which has developed over millions of years. During these years, oil and gas has gradually been formed from organic material in a time-consuming process requiring high pressure, high temperature and limited inflow of oxygen. The oil and gas is contained within small void spaces, or pores, of the reservoir. In a good reservoir, these pores are connected to make up a continuous network where the fluids are able to flow. For this reason, a petroleum reservoir is regularly called a porous rock. The oil and gas can be extracted by drilling wells into this rock. The extraction can often be achieved to a certain extent by exploiting natural pressure differences, however, this driving force will decline as more oil is extracted. Therefore, modern technologies have developed multiple methods to enhance the pressure within the reservoir, to recover and increase extraction as natural driving forces decline.

1.1.1 Applications and Challenges for Reservoir Simulation

In recent years, the number of newly discovered petroleum reservoirs has been significantly reduced. This has resulted in an increasing demand to access oil and gas from reservoirs where the the extraction is more challenging. Further, a high amount of competition among oil companies requires them to increase cost efficiency. Among the challenges this offers is to maximize the recovery of oil and gas in a particular reservoir while minimizing expenses. To achieve this, it is often of vital importance to acquire knowledge of how the fluids in the reservoir rock are able to flow under changing conditions. In particular, this can help the oil companies to position wells in order to maximize extraction. This leads to a continuous demand for the technology to produce more accurate reservoir models and flow patterns.

The movement of fluids in a porous rock is driven by processes taking place on a large span of spatial scales. On the large scale the flow is governed by global forces like pressure differences and gravity, while on the pore-scale the flow is determined by tiny flow paths. While a hydrocarbon reservoir may extend over several square kilometers, the pores are often on the micrometers scale. It is obviously not possible to embed pore-scale details into a model of the entire reservoir. However, the goal in our context is to predict global flow patterns rather than micro-scale fluid displacements. For this purpose, the rock model is described by petrophysical properties like porosity, which is a measure of average void space in the rock, and permeability, which measures the rocks ability to allow fluids to pass through it. In petroleum reservoirs, the spatial distributions of these properties follow complex statistical and empirical relationships, and often exhibit great variations over short distances. This makes it a substantial challenge to predict the fluid flow. In Figure 1.1 you see the permeability field of a rock model visualized.

Among the important tools used to predict flow patterns is *reservoir simulation*, where a mathematical model of the fluid flow is used together with a model of the reservoir rock to predict the flow characteristics. Progress in modern technologies such as reservoir characterization and data integration techniques has provided increasingly complex and detailed models of the reservoirs. Today, these models typically have a resolution down to the meter scale. Considering the sizes of the reservoirs, this amounts to massive volumes of data, or even tens of millions of cells. To simulate flow on these models is a major challenge. Together with a continuous demand from the industry to predict increasingly accurate flow characteristics, this has contributed to much effort devoted to improving reservoir simulation techniques. However, despite major advances, they have not been able to keep up with the increasingly detailed reservoir descriptions and the requirements by

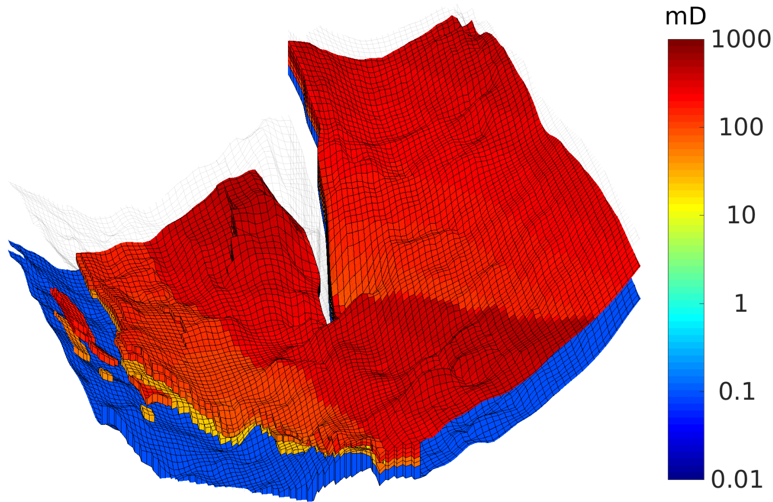


Figure 1.1: *Visualization of the permeability field for a geological rock model of the Johansen formation, located offshore the south-west coast of Norway. The formation has been considered a candidate location for CO_2 storage.*

the industry. Simulating flow directly on the scale provided by todays rock models are typically too computational demanding, even with state-of-the-art computer power. We say that there exists a resolution gap between the provided rock models and the capability of simulation technologies. The traditional approach to deal with this resolution gap is through *upscaling techniques* [45].

The strategy behind upscaling techniques is to use the original rock model to create a reduced problem. The objective is for the resulting coarse model to embed high quality upscaled properties which can be used to compute relatively accurate flow solutions. But upscaling techniques tend to have problems in computing accurate and robust solutions to models that exhibit high heterogeneity with no clear scale separation, which is the normal case for reservoir rocks.

1.1.2 Multiscale Methods

More recently, so-called *multiscale methods* have been developed in attempt to overcome the shortcomings of upscaling techniques. In the same way as with upscaling, these methods rely on a coarse partition of the underlying fine grid to create a reduced problem. But in contrast to upscaling, the methods produce flow solutions on the fine scale.

To describe the elementary idea behind multiscale methods, we consider the

variable-coefficient Poisson's equation,

$$\nabla \cdot \big(\boldsymbol{K}\nabla p\big) = -q,$$

where p is the fluid pressure, q is the source terms, and $\boldsymbol{K}$ is the permeability tensor that may undergo large variations over short distances. The above equation is discretized using a cell-centered finite-volume method, resulting in the linear system of equations for the pressure solution,

$$\boldsymbol{A}\boldsymbol{p} = \boldsymbol{q}.$$

To solve this system, multiscale methods rely on dividing the initial fine-scale problem into several local problems, which are then solved. Their solutions, called basis functions, are used to assemble a *prolongation operator* $\boldsymbol{P}$, that maps quantities on the coarse grid to quantities on the fine grid. In addition, a *restriction operator* $\boldsymbol{R}$ is created, which is an analogous map going the other way. The prolongation and restriction operators are related to the operators by the same names used in traditional *multigrid methods* [41, 42]. The two operators are used together with the fine-scale cell-connections to develop the coarse-scale problem,

$$(\boldsymbol{R}\boldsymbol{A}\boldsymbol{P})\boldsymbol{p}_c = \boldsymbol{R}\boldsymbol{q}.$$

After the coarse solution has been found, the prolongation operator is used once more to reconstruct a pressure approximation on the fine scale,

$$\boldsymbol{p} = \boldsymbol{P}\boldsymbol{p}_c,$$

which can be used to find a fine-scale velocity approximation for the fluid flow. The idea behind this approach is to separate effects determining flow on the local and global scales. While global driving forces are captured by the coarse-scale solution, the local solutions incorporated within the prolongation operator will systematically correct for fine-scale variations in the reservoir rock.

The development of multiscale methods started with the formulation of the multiscale finite-element (MsFE) method by Hou and Wu in 1997 [17]. Since then, several other multiscale methods have been proposed, and much effort has been made to advance these techniques to handle problems of real-world complexity. Todays industry requests methods that can handle very flexible simulation grids, complex flow physics, and are simple to integrate into existing frameworks.

Over the past decade, essentially two multiscale methods have been developed in this direction, the *multiscale finite-volume* (MsFV) method [18, 19, 20] and the *multiscale mixed finite-element* (MsMFE) method [12, 7, 8]. In contrast to the

MsFE method, these two provide conservation of mass for the fine-scale velocity, which is a crucial requirement when solving fine-scale transport problems.

Since the MsFV method was first formulated, it has been extended from an original geometric form to algebraic form [46, 47, 33]. This has played an important role to reduce the complexity of the implementations, permitting the method to be manipulated and extended in an efficient manner, and to allow for simple integration into existing reservoir simulators.

A main focus in the development of the MsFV method has been to expand it from handling simple incompressible flow to more realistic flow physics [30, 31, 21, 15, 27], such as compressible and multi-phase flow, e.g, compressible black-oil models. Along this dimension, the method has come very far. However, the MsFV method has shown to often produce significant errors when applied to models with very heterogeneous properties [32, 44].

This was one of the factors that motivated iterative formulations of the method [14, 47, 34]. It has been shown that these iterative schemes have the desired property of converging to the fine-scale reference solution, granting a systematic approach to increase the accuracy of the solution to a given tolerance. In addition, the schemes can also be used as an efficient linear solver. A key advantage of these formulations compared with multigrid and domain-decomposition methods is that they allow for reconstruction of conservative fluxes at any state of the iteration.

However, a main limitation of the current MsFV formulations is to handle grids of industry complexity. Here, there has been far less progress. A main challenge encountered when attempting to extend the method to handle more complex grids arise from the fact that it relies on a primal-dual coarse partitioning. An important step was made in this direction when O. Møyner and K-A. Lie extended the MsFV method in an implementation that could handle Stratigraphic Grids with faults and wells [40]. In this work, they present coarsening methods that create the required primal-dual partition for a wide class of stratigraphic grids. Their results show that the method produces accurate results to many complex flow problems. However, in cases with strong heterogeneities, the method may produce very large errors. They conclude that it remains a challenge to create reliable and robust coarsening algorithm for the MsFV method.

On the other hand, the MsMFE method, which does not require a dual coarse grid partitioning, has come far to allow the use of very complex and flexible grids [9, 10, 11]. However, it has proved hard to extend the method to realistic flow physics, and still today the MsMFE formulations are primarily able to produce reliable results only on incompressible and weakly compressible flow models.

The multiscale Two-point Flux Approximation (MsTPFA) method [37] was

proposed as an attempt to combine the best features of the MsFV and MsMFE methods. It is derived based mainly on the same principles as MsFV, but removes the need for a dual coarse partition. This makes it much more flexible with respect to the coarse partitioning. The MsTPFA method was shown to be significantly more robust than MsFV on models with high heterogeneity. However, it is less accurate on problems with smooth heterogeneities, and is somewhat intricate to implement due to the use of both grid-specific partition-of-unity functions and local flow problems.

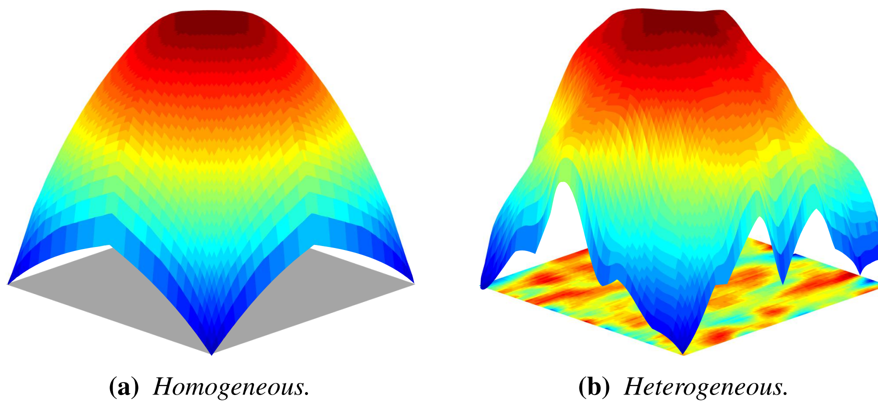
1.1.3 Restricted-Smoothed Basis Functions

A newly proposed approach is the *multiscale restricted-smoothed basis* (MsRSB) method [38, 39]. It is connected to the MsTPFA method in the sense that it is based on the underlying principles of the MsFV method, and does not require a dual coarse partitioning. However, it applies a quite different strategy to create the prolongation operator.

The prolongation operator is formed through what we call a restricted-smoothing iterative procedure applied to the basis functions (localized problems). The pressure values of each basis function is initialized as the characteristic function on its corresponding coarse subdomain. A relaxed Jacobi smoother is then iteratively applied to the basis functions, making them gradually adapt to the local permeability field of the rock. The iteration is restricted to force the basis functions to remain within their defined support regions, and to preserve partition of unity for the union of basis functions. In Figure 1.2 you see two such basis functions resulting from both homogeneous and heterogeneous permeability fields.

A key advantage of the iterative approach is that the basis functions can be modified inexpensively by applying a few extra steps on the existing ones, which is advantageous when faced with dynamic mobility changes in the rock. The MsRSB method has proved to be accurate and robust when compared with other existing multiscale methods [38, 39, 36]. Like the MsMFE method it is flexible concerning the fine and coarse grids, and like the MsFV method it is can handle complicated flow physics used by the industry. Further, the method is formed in a fully algebraic manner, is easy to implement and relatively cheap to compute.

At the time of writing, the MsRSB method is considered the state-of-the-art multiscale method. For a good review of the development of multiscale method, we refer to the recent article K-A. Lie et al. (2016) [29].



(a) *Homogeneous.* **(b)** *Heterogeneous.*

Figure 1.2: *A basis function for a two-dimensional rock model with homogeneous and heterogeneous permeability fields, as computed by the MsRSB method.*

1.1.4 Parallel Computing

As already established, reservoir simulations are in general very computationally demanding, even when the most effective solvers are used. It is therefore desirable to harness the computational power of modern supercomputers. These computers consist of a large number of processing units, or cores, each able to perform their own computations.

Designing a program to efficiently take advantage of this structure offers two main challenges. Firstly, it relies on distributing the computations evenly among the processing units, to achieve so-called *load balance*. In particular, an effective distribution should minimize the work executed by any processing units. The unit distributed the largest amount of work will be a typical weakest link, preventing the full program to achieve a better run-time. Secondly, it relies on minimizing the communication carried out between the processing units, since sending and receiving of data typically consume a significant fraction of the total runtime, especially when using a large number of cores. To what extent such a division can be achieved will typically rely on the underlying algorithm to be computed. In most situations, an optimal load-balance can not be achieved at the same time as minimizing the communication cost, and finding a good compromise between the two consideration can be hard. An algorithm ideally suited for parallelism will consist of a large number of entirely independent computations, but this is rarely the situation in practice. The growing use of supercomputers has therefore resulted in an increasing importance to develop algorithms well suited for multiprocessor computations.

To measure how efficient a program utilizes the available processing units, we use the so-called *speedup*. Let t_p be the time it takes to run a given program on p processing cores. The speedup s_p is defined as

$$s_p = \frac{t_1}{t_p},$$

that is, it equals the ratio between the runtime on a single core and on p cores. The best we can hope to achieve is for the program to run p times as fast on p cores than on one, which we call a perfect speedup. In practice, however, the speedup typically declines by increasing the number of processing cores. When we do this, each core is assigned less work while the communication overhead typically increases. How well the speedup scales when adding more cores depends on the total amount of work to be performed and by how fast the communication volume increases as a function of cores. Figure 1.3 illustrates the difference in perfect speedup and what we normally see in practice. As you can see, adding more processing units will at some point stop being beneficial.

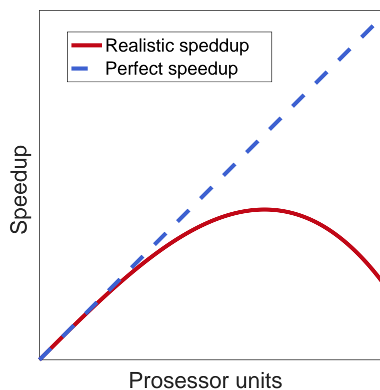


Figure 1.3: *Perfect speedup and realistic speedup as functions of the number of processing units.*

The hardware structure of the supercomputer impacts how the parallelization should and must be carried out. Here, the main dividing line runs between *shared memory machines* and *distributed memory systems*.

As the name implies, a shared memory machine consists of multiple cores that have access to the same memory. On these machines, the *OpenMP language extension* [5] is widely used to realize parallel computations in a quick and simple manner. The OpenMP API offers a framework existing of a collection of compiler directives, environment variables, and library routines which can be used to employ parallel computations. It realizes multithreading under a fork-join approach,

where processes, or threads as we will call them, are spawned and handled by the compiler on the available cores. This approach often allows parallel computation to be accomplished without extensive changes to the sequential code.

A distributed memory system consists of multiple *nodes* who each have their own private memory. Here, a node is a self-contained computer unit that typically consist of multiple processing cores. When executing a program on multiple nodes, one or more separate instance of the program will be spawned on each of them. Communication between the nodes must be carried out through explicit message-passing. Therefore, parallelizing a program for distributed memory systems usually requires extensive changes in the sequential code. The *Message Passing Interface* (MPI) [3] is a standard portable message-passing system used for the communication. MPI provides syntax and semantics of library routines for C, C++ and Fortran, and offers both point-to-point and collective communication operators. Because of its portability and efficiency it has become the de factor standard for message passing. Using MPI is more comprehensive than OpenMP, since the developer must take into account both load balance and communication, which for OpenMP is handled by the compiler.

A supercomputer normally consists of a cluster of nodes that communicate through a high-speed interconnect. A much used parallelization strategy is therefore to apply a hybrid approach of the two models considered above, using a shared memory programming model within each node, and a message massing model to communicate across them. This often leads to a more efficient execution than the pure message-passing alternative. Note that the message passing model can be used both on shared and distributed memory machines, while the shared memory model is only applicable on shared memory machines.

In this thesis we focus on supercomputer applications. However, today's desktop computers are also multiprocessing machines, typically with 2, 4 and also 8 processing cores. Even smartphones normally have more than one core. The importance of efficient parallel computations is therefore not limited to massively parallel supercomputing, but emerges also in the everyday use of modern technology.

1.1.5 Matlab Reservoir Simulation Toolbox

The *Matlab Reservoir Simulation Toolbox* (MRST) [4, 28] has been an important tool for the work presented in this thesis. MRST is a Matlab toolbox developed by the Computational Geoscience group in the Department of Applied Mathematics at SINTEF ICT, and is primarily intended as a toolbox for rapid prototyping

and demonstration of new simulation methods and modeling concepts on unstructured grids. The toolbox contains a set of data structures for representing reservoir rock models, including grids, rock properties, and forcing terms such as gravity, boundary conditions and source terms. It also contains several solvers, including the MsRSB method, and visualization functionality. In this thesis it has been used to construct reservoir models together with the corresponding system of equations to be solved, testing new solving strategies, and to create most visual illustrations throughout this presentation.

1.2 Structure of the Thesis

This thesis explores how to improve the computational efficiency of constructing the prolongation operator used by the MsRSB method. Here, two prime subjects have been investigated.

The first subject concerns how to advance the underlying construction algorithm. We explore how to improve the efficiency of the smoothing step, in attempt to reduce the computational cost needed to reach a predefined quality. In particular, we aim to develop Gauss-Seidel-type smoothers with higher convergence rate than the originally used relaxed Jacobi smoother.

The second subject regards developing a program that constructs the prolongation operator on distributed memory systems. Here, a hybrid approach is investigated, which combines the use of MPI and openMP.

The rest of the thesis is organized as follows.
Chapter 2 starts by presenting a flow model and derives the system of equations to be solved, before describing the Jacobi and Gauss-Seidel methods. It continues by presenting the MsRSB method and the algorithm used to construct the prolongation operator. The chapter ends by demonstrating the use of the MsRSB method by applying in on a model-problem. After this, Chapter 3 starts by presenting a modified version of the construction algorithm presented in Chapter 2, followed by an analysis of its application. It continues by considering replacing the relaxed Jacobi smoother used in the construction algorithm by the Gauss-Seidel method, and shows that this requires modifications to provide an efficient alternative. Two alternative smoothers are then presented, both of which are modified versions of the Gauss-Seidel method. The chapter concludes by presenting numerical results, comparing the modified construction algorithm to its original, and examining the new smoothing approaches. Chapter 4 presents the C++ program that constructs the prolongation operator, which enables computation on distributed memory system through the use of explicit message-passing. The beginning of the chapter

describes sparse data structures that are used, before presenting the program when executed sequentially. Thereafter follows a discussion of how the computational work is distributed among MPI processes, and the message-passing program is then introduced. Runtime results from testing the program on the supercomputer Vilje are then presented, followed by introducing a strategy of how to reduce the communication cost. The chapter ends with a discussion of the results and ideas for future work. The thesis concludes by a summary and discussion of the main results, presented in Chapter 5.

Chapter 2

Problem Derivation and Method Presentation

This chapter is organized as follows. Section 2.1 gives a brief introduction to the governing equations for flow in porous media, which are the model equations for this thesis. Section 2.2 presents the Jacobi method and various Gauss-Seidel methods. A Jacobi iteration is used in the procedure that constructs the prolongation operator, and later on we discuss changing it to Gauss-Seidel. In Section 2.3 we describe the coarse grid and provide a framework that enables us to relate to different parts of the grid which we use when formulating the MsRSB method. Section 2.4 then presents the MsRSB method by defining the coarse-scale system of equations to be solved, and illustrates how its solution is used to construct a fine-scale flux approximation. The algorithm that constructs the prolongation operator is presented in Section 2.5. The chapter concludes by illustrating the MsRSB method on a model-problem, which is the content of Section 2.6.

2.1 Flow Model and Discretization

In the following we will consider a single-phase, incompressible flow model, where the permeability and the source terms are constant in time. The model is derived from *Darcy's law* and *the fundamental law of mass conservation*. We will then show how the system is discretized from a two-point flux-approximation to arrive at the system of linear equations for the steady state pressure distribution.

2.1.1 Flow Model

Darcy's law can be stated as follows,

$$\boldsymbol{v} = -\frac{\boldsymbol{K}}{\mu}\big(\nabla p - g\rho\nabla z\big).$$

Here, $\boldsymbol{v}$ is the *macroscopic Darcy velocity*, $\boldsymbol{K}$ is the rock permeability, μ is the *dynamic viscosity* of the fluid, p is the pressure, g is the gravitational acceleration, ρ is the fluid density and z is the vertical coordinate. Henceforth, μ will without lack of generality be set to unity. We can continue by either introducing the fluid potential $\boldsymbol{\Phi} = p - g\rho z$, or by neglecting the gravitational force. We chose to do the latter, and the above equation simplifies to

$$\boldsymbol{v} = -\boldsymbol{K}\nabla p.$$

The fundamental law of mass conservation on differential form can be stated as follows,

$$\frac{\partial}{\partial t}(\phi\rho) + \nabla\cdot(\rho\boldsymbol{v}) = \rho q,$$

where ϕ is is the *effective porosity* of the rock and q is the fluid source term. We assume incompressible fluid and constant porosity, $\frac{\partial}{\partial t}(\phi\rho) = 0$, and the relationship becomes

$$\nabla\cdot\boldsymbol{v} = q.$$

Proceeding by introducing Darcy's law into the above relation we arrive at the equation

$$\nabla\cdot\Big(\boldsymbol{K}(\boldsymbol{x})\nabla p(\boldsymbol{x})\Big) = -q(\boldsymbol{x}), \qquad \boldsymbol{x}\in\boldsymbol{R}^d,\ \boldsymbol{K}(\boldsymbol{x})\in\mathbb{R}^d\times\mathbb{R}^d. \tag{2.1}$$

This is a generalization of the *Poisson's equation* with varying coefficients. The dimension d will be 2 or 3 for all cases considered in this paper. Note that the permeability $\boldsymbol{K}$ can in the most general case be a full tensor.

We will continue by deriving the discretized equivalent of equation (2.1), after a few words about the grid. The fine grid is a *tessellation* of the planar or volumetric object that constitutes the model of the rock. The tessellation results in a set of contiguous simple shapes which we call *cells*. In general, a cell will be a polygon or a polyhedron in two and three dimensions, respectively. We will use numbering

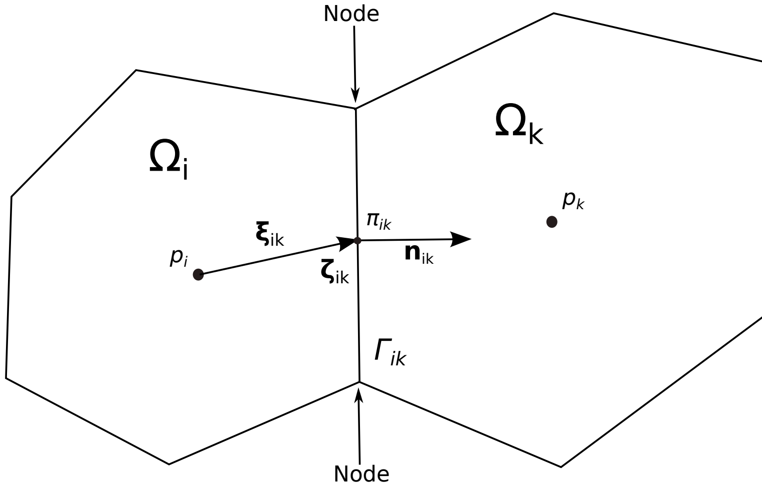


Figure 2.1: *Example of two connected cells resulting from a grid-partitioning.*

of the cells from 1 to n, and relate to a certain cell Ω_i by its number. Let F be the set of cell-numbers,

$$F = [1, 2, \ldots, n-1, n],$$

and denote the partitioning of the grid as the set of cells,

$$\{\Omega_i | i \in F\}.$$

We say that two cells Ω_i and Ω_k are *connected* if they share a face, that is, if $(\partial\Omega_i \cap \partial\Omega_k \neq \emptyset)$. Further, we say that two cells are *neighbors* if they share a node. Here, a node refers to a connection point of the tessellation that makes up the grid. The face between two connected cells Ω_i and Ω_k is denoted Γ_{ik}. Figure 2.1 shows two such cells.

