Enhancing OPM-based Reservoir Simulation via PETSc integration

Jørgen Kvalsvik

Master of Science in Computer Science
Submission date: June 2015
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

GPUs for high-performance computing has been receiving a lot of attention lately. Major numerical libraries are starting to offer GPU-accelerated support for kernels, but how efficient they are when used with real-world applications is largely unknown.

This project includes implementing support for PETSc solvers in the free software reservoir simulator OPM (Open Porous Media), testing and measuring the performance of different algorithms and implementations for CPU and GPU in a real-world application with a real data set.

Abstract

Numerical libraries have different properties and performance depending on the problem and data set at hand, and support different features such as parallelisation, co-processor acceleration, debugging and profiling. Having several libraries available can be beneficial for a complicated and performance sensitive software project. Utilising a new library is non-trivial, as libraries have different design philosophies and semantics, so properly integrating them is important for usability and adoption.

In this thesis, the well-established PETSc library was integrated with the Open Porous Media (OPM), and an example application was adapted to use PETSc for numerical computation. The application performance was measured with several configurations using a real-world data set.

We found that the fastest PETSc configuration was approximately 48% faster than the fastest unmodified configuration, and that PETSc is a viable alternative numerics library for OPM. We did not observe any benefit when performing numerics on the GPU for this particular workload.

The integration described in this thesis is proposed as a draft for a unified interface for linear algebra and numerics libraries for OPM, with the goal of supporting even more third party libraries.

This work was done in collaboration with Statoil.

Sammendrag

Numerikkbiblioteker har forskjellige egenskaper og ytelse avhengig av problemet og datasettet, samt øvrige egenskaper som støtte for parallelisering, co-prosessorerakselerasjon, debugging og profilering. Det er dermed attraktivt å kunne benytte flere bibilotek for kompliserte og ytelsessensitive prosjekter. God integrasjon er viktig for brukervennlighet og adopsjon, ettersom det kan være ikke-trivielt å bruke et nytt bibliotek grunnet forskjellig designfilosofi og semantikk.

I denne oppgaven ble det veletablerte biblioteket PETSc integrert i Open Porous Media (OPM), og en eksempelapplikasjon ble omskrevet til å benytte PETSc for numeriske beregninger. Applikasjonsytelsen ble målt med et representativt datasett og flere konfigurasjoner.

Målingene viser at den raskeste PETSc-konfigurasjonen var omtrent 48% raskere enn den raskeste umodifiserte versjonen, og at PETSc dermed er et egnet alternativ for numeriske beregninger i OPM. Ingen målinger viste forbedring for denne oppgaven ved å gjøre numerikken på en grafikkprosessor.

Integrasjonen beskrevet i denne oppgaven er et foreslått utkast for et unifisert grensesnitt for lineær algebra- og numerikkbibilotek i OPM, hvis mål er å støtte enda flere tredjepartsbibilotek.

Dette arbeidet ble utført i samarbeid med Statoil.

Acknowledgements

I would like to thank Dr. Anne C. Elster for being my supervisor, introducing me to PETSc and organising the project with Statoil.

I would also like to thank Dr. Alf B. Rustad at Statoil Research for the opportunity to work with him, the OPM community, and for all the feedback and technical assistance I received. A special thanks to him and the Statoil Research Centre at Rotvoll for providing a data set for me to experiment with.

I would like to thank my fellow students at the HPC-lab for their general support and assistance, with a special mention to Imre Kerr and Bjør Åge Tungesvik for all the help and the interesting discussions. Many thanks to Matthew Greening for helping me proof read.

Finally, I would like to thank NTNU, AMD and NVIDIA, and Statoil for their support of the HPC-lab, which allowed me to work on this thesis.