2.1.2 Two-Point Flux-Approximation

We continue by discretizing the equation (2.1) by regarding the cells as control volumes. The flux across the cell-face in Figure 2.1 is denoted v_{ik}, and is defined by

$$v_{ik} = \int_{\Gamma_{ik}} \boldsymbol{v} \cdot \boldsymbol{n} \, ds, \quad \Gamma_{ik} = \partial\Omega_i \cap \partial\Omega_k.$$

Here, $\boldsymbol{n}$ is the normal vector pointing into cell Ω_k. We approximate this flux using the *midpoint rule*,

$$v_{ik} \approx A_{ik}\boldsymbol{v}(\boldsymbol{\zeta}_{ik}) \cdot \boldsymbol{n}_{ik},$$

where A_{ik} is the area of the face, $\boldsymbol{\zeta}_{ik}$ is the centroid of the face, and $\boldsymbol{n}_{ik}$ is its corresponding normal vector. Applying Darcy's law in the above equation we obtain

$$v_{ik} \approx -A_{ik}(\boldsymbol{K}\nabla p)(\boldsymbol{\zeta}_{ik}) \cdot \boldsymbol{n}_{ik}. \tag{2.2}$$

Define p_i to be the pressure in the center of Ω_i. Further, define π_{ik} to be the pressure at the face centroid, and $\boldsymbol{\zeta}_{ik}$ the vector from the cell center to the face centroid. We approximate $\nabla p(\boldsymbol{\zeta}_{ik})$ by assuming a linear change in pressure between the center of Ω_i and $\boldsymbol{\zeta}_{ik}$,

$$\nabla p(\boldsymbol{\zeta}_{ik}) \approx -\frac{(p_i - \pi_{ik})\boldsymbol{\xi}_{ik}}{|\boldsymbol{\xi}_{ik}|^2},$$

where $\boldsymbol{\xi}_{ik}$ is the vector from the center of Ω_i to $\boldsymbol{\zeta}_{ik}$. Inserting the above equation into (2.2) we arrive at

$$v_{ik} \approx A_{ik}\boldsymbol{K}_i\frac{(p_i - \pi_{ik})\boldsymbol{\xi}_{ik}}{|\boldsymbol{\xi}_{ik}|^2} \cdot \boldsymbol{n}_{ik} = \tilde{T}_{ik}(p_i - \pi_{ik}), \quad \tilde{T}_{ik} = A_{ik}\boldsymbol{K}_i\frac{\boldsymbol{\xi}_{ik}}{|\boldsymbol{\xi}_{ik}|^2} \cdot \boldsymbol{n}_{ik}.$$

Here, $\boldsymbol{K}_i$ is the permeability inside cell Ω_i. We name $\tilde{T}_{ik}$ the *half-transmissibility* from Ω_i to Ω_k. Imposing continuity of flux across the face, i.e, $v_{ik} = -v_{ki}$, gives us

$$\tilde{T}_{ik}^{-1}v_{ik} = p_i - \pi_{ik}, \quad -\tilde{T}_{ki}^{-1}v_{ik} = p_k - \pi_{ik}.$$

We eliminate $\pi_{i,k}$, and get

$$v_{ik} = \frac{1}{\tilde{T}_{ik}^{-1} + \tilde{T}_{ki}^{-1}}(p_i - p_k) = T_{ik}(p_i - p_k), \quad T_{ik} = T_{ki} = \frac{1}{\tilde{T}_{ik}^{-1} + \tilde{T}_{ki}^{-1}}, \tag{2.3}$$

where T_{ik} is the *transmissibility* associated with the connection between the two cells. We require mass conservation on each cell, which on integral form can be stated as follows,

$$\int_{\partial\Omega_i} \boldsymbol{v} \cdot \boldsymbol{n}\, ds = \int_{\Omega_i} q\, d\boldsymbol{x}.$$

Inserting (2.3) into this equation, the following system is obtained,

$$\sum_k T_{ik}(p_i - p_k) = q_i, \quad \forall\, i \in F.$$

The system matrix $\boldsymbol{A} = \{a_{ij}\}$ can now be assembled from

$$a_{ij} = \begin{cases} \sum_k T_{ik} & \text{if } j = i, \\ -T_{ij} & \text{if } j \neq i, \end{cases}$$

and we arrive at the resulting linear system of equations,

$$\boldsymbol{A}\boldsymbol{p} = \boldsymbol{q}. \tag{2.4}$$

We will now state important properties of the linear system. Firstly, each row of $\boldsymbol{A}$ sums to zero,

$$\sum_{k=1}^{n} a_{ik} = 0, \quad \forall\, i \in F. \tag{2.5}$$

Secondly, an element a_{ik} of the system matrix is zero if cell i and k are not connected,

$$a_{ik} = 0, \quad \text{if } \ k \neq Q_i. \tag{2.6}$$

Here, Q_i is the set of all cell-numbers connected to Ω_i,

$$Q_i = \big\{k \big| (\partial\Omega_i \cap \partial\Omega_k \neq \emptyset), i \neq k\big\}.$$

And lastly, the linear system is symmetric and its solution is defined up to an arbitrary constant.

As already mentioned, the system of equations (2.4) resulting from a typical rock model is often very computationally demanding to solve directly. This has two main reasons. Firstly, the system is normally very large, which is a result of the vast spatial size of the reservoir and the relatively detailed model provided. Secondly, the permeability field of the rock is typically highly heterogeneous, and may undergo vast variations over short distances. This results in the system matrix $\boldsymbol{A}$ having highly variable coefficients, and will therefore typically be very ill-conditioned. In Section 2.4, the MsRSB method is introduced which can be used to develop a reduced system in order to provide an approximate or full solution in an computationally efficient manner. Keep in mind that the MsRSB method can be applied to several more complicated flow models than the one derived here,

including multi-phase and compressible flow models. In fact, the MsRSB method can be applied to any flow model where one can isolate a pressure equation.

Note that if the flow model derived above is applied to a rock model with homogeneous permeability, the resulting problem to be solved would be the simple Poisson's equation.

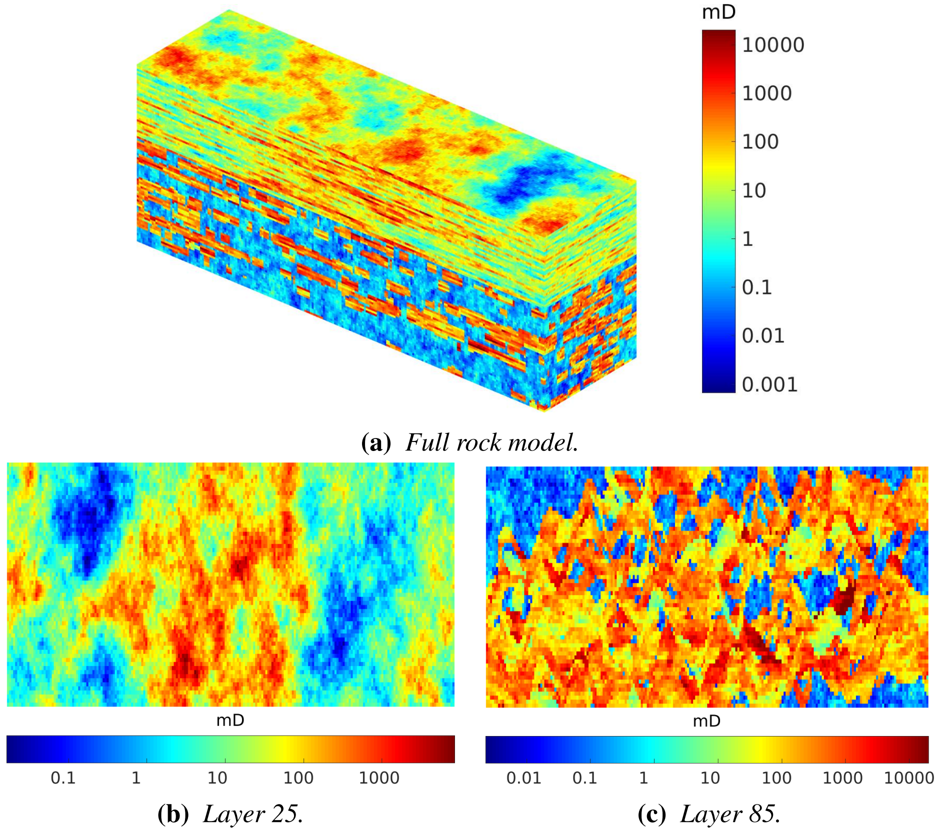
2.1.3 SPE 10 Dataset

A rock model we will use as a test case in this thesis is the 10th Comparative Solution Project [13], published by the Society of Petroleum Engineers. We will refer to it as the SPE 10 dataset. The model uses a Cartesian grid consisting of $60 \times 220 \times 85$ cells, each of size $20 \times 10 \times 2$ feet. It was designed as a challenging benchmark for upscaling methods, and is today frequently used to test multiscale methods. Figure 2.2a visualizes the permeability field of the SPE 10 dataset.

The SPE 10 dataset consists of two rock-formations that have qualitatively different permeability fields. Both of them have large heterogeneities, varying with 8-12 orders of magnitude. The upper 35 layers represent a geostatistical realization of the Tarbert formation. The permeability of this formation follows a lognormal distribution, which gives relatively smooth variations in the heterogeneity and good communication in all spatial directions. The lower 50 layers represent the Upper Ness formation. It combines long correlation lengths of high permeability with rapidly abrupt changes. Approximately 2.5% of its cells have zero porosity and are therefore regarded as being inactive, allowing no fluid to be stored or pass through them. Upper Ness has proved to be a very challenging formation for upscaling and multiscale techniques to solve correctly. Figure 2.2b and 2.2c show the permeability fields of Layers 25 and 85, respectively, and illustrate the qualitatively different structure of the two rock-formations.

2.2 Jacobi and Gauss-Seidel Methods

In Section 2.5 we will see that the prolongation operator used by MsRSB is created by an iterative algorithm which applies the relaxed Jacobi method to make the prolongation operator so-called algebraically smooth. In Chapter 3, we present alternative smoothers used in this construction which is based on modified Gauss-Seidel methods. This section is therefore included to provide background knowledge of both Jacobi and Gauss-Seidel methods, which we will see are very closely related. Although they have slow convergence rate in general, they inhabit the



(a) *Full rock model.*

(b) *Layer 25.*

(c) *Layer 85.*

Figure 2.2: *Horizontal permeability field of the SPE 10 rock model. A logarithmic color-scale is used.*

smoothing property of rapidly dampening the high-frequency components of the residuals. We will return to the purpose of the smoother in Section 2.5.

Consider solving our system of linear equations $\boldsymbol{A}\boldsymbol{p} = \boldsymbol{q}$ with the Jacobi method, defined by the following iterative procedure. Denote $\boldsymbol{p}^k$ the approximation of the solution $\boldsymbol{p}$ at step k. The ith component of the next approximation is chosen to eliminate the ith component of the residual vector

$$\tilde{\boldsymbol{r}}^k = \boldsymbol{q} - \boldsymbol{A}\tilde{\boldsymbol{p}}_i^{k+1},$$

where

$$\tilde{\boldsymbol{p}}_i^{k+1} = \left(p_1^k, \quad p_2^k, \quad \cdots, \quad p_{i-1}^k, \quad p_i^{k+1}, \quad p_{i+1}^k, \quad \cdots, \quad p_{n-1}^k, \quad p_n^k\right)^T.$$

On component form, the next approximations reads

$$p_i^{k+1} = \frac{1}{a_{ii}}\left(q_i - \sum_{j=1}^{i-1} a_{ij}p_j^k - \sum_{j=i+1}^{n} a_{ij}p_j^k\right), \quad i \in F. \tag{2.7}$$

A relaxed version of the Jacobi method is defined by

$$\begin{aligned} p_i^{k+1} &= (1-\omega)p_i^k + \frac{\omega}{a_{ii}}\left(q_i - \sum_{j=1}^{i-1} a_{ij}p_j^k - \sum_{j=i+1}^{n} a_{ij}p_j^k\right) \\ &= p_i^k + \frac{\omega}{a_{ii}}\left(q_i - \sum_{j=1}^{n} a_{ij}p_j^k\right), \qquad i \in F. \end{aligned}$$

where ω is the *relaxation factor*.

To motivate the Gauss-Seidel (GS) iteration, assume that we solve the equations (2.7) sequentially, starting at $i = 1$. Gauss-Seidel iteration is the result of changing the formula to immediately start using the updated components of $\boldsymbol{p}^{k+1}$ in the computation of the consecutive elements. On component form, a forward Gauss-Seidel sweep reads

$$p_i^{k+1} = \frac{1}{a_{ii}}\left(q_i - \sum_{j=1}^{i-1} a_{ij}p_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}p_j^k\right), \quad i \in F.$$

A backward Gauss-Seidel sweep is defined analogously, but the updates are executed in the opposite order, starting at the last element and ending at the first. On component form, it is given by

$$p_i^{k+1} = \frac{1}{a_{ii}}\left(q_i - \sum_{j=1}^{i-1} a_{ij}p_j^k - \sum_{j=i+1}^{n} a_{ij}p_j^{k+1}\right), \quad i \in F.$$

The *successive overrelaxation* (SOR) method is a relaxed version of the forward Gauss-Seidel method. On component form, it reads

$$p_i^{k+1} = \frac{\omega}{a_{ii}}\left(q_i - \sum_{j=1}^{i-1} a_{ij}p_j^k - \sum_{j=i+1}^{n} a_{ij}p_j^{k+1}\right) + (1-\omega)x_i^k, \quad i \in F.$$

In the cases of the relaxed formulations, choosing an appropriate ω can improve the convergence rates and smoothing properties of the iterations. However, a good choice if often hard to find and depends on the properties of $\boldsymbol{A}$. The relaxation factor is typically located within the interval $\omega \in [0, 2]$, where $\omega < 1$ is called

under-relaxation, while $\omega > 1$ is called over-relaxation. Although under-relaxation typically achieves best results for the Jacobi method, $\omega \in [1, 2]$ has been found to be the optimal choice for the Gauss-Seidel method on some particular problems.

To express the above procedures in matrix notation we use the following decomposition of the system matrix [41, p. 103],

$$\boldsymbol{A} = \boldsymbol{D} - \boldsymbol{L} - \boldsymbol{U},$$

where $\boldsymbol{D}$ is the diagonal, $-\boldsymbol{L}$ the strict lower, and $-\boldsymbol{U}$ the strict upper part of $\boldsymbol{A}$. The relaxed Jacobi method in matrix form can be stated as follows,

$$\begin{aligned}\boldsymbol{p}^{k+1} &= \omega \boldsymbol{p}_{\mathrm{J}}^{k+1} + (1-\omega)\boldsymbol{p}^k \\ &= \omega(\boldsymbol{p}^k - \boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{p}^k + \boldsymbol{D}^{-1}\boldsymbol{q}) + (1-\omega)\boldsymbol{p}^k \\ &= \boldsymbol{p}^k - \omega \boldsymbol{D}^{-1}(\boldsymbol{A}\boldsymbol{p}^k - \boldsymbol{q}),\end{aligned}$$

where $\boldsymbol{p}_J^{k+1}$ is the non-relaxed Jacobi update, obtained by choosing $\omega = 1$. In matrix form the forward Gauss-Seidel method can be stated in the following way,

$$(\boldsymbol{D} - \boldsymbol{L})\boldsymbol{p}^{k+1} = \boldsymbol{U}\boldsymbol{p}_k + \boldsymbol{q},$$

and SOR as

$$(\boldsymbol{D} - \omega\boldsymbol{L})\boldsymbol{p}^{k+1} = \big(\omega\boldsymbol{U} + (1-\omega)\boldsymbol{D}\big)\boldsymbol{p}^k + \omega\boldsymbol{q}.$$

Note that these methods can also be interpreted as fixed point iterations on the system $\boldsymbol{A}\boldsymbol{p} = \boldsymbol{q}$ after a preconditioning matrix $\boldsymbol{M}$ has been applied. For relaxed Jacobi, the preconditioning matrix is

$$\boldsymbol{M}_{\mathrm{JA}} = \omega\boldsymbol{D},$$

while for Gauss-Seidel it is

$$\boldsymbol{M}_{\mathrm{GS}} = \boldsymbol{D} - \boldsymbol{L}.$$

If we were to actually solve our system $\boldsymbol{A}\boldsymbol{p} = \boldsymbol{q}$ with one of these methods, we would have to adjust it so that the solution is defined uniquely. This can be achieved by fixing the pressure in a single cell, i.e, modify a single row in $\boldsymbol{A}$ by setting all its entries equal to zero except the diagonal which we set to unity. The matrix would now be *irreducibly diagonally dominant*, which is a sufficient criteria for the Jacobi and Gauss-Seidel methods to converge [41, p. 118]. In general, the asymptotic convergence rate depends on the spectral radius of $\boldsymbol{A}$, but is of less importance when the methods are used as smoothers.

2.2.1 Red-black Gauss-Seidel

As indicated by the existence of both the forward and backward Gauss-Seidel methods, the order in which the elements are updated by GS matters, in contrast to the Jacobi method. To see how the ordering impacts the next approximation, consider applying a single step of the forward and backward GS methods to some initial guess $\boldsymbol{p}^0$. The first and the last element of $\boldsymbol{p}^1$ will be computed from the following expressions,

$$\text{Forward GS} \quad p_1^1 = \frac{1}{a_{11}}\left(q_1 - \sum_{j=2}^{n} a_{1j}p_j^0\right), \quad p_n^1 = \frac{1}{a_{nn}}\left(q_n - \sum_{j=1}^{n-1} a_{1j}p_j^1\right),$$

$$\text{Backward GS} \quad p_1^1 = \frac{1}{a_{11}}\left(q_1 - \sum_{j=2}^{n} a_{1j}p_j^1\right), \quad p_n^1 = \frac{1}{a_{nn}}\left(q_n - \sum_{j=1}^{n-1} a_{1j}p_j^0\right).$$

Observe that p_1^1 as computed by forward GS coincides with the Jacobi method. The same is true for p_1^n when computed by backward GS. On the other hand, p_1^n is computed solely from updated elements in the case of forward GS, which is the same case for p_1^1 when computed by the backward method. Picture Gauss-Seidel as iteratively sweeping through the elements of the column vector $\boldsymbol{p}$, updating each element based on the latest update of all other values of the vector. Forward GS sweeps through $\boldsymbol{p}$ starting the top and moving downwards, while backward GS starts at the bottom and moves upwards. In general, one can imagine the sweep being done in any order. This can be expressed by renumbering the elements in $\boldsymbol{p}$, perturbating the matrix accordingly, and applying the forward GS method, or as we will do, by defining the order with the vector $\boldsymbol{\nu}$. The vector contains the set of integers between 1 and n ordered the way we wish to update the elements. A Gauss-Seidel step with this ordering can now be expressed as

$$p_{\nu_i}^{k+1} = \frac{1}{a_{\nu_i\nu_i}}\left(q_{\nu_i} - \sum_{j=1}^{i-1} a_{\nu_i\nu_j}p_{\nu_j}^{k} - \sum_{j=i+1}^{n} a_{\nu_i\nu_j}p_{\nu_j}^{k+1}\right), \quad i \in F.$$

A common version of a non-trivial ordered Gauss-Seidel iteration is the *red-black* Gauss-Seidel method. Consider solving the system (2.4) on a two-dimensional Cartesian grid, resulting in the regular five point stencil. Color the cells red and black in the same manner as a checkerboard, as illustrated by Figure 2.3a. A GS update of a single cell is now computed solely from cells of opposite color, as illustrated by Figure 2.3b. Consider applying the GS sweep ordered so that all black cells are updated before all the red ones. The updates of the black cells will coincide with a Jacobi update, since they are computed solely from values that

Figure 2.3: *(**a**) Red-black colored Cartesian grid, and (**b**) the regular five-point stencil.*

were updated in the previous step, while the updates of the red cells will coincide with what a Jacobi update would result in at the next step, since they are computed solely from values that have been updated in the ongoing step. This results in sort of a leaping-frog Jacobi method which approaches convergence twice as fast as regular Jacobi. The red-black Gauss-Seidel method is popular in parallel computing because it allows the values belonging to cells of the same color to be computed in parallel. In contrast, the forward and backward Gauss-Seidel method generally require the elements to be updated sequentially.

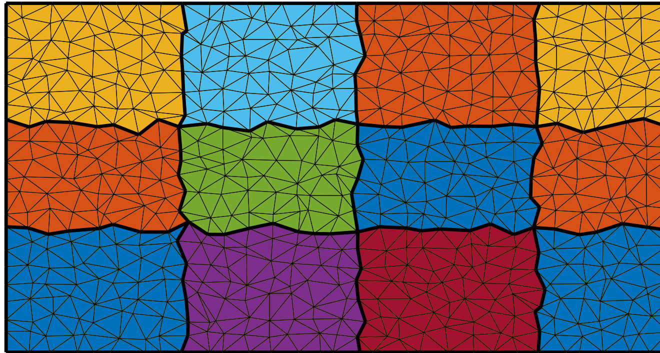


Figure 2.4: *A triangular grid partitioned into blocks.*

2.3 Coarse Grid

An important step towards developing a reduced system from Equation (2.4) is to partition the fine grid into coarse *blocks*, each of which consists of a number of contiguous cells. Figure 2.4 shows an example of such a partition, where 1200 fine cells have been divided into twelve blocks outlined by the bold black lines. This results in an average of 100 cells in each block, and we say that this coarse grid has an *upscaling factor* of 100. The partition of cells that defines the blocks

can be constructed in various ways, and in Figure 2.4 a uniform partitioning was used. In practice, the choice of partitioning can have huge impact on the accuracy when solving the model, be it by an upscaling or multiscale solver. To find an efficient partition, it should be tailor made to fit the reservoir rock. Consider applying an upscaling technique to a rock consisting of several *facies*. Here, a facies is a separate part of the rock that has specific characteristics. The difference in permeability between different facies might be large, while each facies is typically relatively homogeneous. In this situation it would be advantageous to let the face of the blocks follow the facies boundaries, to achieve a good approximation of the permeability in each block.

Before we continue by presenting the multiscale method, we will provide a set of variables that enables us to relate to different parts of the fine grid and the coarse blocks, which is the main content of this section. We also introduce the *prolongation operator*, the *basis functions*, and the *restriction operator* which are used in the multiscale formulation presented in the next chapter.

The grid is partitioned into m blocks, numbered from 1 to m. Denote the coarse partitioning by

$$\{\overline{\Omega}_j |\ j \in [1, \ldots, m]\},$$

and let C_j be the set of all cell-numbers belonging to each subdomain $\overline{\Omega}_j$,

$$C_j = \{i | \boldsymbol{\chi}_i \in \overline{\Omega}_j\}.$$

Here, $\boldsymbol{\chi}_i$ is the coordinate of the center of cell i. Define the prolongation operator

$$\boldsymbol{P} : \{\overline{\Omega}_j\} \rightarrow \{\Omega_i\},$$

which maps quantities on the coarse grid to quantities on the fine grid, and the restriction operator

$$\boldsymbol{R} : \{\Omega_i\} \rightarrow \{\overline{\Omega}_j\},$$

which is an analogous map going the other way. The prolongation operator will for discrete problems be represented by an $n \times m$ matrix, where the row-indices correspond to cell-numbers, and the column-indices correspond to block-numbers. The restriction operator will be represented by an $m \times n$ matrix, where the row-indices correspond to block-numbers, and the column-indices correspond to cell-numbers. We call column j of $\boldsymbol{P}$ the basis function number j, and denote it $\boldsymbol{P}_j$. We say that the basis function $\boldsymbol{P}_j$ belongs to block $\overline{\Omega}_j$. Each basis function has a support region defined by I_j, which contains the set of cell-numbers where the basis function

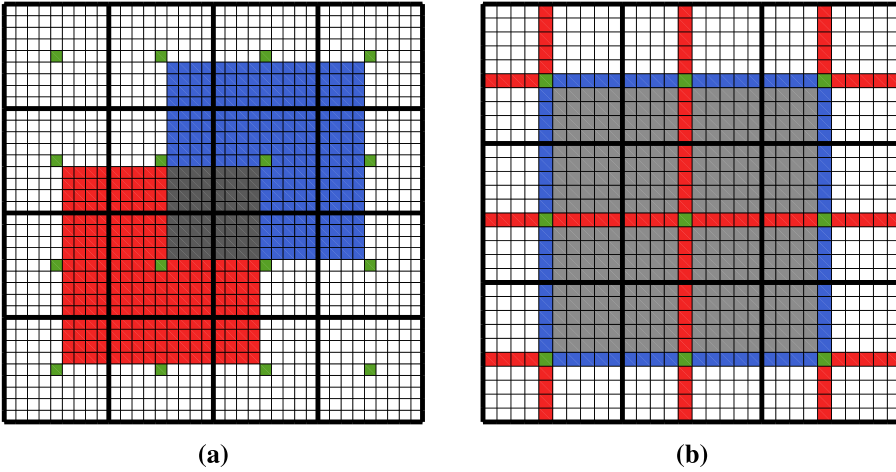


Figure 2.5: ***(a)*** *A grid with two support regions highlighted, and* ***(b)*** *a grid with one support region highlighted together with union boundary cells.*

is allowed to be nonzero. Figure 2.5a shows an example of a Cartesian grid with 16 blocks divided by the bold black lines. Two support regions are colored red and blue, with an overlapping region colored grey.

Let B_j^0 be the set of cells on the boundary of support region j. That is, all cells that are connected to support region j, but are not themselves contained in it,

$$B_j^0 = I_j^{\complement} \cap Q_{I_j}.$$

Here, $I_j^{\complement}$ is the complement of I_j and Q_{I_j} is the set of cells connected to any cell in I_j. Further, let B_j^1 be the set of all cells on the second boundary of support region j. That is, all cells that are connected to B_j^0, but are not themselves contained in neither B_j^0 nor I_j. Define U to be the set of cells that are members of one or more support region boundaries,

$$U = B_1^0 \cup B_2^0 \cup \cdots \cup B_{m-1}^0 \cup B_m^0.$$

We will refer to U as the union boundary. Note that the union boundary will play a very important role in the upcoming discussions regarding the construction of the prolongation operator. We will see that what happens to these cells contribute to complicate this procedure. Figure 2.5b shows an example of a Cartesian grid with 9 blocks, where the middle support region is colored grey, and its support boundary is colored blue. Cells that belong to the boundary of other support regions than the middle one are colored red. Together, the red, blue and green cells belong to U.

And lastly, let χ_j^c denote the center of block j. To determine the support region of this block, we use a local *triangulation* which is created based on the centers of all its neighboring blocks, as well as the centroid of each coarse face which is shared among any two of them. The support region is defined as all cells whose centroids lie within this triangulation. In this thesis, the block centers will in most cases coincide with its centroid. The green cells in Figure 2.5 are the block centers, and will always belong to the union boundary.