Trondheim, June 10, 2015

Jørgen Kvalsvik

Contents

List of Figures xiii

List of Tables xv

List of Listings xix

List of Acronyms xxi

1 Introduction 1

1.1 Project goals 1

1.2 Outline 2

2 Background 3

2.1 Numerical methods 3

2.1.1 Linear algebra 3

2.1.2 System of linear equations 10

2.1.3 Numerical solvers 11

2.1.4 Direct methods 11

2.1.5 Iterative methods 12

2.1.6 Conjugent Gradient 13

2.1.7 Generalised Minimal Residual 14

2.1.8 Algebraic Multigrid Method 16

2.2 High-performance computing 16

2.2.1 Amdahl's Law 17

2.2.2 Gustafson's Law 17

2.2.3 Amdahl's Law and Gustafson's Law for multicore systems 18

2.2.4 MPI 19

2.3 Numerical software 21
2.3.1 BLAS and LAPACK 21
2.3.2 DUNE 22
2.3.3 Hypre 22
2.3.4 Fluent 22
2.4 PETSc 22
2.5 C++ 25
2.5.1 Complex objects 25
2.5.2 Constructor and destructor 26
2.5.3 The `this` pointer 27
2.5.4 Inheritance 28
2.5.5 Overloading 29
2.5.6 Template programming 30
2.5.7 Move semantics 31
2.5.8 Resource Acquisition is Initialisation 32
2.6 Further reading 32

3 Upscaling **35**
3.1 Reservoir engineering, permeability and Darcy's Law 35
3.2 Grid 37
3.3 Basics of upscaling 39
3.4 Program design 41
3.5 Scalability 43

4 Integrating PETSc with OPM **47**
4.1 On development 47
4.2 Motivation for a library 48
4.3 Types and containers 49
4.4 High-level interface 50
4.5 Common and mixins 51
4.6 Vector 53
4.7 Matrix 55
4.8 Solver 57
4.9 Porting *upscale_relperm* to use PETSc 60
4.10 Bugs 61

5 Results and Measurements **65**

5.1 Configuration 65
5.2 Differences 66
5.3 Computation time 69

6 Conclusions and Future Work **75**
6.1 Contributions 75
6.2 Conclusions 76
6.3 Future Work 76

References **79**

Appendices

A Code snippets **83**

B Programs **99**

C Tables & figures **105**

List of Figures

2.1 Matrices and vectors . . . 4
2.2 Block matrix . . . 6
2.4 Linear system and augmented matrix . . . 10
2.5 $Ax = b$ . . . 10
2.6 A generalised $Ax = b$ problem . . . 11
2.7 Speedup: Amdahl's Law . . . 19
2.8 Speedup: Gustafson's Law . . . 20
2.9 MPI communicators . . . 21
2.10 PETSc class relationship . . . 24

3.1 Porous rock . . . 36
3.2 The Norne reservoir . . . 37
3.3 Simple 3D Cartesian grid . . . 38
3.4 3D hexahedral grid . . . 38
3.5 Upscaling . . . 39
3.6 Upscaled relative permeability . . . 40
3.7 Diff satpoints . . . 45
3.8 Upscaler-relperm parallel design . . . 46

5.1 *upscale_relperm* output . . . 67
5.2 PETSc timings . . . 72
5.3 DUNE timings . . . 72

C.1 PETSc timings . . . 106
C.2 DUNE timings . . . 106
C.3 *upscale_relperm* output . . . 107

List of Tables

2.1 Linear algebra operations 6
2.2 Axioms for vector spaces 9
2.3 Achievable speedup according to Amdahl's Law 18
2.4 PETSc components 23

3.1 Execution time: *upscale_relperm* 42

4.1 Workstation 48
4.2 Containers in DUNE 2.3 49
4.3 Interface: `Vector` 54
4.4 Interface: `Matrix` 56
4.5 Interface: `Builder` 58
4.6 Interface: `Solver` 59

5.1 Testing system 66
5.2 PETSc output difference I 68
5.3 PETSc output difference II 69
5.4 PETSc timings 71
5.5 DUNE timings 73
5.6 Single process timings 73