2.4 Multiscale Restricted-Smoothed Basis Formulation

We have already introduced the restriction operator $\boldsymbol{R}$ and the prolongation operator $\boldsymbol{P}$. While the prolongation operator is the topic of the next chapter, we now define the restriction operator as a *control-volume summation*,

$$\boldsymbol{R}_{ji} = \begin{cases} 1, & \text{if } \boldsymbol{\chi}_i \in \overline{\Omega}_j, \\ 0, & \text{otherwise.} \end{cases}$$

Applying this operator to quantities on the fine grid will produce a single quantity corresponding to each block, which equals the sum of fine-scale quantities contained within the block.

We will now relate the physical quantities associated with the coarse grid to physical quantities associated with the fine grid by using $\boldsymbol{P}$ and $\boldsymbol{R}$, starting from the fine-scale system of equations derived in Section 2.1,

$$\boldsymbol{Ap} = \boldsymbol{q}. \tag{2.8}$$

Here, $\boldsymbol{p}$ and $\boldsymbol{q}$ denote the pressure and source terms in the fine cells, respectively. Likewise, let $\boldsymbol{p}_c$ and $\boldsymbol{q}_c$ denote the pressure and source terms in the coarse blocks. The prolongation operator can be used to compute a fine-scale pressure approximation $\boldsymbol{p}^{\text{ms}}$ from the coarse-scale pressure solution,

$$\boldsymbol{p}^{\text{ms}} = \boldsymbol{P}\boldsymbol{p}_c. \tag{2.9}$$

Inserting the fine-scale approximation into equation (2.8) and applying the restriction operator on both sides, we get

$$(\boldsymbol{RAP})\boldsymbol{p}_c = \boldsymbol{Rq} \equiv \boldsymbol{q}_c. \tag{2.10}$$

This defines the coarse-scale system of equations, where the coarse system matrix is $\boldsymbol{A}_c = \boldsymbol{RAP}$. An approximate pressure solution on the fine scale can be found

by solving the coarse system and applying the prolongation operator to the coarse pressure solution as in equation (2.9),

$$\boldsymbol{p}^{\text{ms}} = \boldsymbol{P}\boldsymbol{A}_c^{-1}\boldsymbol{q}_c = \boldsymbol{P}(\boldsymbol{R}\boldsymbol{A}\boldsymbol{P})^{-1}\boldsymbol{R}\boldsymbol{q}. \tag{2.11}$$

By construction, the flux induced by the coarse scale pressure solution preserves mass globally on the coarse grid. But when the prolongation operator is applied to obtain the pressure on the fine scale, conservation of mass is no longer ensured on the fine scale from a direct application of Darcy's law. That is, the discrete fluxes obtained from,

$$v_{ij}^{\text{ms}} = -T_{ij}(p_i^{\text{ms}} - p_j^{\text{ms}}), \tag{2.12}$$

are not conservative everywhere on the fine grid. This results from the fact that $\boldsymbol{p}^{\text{ms}}$ is not an exact solution to the system (2.8). To obtain fluxes that are mass conservative everywhere on the fine scale, we can exploit that the fluxes computed by equation (2.12) are conservative over the coarse block interfaces. Define $\overline{\boldsymbol{p}}^{\text{ms}}$ to be a reconstructed pressure solution on the fine scale. Solving the local problems

$$\nabla \cdot \Big(\boldsymbol{K}(\boldsymbol{x})\nabla\overline{p}^{\text{ms}}(\boldsymbol{x})\Big) = -q(\boldsymbol{x}), \quad \boldsymbol{x} \in \overline{\Omega}_j,$$

for each coarse block with the Neumann boundary conditions

$$\nabla\overline{p}^{\text{ms}} \cdot \boldsymbol{n} = -v^{\text{ms}}, \quad \boldsymbol{x} \in \partial\overline{\Omega}_j,$$

allows us to apply Darcy's law to $\overline{\boldsymbol{p}}^{\text{ms}}$ to obtain a velocity field which is mass conservative everywhere on the fine grid. This approach does not provide an exact solution to Darcy's law. However, it does provide a conservative approximation of the flux that can often be used to solve fine-scale transport problems to a sufficient degree of accuracy without the exact pressure solution being known.

To help provide a physical interpretation of the coarse system, we first define the $n \times m$ matrix $\boldsymbol{F}$ as

$$\boldsymbol{F}_{ij} = \sum_{k=1}^{n} T_{ik}(\boldsymbol{P}_{ij} - \boldsymbol{P}_{kj}) = (\boldsymbol{A}\boldsymbol{P})_{ij},$$

and denote $\boldsymbol{F}_j$ column number j of $\boldsymbol{F}$. The physical interpretation of $\boldsymbol{F}_j$ is the net flux out of all fine cells induced by basis function number j. Applying the control volume summation operator to $\boldsymbol{F}$ results in the coarse system,

$$\boldsymbol{R}\boldsymbol{F} = \boldsymbol{R}\boldsymbol{A}\boldsymbol{P} = \boldsymbol{A}_c.$$

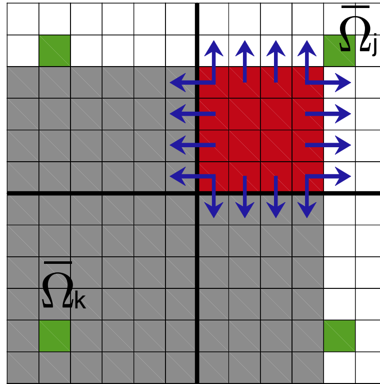


Figure 2.6: *Interpretation of* $(\boldsymbol{A}_c)_{jk}$*. The red cells are the intersection of block* j *and the support region to basis function* k*. The blue arrows illustrate flux out of the area.*

The connection strength $(\boldsymbol{A}_c)_{jk}$ from block j into to block k can thus be expressed as the sum of fluxes induced by basis function k after having removed all fluxes belonging to cells outside of block j,

$$(\boldsymbol{A}_c)_{jk} = \boldsymbol{R}_{j*}\boldsymbol{F}_k = \boldsymbol{R}_{j*}\boldsymbol{A}\boldsymbol{P}_k.$$

Here, $\boldsymbol{R}_{j*}$ is row number j of the restriction operator. In other words, $(\boldsymbol{A}_c)_{jk}$ will equal the flux out of the area contained inside both the support region of basis function k and coarse block j, as illustrated by Figure 2.6.

The fine-scale pressure solution obtained from a single solver of equation (2.10) and (2.9) may or may not satisfy the error tolerance required in a certain application. While the pressure approximation on the coarse scale often achieves high accuracy, errors associated with fine-scale variations will normally still be present. In order to provide a solution with error below some predefined tolerance, an *iterative multiscale formulation* has been presented [14]. Similar to multigrid methods, it exploits the fact that well known iterative techniques rapidly reduce high-frequency errors, even though they converge slowly to the full solution. In combination with an accurate pressure solution on the coarse scale, this iterative scheme efficiently reduces both local and global error and converges to the fine-scale reference pressure solution. Let $\boldsymbol{p}^k$ denote the fine-scale pressure approximation after k iterations, and define the residual

$$\boldsymbol{r}^k = \boldsymbol{q} - \boldsymbol{A}\boldsymbol{p}^k.$$

Let $\boldsymbol{z}^k = S(\boldsymbol{r}^k)$ be the result of applying a smoother to the residual. The smoother is applied to reduce local error, and should be inexpensive to compute, while reducing the local error. An iterative scheme will be stated next, motivated by the

following,

$$\begin{aligned}\boldsymbol{p} &= \boldsymbol{A}^{-1}\boldsymbol{q} \\ &= \boldsymbol{A}^{-1}\boldsymbol{q} + \boldsymbol{p}^k - \boldsymbol{p}^k \\ &= \boldsymbol{p}^k + \boldsymbol{A}^{-1}(\boldsymbol{q} - \boldsymbol{A}\boldsymbol{p}^k) + \boldsymbol{z}^k - \boldsymbol{z}^k \\ &= \boldsymbol{p}^k + \boldsymbol{A}^{-1}(\boldsymbol{r}^k - \boldsymbol{A}\boldsymbol{z}^k) + \boldsymbol{z}^k.\end{aligned}$$

Using that $\boldsymbol{A}^{-1} \approx \boldsymbol{P}\boldsymbol{A}_c^{-1}\boldsymbol{R}$, the iterative scheme is given by

$$\boldsymbol{p}^{k+1} = \boldsymbol{p}^k + \boldsymbol{P}\boldsymbol{A}_c^{-1}\boldsymbol{R}(\boldsymbol{r}^k - \boldsymbol{A}\boldsymbol{z}^k) + \boldsymbol{y}^z.$$

2.5 Constructing the Prolongation Operator

We will now introduce the original algorithm used by the MsRSB method to construct the prolongation operator. As already mentioned, this is done by an iterative process where the basis functions are initialized as the characteristic function on each block,

$$p_{ij}^0 = \begin{cases} 1 & \text{if } i \in C_j, \\ 0 & \text{otherwise,} \end{cases} \qquad i \in F,\ j \in [1, 2, \dots, m].$$

A characteristic function is illustrated in Figure 2.7. The purpose of the iterative procedure is to reduce the residual norms $||\boldsymbol{A}\boldsymbol{P}_j||_1$ as much as possible. If this is achieved, it means that the basis functions have adapted to the local permeability field of the rock, and will incorporate this information when $\boldsymbol{P}$ is applied to the coarse pressure solution $\boldsymbol{p}_c$.

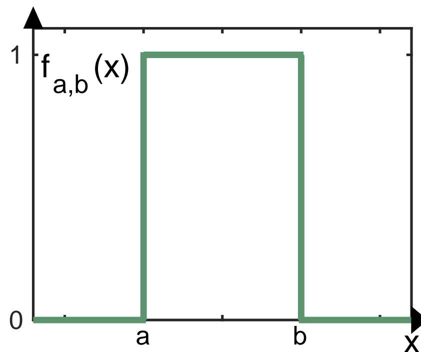


Figure 2.7: *A one-dimensional characteristic function, here denoted $f_{a,b}(x)$. It is zero everywhere except in the interval $[a, b]$, where it equals unity.*

In addition to be as smooth as possible, the basis functions are required to satisfy two conditions. Firstly, they have to stay local. That is, they must be zero outside of their defined support regions,

$$p_{ij} = 0, \quad \forall\, i \in I_j^{\complement},\; j \in [1, \ldots, m]. \tag{2.13}$$

Secondly, we require the prolongation operator to satisfy partition of unity on all cells, also called the normality condition,

$$\sum_{j=1}^{m} p_{ij} = 1, \quad \forall\, i \in F. \tag{2.14}$$

The original algorithm used by the MsRSB method to construct the prolongation operator uses the relaxed Jacobi method to reduce the residuals $||\boldsymbol{AP}_j||_1$. It is stated in Algorithm 1 [38], which takes as input the existing prolongation operator. In the next chapter, Algorithm 1 will be reformulated and discussed in more detail. Here, we will illustrate its application on a one-dimensional permeability field, to give an understanding of what it does.

The permeability is shown in Figure 2.8a. In Figure 2.8b you can see the development of a single basis function from its initial declaration to its converged state, with three intermediate results. Figure 2.8c shows the residual vector $|\boldsymbol{AP}_j|$ in each of the cells after 0, 2 and 4 iterations, as well as for the converged result. You can see that the residuals decrease rapidly, but do not converge to zero. That they do not converge to zero is a results from constraining the basis functions to satisfy partition of unity. If we were not to enforce this constraint, however, all basis functions would merely converge to zero. The simple explanation for this is that it would correspond to finding the pressure solution with no source-terms, and with Dirichlet boundary conditions enforcing zero pressure on the boundary given by equation (2.13).

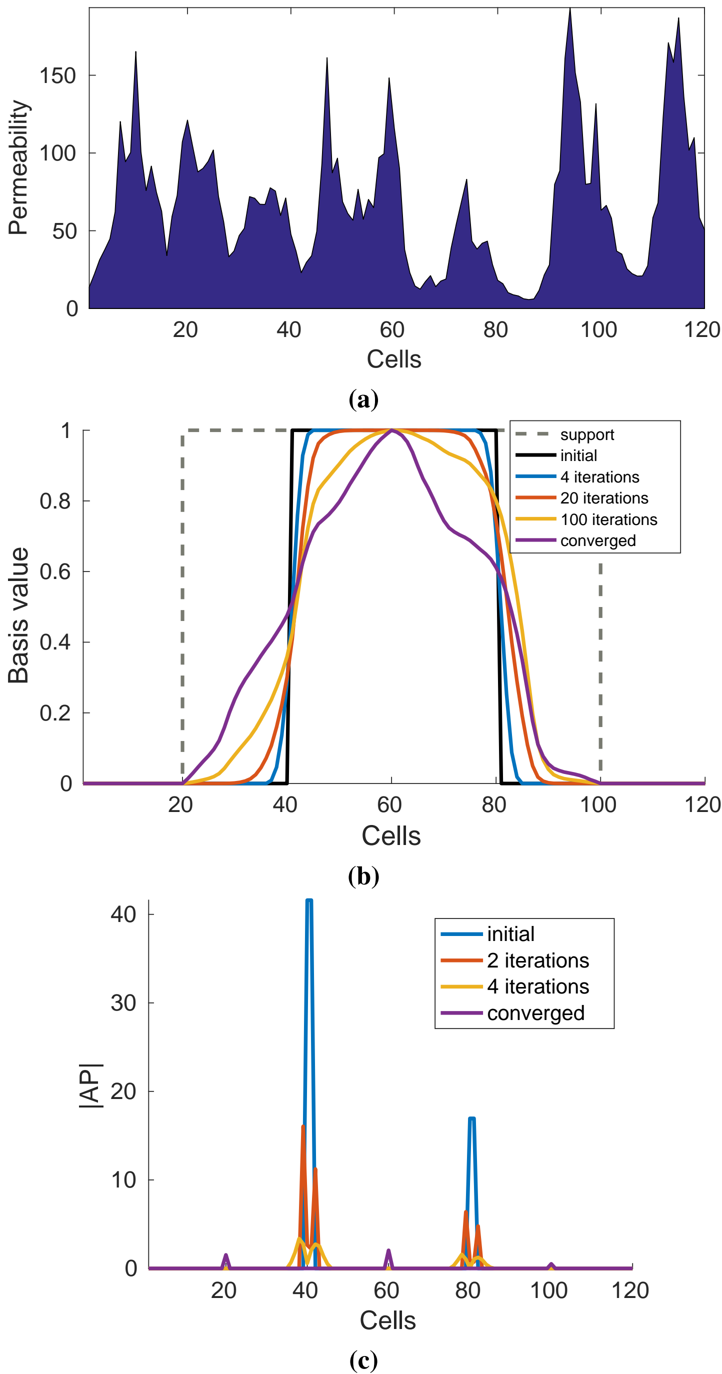


Figure 2.8: *(**a**) One-dimensional permeability field over 120 file cells, (**b**) development of a single basis function, and (**c**) resulting residuals* $|\boldsymbol{AP}_j|$ *in each cell.*

Algorithm 1. Constructing the prolongation operator

1. Compute the non-modified increments,

$$\hat{h}_{ij} = -\frac{\omega}{a_{ii}} \sum_{l=1}^{n} a_{il} p_{lj}^{k}, \quad i \in F,\ j \in [1, 2, \ldots, m].$$

2. Modify the increments to avoid growth outside of the support regions,

$$\overline{h}_{ij} = \begin{cases} \hat{h}_{ij} & \text{if } i \in I_j, \\ 0 & \text{if } i \notin I_j, \end{cases} \quad i \in F,\ j \in [1, 2, \ldots, m].$$

3. For each cell on the union boundary, compute the sum of all increments belonging to it,

$$u_i = \sum_{l=1}^{m} \overline{h}_{il}, \quad i \in U.$$

4. Modify the increments to preserve partition of unity,

$$h_{ij} = \begin{cases} \frac{\overline{h}_{ij} - u_i p_{ij}^{k}}{1 + u_i}, & \text{if } i \in I_j, i \in U, \\ \overline{h}_{ij} & \text{Otherwise}, \end{cases} \quad i \in F,\ j \in [1, 2, \ldots, m].$$

5. Update the basis functions,

$$p_{ij}^{k+1} = p_{ij}^{k} + h_{ij}, \quad i \in F,\ j \in [1, 2, \ldots, m].$$

6. Define the error as

$$e = \max_{ij} \left(|h_{ij}| \right), \quad i \in F \cap U^{\complement},\ j \in [1, 2, \ldots, m].$$

If $e <$tol, return $\boldsymbol{P} = \boldsymbol{P}^{k+1}$, otherwise go to Line 1.

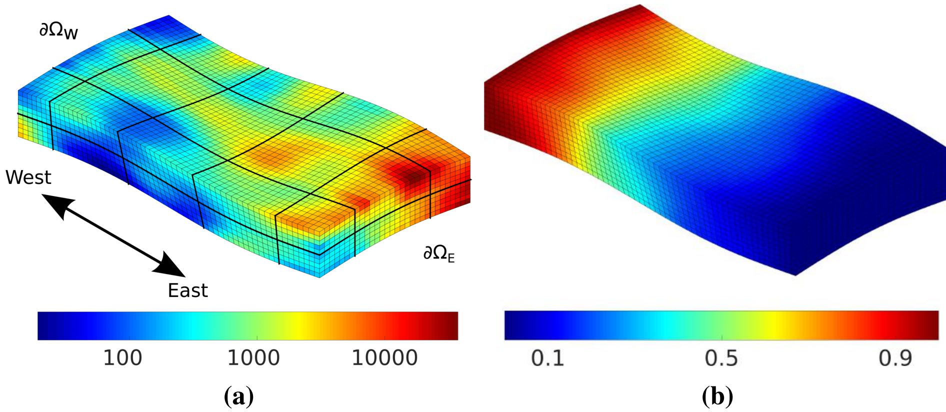


Figure 2.9: *(a) Permeability field and (b) fine-scale pressure solution of the model-problem.*

2.6 Multiscale Method Application

We will now illustrate the MsRSB method by applying it to a model-problem. Here we use a stratigraphic grid with curved boundaries, consisting of $60 \times 30 \times 10$ cells. The grid is partitioned into $5 \times 3 \times 2$ blocks, giving an upscaling factor of 600. The permeability field is the realization of a lognormal isotropic distribution, with the average permeability equal to 1000mD. The field has been made fairly heterogeneous, with permeability varying with three orders of magnitude. Dirichlet boundary conditions are assigned on the west and east faces of the rock, with a constant pressure of 1 on the west side, and 0 on the east side. This will drive the flow from west to east. The remaining outer faces are assigned to have no-flow boundaries,

$$p(\boldsymbol{x}) = \begin{cases} 1, & \text{if } \boldsymbol{x} \in \partial\Omega_{\mathrm{W}}, \\ 0, & \text{if } \boldsymbol{x} \in \partial\Omega_{\mathrm{E}}. \end{cases}$$
$$\boldsymbol{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) = 0, \quad \text{if } \boldsymbol{x} \in \partial\Omega \cap (\partial\Omega_{\mathrm{W}} \cup \partial\Omega_{\mathrm{E}})^{\complement}.$$

Here, $\boldsymbol{n}(\boldsymbol{x})$ is a vector normal to the boundary $\partial\Omega$, and $\partial\Omega_{\mathrm{W}}$ and $\partial\Omega_{\mathrm{E}}$ are the west and east parts of $\partial\Omega$, respectively. The permeability field is visualized in Figure 2.9a, where the coarse grid is highlighted by the black lines.

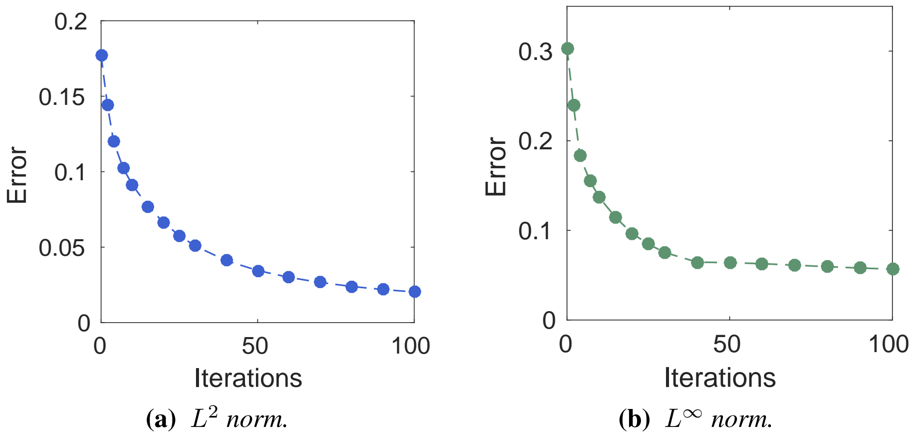
For this flow problem, we compute a fine-scale pressure approximation using the MsRSB method. Note that we do not use the iterative multiscale formulation, and the pressure approximation is found by simply solving the coarse system (2.10) and applying the prolongation operator to the result as in equation (2.11). We will measure the accuracy of $\boldsymbol{p}^{\mathrm{ms}}$ by comparison with the fine scale reference

solution, denoted $\boldsymbol{p}^{\text{fs}}$, which you can see visualized in Figure 2.9b. To compare the two, we use the scaled L^2 and L^∞ norms,

$$||\boldsymbol{p}^{\text{fs}} - \boldsymbol{p}^{\text{ms}}||_2 = \sqrt{\frac{\sum_{i\in F} |\boldsymbol{p}_i^{\text{fs}} - \boldsymbol{p}_i^{\text{ms}}|^2}{\sum_{i\in F} |\boldsymbol{p}_i^{\text{fs}}|^2}}, \qquad ||\boldsymbol{p}^{\text{fs}} - \boldsymbol{p}^{\text{ms}}||_\infty = \frac{\max_{i\in F} |\boldsymbol{p}_i^{\text{fs}} - \boldsymbol{p}_i^{\text{ms}}|}{\max_{i\in F} |\boldsymbol{p}_i^{\text{fs}}|}.$$

In the context of this thesis it is of particular interest to observe how the errors vary as a function of smoothing iterations applied to the prolongation operator. We therefore compute the errors when varying the number of iterations from 0 to 100. In Figure 2.10 you see the result.

When zero smoothing iterations are applied, the resulting multiscale pressure approximation will simply be equal to the coarse scale pressure solution. In our example, this results in an error relative to the fine scale solution of 0.18 and 0.30 in the L^2 and L^∞ norms, respectively. But when applying more and more smoothing steps to the prolongation operator, we gradually and systematically incorporate the fine scale variations into the multiscale approximation, resulting in a decreasing error. After having applied 100 smoothing iterations, the error has decreased to 0.020 and 0.057 in the L^2 and L^∞ norms, respectively. As you can see from Figure 2.10, there is not much to gain by applying more than 100 smoothing steps. If one requires a better pressure approximation, this can be achieved by using the iterative multiscale formulation or by reducing the upscaling factor.



(a) *L^2 norm.* **(b)** *L^∞ norm.*

Figure 2.10: *Error measures of the multiscale pressure solution as a function of smoothing iterations applied to the prolongation operator.*

Chapter 3

Improving the Construction of the Prolongation Operator

In this chapter we present two primary contributions to improve the original algorithm that constructs the prolongation operator used in the MsRSB method. The first contribution is a modification to the original algorithm that was presented in the previous chapter. We will see that the suggested modified algorithm leads to a more numerically stable implementation. The second contribution is two new smoothing approaches, both of which are modified versions of the Gauss-Seidel method.

The chapter is organized as follows. Section 3.1 presents the modified version of the construction algorithm which was initially stated in Section 2.5. Section 3.2 continues by analyzing the algorithm, in particular by discussing how the Jacobi smoothing step works. In Section 3.3, we consider replacing the Jacobi smoother by a straightforward Gauss-Seidel smoother, and argue that it will lead to a more expensive implementation. As an important step towards modifying the Gauss-Seidel smoother to become more efficient, Section 3.4 consider applying the red-black ordered Gauss-Seidel method as a smoother. In Section 3.5 and 3.6 we present the two new smoothing procedures. These can be applied directly into the algorithm presented in Section 3.1 by replacing the Jacobi smoother. The chapter concludes by presenting numerical results in Section 3.7. Here, the algorithm presented in Section 3.1 is first compared to its original formulation, followed by comparisons of the newly presented smoothers and the relaxed Jacobi method.

3.1 A Change to the Original Construction Algorithm

Before we discuss the construction of the prolongation operator in more detail, we will introduce a modified version of the original algorithm that was presented in the previous chapter. Assuming that requirements (2.13) and (2.14) hold, the new version will produces an equivalent updated prolongation operator. The main difference between the two versions is when the increments are added to update basis values. The new algorithm is stated in Algorithm 2.

Keep in mind that a straightforward implementation of Algorithm 2 would be inefficient because a large amount of redundant computations is carried out. In particular, the values belonging to cells outside of their respective support region are never required to be computed since they have the constant value zero. In Chapter 4.3 the algorithm is restated in a form more similar to the implementation. However, to understand how the algorithm works it is instructive to consider it on its current form.

We end this section by proving that Algorithms 1 and 2 produce the same updates to the basis functions, provided that requirements (2.13) and (2.14) holds after the previous step.

Proof. It is easy to see that the two algorithms produce equivalent updates to values that do not belong to the union boundary. We will show that this is also the case for the cells on the union boundary. Notice that

$$\overline{g}_{ij} = p_{ij}^k + \overline{h}_{ij},$$

and

$$s_i = \sum_{l=1}^{m} \left(p_{il}^k + \overline{h}_{il}\right) = 1 + u_i,$$

where it is used that

$$\sum_{l=1}^{m} p_{il}^k = 1.$$

Assume that we update the value of cell i belonging to basis function j, where the cell also belongs to the union boundary. By the first algorithm we get that its updated value is

$$p_{ij}^{k+1} = p_{ij}^k + h_{ij} = p_{ij}^k + \frac{\overline{h}_{ij} - u_i p_{ij}^k}{1 + u_i} = \frac{p_{ij}^k + \overline{h}_{ij}}{1 + u_i} = \frac{\overline{g}_{ij}}{s_i} = g_{ij}$$

which is the same expression as provided by the second algorithm. □

Algorithm 2. Constructing the prolongation operator - Second formulation

1. Compute the non-modified updated values,

$$\hat{g}_{ij} = p_{ij}^k - \frac{\omega}{a_{ii}} \sum_{l=1}^{n} a_{il} p_{lj}^k, \quad i \in F,\ j \in [1, 2, \ldots, m].$$

2. Modify the updated values to avoid growth outside of the support region,

$$\overline{g}_{ij} = \begin{cases} \hat{g}_{ij} & \text{if } i \in I_j, \\ 0 & \text{if } i \notin I_j, \end{cases} \quad i \in F,\ j \in [1, 2, \ldots, m].$$

3. For each cell on the union boundary, compute the sum of all values belonging to the cell,

$$s_i = \sum_{l=1}^{m} \overline{g}_{il}, \quad i \in U.$$

4. Modify the updated values to preserve partition of unity,

$$g_{ij} = \begin{cases} \frac{\overline{g}_{ij}}{s_i}, & \text{if } i \in I_j, i \in U, \\ \overline{g}_{ij} & \text{Otherwise}, \end{cases} \quad i \in F,\ j \in [1, 2, \ldots, m].$$

5. Update the basis functions,

$$p_{ij}^{k+1} = g_{ij}, \quad i \in F,\ j \in [1, 2, \ldots, m].$$

6. Define the error as

$$e = \max_{ij} \left(|p_{ij}^{k+1} - p^k| \right), \quad i \in F \cap U^{\complement},\ j \in [1, 2, \ldots, m].$$

If $e <$tol, return $\boldsymbol{P} = \boldsymbol{P}^{k+1}$, otherwise go to Line 1.

3.2 Analyzing the Construction Algorithm

The content of a single step of Algorithm 2 (and 1) can be summarized as follows. Apply the smoother to each basis function to reduce the residuals $||\boldsymbol{A}\boldsymbol{P}_j||_1$.

Then modify the result to ensure that the two requirements (2.13) and (2.14) hold. To show that the requirements hold, we will state two properties of the Jacobi smoother performed in Line 1.

Jacobi smoother property 1. *The Jacobi smoother preserves partition of unity.*

That is, if we update the basis functions immediately after Line 1 of the above algorithm has been executed, partition of unity is preserved.

Proof. We have that

$$\sum_{j=1}^{m} p_{ij}^{k+1} = \sum_{j=1}^{m} \left(p_{ij}^{k} - \frac{\omega}{a_{ii}} \sum_{l=1}^{n} a_{il} p_{lj}^{k} \right) = 1 - \frac{\omega}{a_{ii}} \sum_{j=1}^{m} \sum_{l=1}^{n} a_{il} p_{lj}^{k}$$
$$= 1 - \frac{\omega}{a_{ii}} \sum_{l=1}^{n} a_{il} \sum_{j=1}^{m} p_{lj}^{k} = 1 - \frac{\omega}{a_{ii}} \sum_{l=1}^{n} a_{il} = 1,$$

where equations (2.5) and (2.14) were used. □

However, the Jacobi smoother does not prevent the basis functions to grow outside of their support regions. We will now state a second property of the Jacobi smoother.

Jacobi smoother property 2. *For each basis function, the Jacobi smoother preserves zero value on all cells outside of its support region, except on its support boundary. That is, if requirement (2.13) holds at step k, and we apply the Jacobi smoother, we have that*

$$p_{ij}^{k+1} = 0, \quad \forall\, i \in \left(I_j \cup B_j^0\right)^{\complement},\ j \in [1, \ldots, m]. \tag{3.1}$$

Proof. Assume that cell i does not belong to neither the support region nor the support boundary of a particular basis function j,

$$i \in \left(I_j \cup B_j^0\right)^{\complement}.$$

and consider applying the Jacobi smoother to p_{ij}^{k}. We get

$$p_{ij}^{k+1} = p_{ij}^{k} - \frac{\omega}{a_{ii}} \sum_{l=1}^{n} a_{il} p_{lj}^{k} = -\frac{\omega}{a_{ii}} \sum_{l=Q_i} a_{il} p_{lj}^{k},$$

Here, property (2.6) and requirement (2.13) were used. Further, since cell i is not on the boundary of support region j, we have that

$$Q_i \subset I_j^{\complement}. \tag{3.2}$$

From (2.13) we therefore have that

$$p_{lj}^k = 0, \quad \forall\, l \in Q_i,$$

and $p_{ij}^{k+1} = 0$ follows. □

For a particular basis function, the Jacobi smoother is not guaranteed to preserve zero value on cells belonging to its support boundary, which breaks requirement (2.13). This is because equation (3.2) is not true on these cells. Line 2 of the algorithm is therefore needed to enforce that this requirement is preserved. Because of the Jacobi smoother property 2 however, this will only affect values on the union boundary. But when setting the values of cells on the boundaries to be zero, it breaks the requirement of partition of unity on the same cells. Line 3 and 4 are therefore required to explicitly enforce partition of unity on the union boundary. After Lines $1-4$ have been carried out, requirements (2.13) and (2.14) are again preserved on the entire grid.

We will now illustrate the algorithm by applying it to a simple two-dimensional model and look at how a single basis function develops. The grid is Cartesian, consists of 27×27 cells, and is divided into 9 equally large blocks. It is shown in Figure 3.1a, where the union boundary cells are outlined in red, and the cell-centers of the blocks are colored green. We take a look at how the basis function belonging to the middle block develops. It is initialized as the characteristic function, as shown in Figure 3.1b. Here, the inner black line is the boarder of the block, and the outer black line is the boarder of the support region to the basis function. How the basis function develops during the first six steps is shown in Figures 3.1c-3.1h. Observe that it grows towards the support boundary, one cell-width at each step. After four steps it has reached the support boundary, and in the fifth iteration it would continue to grow outside of it, was it not for that the cells outside of the support region are set to zero. Figure 3.1i shows the converged basis function.

That the basis function grows one cell-width at each step has to do with the nature of the Jacobi update. Considering a cell of zero value, the only way it can turn non-zero is if it is connected to a cell that was nonzero in the previous step. This is exactly the same feature that makes the Jacobi smoother property 2 hold.

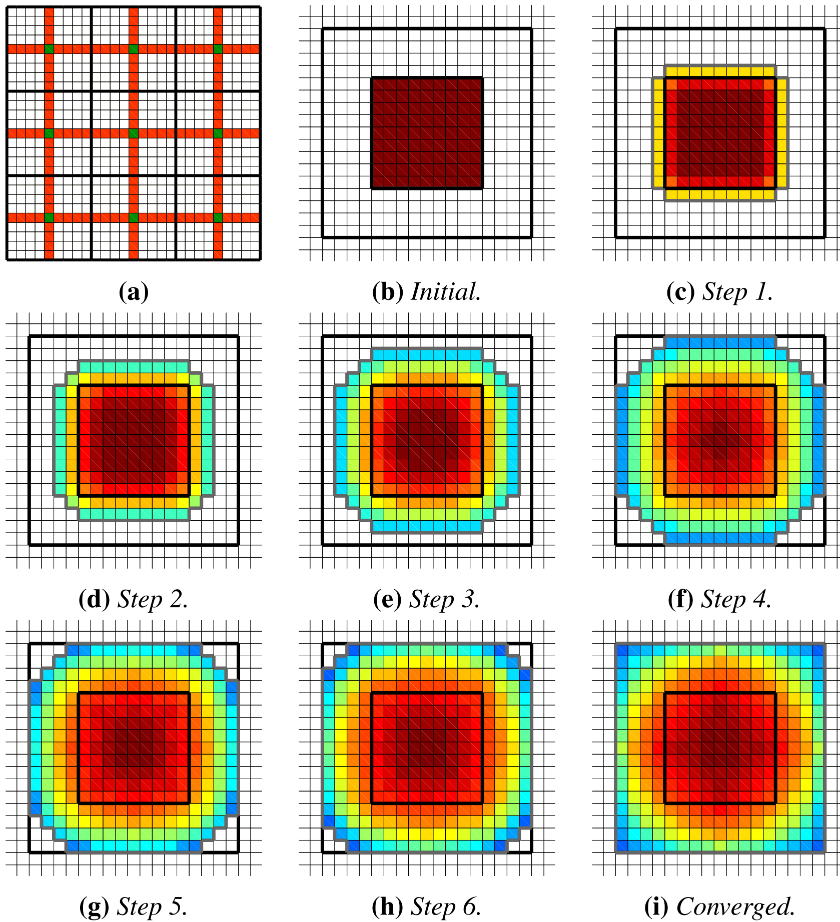


Figure 3.1: ***(a)*** *Example-grid with union boundary and block centers highlighted, and* ***(b-i)*** *development of a basis function for a homogeneous permeability field. The grey line highlights the frontier of the basis function.*

3.3 Gauss-Seidel as Smoother

Wishing to decrease the computational cost needed to create the prolongation operator, it is natural to consider changing Algorithm 2 to use a Gauss-Seidel smoother instead of Jacobi. By applying this change, the hope is that the residuals $||\boldsymbol{AP}_j||_1$ are reduced faster, so that fewer steps are needed to reach a certain quality of the prolongation operator. Assume that this change is carried out, so that Line 1 of Algorithm 2 reads

1. Compute the non-modified updated values,

$$\hat{g}_{ij} = p_{ij}^k - \frac{\omega}{a_{ii}}\left(\sum_{l=1}^{i-1} a_{il}\hat{g}_{lj} + \sum_{l=i}^{n} a_{il}p_{lj}^k\right), \quad i \in F,\ j \in [1, 2, \ldots, m]. \tag{3.3}$$

Here we have stated the slightly more general SOR method. Choosing $\omega = 1$ would result in the forward Gauss-Seidel method. It is straightforward to prove that partition of unity is preserved after Line 1 has been applied, as it is for Jacobi. However, when looking into the consequences of the change a bit further, we will realize that it has a prominent disadvantage over Jacobi.

The disadvantage results from that Jacobi smoother property 2 does not hold in general for the GS smoother. Therefore, if the above change is applied to the original algorithm with no other modification, partition of unity would not be guaranteed to be preserved. To correct for this one would have to force partition of unity on more cells than the ones on the union boundary.

To illustrate what happens, consider the same test-case as we did in the previous section. Figure 3.2a shows the same grid where the support regions of the bottom left and top right basis functions are highlighted in grey, and their boundary cells are colored red. As we discussed in Chapter 2, the GS method depends on the order in which the elements are updated. In this example we update the cells from left to right, top to bottom. That is, the first cell to be updated is the one in the south-west corner and the last one is in the north-east corner of the grid. This is the ordering that appears naturally from applying forward GS to the way the cells are numbered by the program that was used. We take a look at the result of applying a sweep of Equation (3.3) to the initial basis functions, in particular the bottom left and the upper right ones. The results are shown in Figure 3.2b and 3.2c, respectively. Here, we see that the bottom left basis function has grown to be nonzero on the entire grid. When preserving locality we must therefore adjust every cell outside of its support region to be zero, which removes the partition of unity requirement on the same cells. This leads to Lines 4 and 5 of Algorithm 2 having to be applied to all cells outside of the support region of the same basis function. The upper right basis function, however, has only grown one cell-width, just like for the Jacobi smoother.

Why this happens results from the fact that the GS smoother uses the most recently updated values. Consider a situation where the value of a cell belonging to B_j^1 is updated after the value of a connected cell on B_j^0. Now the cell on B_j^1 is no longer ensured to remain zero, since its computation is based on a connected cell which can be nonzero. This effect can again propagate further away from the

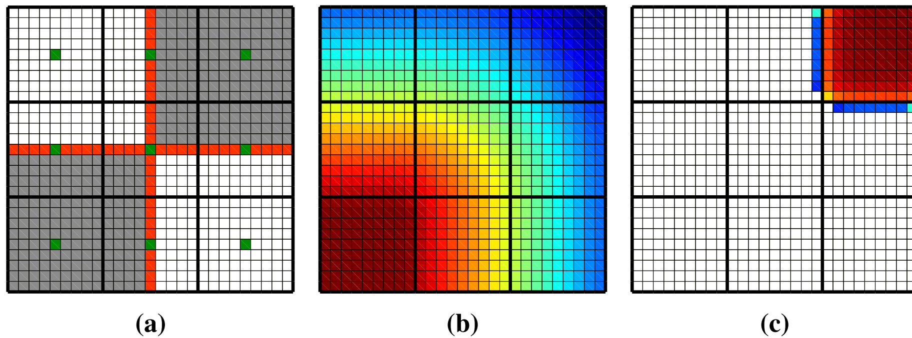


Figure 3.2: *Example-grid with **(a)** support regions and boundaries for the bottom-left and top-right basis functions highlighted, and **(b,c)** resulting basis functions after one sweep of Gauss-Seidel.*

support region, and will in some cases, such as illustrated in Figure 3.2b, propagate throughout the entire grid. The asymmetric development of the two basis functions is a result of the order in which we apply the updates. If we had updated the cells from right to left, top to bottom, we would see the developments of the two basis functions switched.

The disadvantage of a direct change to Gauss-Seidel should now be clear. It results in a higher computational cost in order to preserve partition of unity. To minimize or remove this disadvantage, we will consider applying the Gauss-Seidel smoother in a different order, which is the topic of the next three sections.

We end this section by mentioning another disadvantage of the Gauss-Seidel smoother which was discovered during the numerical investigation. When using the same ordering as above, it was discovered that even when the extra computational cost was used to ensure partition of unity, the basis functions did not converge to the same solutions as the original algorithm. In fact, it did not reduce $||\boldsymbol{AP}_j||_1$ as much as Jacobi. We will not discuss this further, but it is believed to be an effect of the asymmetrical development of the basis functions illustrated by Figure 3.2b and 3.2c.

3.4 Red-black Gauss-Seidel Smoother

We will now consider the application of a red-black Gauss-Seidel sweep as smoother, and see that this approach reduces the problem discussed in the previous section. But it still does not meet the Jacobi smoother property 2 entirely.

We look at the same test problem as earlier, although now with 15×15 cells.

Figure 3.3a shows the grid in red-black coloring, where the support region of the middle block is highlighted by the white line. Figure 3.3b shows B_j^0 and B_j^1 colored in dark and light green, respectively, and the support region is colored grey.

Consider applying the GS smoother to the basis function belonging to the highlighted support region, ordered so that all black cells are updated before all red cells. After the black cells have been updated, zero value is no longer ensured on half the cells on B_j^0. These are the black cells outside of the support region in Figure 3.3c. When continuing by updating the red cells, zero value will no longer be ensured on the remaining half of the cells on B_j^0. In addition, zero value is no longer ensured on cells belonging to B_j^1 that are connected to a black cell on B_j^0. These are the red cells belonging to B_j^1 in Figure 3.3c. However, zero value is ensured on all remaining cells that do not belong to the support region.

The result is that when applying this smoother, partition of unity must be enforced on a subset of U_1 in addition to the whole of U, where U_1 is defined as the union of cells on the second boundary,

$$U_1 = B_1^1 \cup B_2^1 \cup \cdots \cup B_m^1.$$

While this is a substantial improvement of the ordering considered in the previous section, we are still able to do better.

Consider if Lines $2-5$ of Algorithm 2 are applied after the black cells have been updated, and yet again after the red cells have been updated. By doing this, zero value on B_j^1 would always be preserved, and the cells on U are the only ones that require modification to preserve partition of unity. Also, the number of operations required to preserve partition of unity would not increase, since Lines 3

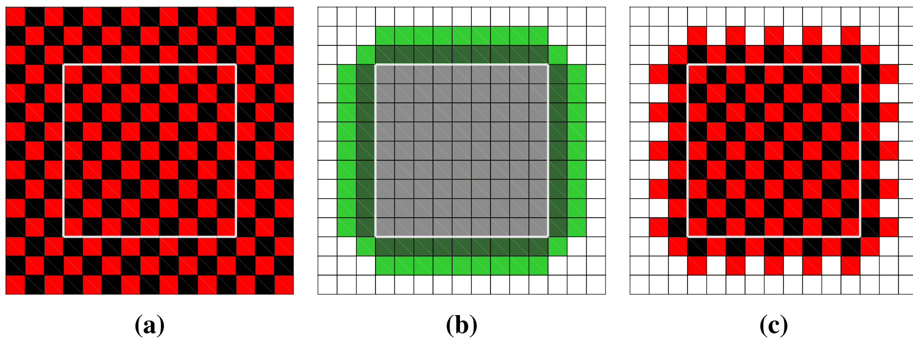


Figure 3.3: *Example-grid **(a)** in red-black coloring, **(b)** with a support region and support boundaries B_j^0 and B_j^1 highlighted, and **(c)** nonzero updated values colored red and black.*

and 4 of the original algorithm would only have to be applied to half the cells on U in each normalization operation. Still, when it comes to computational cost it can be a disadvantage to apply a partition of unity operation twice in every iteration, rather than once as for the original algorithm. In the next section we therefore introduce another approach.

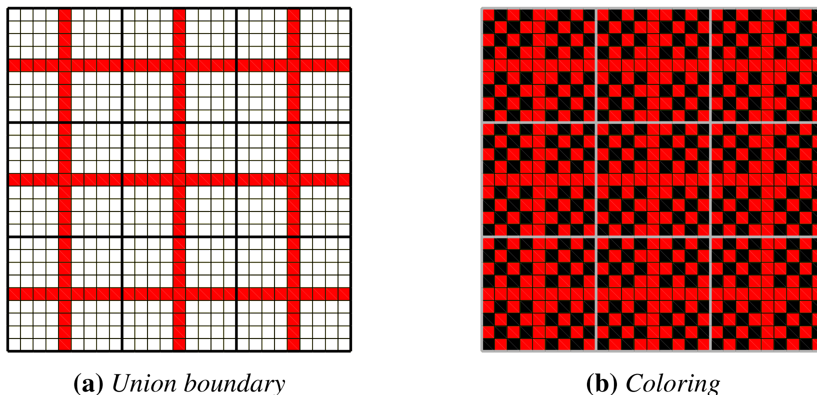
3.5 Partially Red-black Gauss-Seidel Algorithm

We will now present an alternative procedure which is a compromise between the Jacobi smoother and the red-black Gauss-Seidel smoother. We name the new approach the *partially red-black Gauss-Seldel* algorithm. Like the original algorithm, this method requires partition of unity to be enforced only on U, and during a single operation in each step. Like the red-black ordered Gauss-Seidel smoother, it uses coloring of the cell, which will be explained next.

Start by coloring all cells on U red. Then iterate through the remaining cells and color them in the following way. If the cell is not connected to any black cells, color it black. Otherwise, color it red. The result of applying this coloring to our previous example is shown in Figure 3.4b. Figure 3.4a shows the corresponding union boundary.

Let R and K be the set of all red and black cells, respectively. In the same way as for the red-black ordered Gauss-Seidel method, all black cells will be updated before all red cells. Each cell-value will be incremented using relaxed Jacobi update. The partially red-black Gauss-Seldel is stated in Algorithm 3.

After Line 1 in Algorithm 3 has been executed, requirement (2.13) and (2.14)



(a) *Union boundary* **(b)** *Coloring*

Figure 3.4: *Example-grid with partially red-black coloring.*

Algorithm 3. Partially red-black Gauss-Seidel

1. Compute the non-modified updated values to all black cells,

$$\hat{g}_{ij}^{1/2} = \begin{cases} p_{ij}^k - \frac{\omega}{a_{ii}} \sum_{l=1}^n a_{il} p_{lj}^k & \text{if } i \in K, \\ p_{ij}^k & \text{if } i \in R, \end{cases} \quad i \in F,\ j \in [1, 2, \ldots, m].$$

2. Compute the non-modified updated values to all red cells,

$$\hat{g}_{ij} = \begin{cases} \hat{g}_{ij}^{1/2} & \text{if } i \in K, \\ \hat{g}_{ij}^{1/2} - \frac{\omega}{a_{ii}} \sum_{l=1}^n a_{il} \hat{g}_{lj}^{1/2} & \text{if } i \in R, \end{cases} \quad i \in F,\ j \in [1, 2, \ldots, m].$$

3. Continue by performing Lines $2 - 6$ as written in Algorithm 2.

are preserved. This follows from the fact that the Jacobi updates preserves partition of unity, and that no cells on the union boundary have been updated yet. After Line 2 has been executed, requirement (2.14) is yet again preserved. But in the same way as the original procedure, elements on the support boundary of each basis function are no longer ensured to be zero. Lines 2 – 6 in Algorithm 2 are therefore still needed to ensure that this requirement holds.

Note that the partially red-black ordered GS algorithm applies exactly the same arithmetic operations as the original. The only differences are the order in which the values are updated and, in the case of Line 2, what latest updates are used. Regarding memory usage, the new algorithm suffers no disadvantage from the original one. Rather, a slight amount of memory can be saved in the buffer where updated elements are stored. The algorithm requires an additional setup computation to find the coloring of the grid. This operation is straightforward and can be computed by a single sweep through the grid, where it colors the cells by the rules stated above. The coloring procedure can be applied to any type of grid since it only uses topology information.

3.6 Boundary Last Gauss-Seidel Algorithm

A simpler approach than the one presented above was found, which also fixes the problem of the straightforward Gauss-Seidel smoother. The approach is explained by the following. First apply the GS update to all cells that do not belong to U. Then apply the Jacobi update to the cells that do belong to U. We call the new

approach the *boundary last Gauss-Seidel* algorithm. It is stated in Algorithm 4.

Algorithm 4. Boundary last Gauss-Seidel

1. Compute the non-modified updated values by applying the Gauss-Seidel smoother to all cells that do not belong to U,

$$\hat{g}_{ij}^{1/2} = \begin{cases} p_{ij}^k - \frac{\omega}{a_{ii}}\left(\sum_{l=1}^{i-1} a_{il}\hat{g}_{lj}^{1/2} + \sum_{l=i}^{n} a_{il}p_{lj}^k \right), & \text{if } i \in U^{\complement}, \\ p_{ij}^k & \text{if } i \in U, \end{cases}$$
$$i \in F,\ j \in [1, 2, \ldots, m].$$

2. Compute the non-modified updated values by applying the Jacobi smoother to all cells that do belong to U,

$$\hat{g}_{ij} = \begin{cases} \hat{g}_{ij}^{1/2} & \text{if } i \notin U, \\ \hat{g}_{ij}^{1/2} - \frac{\omega}{a_{ii}} \sum_{l=1}^{n} a_{il}\hat{g}_{lj}^{1/2} & \text{if } i \in U, \end{cases} \quad i \in F,\ j \in [1, 2, \ldots, m].$$

3. Continue by performing Lines $2 - 6$ as written in Algorithm 2.

The idea behind this algorithm is the same as with the partially red-black algorithm. After Line 1 has been executed, requirement (2.13) and (2.14) are preserved. After Line 2 has been executed, Lines 2-6 of Algorithm 2 must be carried out to again preserve the two conditions. Note that the GS smoother applied in Line 1 can be executed in any order. The only restriction is that it must not update any cells on U.

Algorithms 3 and 4 both have their own advantage. The advantage of Algorithm 4 is that it allows the use of a larger fraction of recently updated values in the computation of consecutive updates. However, a disadvantage is that Line 1 must be computed sequentially, making it less suitable for parallelism. Algorithm 3 does not suffer from this drawback, where both Lines 1 and 2 can be computed in parallel. In addition, Algorithm 3 produces a more symmetric development of the basis functions, which possibly makes it more numerically stable.

Before presenting numerical results, we briefly comment on another approach that was investigated. It was named the *varying ordered Gauss-Seidel* algorithm, and explained briefly, it is based on applying the Gauss-Seidel method to the entire grid, but let each basis function update its values in its own order. The ordering is chosen to prevent the basis functions to grow past U. However, for partition of

unity to be preserved it is required in general that the Gauss-Seidel method must be applied in the same order to all basis functions. Hence, the approach would require partition of unity to be enforced on the entire grid. For this reason, it was discarded.

3.7 Numerical Results

We end the chapter by presenting numerical results of the new algorithms we have presented. Section 3.7.1 presents numerical evidence showing that the algorithm that constructs the prolongation operator presented in Section 3.1 leads to a more numerically stable implementation than the original formulation. In Section 3.7.2 we compare the different smoothing procedures, and show that the procedures presented in this chapter offers an improvement over the original relaxed Jacobi smoother.

3.7.1 Comparison of Construction Algorithms

We will now compare the original algorithm used to construct the prolongation operator [38] stated in Section 2.5 to the new formulation presented in Section 3.1, refereed to by Algorithms 1 and 2, respectively. Although the differences between the two might seem insignificant, numerical experiments suggest that an implementation based on Algorithm 2 is less prone to round-off errors introduced by floating point operations.

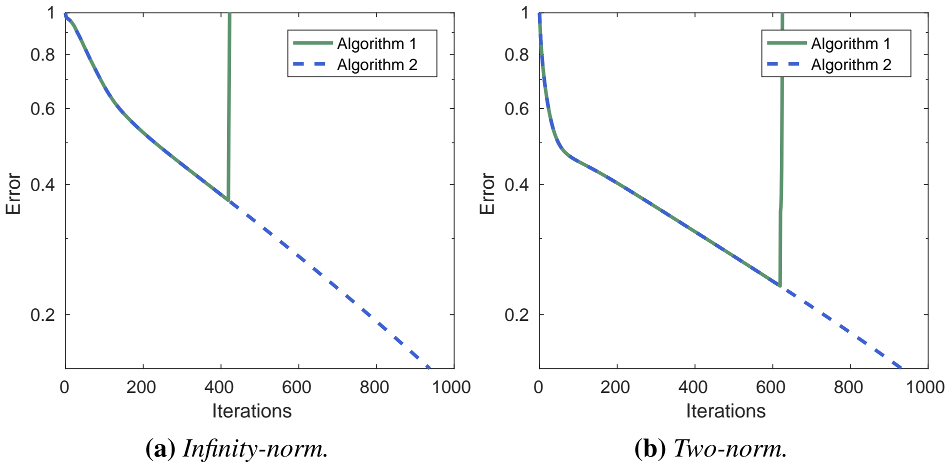
To investigate their numerical stability, we consider three different programs that construct the prolongation operator. The first is implemented in Matlab as part of the MRST toolbox, which explicitly enforces partition of unity on the entire grid in every step. It makes no attempt to optimize efficiency, and is mainly used to explore the method. The second one is implemented in C++ and is based on Algorithm 1. It was created by Olav Møyner during the fall of 2015. Because of the introduction of round-off errors, this program performs a partition of unity operation of the entire grid regularly. If this operation is not performed frequently enough, the prolongation operator diverges. In the tests presented here, this operation is performed at every 100th step, and comes in addition to the computations contained in the algorithm as stated in Section 2.5. The third program is originally presented in this thesis, and will be outlined in Chapter 4. It is implemented in C++ and is based on Algorithm 2. Unlike the second program, it does not perform any additional operation to preserve partition of unity than what is stated in the algorithm.