C.1 Workstation 105
C.2 Testing system 108

List of Algorithms

2.1 Conjugent Gradient 14
2.2 *k*-th Arnoldi iteration 15
2.3 Generalised Minimal Residual 15
2.4 General AMG 16
3.1 OPM *upscale* design 35
3.2 OPM *upscale_relperm* design 41

List of Listings

2.1 Class member accessability 26
2.2 Inheritance 28
2.3 Private inheritance 30
2.4 Template programming 31
2.5 move without copy 32
2.6 RAII managing a heap-allocated array 33
4.1 Ported *IncompFlowSolverHybrid* matrix setup 62
4.2 PETSc-ViennaCL patch 63
4.3 *upscale_relperm* patch 64
A.1 `uptr` 83
A.2 Vector as a raw PETSc handle 83
A.3 Managed resource from PETSc handle 84
A.4 Deleter 84
A.5 Deleter with decltype 84
A.6 Value oriented `dot` 84
A.7 Variadic arguments to `solve` 84
A.8 Solving with DUNE 85
A.9 Solving with PETSc 86
A.10 Solving a linear system with OPM-PETSc 87
A.11 Solving a linear system with PETSc. 87
A.12 Removed DUNE code 88
B.1 matrix-diff.hs 100
B.2 make-result-tables.sh 101
B.3 opm-bench.sh I 102
B.4 opm-bench.sh II 103
B.5 extract-timing.pl 104

List of Acronyms

AMG Algebraic Multigrid Method.

BLAS Basic Linear Algebra Subprograms.

CG Conjugate Gradient method.

DUNE Distributed and Unified Numerics Environment.

FFI Foreign Function Interface.

GMRES Generalised Minimal Residual Method.

ILU Incomplete Lower Upper factorisation.

ISTL Iterative Template Solver Library.

LAPACK Linear Algebra Package.

MPI Message Passing Interface.

PETSc Portable, Extensible Toolkit for Scientific Computation.

POD Plain Old Data.

RAII Resource Acquisition Is Initialisation.

SOR Successive over-relaxation.

SSOR Symmetric SOR.

Chapter 1

Introduction

Solving systems of linear equations is a core component in several aspects of petroleum reservoir simulation, and simulator performance largely depends on the algorithm chosen and the speed of the numerical kernels.

The Open Porous Media project is a collection of free libraries and programs developed by several industrial partners and research institutions, which largely depends on DUNE for numerical computation. A component of the OPM project is the *upscaler*, a component that coarsens flow properties of the reservoir in order to reduce the computational load in subsequent phases of simulation.

In this thesis, the numerical component DUNE has been replaced by the alternative component PETSc, developed by the Argonne National Laboratory, which has been integrated into the project, and the test application *upscale_relperm* has been ported to utilise this new backend.

1.1 Project goals

The main goal of the project is to provide an alternative to DUNE and UMFPACK for numerical computation in OPM. This is motivated by supporting more algorithms, more matrix representation formats, more third party packages, and more debugging and profiling tools. PETSc has for many years been one of the leading projects for scientific computations and is well established in the community. Additionally, improvements and additions to

the PETSc project will directly benefit OPM, and contributions from the OPM project will also benefit other projects relying on PETSc.

With support for more algorithms comes the potential for improved performance. While optimisation is not a goal for this thesis in itself, it is a potential benefit for some inputs or classes of problems.

1.2 Outline

The rest of the thesis is structured as follows:

Chapter 2: Relevant background material and concepts related to this thesis, including a brief introduction to linear algebra, numerical methods and the C++ programming language.

Chapter 3: The upscaling process and the OPM *upscale* family of programs are introduced, as well as basics of petroleum engineering and grids.

Chapter 4: The main work of this thesis is described, including decisions and considerations made integrating PETSc into the OPM project, and the final design of the necessary components to effectively use PETSc in OPM.

Chapter 5: A presentation of experimental results and measurements of various DUNE and PETSc driven runs of *upscale_relperm*.

Chapter 6: The contributions and conclusion of the thesis are presented, as well as topics for future work.