To keep it brief, we show results only for a single test-case, but note that the same main result was confirmed in several other numerical tests. The test case is Layer 35 of the SPE 10 dataset, consisting of 60×220 cells and divided into 6×11 blocks, giving an upscaling factor of 200. The extra normalization performed by the second program is initially turned off. After each step in the construction, the distance from the converged prolongation operator is computed. This distance is measured in the following norms,

$$||\boldsymbol{P}^k - \boldsymbol{P}^*||_\infty = \frac{\max_{ij} |\boldsymbol{P}^k_{ij} - \boldsymbol{P}^*_{ij}|}{\max_{ij} |\boldsymbol{P}^0_{ij} - \boldsymbol{P}^*_{ij}|}, \quad ||\boldsymbol{P}^k - \boldsymbol{P}^*||_2 = \sqrt{\frac{\sum_{j=1}^m \sum_{i=1}^n |\boldsymbol{P}^k_{ij} - \boldsymbol{P}^*_{ij}|^2}{\sum_{j=1}^m \sum_{i=1}^n |\boldsymbol{P}^0_{ij} - \boldsymbol{P}^*_{ij}|^2}}.$$

Here, k is the number of steps and $\boldsymbol{P}^*$ is the approximate converged prolongation operator computed numerically. Figure 3.5 shows the distance as a function of the number of steps for the two programs. Observe that the two algorithms steadily approach convergence in the beginning of the iteration. After around 400 and 600 iterations, however, the distance of Algorithm 1 rapidly blows up and diverges. Meanwhile, Algorithm 2 continues to approach convergence steadily.

Next, we will show that the round-off errors introduced by Algorithm 1 are present even when the extra normalization procedure is performed every 100th step. To show this, we look at the discrepancy between the prolongation operator as computed by the Matlab program and the two C++ programs after 100, 1,000, 10,000 and 20,000 steps. Since the Matlab program explicitly normalizes the entire grid in every step, it will not suffer from any propagation in round-off errors associated with preserving the normality condition. Table 3.1 shows the re-



(a) *Infinity-norm.* **(b)** *Two-norm.*

Figure 3.5: *Error measure between diverged prolongation operator and the prolongation operator produced by Algorithms 1 and 2 as a function of smoothing steps.*

sults. Here, the discrepancy between the prolongation operators is measured in the scaled L^∞ norm. The discrepancy of Algorithm 1 starts off relatively large, but decreases as the number of iterations grows, and converges to the same solution as the MRST program within machine precision. The discrepancy of Algorithm 2 is significantly lower than that of Algorithm 1 in the beginning, and remains lower also after 10,000 iterations. Further, it remains stable, although slightly increasing. After 20,000 iterations the discrepancy of Algorithm 2 is larger than that of Algorithm 1. This is expected, since the prolongation operator produced by Algorithm 2 will deviate slightly from the partition of unity condition as it has never been explicitly enforced on any cells but the ones on the union boundary. To preserve partition of unity it might therefore be expedient to end the iteration by enforcing partition of unity on all cells.

The reason why Algorithm 2 is more numerically stable than Algorithm 1 can be explained by the difference in how they preserve partition of unity. First, consider Algorithm 1. It is well known that subtracting two numbers of approximately the same value is highly prone to round-off errors. This is exactly what happens in Line 4, when performing the subtraction $\overline{h}_{ij} - u_i p_{ij}^k$. When approaching convergence, this value approaches zero, and the algorithm ends up subtracting several approximately equal numbers in each iteration. This introduces a significant round-off error, which results in a deviation from preserving the normality conditions on the union boundary. Further, a single step preserves the normality condition on the union boundary only if it was preserved at the last step. This causes the deviation to propagate, and will increase in every iteration caused by further round-off error. When using Algorithm 2, on the other hand, both of these issues are resolved. The subtraction of two approximately equal numbers is removed, and the partition of unity requirement is enforced on the union boundary in a way that does not require it to hold in the last step. The last point prevents round-off errors associated with this operation to propagate. It is also worth noticing that

Table 3.1: *Discrepancy in the prolongation operator resulting from implementations of Algorithm 1 and 2, with respect to the Matlab program, measured in the scaled L^∞ norm.*

Iterations	Discrepancy	
	Algorithm 1	Algorithm 2
100	$1.4832 \cdot 10^{-3}$	$1.5261 \cdot 10^{-7}$
1000	$8.0641 \cdot 10^{-4}$	$2.5573 \cdot 10^{-7}$
10000	$3.4710 \cdot 10^{-6}$	$2.7195 \cdot 10^{-7}$
20000	$3.3250 \cdot 10^{-8}$	$2.7196 \cdot 10^{-7}$

Algorithm 2 performs fewer arithmetic operations in order to preserver partition of unity. This can be observed in Line 4 of the two algorithms. Where Algorithm 1 performs four operations to each value belonging to the union boundary, Algorithm 2 performs only one. This saves the latter a bit of computational work.

3.7.2 Changing the Smoother

We will now show results from numerical experiments where the different smoothing procedures are compared, by studying the rate of which they reduce the residuals $||\boldsymbol{AP}_j||$. To evaluate the quality of the prolongation operator, we use

$$\tau_k = \frac{\sum_{j=1}^{m} ||\boldsymbol{AP}_j^k||_1}{\sum_{j=1}^{m} ||\boldsymbol{AP}_j^0||_1},$$

where k is the number of iterations. Here, a smaller τ indicates a higher quality than a larger one. As mentioned earlier, this value does not converge to zero. Therefore, we measure the distance from convergence by

$$\lambda^k = \frac{\tau^k - \tau^*}{\tau^0 - \tau^*},$$

where τ^* is the approximately converged value computed numerically. The relaxation factor used by the Jacobi smoother was originally set to $\omega = 2/3$, but in the report [22] we suggested that $\omega = 0.95$ is a better choice. We therefore consider both of these values and compare them with the non-relaxed versions of Algorithm 3 and 4, which have been implemented in Matlab. The iterations are interrupted when either of the following conditions are true,

$$|\tau^k - \tau^{k-1}| < 10^{-8}, \quad \text{or} \quad \tau^{k-1} < \tau^k.$$

That is, the iterations are stopped when the change in the quality of the prolongation operator is sufficiently small, or when the quality decreases. In the test cases, τ_k was strictly decreasing, so the right stopping criteria was never true. We are interested in finding by how much we can reduce the number of iterations when switching from the Jacobi smoother to one of the new ones, while achieving the same quality of the prolongation operator. Therefore, in addition to show plots of λ^k, we include plots of the following. Let τ_k^{J} and τ_k^{RB} be the quality measure obtained by Jacobi and the partially red-black smoother, respectively, after k number of iterations. Here, the Jacobi smoother uses the relaxation factor $\omega = 0.95$. For each k, we find

$$s(k) = \min\left[\tilde{k} \middle| \tau_{\tilde{k}}^{\mathrm{RB}} < \tau_k^{\mathrm{J}}\right],$$

and will show the additional plot $\frac{s(k)}{k}$ as a function of k. The same plot is shown for the BLGS smoother.

The different smoothers were initially tested on several homogeneous rocks on two-dimensional Cartesian grids of various sizes and upscaling factors. In addition, they were tested on different parts of the SPE 10 dataset. We will now present the results of three cases. Note that the other test cases gave similar results. The first case is the rock presented in the example from Section 2.6, which uses a stratigraphic grid and has upscaling factor 600. The second case is Layers 20-29 of the SPE 10 dataset, consisting of $60 \times 220 \times 10$ cells and is divided into $6 \times 11 \times 2$ blocks, giving an upscaling factor of 1,000. The third case is Layer 85 of the SPE 10 dataset, consisting of 60×220 cells and is divided into 6×11 blocks, giving an upscaling factor of 200.

Table 3.2 shows τ^* computed in the three test cases, and the number of iterations required by the different smoothers to reach the stop criteria. Note that, since we consider smoothers, we are mainly interested in the initial reduction in τ, and not the asymptotic convergence rates. Normally, performing 100 iterations would be enough to obtain a good quality prolongation operator. Here, a high amount of iterations were executed to ensure that the algorithms converge to the same solution, and to get a good approximation of the quality measure τ^* for the converged prolongation operator. Figure 3.6 shows plots of $\lambda(k)$ and $s(k)$.

In all cases, the Jacobi smoother converge faster with $\omega = 0.95$ than with $\omega = 2/3$. However, on some highly heterogeneous parts of the SPE 10 dataset, $\omega = 2/3$ reduces λ faster during the very start of the iteration. On Layer 85, for example, choosing $\omega = 2/3$ achieves a higher quality measure during the first 17 iterations. After that, $\omega = 0.95$ provided best results of the two. This suggest that it can be beneficial to apply the relaxation factor in a dynamic manner.

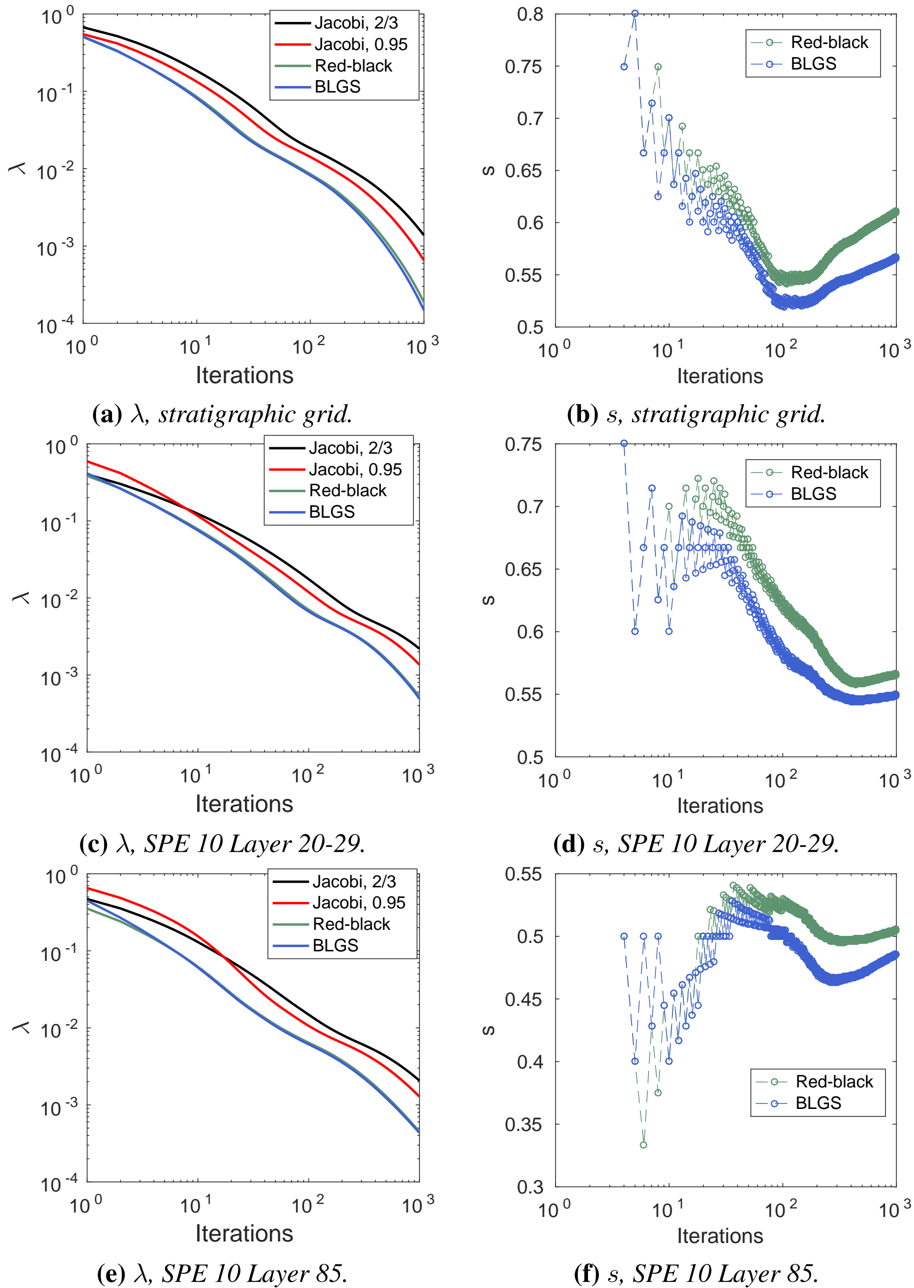
In all cases considered, the BLGS smoother reaches the stopping criteria fastest, and achieves the best asymptotic convergence rate. However, the difference between BLGS and the partially red-black GS smoother is small. During the first few iterations, the partially red-black GS smoother reduces λ fastest of all, except

Table 3.2: *Number of iterations required to reach tolerance for three test-cases, and the converged quality measures τ^*.*

Case	Iterations				τ^*
	Jacobi $(2/3)$	Jacobi (0.95)	Red-black	BLGS	
Stratigraphic grid	4798	3586	2494	2322	$6.44 \cdot 10^{-2}$
SPE 10 Layers 20-29	7950	6037	3858	3742	$1.53 \cdot 10^{-2}$
SPE 10 Layer 85	10210	8299	5395	5293	$1.51 \cdot 10^{-2}$

for a single problem on the SPE 10 dataset, where it was beaten by Jacobi with $\omega = 2/3$ in the first 10 iterations.

The results are evidence that one can reduce the number of iterations by 30-50% by switching from the relaxed Jacobi smoother to either the BLGS or partially red-black GS smoother, and expect to achieve the same quality of the prolongation operator on many problems. But we conclude this chapter with a warning. The relaxation factor can be interpreted as the step length taken at each iteration. In numerical analysis it is well known that increasing the step length often results in a less stable approach. Considering this, it is likely that the Jacobi smoother with $\omega = 2/3$ is more stable than the others, which might lead it to converge in cases where the other smoothers diverge. In particular, this situation might occur for very heterogeneous rocks. However, the smoothers were tested on the SPE 10 Layer 85, which is considered to have a very challenging heterogeneity. Also here the new smoothers proved to be stable, and provided a faster improvement in the quality measure of the prolongation operator. But we recommend to carry out more tests on a wider variety of rock models. In particular, the newly proposed procedures should be tested on unstructured and highly complex grids, and their stability should be investigated on more rock-models with challenging permeability fields. It is also worth investigating if relaxed versions of the new smoothers can improve them further.



(a) *λ, stratigraphic grid.* **(b)** *s, stratigraphic grid.*

(c) *λ, SPE 10 Layer 20-29.* **(d)** *s, SPE 10 Layer 20-29.*

(e) *λ, SPE 10 Layer 85.* **(f)** *s, SPE 10 Layer 85.*

Figure 3.6: *The left figures show the quality measure of the prolongation operator, λ, as a function of the number of steps by the four smoothers. The right figures show the fraction of number of iterations required for the BLGS and partially red-black GS smoothers to reach equally high quality, s, as the Jacobi smoother, as a function of number of steps applied with the Jacobi smoother and $\omega = 0.95$.*

Chapter 4

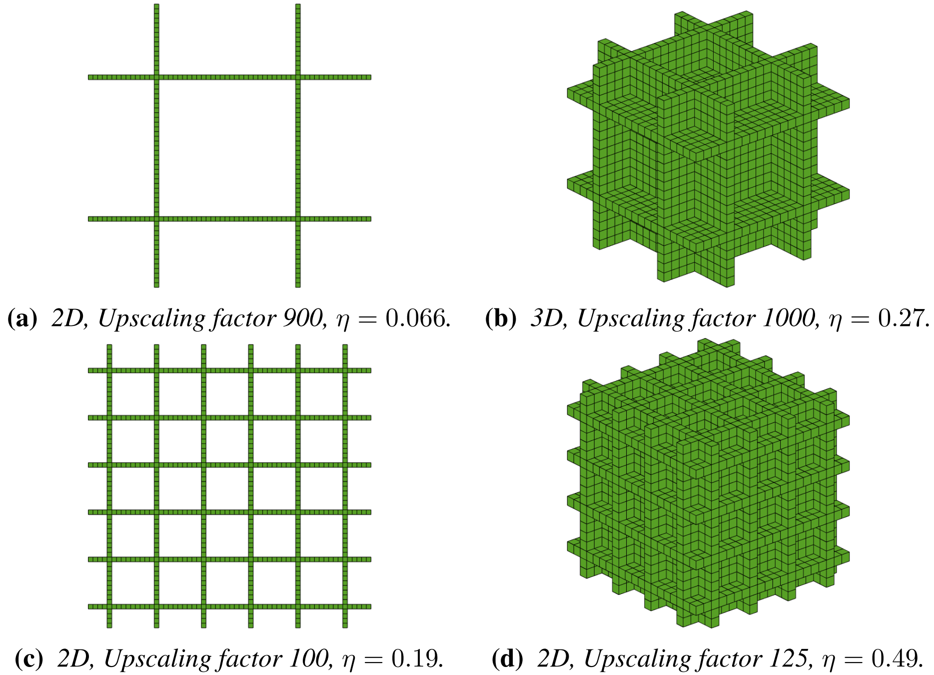
Constructing Prolongation Operator on Distributed Memory Systems

In the fall of 2015, a program that constructs the prolongation operator used by the MsRSB method was created by Olav Møyner, a Ph.D. candidate working for the Department of Applied Mathematics at SINTEF. The program is written in C++ and uses the OpenMP language extension to realize parallel computation on shared memory machines. It is included in MRST, where it has been made available through the use of MEX files [1]. The C++ program we present in this chapter has been based on Olav's implementation, but extends it by applying the Message Passing Interface (MPI) to achieve parallel computation on distributed memory systems.

While the main revision from the shared memory program is the expansion to message-passing, additional improvements were made during the development of the program we present. These have in part been discussed in Chapter 3.7.1, where it was shown that the message-passing program is based on a more numerically stable algorithm.

The chapter is organized as follows. Section 4.1 starts by making a couple of points about the union boundary, U, which will be important when analyzing the performance of the message-passing program. In Section 4.2 we present two sparse data formats that the program use, before presenting the sequential version of the program in Section 4.3. Section 4.4 starts by describing the distribution of computational work and how the message-passing is performed, before presenting the message-passing program. In Section 4.6 we present test results, and at the

end a modification of the underlying algorithm that aims to reduce communication cost. The chapter concludes by a discussion of the results and ideas to improve the message-passing program, which is the topic of Section 4.6.



(a) *2D, Upscaling factor 900,* $\eta = 0.066$. **(b)** *3D, Upscaling factor 1000,* $\eta = 0.27$.

(c) *2D, Upscaling factor 100,* $\eta = 0.19$. **(d)** *2D, Upscaling factor 125,* $\eta = 0.49$.

Figure 4.1: *Union boundary cells,* U*, for square fine and coarse grids, in two and three dimensions.*

4.1 The Union Boundary

The union boundary has been an important topic of discussion in the previous chapter, and we will see that it is just as important in this one. In particular, the fraction of cells that belong to the union boundary will be essential when analyzing the performance of the message-passing program. We therefore define this fraction by η,

$$\eta = \frac{|U|}{|F|}.$$

In this section, we make two points; η increases when the upscaling factor decreases (i.e when adding more blocks), and η is significantly larger on a three-dimensional grid than on a two-dimensional one, if the upscaling factor is the

same. The first point is pretty self explanatory, but the second one might not be as obvious.

In general, η will also depend on the details of the fine grid and how it is partitioned into blocks, so for simplicity we will consider the following case; a square Cartesian grid distributed into equally large, square blocks. Figure 4.1 shows the union boundary for grids of this type in both the two- and three-dimensional case. By merely a visual comparison of the grids in two and three dimensions that have approximately equal upscaling factor, it is clear that η is significantly higher for the three-dimensional grid. Their precise values can be seen in the figure. Denote r to be the upscaling factor. For the particular types of grids of we consider, $\eta(r)$ can be computed analytically, and in fact η is $r^{1/6}$ times larger in the three-dimensional case, for a constant r. Figure 4.2 shows $\eta(r)$ for both dimensions.

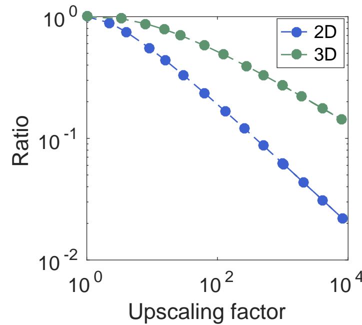


Figure 4.2: *Ratio of cells that belong to the union boundary, η, as a function of the upscaling factor for square fine and coarse grids, in two and three dimensions.*

4.2 Sparse Data Formats

The message-passing program uses two sparse data formats which we will here describe. The first is used to represent matrices, and the second to represent graphs. Zero-based indexing is used in both cases.

Recall that the primary operation carried out to construct the prolongation operator is the Jacobi sweep,

$$\boldsymbol{P}^{k+1} = \boldsymbol{P}^k - \omega \boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{P}^k = \boldsymbol{P}^k - \omega \boldsymbol{M}\boldsymbol{P}^k. \tag{4.1}$$

Among the input parameters used by the message-passing program is the matrix $\boldsymbol{M} = \boldsymbol{D}^{-1}\boldsymbol{A}$, which is computed by MRST routines in Matlab and written to text file in a sparse format. This format is inspired by the *Compressed sparse row / column* (CSR/ CSC) format, but is not quite the same.

Firstly, the diagonal elements of $\boldsymbol{M}$ are not stored as they all equal unity. Define the maximum number of nonzero entries in any row of $\boldsymbol{M}$, excluding the diagonal, to be r,

$$r = \max_{i \in F} \sum_{j=1,\, j \neq i}^{n} \kappa(m_{ij}), \quad \kappa(x) = \begin{cases} 1 & \text{if } x \neq 0, \\ 0 & \text{if } x = 0. \end{cases}$$

The matrix is stored in two arrays that we name $\boldsymbol{a}$ and $\boldsymbol{b}$, both of length $r \times n$. Each array has r entries reserved for every row in $\boldsymbol{M}$, so that entries in the interval $[i \times r,\ i \times r + r - 1]$ hold data about row number i of the matrix. The array $\boldsymbol{a}$ holds the column-indices where nonzero entries in the corresponding row are stored, and $\boldsymbol{b}$ holds the belonging matrix-entries. Let k be the number of nonzero entries on the off-diagonal in a specific row i. If $k < r$, the array $\boldsymbol{a}$ will hold the number -1 in the entries within the interval $[i \times r + k,\ i \times r + r - 1]$, expressing that the reserved space is not needed. That is, obtaining the number -1 from $\boldsymbol{a}$ when iterating over $\boldsymbol{M}$ means that we have reached the end of the current row. The corresponding entries of $\boldsymbol{b}$ will be 0.

The format allows for efficient iterations over the rows, making it suitable when computing (4.1). It also grants fast lookup of a row from its number, which is an advantage when it is computed in parallel. In terms of memory usage, the storage format is suitable when the average number of nonzero entries in the rows of $\boldsymbol{M}$ is close to r. That is, when

$$r - \frac{1}{n} \sum_{i=1}^{n} \Big(\sum_{j=1,\, j \neq i}^{n} \kappa(m_{ij}) \Big)$$

is small. This is the typical case for system matrices obtained from the discretization derived in Section 2.1. On a Cartesian grid in particular, the redundant storage allocated by the format results solely from global boundary cells with fewer connections than the interiors.

Example. *Example: Let the matrix be*

$$\boldsymbol{M} = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -0.2 & 1 & -0.8 & 0 & 0 \\ -0.6 & 0 & 1 & 0 & -0.4 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -0.7 & -0.3 & 1 \end{pmatrix}.$$

Here, $n = 5$ and $r = 2$, so 10 elements will be preserved for each of the two arrays. The matrix will be defined by n, r and the two arrays

$$\boldsymbol{a} = \begin{pmatrix} 1 & -1 & 0 & 2 & 0 & 4 & 1 & -1 & 2 & 3 \end{pmatrix},$$

$$\boldsymbol{b} = \begin{pmatrix} -1 & 0 & -0.2 & -0.8 & -0.6 & -0.4 & -1 & 0 & -0.7 & -0.3 \end{pmatrix}.$$

We will see that the message-passing program relies on an effective division of computational work among the processing units. This division is determined by partitioning of a graph that represents partially dependent computations. The graph is stored using the *compressed storage format* (CSR), which is a commonly used format to store sparse graphs. It is described in the following.

Let y be the number of vertexes and z the number of edges in the graph. The adjacency structure will be stored using two arrays, $\boldsymbol{y}$ of length $y+1$ and $\boldsymbol{z}$ of length $2z$. The list of vertexes that vertex number i is connected to are found within the interval $\left[\boldsymbol{y}(i),\ \boldsymbol{y}(i+1)-1\right]$ of $\boldsymbol{z}$. In other words, $\boldsymbol{z}$ stores the edges of all vertexes ordered from vertex number 0 to vertex number $y-1$, and $\boldsymbol{y}$ holds the information of where in $\boldsymbol{z}$ one vertex ends and another begins.

Example. *Figure 4.3 shows an example of a graph with 8 vertexes and 14 edges.*

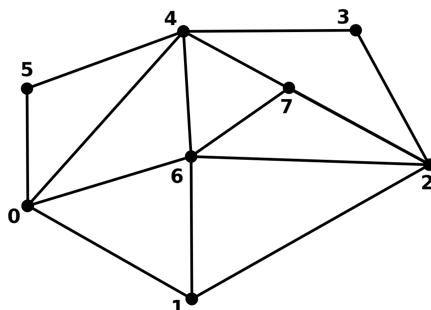


Figure 4.3: *Example of a graph.*

Below are the two arrays $\boldsymbol{y}$ *and* $\boldsymbol{z}$ *for the above graph.*

$$\boldsymbol{y} = \begin{bmatrix} 0 & 4 & 7 & 11 & 13 & 18 & 20 & 25 & 28 \end{bmatrix}.$$

$$\boldsymbol{z} = \begin{bmatrix} 1 & 4 & 5 & 6 & 0 & 2 & 6 & 1 & 3 & 6 & 7 & 2 & 4 & 0 & 3 & 5 & 6 & 7 \\ & & & & & & & & & & & 0 & 4 & 0 & 1 & 2 & 4 & 7 & 2 & 4 & 6 \end{bmatrix}.$$

Notice that in the above example, the list of vertexes that each vertex is connected to is stored in sorted order. This is not a requirement by the CSR format, but has computational benefits when performing search through the graph. For this reason, the sorted ordering of $\boldsymbol{z}$ is used in the implementation implementation of the message-passing program.