Appendix A: Relevant code snippets from the PETSc integration.

Appendix B: Various helper and utility programs

Appendix C: A collection of tables and figures for ease-of-access.

Chapter 2
Background

This Chapter introduces the basic concepts needed to understand the material of this thesis. A brief overview of linear algebra and numerical methods is presented in Section 2.1. Some important concepts for high-performance computing are presented in Section 2.2. Section 2.3 introduces and describes some key pieces of numerical software either used in or considered for this thesis. The chosen library and some key aspects of it are described in Section 2.4. Finally, Section 2.5 describes the relevant features and idioms that form the tools for this thesis in detail.

Some parts of this chapter are built on the background chapter of my project in the autumn of 2014.

2.1 Numerical methods

2.1.1 Linear algebra

Linear algebra is a branch of mathematics that mainly concerns vector spaces and linear mappings between such spaces, and forms a foundation for numerical methods. This section gives a brief introduction to the key concepts and terms required to understand the algorithms discussed and the motivations for this thesis.

A linear equation is an equation where each term is either constant or the prodct of a constant and a single variable. In general, a linear equation is an

$$\begin{bmatrix} 0.5 & 12 & 4 & -5 \\ 3 & 0.1 & 1 & 2 \end{bmatrix} \qquad \begin{bmatrix} 2 & \pi \\ x & \frac{1}{2} \\ 4 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2x & e^{-x} \\ 1 & e^{3x} \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 2 & 3 & 5 & 8 & 13 \end{bmatrix}$$

Figure 2.1: Three matrices and a vector. Dimensions 2×4, 3×2, 2×2 and 1×7 respectively.

equation of the form of Equation 2.1.

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = b \tag{2.1}$$

The main tools of linear algebra are matrices and vectors, see Figure 2.1. A matrix is a rectangular array of numbers or functions, called **entries**, and a vector is a one-dimensional sequence of entries. A matrix is denoted with a capital letter, A, with its element denoted with an ij subscript. a_{ij} denotes an element at the ith row and jth column of the matrix A. Vectors are denoted with a lowercase letter. A matrix with m rows and n columns, or m-by-n, is typically written as $m \times n$, where $m \times n$ are called the **dimensions** of the matrix. A matrix with $m = 1$ is called a row vector, $n = 1$ is called a column vector and $m = n$ is called a square matrix. A matrix where every element is either zero or positive is called a **positive matrix** or **non-negative matrix**, and analoguously, a matrix that has only zero or negative elements is called a **negative matrix**. A matrix where every element is zero is called a **zero matrix** or a **null matrix**.

There are several fundamental arithmetic operations on matrices and vectors. For easier reference, the operations, their notations, their names and their result is presented in Table 2.1.

Both vectors and matrices can be multiplied with a constant, *scaled*, which is an element-wise multiplication of the constant and the entries of the structure. There are two binary operations for pairs of vectors, the **cross**

product $u \times v$ and the **dot product** $u \cdot v$. The cross product of two **linearly independent** vectors is a vector peripendicular on the plane they create. The dot product is defined algebraicly as $u \cdot v = \sum_{i=1}^{n} u_i v_i$. Geometrically this is defined through the cosine of the angle between the vectors.

Definition 2.1. **Linear dependence** A set of vectors V is **linearly dependent** if one of the vectors of the set can be defined as a combination of the other vectors.

Pairs of matrices can be **added** and **subtracted** by performing the binary operation element-wise, i.e. $A + B = C \Rightarrow c_{ij} = a_{ij} + b_{ij}$. Since vectors can be considered matrices with $n = 1$, this also applies to vectors. Addition and substraction will produce a matrix for matrices and a vector for vectors.

Matrix multiplication is a binary operation that produces a matrix. There are several different definitions for what constitues matrix multiplication, but the most common one, which is the one used in this thesis, is $AB = C \Rightarrow c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$.

The unit vector e is a vector of length 1, the simplest being the vector e_i where all entries are zero except for the ith, which is 1. The general notion of **norm** is a function that assigns a *length* to a vector. In a eucledian space, the norm is $||x|| = \sqrt{x_1^2 + \cdots + x_n^2}$.

An identity matrix is the $n \times n$ matrix I_n where the **main diagonal** entries are 1 and all other entries are zero. The ith column an identity matrix is the unit vector e_i. The identity matrix has the property that $AI_n = I_n A = A$. Finally, **matrix-vector multiplication**, which can be considered a special case of matrix-matrix multiplication, where the second operand is a column vector, results in a new a vector. Matrix-vector multiplication is an important part of many scientific and engineering applications, and a lot of research goes into optimising these computations. An example of this is autotuning frameworks and knowledge databases [8].

A **block matrix** is a matrix that can considered broken down into **blocks** or **submatrices**. Submatrix entries are denoted with capital letters, and zero matrices are denoted with 0.

Table 2.1: The operations of linear algebra and their results. Lower-case symbols are vectors, upper-case symbols are matrices and Greek symbols are scalars.

	Symbol	Name	Expression	Result
Vector				
	α	Scale	$\alpha u = w$	Vector
	$+$	Addition	$u + v = w$	Vector
	-	Subtraction	$u - v = w$	Vector
	$\times$	Cross product	$u \times v = w$	Vector
	$\cdot$	Dot product	$u \cdot v = \beta$	Scalar
Matrix				
	α	Scale	$\alpha A = C$	Matrix
	$+$	Addition	$A + B = C$	Matrix
	-	Subtraction	$A - B = C$	Matrix
	$\cdot$	Multiplication	$AB = C$	Matrix
Matrix-vector				
	$\cdot$	Multiplication	$Av = w$	Vector

$$M = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} B = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} C = \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix} D = \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$$

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Figure 2.2: Block matrix notation.

Pairs of vectors can be **orthogonal** and **orthonormal**.

Definition 2.2. **Orthogonal** Two vectors u and v are **orthogonal** to eachother if they are perpendicular. If two vectors are peripendicular, $u \perp v$, then their dot product is zero.

$$u \cdot v = 0 \tag{2.2}$$

Definition 2.3. **Orthonormal** If u and v are orthogonal and unit vectors, they are considered **orthonormal**.

A **diagonal** of a matrix A is a collection of entries a_{ij} laid out diagonally in A. The main diagonal has already been mentioned, and can help create the **transpose**. Another common diagonal is the **antidiagonal**.

Definition 2.4. **Main diagonal** The entries a_{ij} where $i = j$ form the **main diagonal** of the matrix A. The following three matrices have their main diagonal indicated with 1s in red.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

Definition 2.5. **Antidiagonal** The **antidiagonal** of a square matrix A of dimension n are the entries a_{ij} where $i + j = n + 1$.

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

Definition 2.6. **Transpose** The tranpose of a matrix A, denoted A^T, is the matrix created by any of the equivalent actions:

- Reflect A over its main diagonal
- Make the rows of A the columns of A^T

- Make the columns of A the rows of A^T

This yields the matrix A^T so that the entries of A^T can be described by Equation 2.3.

$$a_{ij}^T = a_{ji} \tag{2.3}$$

From the transpose we can describe the **symmetric** matrix, which is a prerequisite for some numerical algorithms. Since the symmetric matrix A is symmetric with respect to its main diagonal, the entries $a_{ij} = a_{ji}$ for all indices i and j.

Definition 2.7. Symmetric matrix The matrix A is **symmetric** if it is identical to its transpose A^T. Formally:

$$A = A^T \tag{2.4}$$

The following concepts, **vector space**, **basis** and **span**, are of general interest in linear algebra, and they form an important foundation for some applications. Additionally, **conjugates** and **positive-definite matrices** are requirements for some algorithms.