4.3 Sequential Program

In the next section we present the message-passing program. But before we get this far, it is constructive to consider the program when executed on a single processor, where operations related to distribution and sending of data are not required. It is stated in Program 1, where m_{il} are entries of the matrix $\boldsymbol{M}$, and H_i is defined as the set of support regions of which cell i belongs to,

$$H_i = \{j|\ i \in I_j\}.$$

For brevity, the stopping criterion is omitted from Program 1.

Program 1. Constructing prolongation operator - Single process

1. Read problem data from text files.
2. Create data structures used in the iteration procedure.
3. Until convergence, do,

   (a) Compute the non-modified increments,

   $$g_{ij} = p_{ij} + \sum_{l \in Q_i \cap I_j} m_{il} p_{lj}, \quad i \in I_j,\ j \in [1, 2, \dots, m].$$

   (b) Update basis functions,

   $$p_{ij} := p_{ij} - \omega g_{ij}, \quad i \in I_j,\ j \in [1, 2, \dots, m].$$

   (c) For each cell on the union boundary, compute the sum of all values belonging to the cell,

   $$s_i = \sum_{l \in H_i} p_{il}, \quad i \in U.$$

   (d) Modify the updated values to preserve partition of unity,

   $$p_{ij} := \begin{cases} \frac{p_{ij}}{s_i}, & \text{if } i \in I_j \cap U, \\ p_{ij} & \text{Otherwise,} \end{cases} \quad i \in I_j,\ j \in [1, 2, \dots, m].$$

4. Write the basis functions to text file.

The main change from Algorithm 2 to Program 1 is that redundant work has been removed. In particular, notice that the program utilizes the matrix sparsity and the limited support of the basis functions. Because cells outside of each support region are never updated, Line 2 of Algorithm 2 is not needed in Program 1.

The message-passing program we present in the next section can also realize parallel computation solely by the use of OpenMP threads, if executed in a shared memory environment. By doing this, it is executed as stated in Program 1, where parallel computations of each line is realized by OpenMP's pragma-calls. This brings us to an important important topic; the use of MPI and OpenMP can be combined in a hybrid approach when executed on a cluster of shared-memory computer units, or nodes, such as the supercomputer Vilje, which is used to test the program in Section 4.5. We will explain this approach based on the circumstances on Vilje.

Vilje has a large amount of nodes, each of which has 16 processing units, or cores, that have access to the same memory. If multiple nodes are used in an application, OpenMP can still be used to manage cores belonging to the same node, but MPI must be used to handle message-passing between the nodes. Recall that MPI realizes parallel computation by spawning a separate instance of the program on each core, or more generally, on each subset of shared memory cores. Here, each instance of the program is called a process. For example, assume that we wish to run our program on 4 nodes, i.e, 64 cores. One viable approach is to open one MPI process on each node, and let each of them manage 16 OpenMP threads to utilize all the cores on their respective node. Other viable approaches to utilize all the cores in this particular case are to open 2, 4 or 8 MPI processes on each node with respective 8, 4 or 2 OpenMP threads on each process. If we open 16 MPI processes on each node, each core is managed by its own process, which is the pure MPI approach.

4.4 Message-Passing Program

A primary question that arises when expanding the program to distributed memory is how to divide the computational work among the MPI processes. Recall that to efficiently take use of the available computer power, the work must be distributed so that the processing units are load balanced at the same time as the communication cost is kept as small as possible. An effective way to enable such a division can be realized if we are able to divide the work into a large number of parts that to a high degree can be computed independently. Creating the prolongation operator allows a division of this type, because the majority of the work applied to each

basis function can be computed separately. We have therefore chosen the natural approach to distribute the work performed on each basis functions among the processes.

A remaining task is to find an effective distribution of the basis functions among the processes. As the first step to accomplish this we create a graph to represent the computational work of each basis function, and the computational dependence between each two of them. This graph is described in Section 4.4.1. The distribution of basis functions is then determined from a partitioning of this graph, that aims to achieve a good load balance and low communication cost. This is the topic of Section 4.4.2. How the communication between the processes is carried out is then described in Section 4.4.3, before the message-passing program is presented.

4.4.1 Graph Representing Computational Work

In the graph that describes the computational work and dependence of the basis functions, a weighted vertex represents the work related to a specific basis function, and a weighted edge represents the dependence between two of them. We will now describe how these weights are determined.

If we consider a singe iteration step of Program 1, there are three main factors that determine the amount of computational work performed on a basis function; how many cells there are in its support region, how many nonzero elements there are in the corresponding rows of $\boldsymbol{M}$, and how many of its cells belong to the union boundary (additional work is carried out to normalize these cells). When evaluating the work associated with a basis function, we ignore the two latter factors and consider only the first. That is, the weight μ_j of the vertex representing basis function number j is set equal to the number of cells in its support region, which we denote $|I_j|$,

$$\mu_j = |I_j|.$$

To determine the computational dependence between two basis functions, consider updating one specific basis function by Program 1, and observe that it can be computed independently except when finding the sums in line 3c,

$$s_i = \sum_{l \in H_i} p_{il}, \quad i \in U,$$

which are used in the next line to normalize the cell-values on its union boundary. To compute a specific s_i, the required data is all (possibly) nonzero values belonging to the cell, which are found in the collection of basis functions that have the

particular cell in their support regions,

$$\{p_{il} | \; l \in H_i\}.$$

In other words, the factor that determines the computational dependence between two basis functions is the number of union boundary cells contained in both of their support regions. We therefore define

$$\gamma_{jk} = |I_j \cap I_k \cap U|,$$

to be the weight of the edge between basis function j and k.

The graph resulting from a small Cartesian grid is shown in the following example.

Example. *We consider a grid that consists of* 28×20 *cells and is partitioned into* 7×4 *equally large blocks. The grid is shown in Figure 4.4c, where the union boundary cells are colored red, and the cell centers of the blocks are colored green. As you can see, the center of the exterior blocks have been moved from the block centroid to the global boundary. To understand the purpose of this, remember how we defined the support regions in Section 2.3. When we move the center of these blocks to the global boundary, we at the same time expand the surrounding support regions to contain the cells close to the global boundary. These cells would otherwise be contained within fewer support regions than the ones in the interior of the grid. This is exemplified by Figure 4.4a and 4.4b, where you see a closeup of the south-west corner of the grid with a highlighted support region before and after the change. Figure 4.5 shows the resulting graph that represents the computation work and dependence of the basis functions.*

4.4.2 Graph Partitioning

After having created the graph described above, the next task is to find a partitioning that meets the two objectives stated in the beginning of this section in the best possible way. This is a standard graph partitioning problem found in many applications, and several algorithms exist that are able to find good partitions. The problem can be formulated as follows. Distribute the vertexes into k partitions so that the sum of the vertex weights within the partitions are approximately equal. At the same time, minimize the total communication cost resulting from the partition. Here, k is the number of processes to which the computations are distributed.

The message-passing program uses the software package METIS [2, 23] to find a good partitioning. METIS can partition graphs using either the *multilevel*

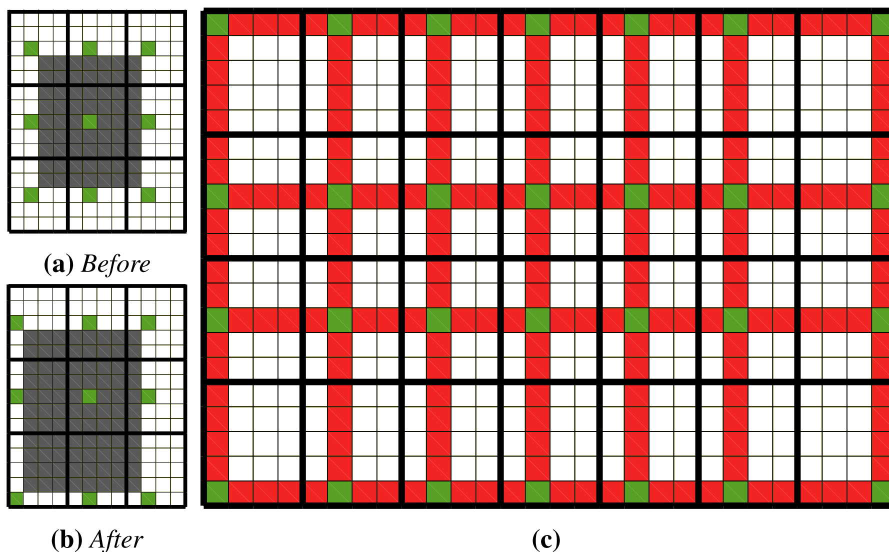


Figure 4.4: *(a,b) A support region highlighted in grey before and after the center of exterior blocks are moved, and* **(c)** *the example grid.*

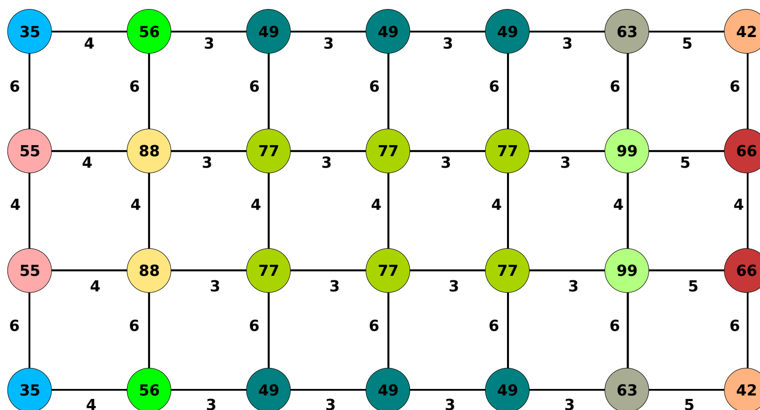


Figure 4.5: *Graph resulting from the grid in Figure 4.4c with vertex weights μ_j and edge weights γ_{jk}. Vertexes with the same weights have been highlighted with the same color.*

recursive bisection (MLRB) algorithm [24] or the *multilevel k-way partitioning* (MLkP) algorithm [25]. Both of these are able to provide high-quality partitions, but MLkP offers more options such as minimizing alternative objective functions, enforcing contiguous partitions, etc. METIS also allows the user to chose between two partitioning objectives; minimizing the *edgecut* or minimizing the total communication volume.

The current implementation of the message-passing program uses the multi-level recursive bisection algorithm with minimizing the edgecut as the objective function.

In general, the total communication cost depends on three factors; the total communication volume, the maximum volume of data any particular processor has to send and receive, and the number of messages a processor has to send and receive [23, 16]. Minimizing the edgecut is the traditional objective used by graph partitioning algorithms, and can be stated as follows; minimize the sum of edge weights connecting vertexes in different partitions. In general, the edge-cut objective does not provide the true minimization of the total communication cost [16], but will often provide a good approximation. The alternative objective function to explicitly minimizing the total communication volume [23, p 24] is often able to create better partitions of graphs with high variation of edges connecting each vertex, called *vertex degrees*. However, on many problems the two objective functions are comparable, and minimizing the edgecut is the cheapest to compute.

For the message-passing program, a brief comparison of MLRB and MLkP was carried out, but neither of them proved to outperform the other in terms of providing a partitioning that reduces the runtime of the iterative procedure. MLRB was therefore chosen because it is cheaper to compute. The alternative objective function of minimizing the total communication volume has not yet been tested, and is worth investigating. However, grids with relatively uniform coarsening as the ones considered in this thesis result in graphs with low variation of vertex degrees, which suggests that there might not be much to gain from applying this change. Figures 4.6 and 4.7 show some resulting partitions of the example-grid considered in Section 4.4.1 and of the SPE 10 dataset, respectively.

4.4.3 Message-Passing

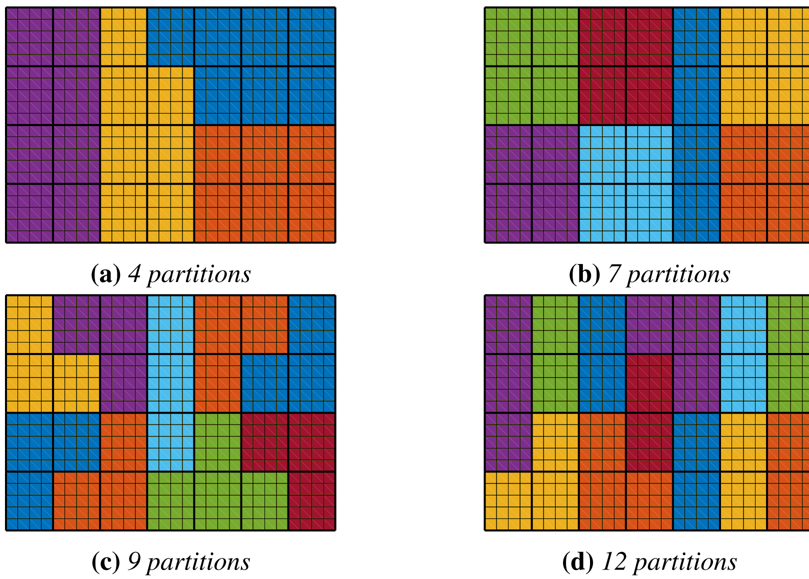
To express the message-passing performed by the program between MPI processes during the iteration procedure, we will define some additional variables.

Let X_c be the set of basis functions that belong to MPI process c, and D_c the set of cells that are located on the same process,

$$D_c = \{i \in I_j |\ j \in X_c\}.$$

Define E_c to be the subset of the union boundary cells that are found solely on process c,

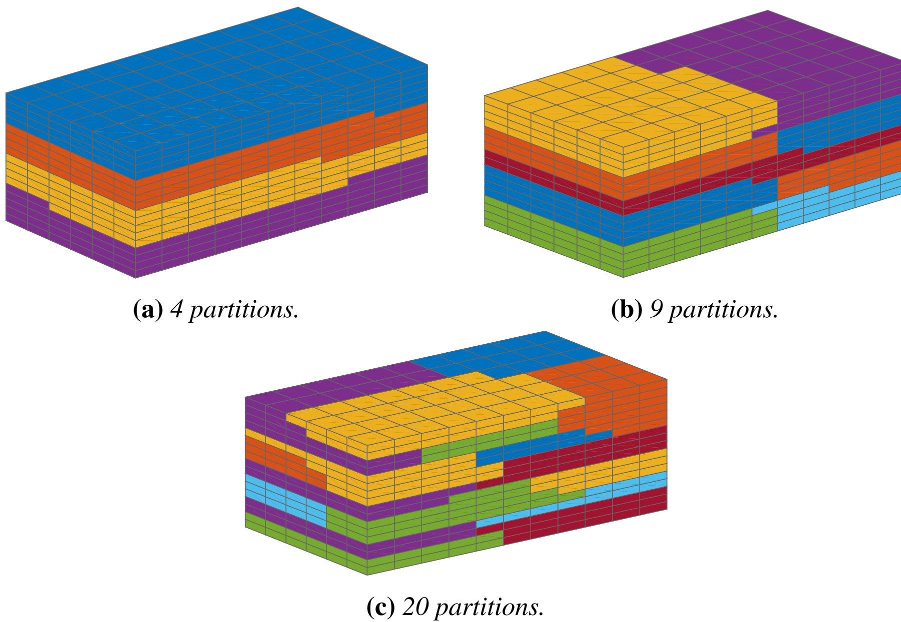
$$E_c = \{i \in U \cap D_c |\ H_i \subset X_c\}.$$



(a) *4 partitions* **(b)** *7 partitions*

(c) *9 partitions* **(d)** *12 partitions*

Figure 4.6: *Partitions of the example-grid considered in Section 4.4.1 resulting from the multilevel recursive bisection algorithm.*



(a) *4 partitions.* **(b)** *9 partitions.*

(c) *20 partitions.*

Figure 4.7: *Partitions of the SPE 10 dataset with* $6 \times 11 \times 17$ *coarse blocks, resulting from the multilevel recursive bisection algorithm.*

Further, define F_c to be the subset of union boundary cells that are found on process c, but also on at least one other process,

$$F_c = U \cap D_c \cap E_c^{\complement}.$$

Figure 4.8 illustrates the sets X_c, D_c, E_c and F_c for a process that has been distributed eight basis functions.

For the process to complete the update of its basis functions, it must obtain the sums s_i belonging to its union boundary cells, that is, the cells in $E_c \cup F_c$. The sums that belong to E_c can be completed locally, but to find the complete sums of cells belonging to F_c, it must acquire data from the other processes that hold these cells. For process c, let Y_c be the set of these processes,

$$Y_c = \{k |\ F_c \cap F_k \neq \emptyset, k \neq c\}.$$

We can now describe how each process obtains the full sums $\{s_i |\ i \in E_c \cup F_c\}$. Each process computes the part of its sums s_i that are available locally. We denote these sums by $\tilde{s}_i$,

$$\tilde{s}_i = \sum_{l \in H_i \cap X_c} p_{il}, \quad i \in E_c \cup F_c.$$

The process then sends the incomplete sums $\{s_i |\ i \in F_c\}$ to all processes in Y_c, and receives data from the same processes. After the message-passing has been

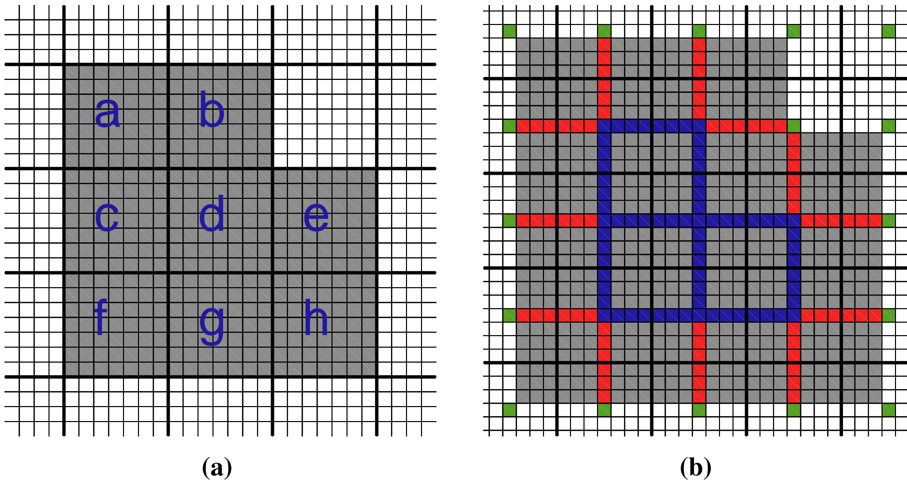


Figure 4.8: ***(a)*** *Highlighted blocks distributed to a process, giving $X_c = \{a, b, c, d, e, f, g, h\}$.* ***(b)*** *Highlighted cells belonging to D_c, E_c and F_c. Here, cells in E_c are blue, cells in F_c are red, and D_c are cells in both E_c and F_c, and the grey ones. Neighboring block-centers are colored green.*

completed, each process has the required information to complete the sums and normalize their cells.

The above approach results in a higher communication volume than what is actually required to complete the computations. The reason is that each processes sends the same package of data $\{s_i|\ i \in F_c\}$ to all its dependent processes, whereas each of the receiving ones only require a subset of these values. There is nothing that prevents the program to be revised to reduce the communication volume to the minimum of what the computations needs. However, this would require each process to send custom-made data packages to each of the receiving processes. Note that there is a mismatch between the communication volume resulting from the above approach and the objective used to distribute the basis functions. The partitioning of the graph discussed in Section 4.4.1 by the objective discussed in Section 4.4.2 aims to (approximately) minimize the number of cells in U that belong to each pair of processes, which we denote T_v,

$$T_\text{v} = \sum_{c \neq e} |D_c \cap D_e \cap U|. \tag{4.2}$$

However, the communication volume of the above approach is determined from

$$\tilde{T}_\text{v} = \sum_{c} |F_c|. \tag{4.3}$$

A distribution that minimizes (4.2) and (4.3) is generally not the same, and the partitioning is therefore determined by the wrong criteria. However, if the program is revised to decrease the communication volume to the minimum requirement as discussed above, equation (4.2) is the correct objective to minimize.

The sending and receiving of $\{s_i|\ i \in F_c\}$ is carried out by the MPI functions `MPI_Isend()` and `MPI_Irecv()`, which are *non-blocking* one-to-one communications. In message-passing, blocking communication calls are functions that do not return before the communication has been completed. Non-blocking communication calls, on the other hand, return immediately after the request has been executed, even if the communication is not finished. This allows the program to continue performing its tasks while the message-passing is carried out. In our case, non-blocking communications were chosen to avoid the program to *deadlock*, which occurs when a process waits for a message to be sent or received from another one, but this never happens. For example, if blocking communication calls are used for communication among two processes, and each of them starts by sending a message to the other, both of them will stop and wait for the other process to receive the message, but it never does. Now we are ready to present the message-passing program, which is stated in Program 2.

Program 2. Constructing prolongation operator - Message-passing

1. If $c = 0$, do
   (a) Read the problem data from text files.
   (b) Create the graph representing the basis functions computations.
   (c) Create the partitioning of the basis functions.
   (d) Distribute the problem data to the rest of the processes.
2. Else, do
   (a) Receive problem data from process number 0.
3. Create the data structures used in the iteration procedure.
4. Until convergence, do,
   (a) Compute the non-modified increments,
   $$g_{ij} = p_{ij} + \sum_{l \in Q_i \cap I_j} m_{il} p_{lj}, \quad i \in I_j,\ j \in X_c.$$
   (b) Update the basis functions,
   $$p_{ij} := p_{ij} - \omega g_{ij}, \quad i \in I_j,\ j \in X_c.$$
   (c) Compute the local sums $\tilde{s}_i$,
   $$\tilde{s}_i = \sum_{l \in H_i \cap X_c} p_{il}, \quad i \in E_c \cup F_c.$$
   (d) Send the sums $\{\tilde{s}_i | i \in F_c\}$ to the processes in Y_c. Receive data from the same processes.
   (e) Complete the sums $\{\tilde{s}_i | i \in F_c\}$ and set $\boldsymbol{s} = \tilde{\boldsymbol{s}}$.
   (f) Modify the updated values to preserve partition of unity,
   $$p_{ij} := \begin{cases} \frac{p_{ij}}{s_i}, & \text{if } i \in E_c \cup F_c, \\ p_{ij} & \text{Otherwise,} \end{cases} \quad i \in I_j,\ j \in X_c.$$
5. Gather the basis functions on a single processing unit and write them to text file.

4.5 Testing the Program

The distributed memory program has been tested on Vilje, which is a cluster produced by NTNU together with met.no and UNINETT Sigma. Table 4.1 summarizes key information about Vilje. For more information, we refer to its web page [6].

Table 4.1: *Information about Vilje.*

System architecture and operating environment	
Manufacturer	SGI
Interconnect	Mellanox FDR infiniband, Enhanced Hypercube Topology
Operating System	SUSE Linux Enterprise Server 11
Compilers	Intel and GNU C and Fortran
MPI library	SGI MPT
Number of nodes	1404
Number of cores	22464
Node details	
Processors per node	2 eight-core processors
Node Type	Intel Xeon E5-2670 (Sandy Bridge)
Processor Speed	2.6 GHz
Cores per Node	16 (dual eight-core)
L3 Cache	20MB / 8 cores
Memory	2GB per core

The wall time used by the program to compute the prolongation operator has been documented on several test-problems. Various problems were chosen to investigate the performance for a wide range of grid-sizes and upscaling factors. The program was initially tested on two-dimensional problems that use a square Cartesian grid both on the fine and coarse scale, with the number of cells varying between 10,000 and 16,000,000, and upscaling factor between 100 and 1600. The program has also been tested on the SPE 10 dataset using a $6 \times 11 \times 17$ coarse grid. This is one of the models that have been used to assess the performance of the MsRSB method [38, 39]. Various grid-types have not been a significant topic in this thesis, but recall that one of the strengths of the MsRSB method is that it handles unstructured polyhedral grids. The distributed memory program has also been developed to handle these types if grids, and we have have therefore also tested the program on unstructured PEBI-grids.

When presenting the results, we separate the times used in the setup procedure (computations that are carried out before the iteration can start, e.g, creating data

structures, distributing the problem data) and in the iteration procedure, which enables us to analyze the two independently. When presenting the iteration times we will show the time used to perform a single iteration. This is done so that the times do not depend on the stopping criteria.

To assess the single-processor performance of the message-passing program, it is compared with the original program created by O. Møyner when executed on a single core. The results are the topic of Section 4.5.1. In Section 4.5.2 we present speedup results of the iteration procedure for five different test cases. The first three use a two-dimensional Cartesian grid of various sizes, the forth is the SPE 10 dataset, and the fifth is an unstructured PEBI-grid. The setup times for the five cases are presented in Section 4.5.3. In Section 4.5.4 we present a modification of the algorithm of which the goal is to reduce the communication cost of the program, and investigate its application on the SPE 10 dataset.

Our primary goal is to asses the computational efficiency of the basis iterations when the problem is already stored in memory, and no work has been done to the functions that read and write data to and from text files. Here, the message-passing program uses the same sequential operations as the original shared memory program. We therefore exclude these times when presenting the results, but the time to read the five test cases is shown in Table 4.2. Here, the fraction of cells belonging to the union boundary, η, is also shown, which play an important role in the discussion of the iteration performance.