Definition 2.8. Vector space A nonempty set of vectors V, such that with any two vectors $a, b \in V$ all their linear combinations are elements of V and satisfy the axioms in Table 2.2, is a **vector space**.

Definition 2.9. Basis A linearly independent set of vectors in V consisting of a maximum possible number n of vectors (in V) is called a **basis**.

$$\begin{gathered}
v_{(1)} = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix} \\
v_{(2)} = \begin{bmatrix} 0 & 1 & \cdots & 0 \end{bmatrix} \\
\vdots \\
v_{(n)} = \begin{bmatrix} 0 & 0 & \cdots & 1 \end{bmatrix}
\end{gathered}$$

Table 2.2: Axioms for vector spaces. The letters in **bold** are vectors, capital letters are vector spaces, and rest are scalars.

Axiom	Consequence
Associativity of addition	$\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$
Commutativity of addition	$\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$
Identity of addition	$\exists \mathbf{0} \in V$ such that $\forall \mathbf{v} \in V : \mathbf{v} + \mathbf{0} = \mathbf{v}$
Inverse of addition	$\forall \mathbf{v} \in V : \exists -\mathbf{v} \in V$ such that $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$
Compatibility	$a(b\mathbf{v}) = (ab)\mathbf{v}$
Identity of addition	$\exists \mathbf{1} \in V : \mathbf{1v} = \mathbf{v}$
Distributivity, vector addition	$a(\mathbf{u} + \mathbf{v}) = a\mathbf{u} + a\mathbf{v}$
Distributivity, field addition	$(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$

Definition 2.10. Span The set of all linear combinations of the vectors $v_{(1)}, \cdots, v_{(n)}$ is called the **span** of these vectors. The span is a vector space.

Definition 2.11. Conjugation A function where some operation is applied, followed by another operation and the reversed initial operation is called a **conjungation**, and is a component of the abstract algebra concept of **inner automorphism**. The right-hand side in Equation 2.5 is a general conjugation for some function $f : G \to G$ defined for all x in the group G, where a is an arbitrary reversible function.

$$f(x) = a^{-1}xa \tag{2.5}$$

The non-zero vectors u and v are conjugate with respect to A if $u^T Av = 0$.

Some square matrices can be characterised by other structural proprties, which can be prequisites for certain numerical algorithms. These properties include **positive-definitness** and **invertability**.

Definition 2.12. Positive-definite matrix A $n \times n$ real matrix A is **positive-definite** if $v^T Av$ is positive for every non-zero column vector v of real numbers and length n.

$$\begin{array}{rcrcr} 2x_1 & + & 5x_2 & = & 2 \\ -4x_1 & + & 3x_2 & = & -30 \end{array} \qquad \left[\begin{array}{cc|c} 2 & 5 & 2 \\ -4 & 3 & -30 \end{array}\right]$$

Figure 2.4: A simple linear system and its corresponding augmented matrix.

$$\begin{bmatrix} 2 & 5 \\ -4 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ -30 \end{bmatrix}$$

Figure 2.5: Figure 2.4 written as $Ax = b$.

Definition 2.13. Invertible matrix The $n \times n$ matrix A is **invertible** if there exists an $n \times n$ matrix B so that Equation 2.6 holds.

$$AB = BA = I_n \tag{2.6}$$

2.1.2 System of linear equations

A system of linear equations, briefly a **linear system**, is a collection of equations with the same set of variables. These systems can be solved (if they have solutions) with the use of matrices, see Figure 2.4.

Many engineering problems are modelled mathematically as linear systems, so solving these systems are important kernels of many applications. We often rely on numerical methods (section 2.1.3) to solve (large) linear systems. Linear systems are often presented as Equation 2.7.

$$Ax = b \tag{2.7}$$

A is known as the **coefficient matrix**, x is known as the **solution vector**, and b is known as the **constant vector**. Figure 2.5 shows the system of linear equations presented in Figure 2.4 written as $Ax = b$.