Table 4.2: *Time to read problem data and η for the five test cases.*

	Test Case	Time to read (ms)	η
1	2D, Small	300	0.19
2	2D, Medium	2700	0.0975
3	2D, Large	66,000	0.049
4	SPE 10	7900	0.32
5	Unstructured	2300	0.70

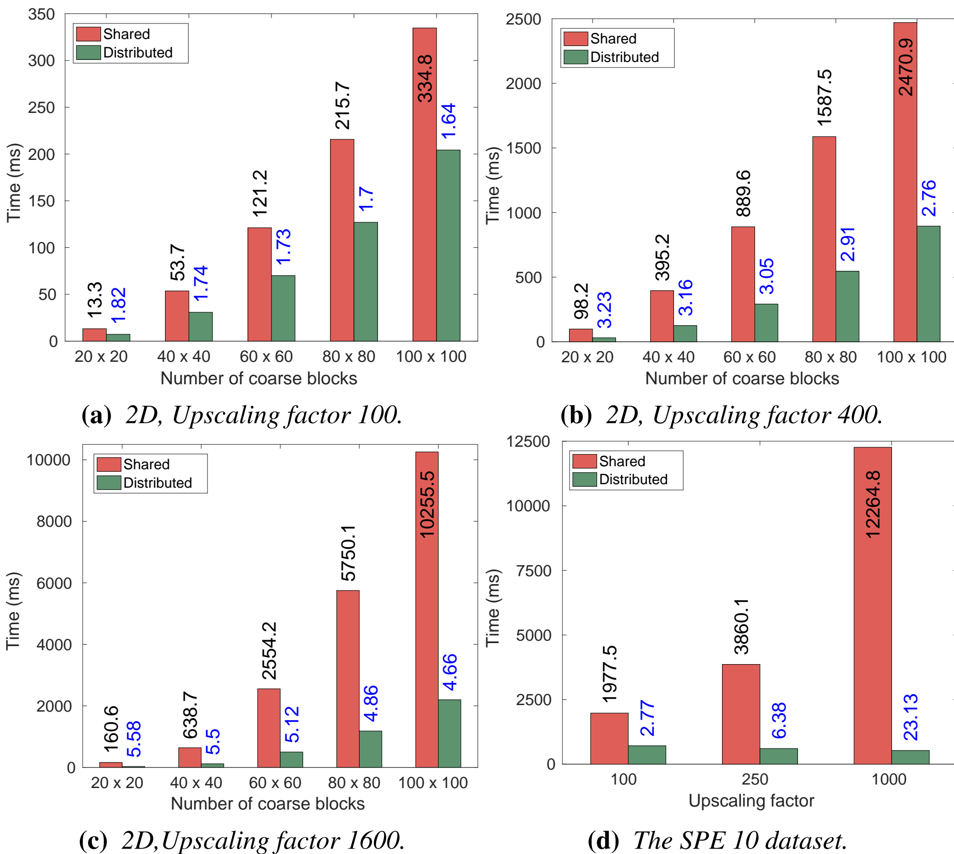
Note that the times we present are in each case results of a single run. The runtimes will always vary slightly in consecutive runs, and repeating the tests will therefore produce slightly different results. However, the difference was observed to be insignificant with respect to assessing the efficiency of the program.

4.5.1 Single-Processor Runtimes

To efficiently use a multiprocessor environment, the single-processor performance is essential. A program might achieve high speedup, but if the sequential execution

is slow it still suggests poor use of computing resources. An impressive speedup is typically harder to achieve for a program that is already efficient in serial than one which is not, but what actually matters is the absolute time used. We therefore compare the single-processor performance of the message-passing program with the program created by O. Møyner, and refer to the two as the new and the original program, respectively. The results are shown in Figures 4.9 and 4.10.

The setup times are significantly lower for the new program, especially for the large problems. In particular, the new program only uses 4.32% of the original time for the SPE 10 dataset with upscaling factor 1000. Without going into the details of the setup procedures, we mention that the improvement is a result of



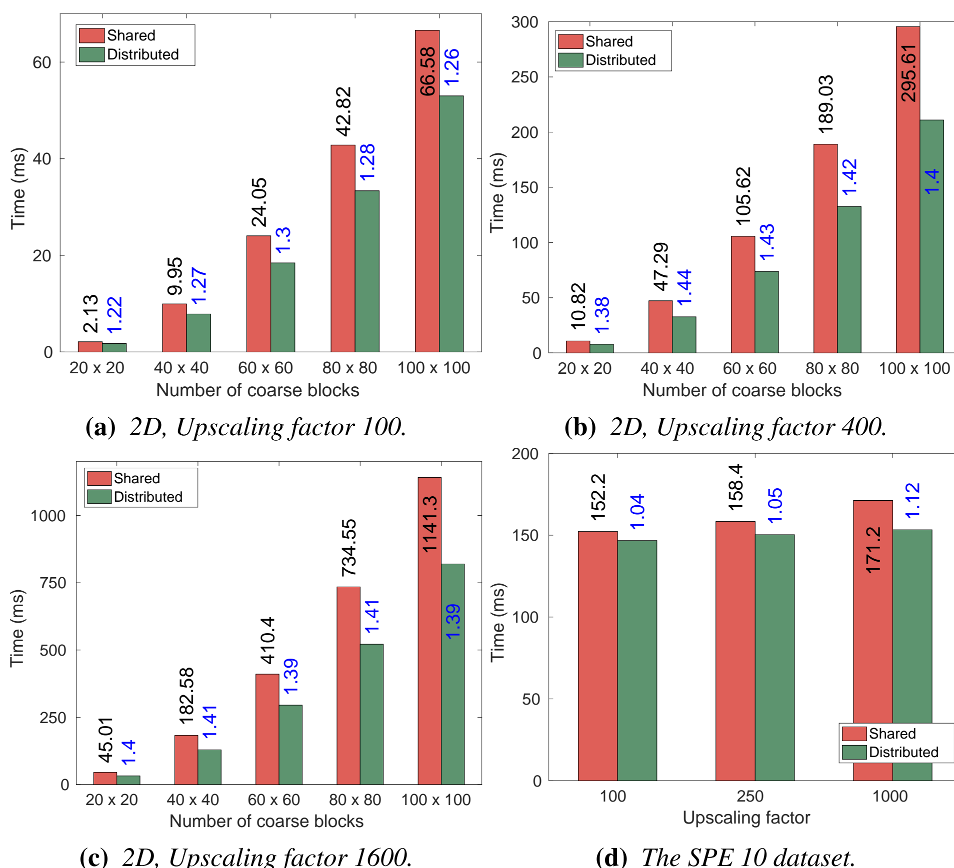
(a) *2D, Upscaling factor 100.*

(b) *2D, Upscaling factor 400.*

(c) *2D,Upscaling factor 1600.*

(d) *The SPE 10 dataset.*

Figure 4.9: *Setup times used by the original (shared memory) program and the new (distributed memory) program when executed on a single core on Vilje. Numbers over the red bars are absolute times used by the original program, and numbers over the green bars are relative speedups of the new program, with respect to the original.*



(a) *2D, Upscaling factor 100.*

(b) *2D, Upscaling factor 400.*

(c) *2D, Upscaling factor 1600.*

(d) *The SPE 10 dataset.*

Figure 4.10: *Times used to perform a single iteration by the original (shared memory) program and the new (distributed memory) program when executed on a single core on Vilje. Numbers over the red bars are absolute times used by the original program, and numbers over the green bars are relative speedups of the new program, with respect to the original.*

avoiding nested loops by instead applying a mapping technique when creating the data structures that are used in the iteration.

Regarding the iteration times, the difference is more modest, but the new program is faster here as well. On the two-dimensional grids the new program achieves speedups in the range of $1.20 - 1.40$ relative to the original. On the SPE 10 dataset the difference is smaller, especially for a small upscaling factor.

To explain these results, we must first understand why the new program is faster than the original. There are three main reasons. Firstly, it performs slightly fewer floating point operations in each iteration. Secondly, it evades having to

perform the extra normalization of the entire grid. Both of these reasons are due to the change in the underlying algorithm, and were discussed in in Chapter 3.7.1. However, the prime place where the new program performs better lies in how it normalizes the union boundary cells. While the original program achieves this by iterating over the entire grid, the new program only iterates over the union boundary cells, which locations are computed during the setup.

The benefit of the new approach will therefore generally depend on η. In fact, if η is high enough, the original approach is expected to perform better, because it avoids a high amount of jumps in the memory when performing the normalization. This explains why the difference between the two programs is lower for the SPE 10 dataset, and also why it decreases when lowering the upscaling factor; as discussed in Section 4.1, η is significantly higher for three-dimensional grids, and increases when lowering the upscaling factor. Table 4.3 shows η for the test problems. Note that in practice one would typically use an upscaling factor of 1000 rather than 100 on the SPE 10 data set, which is also where the best results are achieved.

It is particularly encouraging that the new program also seems to scale better with the problem size, since its intended application is on large problems.

Table 4.3: *The fraction of cells belonging to the union boundary, η, for the test-grids.*

Upscaling factor	100	250	400	1000	1600
			η		
2D	0.19	0.12	0.098	0.063	0.049
SPE 10	0.52	0.42	-	0.32	-

4.5.2 Iteration Speedup

We will now present speedup results on Vilje for five test cases. The use of solely MPI processes has been compared with the different hybrid alternatives of 2,4,8 and 16 OpenMP threads per MPI process. The hybrid approach that in most cases provided the best alternative to pure MPI was the use of 8 OpenMP threads per MPI process (i.e one process on each processor). Results of this combination is presented for the four larger test cases. The times are results of applying 1000 iterations, and no convergence test was applied.

Test Case 1: Small two-dimensional grid

The first model we consider uses a square Cartesian grid with 200×200 cells and 20×20 blocks, giving an upscaling factor of 100. The speedup achieved by the

pure MPI approach is shown in Figure 4.11.

Initially, the speedup is close to perfect, but after reaching a maximum of 41 on 64 cores, it starts to decline. This is not unexpected, because the problem is relatively small and the workload on each core soon becomes modest compared with the overhead introduced by the parallel computing. Figure 4.12 is evidence of this, where you see how the fraction of time used to send and receive data among MPI processes increases as we add more of them. In particular, almost 50% of the total time is used for message-passing when 80 cores are used. Note that if we look at the speedup gain of performing the Jacobi procedure isolated, it is almost perfect all the way up to 80 cores, which is evidence of good load balance.

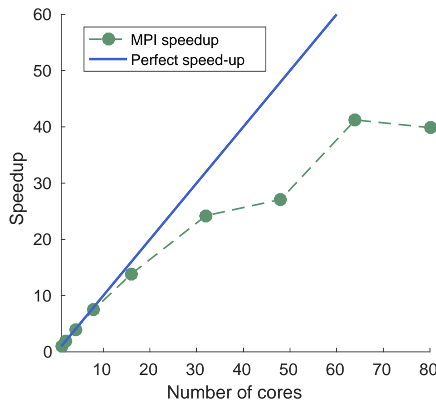


Figure 4.11: *Speedup of the iteration procedure using the pure MPI approach for Test Case 1.*

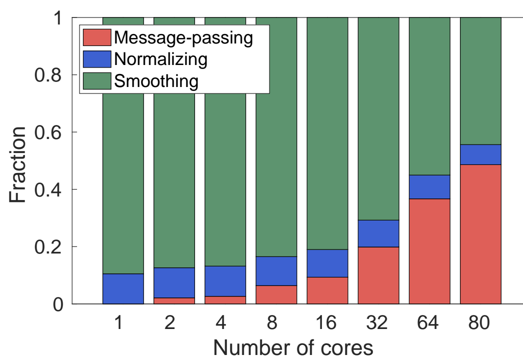


Figure 4.12: *Fraction of time used in different parts of the iteration procedure by the pure MPI approach for Test Case 1.*

Test Case 2: Medium Two-dimensional Grid

We now consider a Cartesian grid with with 800×800 cells and 40×40 blocks, giving an upscaling factor of 400. The speedup for the pure MPI approach and the hybrid approach is shown in Figure 4.13.

An increasing speedup is achieved for a much higher number of cores than on Test Case 1, and it is close to perfect up to 160 cores. Here there is no significant difference between the pure MPI approach and the hybrid approach. After reaching a maximum speedup of 581 on 960 cores, the MPI speedup starts to decline. For the hybrid approach, however, the speedup increases further to reach 685 on 1600 cores.

That the speedup benefits from a higher number of cores here than in Test Case 1 is as expected. Because the problem is larger, there is more computational work to distribute among the cores, and a higher number of cores can be used before the overhead associated with communication becomes a dominating factor. This becomes evident in Figure 4.14. Even on 640 cores, the fraction of time spent to communicate is lower than it is on 80 cores in Test Case 1.

There are two reasons why the hybrid approach performs better on a high number of cores. Firstly, it exploits the shared memory access inside the nodes, which reduces the total communication cost by decreasing both the total communication volume and the number of messages. In particular, message passing is performed between one eight of the number of processes compared to the pure MPI approach,

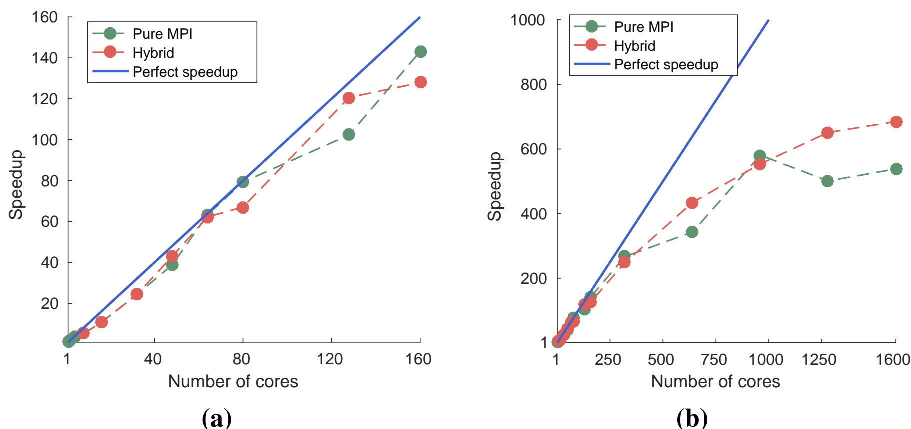


Figure 4.13: *Speedup of the iteration procedure for the pure MPI approach and the hybrid approach with 8 OpenMP threads in each MPI processes, for Test Case 2. Figure **(a)** is a closeup for a small number of cores.*

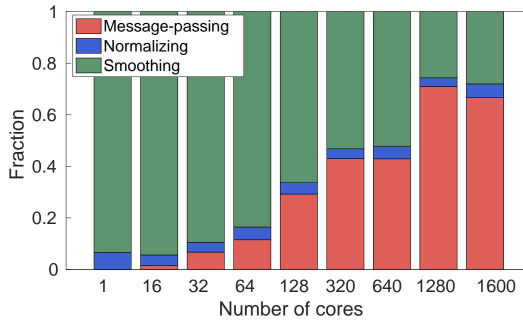


Figure 4.14: *Fraction of time used in different parts of the iteration procedure by the pure MPI approach for Test Case 2.*

and the number of union boundary cells that belong to difference processes is therefore also lower. Secondly, the pure MPI approach relies on distributing the basis functions among all cores, of which there are no more than 1600. The result is that it is harder to obtain a good load balance when the number of cores approaches the number of basis functions. In particular, it is not possible to divide the work between more than 1600 cores, and further, a division among 1600 cores results in a bad load balance since the work performed to each basis function varies. The hybrid alternative, on the other hand, allows eight cores compute a single basis function, and can therefore in theory benefit up until 12,800 cores.

Test Case 3: Large Two-dimensional Grid

We will now present results from the largest problem that was tested, which is a Cartesian grid with 4000×4000 cells divided into 100×100 blocks, giving an upscaling factor of 1600. The speedup for the pure MPI and hybrid approach are shown in Figure 4.15. Here, a solid speedup is achieved all the way up to the maximum of 1600 processors, where 1440 is reached for the hybrid approach.

The outstanding speedup results for this particular problem has several reasons. Firstly, the problem is large enough to provide a high workload to a large number of cores. Secondly, is consist of a high number of basis functions, which makes it easy to achieve a good load balance for a high number of cores. Thirdly, it has the smallest η of the test cases, meaning low dependency between the basis function and as a result, low communication cost. The forth reason we remark is slightly more intricate. We have not discussed memory usage up until now, but for a problem of this magnitude the memory access time may play a significant role.

Note that the speedup on a low number of cores is further away from perfect

here than it is on the previous test cases. In particular, where the smaller cases achieve a close to perfect speedup on 16 cores, this one achieves no more than 10.4. As you see in Figure 4.16, the communication cost is negligible on 16 nodes, and can not be the explanation. It is therefore evidence that memory access time is a limiting factor.

With constant access to fast memory, the speedup gained from adding more cores will suffer if the system does not fetch data from memory fast enough to provide them with the required data to perform computations consecutively. This is a typical bottleneck for large problems. In particular, note that Test Case 1 takes up 23MB on disk, while this test case takes up 2600MB. A single node on Vilje has access to 40MB L3 Cache (fast memory), meaning that Test Case 1 can be stored entirely in the L3 cache, while the present test case must be stored primarily in the main memory (slow memory). This problem will therefore rely on a high amount of fetching data from main memory, which was not an issue for Test Case 1. But when we add more nodes, we also add more L3 Cache. In particular, when using 1600 cores (100 nodes), the access to L3 cache has increased to 4000MB, which decreases or removes the bottleneck of fetching memory for our test case. In other words, the gain in speedup is not only a result of gaining more processing cores, but also from increasing the access to fast memory.

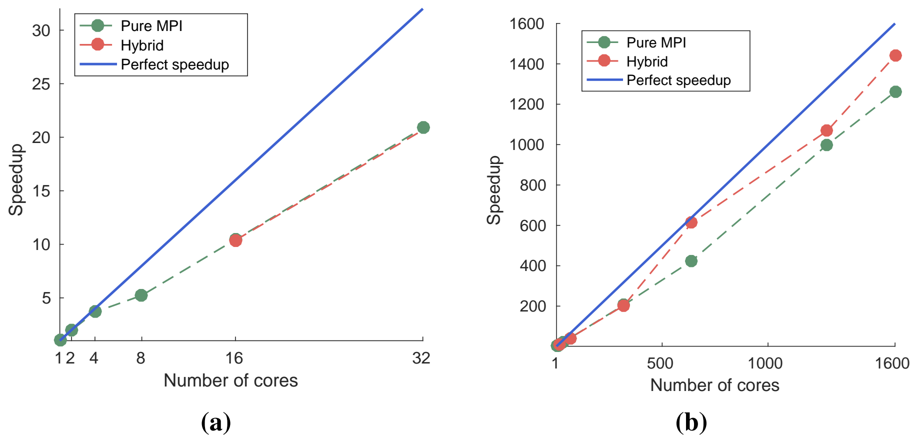


Figure 4.15: *Speedup of the iteration procedure for the pure MPI approach and the hybrid approach with 8 OpenMP threads in each MPI processes, for Test Case 3. Figure **(a)** is a closeup for a small number of cores.*

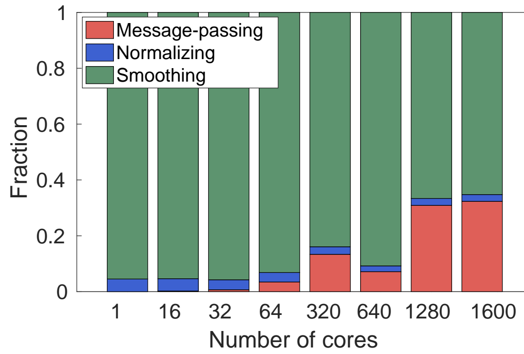


Figure 4.16: *Fraction of time used in different parts of the iteration procedure by the pure MPI approach for Test Case 3.*

Test Case 4: SPE 10 Dataset

We will now present results from the SPE 10 dataset, which uses a Cartesian grid with $60 \times 220 \times 85$ cells and $6 \times 11 \times 17$ blocks, giving an upscaling factor of 1000. The speedup for the pure MPI approach and the hybrid approach is shown in Figure 4.17.

Here the speedup is far less impressive than it was on the previous example when a large number of cores are used. Why this is the case is made clear from Figure 4.18; the fraction of time used in message-passing increases much more

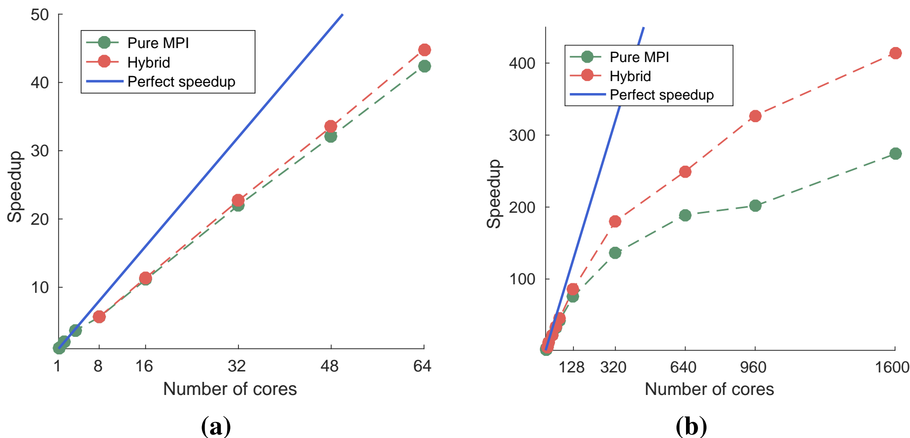


Figure 4.17: *Speedup of the iteration procedure for the pure MPI approach and the hybrid approach with 8 OpenMP threads in each MPI processes, for Test Case 4. Figure* ***(a)*** *is a closeup for a small number of cores.*

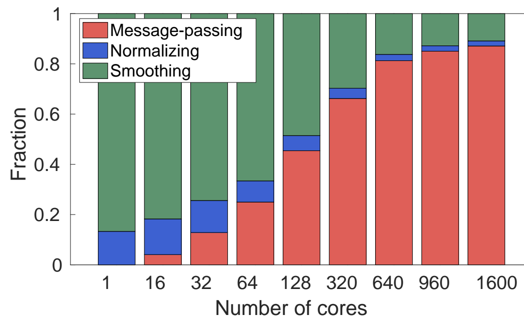


Figure 4.18: *Fraction of time used in different parts of the iteration procedure by the pure MPI approach for Test Case 4.*

drastically. The reason has been discussed earlier; the fraction of cells on the union boundary, η, is significantly larger for three-dimensional problems, which increases the communication volume. The increased number of connections in three dimensions also makes the number of messages to be sent higher. In addition, there are not more than 1122 basis functions to be computed. The pure MPI approach on 1600 cores therefore results in 478 of them not even being used. These two points also explain why the benefit of using the hybrid approach is higher here than on the previous test cases. The hybrid run reached a speedup of 414 on 1600 cores.

Test Case 5: Unstructured Grid

As the last test case we consider an 2.5 PEBI-grid, which is fully unstructured in the horizontal directions and structured in the vertical direction. The grid has approximately 100 cells in each horizontal direction and 25 cells in the vertical direction, with a total of 288,325 cells. It is partitioned according to connection strengths by the use of METIS into 300 blocks. This gives an upscaling factor of 961. The grid is shown in Figure 4.19a, where the blocks have been highlighted. In Figure 4.19b you see a smaller grid of the same type, where the cell-structure is visible.

The speedup for the pure MPI and hybrid approach is shown in Figure 4.20. Here, the pure MPI approach reaches its maximum speedup of 23 on 64 cores. The Hybrid approach performs significantly better, and achieves a speedup of 58 on 160 cores.

Why a higher speedup is not reached from the pure MPI approach has the same reasons as for Test Case 4, but they are now even more prominent. This test case

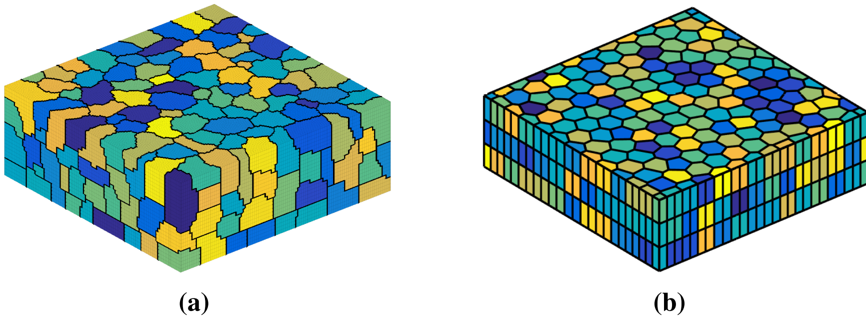


Figure 4.19: *(**a**) Highlighted coarse blocks for Test Case 5, and (**b**) a small 2.5 PEBI-grid.*

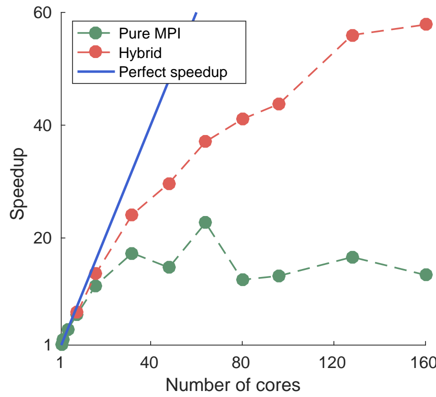
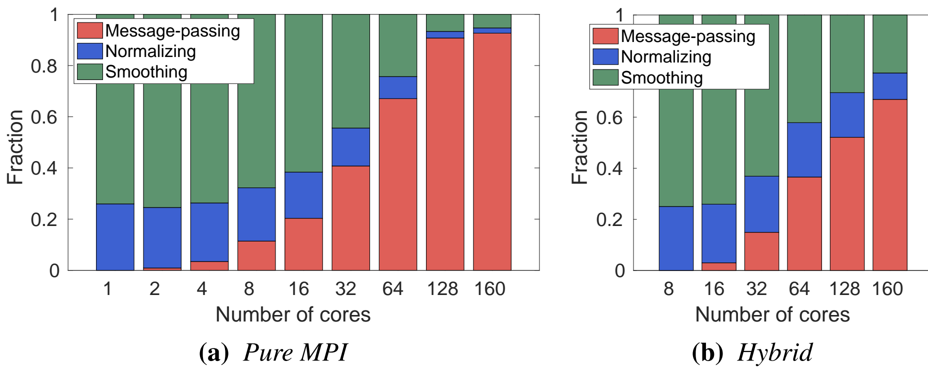


Figure 4.20: *Speedup of the iteration procedure for the pure MPI approach and the hybrid approach with 8 OpenMP threads in each MPI process, for Test Case 5.*



(a) *Pure MPI* **(b)** *Hybrid*

Figure 4.21: *Fraction of time used in different parts of the iteration procedure by the pure MPI and Hybrid approach for Test Case 5.*

has the highest fraction of union boundary cells, with $\eta = 0.70$, which explains the rapid growth in communication cost in Figure 4.21a. For comparison, Figure 4.21b shows the same results for the hybrid runs. Note also that there are only 300 basis functions to be computed, which again makes it hard to get a good load balance for more than a relatively low number of cores.

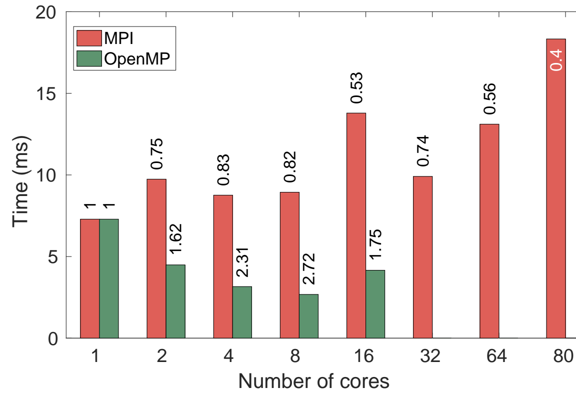
4.5.3 Setup Cost

The times used to perform the setup procedure for the five test cases are shown in Figure 4.22 and 4.23. For the two-dimensional cases, we see that the setup times used by the MPI and hybrid runs are for the most parts close to the time used in serial. Note that when MPI is used, additional computations are performed in the setup to distribute the basis functions. The two most time-consuming steps are to create the partitioning of the basis functions, and to distribute the data among the processes. Both of these are executed by the single MPI process which is responsible to read the problem data from file. Parts of these computations can achieve parallel computation from OpenMP threads. But when a high number of processes are used, the setup time is dominated by distributing the data, and here OpenMP can not be used.

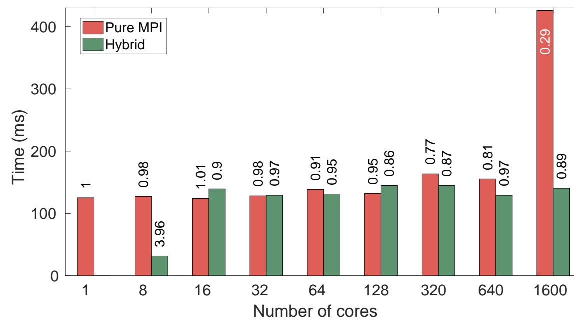
While distributing the problem data results in an additional overhead which is hard to avoid, a speedup is gained in the setup-procedures that follow after the problem has been distributed. These involve creating the data structures that are used in the iteration. For many cases we see that this compensates for the extra time to distribute the problem, and results in approximately the same total setup times. A pure OpenMP run, on the other hand, requires no additional setup computations relative to a single-processor run. Here, we therefore achieve a modest speedup, which results from applying OpenMP threads to parts of the setup computations.

On the three-dimensional test cases, the increase in setup time is higher. The reason has been found to be a large growth in the time it takes to compute the sets F_c, and is therefore directly linked to the high values of η. This is most likely due to an inefficient implementation, and is probably easy to improve.

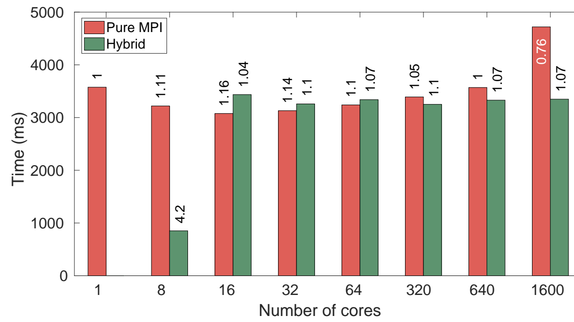
Note that the main focus has been to achieve high efficiency of the basis iterations, and less work has been spent to allow for parallel computation of the setup procedure. In particular, only a few modifications is required to expand the use of OpenMP threads.



(a) *Test Case 1.*

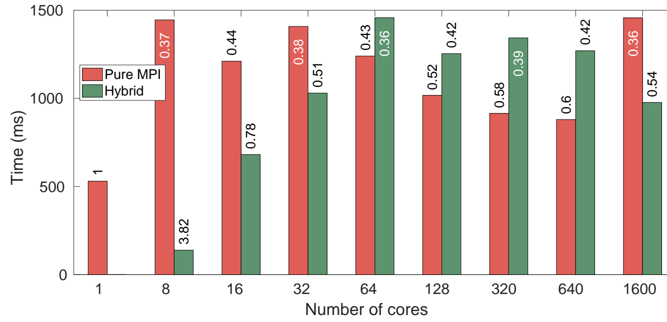


(b) *Test Case 2.*

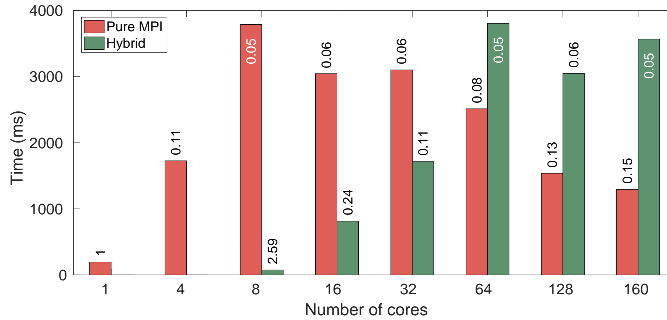


(c) *Test Case 3.*

Figure 4.22: *Setup times for the pure MPI approach and the pure OpenMP approach for Test Case 1, and setup times for the pure MPI approach and the hybrid approach for Test Case 2 and 3. The numbers over the bars are the speedups.*



(a) *Test Case 4.*



(b) *Test Case 5.*

Figure 4.23: *Setup times for the pure MPI approach and the hybrid approach for Test Case 4 and 5. The numbers over the bars are the speedups.*

4.5.4 Modification to Reduce Communication Cost

It is clear that the communication cost of the message-passing program is a bottleneck for achieving high speedup on grids with large η. We therefore introduce a modified version of Algorithm 2, where the goal is to reduce the amount of communication needed to reach the convergence criteria. The idea is simple; in each iteration step, start by performing the Jacobi smoother on all cells that do not belong to the union boundary a predefined number of times, then continue with Line 1 of Algorithm 2. It is stated in Algorithm 5. Here, Line 2 preserves partition of unity, because cell-values on $U^{\complement}$ remain untouched. The procedure requires the same amount of communication in each iteration step step as Algorithm 2, however, the hope is that the additional smoothing of cells in $U^{\complement}$ will lead to fewer steps to reach tolerance, and therefore reduce the total communication cost. The distributed memory program has been extended from Algorithm 2 to Algorithm 5, where Algorithm 2 is retrieved by merely choosing $s = 0$.

Consider running Test Case 4 (SPE 10) on 640 cores. As was presented in

Algorithm 5. Constructing the prolongation operator - $U^{\complement}$ smoothing

1. Set $\hat{\boldsymbol{P}}^0 = \boldsymbol{P}^k$
2. For $t = 1, 2, \ldots, s-1, s$, do
$$\hat{p}_{ij}^{t+1} = \hat{p}_{ij}^{t} - \frac{\omega}{a_{ii}} \sum_{l=1}^{n} a_{il} \hat{p}_{lj}^{t}, \quad i \in F \cap U^{\complement},\ j \in [1, 2, \ldots, m].$$
3. Set $\boldsymbol{P}^k = \hat{\boldsymbol{P}}^s$ and perform line 1 to 5 of Algorithm 2.
4. Define the error as
$$e = \max_{ij} \left(|p_{ij}^{k+1} - p^k| \right), \quad i \in F \cap U^{\complement},\ j \in [1, 2, \ldots, m].$$
If $e <$tol, return $\boldsymbol{P} = \boldsymbol{P}^{k+1}$, otherwise go to step 1.

Section 4.5.2, a hybrid run with 8 OpenMP threads per MPI process reaches a speedup of 250, and is significantly faster than pure MPI. But 53% of the time is still used in message-passing. We have therefore tested if Algorithm 5 offers an improvement on this particular case. The test was carried out in the following way. We enable the convergence test for the original procedure ($s = 0$), and find the number of iterations needed to reach tolerance. We then find how many iterations are required by Algorithm 5 with $s = 1$ to compute a prolongation operator with equally good quality measure τ; recall from Section 3.7.2 that

$$\tau_k = \frac{\sum_{j=1}^{m} ||\boldsymbol{A}\boldsymbol{P}_j^k||_1}{\sum_{j=1}^{m} ||\boldsymbol{A}\boldsymbol{P}_j^0||_1},$$

is used to measure the smoothness of the basis functions. The respective runs are timed, and the result is shown in Table 4.4 for three different tolerances.

Choosing $s = 1$ has significantly reduced the number of iterations, and achieves a speedup compared with $s = 0$ for the particular computation circumstance. The speedup solely results from the fact that the fraction of time used in message-passing has decreased from 0.53 for $s = 0$ to 0.4 for $s = 1$. Note that if we exclude the communication cost, $s = 0$ is faster, which indicates that the original approach is still the better alternative in serial. The speedup is 246 and 343 for $s = 0$ and $s = 1$, respectively, relative to the single-processor run with $s = 0$ (that the speedup for $s = 0$ has decreased from 250 to 246 is because the convergence

Table 4.4: *Number of iterations to reach tolerance for $s = 0$, with respective computation times, and number of iterations to reach tolerance for $s = 1$, with respective speedups relative to times for $s = 0$. The problem is Test Case 4, and the times are results of applying the hybrid approach with 8 OpenMP threads in each MPI process on 640 cores.*

Tolerance	$s = 0$		$s = 1$	
	Iterations	Time (s)	Iterations	Speedup
10^{-2}	90	0.057	55	1.29
10^{-3}	690	0.44	390	1.40
10^{-4}	6890	4.4	4100	1.33

test has been enabled, and requires some extra communication). In general, the best value of s will depend highly on the communication cost. If the fraction of time used in message passing is large enough, it might also be beneficial to set $s > 1$.

The extra smoothing in Line 2 of Algorithm 5 can be interpreted as iterating over $U^{\complement}$ with fixed boundary conditions determined by the cell-values on U. In the first steps, however, the values on U are far from the solution, and therefore provide a poor approximation to the true boundary conditions. But as we apply more steps, U becomes an increasingly good approximation, and more can be gained from its use of boundary conditions in the iteration in Line 2. Considering this, an effective approach might be to determine s in an adaptive manner, with zero value at the start of the iteration, and increasing it as the cell-values of U becomes increasingly good approximations. This approach has not yet been investigated.

We end this section with a comment on the convergence criteria,

$$e = \max_{ij} \left(|p_{ij}^{k+1} - p^k| \right), \quad i \in F \cap U^{\complement}.$$

It was found that this criteria is reached in nearly half the number of iterations for $s = 1$ relative to $s = 0$. However, when comparing the smoothness τ of the two converged prolongation operator, $s = 1$ turns out to be less smooth. This can be understood by looking at the stop criteria; it ignores the change in cell-values on U. Choosing $s = 1$ means that we apply the smoother to cell-values on $U^{\complement}$ twice as many times as for those on U. Consequently, the error on U will be larger when reaching the tolerance for $s = 1$ relative to $s = 0$. To produce an equally smooth prolongation operator, $s = 1$ therefore requires a lower tolerance. An alternative is of course to change the convergence criteria to also include the cell-values on U.

4.6 Results Summary and Improvement Ideas

The message-passing program has shown to provide a considerable speedup in the iteration procedure for a reasonable range of problems. However, the speedup highly depends on the computational dependency of the basis functions. In particular, this limits the speedup on grids with a large number of connections. Here, the hybrid approach offers an improvement over pure MPI, which is a direct result of decreasing the communication cost. Results presented in Section 4.5.4 suggest that the modified approach provided by Algorithm 5 can reduce the total computational dependence required to reach a given quality of the prolongation operator, and thereby offers a possible approach to reduce the total computation time when the communication cost is large. However, the method requires further testing. In particular, the resulting prolongation operator should be examined further, as it might not provide the same results when applied to solve the global flow-problem.

Several improvements can be made to increase the speedup from the iteration procedure. Probably the most considerable one is to reduce the communication volume to the required minimum, which was noted in Section 4.4.3. This will especially improve the speedup on highly dependent computations, such as Test Case 5. It is likely that improvements can be made readily by providing better division of the computational work. This can be achieved from a revision of the graph discussed in Section 4.4.1 and from investigating alternative partitioning routines. In particular, we believe that incorporating the additional computation cost of union boundary cells into the vertex weights of the graph can improve load balance.

We have analyzed the so-called *strong scalability* for the message-passing program. In strong scaling, the problem size stays fixed as the number of processing cores are increased. It would also be of interest to test the programs *weak scalability*. Here, the workload distributed to each core remains fixed, while the problem size is increased and divided among more cores. Weak scalability is of interest because it measures the efficiency in which adding more computer power can be used to solver larger problem. Obtaining large strong scalability is generally the most challenging of the two, which is why we chose to study it in this thesis. Analysis of the weak scalability is left for future work.

Note that to investigate the parallel scalability, we used a lot more computer power than what is typically prescribed for the problems that were considered. For instance, performing 1000 iterations of Test Case 4 (SPE 10) takes about three minutes on a single core, but not much more than half a second when 1600 of them are used. If you have time to wait four more seconds, you are well off with 48 cores.

As we increase the number of cores, the setup time occupies an increasing amount of the total computation time. Here, no speedup is gained when message-passing is used, but will rather result in an increased setup time. The easiest way of improving the setup time is to increase the use of OpenMP threads in this part of the implementation. However, reducing the cost of distributing the problem data among MPI processes requires a different approach.

When the volume of the problem data grows, the time to read the problem data from file soon becomes the most time-consuming step in a parallel application of the message-passing program. The procedure of reading and writing data is for now performed sequentially by a single process. Future work could use a specialized binary format for reading data in parallel. This will in addition reduce the setup time, since it decreases or eliminates the need to distribute the problem data.

We end this chapter by a discussion of Amdahl's law, which says that the highest theoretical speedup s_p a program can achieve from p processing units is determined from

$$s_p = \frac{1}{(1 - t_p) + \frac{t_p}{p}},$$

where t_p is the fraction of the sequential runtime that can benefit from multiple processors. On the other hand, $(1 - t_p)$ is the fraction which is has to be computed sequentially. Assume now that we are able to provide parallel computation of the whole message-passing program, including the setup procedure, except for reading the problem data from file, which must be done sequentially. Again consider using the program to perform 1000 iterations of Test Case 4. When executed on a single processing unit, the fraction of time used to read the problem data from file is 0.05, meaning that $t_p = 0.95$. Amdahl's law tells us that the best speedup we can hope to achieve is

$$s_p = \frac{1}{0.05 + \frac{0.95}{p}},$$

meaning that no mater how many processing units we use, we will not obtain a speedup higher than 20 for the program as a whole. Further, achieving this speedup assumes no extra overhead resulting from the parallel computation, which of course is not realistic in practice. This highlights our previous point; to obtain a better speedup for the message-passing problem as a whole, the most prominent need of improvement is to allow reading the problem data in parallel.

Chapter 5

Concluding Remarks

This thesis has discussed how to enhance construction of the prolongation operator used by the MsRSB method by exploring two avenues; a change from relaxed Jacobi to Gauss-Seidel type smoothing, and assessing the parallel efficiency of the construction algorithm.

During the research, a modified construction algorithm was found that proved to be more numerically stable and easier to implement compared to the preexisting one. This algorithm has therefore been used in the program presented here, and we recommend its use in future implementations. We demonstrated that a direct change to Gauss-Seidel smoothing in the construction algorithm has a significant drawback which renders it an inadequate alternative to Jacobi. But by learning from its failure, we derived two modified Gauss-Seidel smoothing approaches that have shown considerable promise in numerical tests, which suggests that a 40% reduction in the number of iterations can be achieved.

A program which is able to perform the construction algorithm on distributed-memory systems was presented. The program allows utilizing hybrid combinations of explicit message-passing by MPI functionality with shared memory multithreading by the use of OpenMP. Tests performed on the cluster Vilje showed excellent speedup of the iteration procedure on two-dimensional Cartesian grids. However, as we move to three dimensions and grids with more connections, the speedup gain is lessened. This is a consequence from a larger dependency in the computations, which causes a more rapid growth in communication cost. Here, the hybrid approach significantly outperforms runs that use solely MPI-processes, which is a result from a considerable reduce in communication cost. Even for the most challenging test case, a speedup of 58 was obtained on 160 processing

cores. Taking into account that it still remains a task to reduce the communication volume to its required minimum, we conclude that the prolongation construction algorithm is highly suitable for parallel computation, and will certainly not be a bottleneck in a full parallel application of the MsRSB method. We also presented an approach to decrease the computational dependence of the iteration procedure. The method requires more research, but might very possibly be an effective way to decrease the runtime for parallel applications where a large number of processing nodes are used, and where the computational dependence is high. The current message-passing program uses the originally relaxed Jacobi method as smoother, but a switch to one of the proposed Gauss-Seidel type smoothers can be accomplished from a quick change in the existing code.

We end this thesis by mentioning two recent developments in parallel multiscale solvers. In August of 2016, A. Kozlova et al. from Schlumberger published their work on a parallel implementation of the whole MsRSB method in a commercially available simulator [26]. The implementation is still under development, but results are so far promising. However, they conclude that a high speedup is harder to achieve for complex grids, which is in agreement with results we have presented here. In 2015, A. M. Manea et al. at Standford University published an aricle about their work on a parallel implementation of a multiscale solver for structured grids [35], where a MsFE-based algebraic multiscale method is used [43, 47]. Here, the parallel scalability of the multiscale solver is compared against the state-of-the-art algebraic multigrid solver. They conclude that the multiscaler solver shows good scalability, but memory access time becomes a bottleneck when many cores are used on shared memory machines.

Bibliography

[1] Introducing MEX Files, webpage. https://se.mathworks.com/help/matlab/matlab_external/introducing-mex-files.html. Accessed: 26.09.2016.

[2] METIS webpage. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview. Accessed: 29.08.2016.

[3] MPI, webpage. https://computing.llnl.gov/tutorials/mpi/. Accessed: 29.08.2016.

[4] MRST, webpage. http://www.sintef.no/projectweb/mrst/. Accessed: 10.02.2016.

[5] OpenMP, webpage. http://openmp.org/wp/. Accessed: 29.08.2016.

[6] Vilje, webpage. https://www.hpc.ntnu.no/display/hpc/Vilje. Accessed: 23.08.2016.

[7] J.E. Aarnes. On the use of a mixed multiscale finite element method for greaterflexibility and increased speed or improved accuracy in reservoir simulation. *Multiscale Modeling & Simulation*, 2(3):421–439, 2004.

[8] J.E. Aarnes, V. Kippe, and K.A. Lie. Mixed multiscale finite elements and streamline methods for reservoir simulation of large geomodels. *Advances in Water Resources*, 28(3):257–271, 2005.

[9] J.E. Aarnes, S. Krogstad, and K.A. Lie. A hierarchical multiscale method for two-phase flow based upon mixed finite elements and nonuniform coarse grids. *Multiscale Modeling & Simulation*, 5(2):337–363, 2006.

[10] J.E. Aarnes, S. Krogstad, and K.A. Lie. Multiscale mixed/mimetic methods on corner-point grids. *Computational Geosciences*, 12(3):297–315, 2008.

[11] F.O. Alpak, M. Pal, K.A Lie, et al. A multiscale adaptive local-global method for modeling flow in stratigraphically complex reservoirs. *SPE Journal*, 17(04):1–056, 2012.

[12] Z. Chen and T. Hou. A mixed multiscale finite element method for elliptic problems with oscillating coefficients. *Mathematics of Computation*, 72(242):541–576, 2003.

[13] M.A. Christie, M.J. Blunt, et al. Tenth spe comparative solution project: A comparison of upscaling techniques. In *SPE Reservoir Simulation Symposium*. Society of Petroleum Engineers, 2001.

[14] H. Hajibeygi, G. Bonfigli, M.A. Hesse, and P. Jenny. Iterative multiscale finite-volume method. *Journal of Computational Physics*, 227(19):8604–8621, 2008.

[15] H. Hajibeygi, H.A. Tchelepi, et al. Compositional multiscale finite-volume formulation. *SPE Journal*, 19(02):316–326, 2014.

[16] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *International Symposium on Solving Irregularly Structured Problems in Parallel*, pages 218–225. Springer, 1998.

[17] T.Y. Hou and X.H. Wu. A multiscale finite element method for elliptic problems in composite materials and porous media. *Journal of computational physics*, 134(1):169–189, 1997.

[18] P. Jenny, S.H. Lee, and H.A. Tchelepi. Multi-scale finite-volume method for elliptic problems in subsurface flow simulation. *Journal of Computational Physics*, 187(1):47–67, 2003.

[19] P. Jenny, S.H. Lee, and H.A. Tchelepi. Adaptive multiscale finite-volume method for multiphase flow and transport in porous media. *Multiscale Modeling & Simulation*, 3(1):50–64, 2005.

[20] P. Jenny, S.H. Lee, and H.A. Tchelepi. Adaptive fully implicit multi-scale finite-volume method for multi-phase flow and transport in heterogeneous porous media. *Journal of Computational Physics*, 217(2):627–641, 2006.

[21] P. Jenny and I. Lunati. Modeling complex wells with the multi-scale finite-volume method. *Journal of Computational Physics*, 228(3):687–702, 2009.

[22] F. Johannessen. Constructing the prolongation operator to the multiscale restricted-smoothed basis method on distributed memory machines, 2016.

[23] G. Karypis. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 2013.

[24] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[25] G. Karypis and V. Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[26] A. Kozlova, D. Walsh, S. Chittireddy, Z. Li, J. Natvig, S. Watanabe, and K. Bratvedt. A hybrid approach to parallel multiscale reservoir simulator. In *ECMOR XIV-15th European Conference on the Mathematics of Oil Recovery*, 2016.

[27] S.H. Lee, C. Wolfsteiner, and H.A. Tchelepi. Multiscale finite-volume formulation for multiphase flow in porous media: black oil formulation of compressible, three-phase flow with gravity. *Computational Geosciences*, 12(3):351–366, 2008.

[28] K. A. Lie. An introduction to reservoir simulation using MATLAB: User guide for the Matlab reservoir simulation toolbox (MRST), SINTEF ICT, 2015.

[29] K.A. Lie, O. Møyner, J.R. Natvig, A. Kozlova, K. Bratvedt, S. Watanabe, and Z. Li. Successful application of multiscale methods in a real reservoir simulator environment. In *ECMOR XIV-15th European Conference on the Mathematics of Oil Recovery*, 2016.

[30] I. Lunati and P. Jenny. Multiscale finite-volume method for compressible multiphase flow in porous media. *Journal of Computational Physics*, 216(2):616–636, 2006.

[31] I. Lunati and P. Jenny. A multiscale finite-volume method for three-phase flow influenced by gravity. In *Proceedings of XVI international conference on computational methods in water resources (CMWR XVI), Copenhagen, Denmark*, pages 1–8, 2006.

[32] I. Lunati and P. Jenny. Treating highly anisotropic subsurface flow with the multiscale finite-volume method. *Multiscale Modeling & Simulation*, 6(1):308–318, 2007.

[33] I. Lunati and S.H. Lee. An operator formulation of the multiscale finite-volume method with correction function. *Multiscale modeling & simulation*, 8(1):96–109, 2009.

[34] I. Lunati, M. Tyagi, and S.H. Lee. An iterative multiscale finite volume algorithm converging to the exact solution. *Journal of Computational Physics*, 230(5):1849–1864, 2011.

[35] A.M. Manea, J. Sewall, H.A. Tchelepi, et al. Parallel multiscale linear solver for highly detailed reservoir models. In *SPE Reservoir Simulation Symposium*. Society of Petroleum Engineers, 2015.

[36] O Møyner. Construction of multiscale preconditioners on stratigraphic grids. In *ECMOR XIV-14th European Conference on the Mathematics of Oil Recovery*, 2014.

[37] O. Møyner and K. A. Lie. A multiscale two-point flux-approximation method. *Journal of Computational Physics*, 275:273–293, 2014.

[38] O. Møyner and K. A. Lie. A multiscale restriction-smoothed basis method for high contrast porous media represented on unstructured grids. *Journal of Computational Physics*, 304:46–71, 2016.

[39] O. Møyner and K.A Lie. A multiscale restriction-smoothed basis method for compressible black-oil models. *To be appear in SPE J*, 2016.

[40] O. Møyner, K.A. Lie, et al. The multiscale finite-volume method on stratigraphic grids. *SPE Journal*, 19(05):816–831, 2014.

[41] Y. Saad. *Iterative methods for sparse linear systems*. Siam, 2003.

[42] K. Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1):281–309, 2001.

[43] Y. Wang, H. Hajibeygi, and H.A. Tchelepi. Algebraic multiscale solver for flow in heterogeneous porous media. *Journal of Computational Physics*, 259:284–303, 2014.

[44] Y. Wang, H. Hajibeygi, and H.A. Tchelepi. Monotone multiscale finite volume method. *Computational Geosciences*, 20(3):509–524, 2016.

[45] X.H. Wen and J.J. Gómez-Hernández. Upscaling hydraulic conductivities in heterogeneous media: An overview. *Journal of Hydrology*, 183(1):ix–xxxii, 1996.

[46] H. Zhou, H.A. Tchelepi, et al. Operator-based multiscale method for compressible flow. *SPE Journal*, 13(02):267–273, 2008.

[47] H. Zhou, H.A. Tchelepi, et al. Two-stage algebraic multiscale linear solver for highly heterogeneous reservoir models. *SPE Journal*, 17(02):523–539, 2012.