This can be generalised to matrices of arbitrary sizes, as shown in Figure 2.6. Real world problems often have millions or even billions of variables, and the matrices required to model these are as large.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Figure 2.6: A generalised $Ax = b$ problem

2.1.3 Numerical solvers

Sometimes it is intractable, or even impossible, to find a solution, but possible to *approximate* it. Numerical analysis is solving problems in terms of numbers, or numerically, typically using a calculator or computer. Such problems arise from real-world applications in engineering, physics, meteorology, finance, medicine, throughout all the sciences. Particularly relevant for this project are the numerical problems that stem from reservoir - and flow simulations. There are two general techniques for solving these systems numerically; using **direct methods** (2.1.4) or using **iterative methods** (2.1.5).

There are many algorithms and software libraries for numerical analysis available, both free and proprietary. Some of these include BLAS, LAPACK, and DUNE, Trilinos, and PETSc. BLAS, LAPACK and DUNE are described in Sections 2.3.1 and 2.3.2, and PETSc is described in more detail in Section 2.4.

2.1.4 Direct methods

A direct numerical method is an algorithm in which the number of operations can be specified in advance. Direct methods would give exact answers assuming they were performed with infinite-precision arithmetic [7]. In practice, however, we do not use infinite-precision representation of numbers, and every step can introduce new rounding errors, resulting in an accumulation that cannot be corrected. Real-world applications also often has other sources of imprecision, such as slightly imperfect measurement data (e.g. sensor data), contributing further to errors. Additionally, large systems are often very time consuming, if even possible, to solve.

2.1.5 Iterative methods

In contrast to the direct methods, iterative methods, or indirect methods, start from an approximation to the true solution, or an initial guess, acquire better and better approximations from repeating the algorithm as many times as required for achieving the desired accuracy. The error of the current solution is called the **residual**. Increasing the accuracy can be seen as the dual problem of minimising the residual.

Definition 2.14. **Residual** Given a system $Ax = b$, the **residual** r of x is defined by

$$r = b - Ax \tag{2.8}$$

Iterative methods are applied if the **convergence** is fairly rapid, which can be determined with either convergence analysis or a heuristic. They are also typically much faster than direct methods, and enables us to solve systems much larger than available computing power would be able to do with a direct method. There are two main classes of iterative methods; **stationary iterative methods** and **Krylov subspace methods**.

Stationary iterative methods

Stationary iterative methods can in their simplest form be expressed as in Equation 2.9.

$$x^k = Bx^{k-1} + c \tag{2.9}$$

Notice that neither B nor c are dependent on the iteration count k; this gives the class of methods its name. B is an **operator**, a mapping between two vector spaces. The four main stationary iterative methods are Jacobi, Gauss-Seidel, Successive over-relaxation (SOR) [36], and Symmetric SOR (SSOR).

Definition 2.15. **Operator** Let X and Y be two sets. A correspondence or rule which uniquely assigns an element $A(x) \in Y$ to every element x of a subset $D \subset X$ is an operator from A from X into Y.

$$A : D \to Y, \quad \text{where } D \subset X$$

Krylov subspace methods

Krylov methods work by forming a Krylov subspace multiplied with the initial residual then minimise the residual over the subspace.

Definition 2.16. Krylov subspace The **Krylov subspace** $\mathcal{K}_t$ is the subspace that is spanned by $K_t = \begin{bmatrix} b & Ab & A^2b & \cdots & A^{t-1}b \end{bmatrix}$, that is

$$\mathcal{K}_t(A, b) = span\{b, Ab, A^2b, \ldots, A^{t-1}b\} \tag{2.10}$$

Krylov subspace methods include CG and GMRES, discussed in Sections 2.1.6 and 2.1.7.

2.1.6 Conjugent Gradient

The **Conjugate Gradient method (CG)**, developed mainly by Hestenes and Stiefel [12], is a Krylov subspace method for symmeteric and positive-definite matrices.

The goal is to solve Equation 2.7. CG uses a metric function, the quadratic function in Equation 2.11, to decide if it should keep iterating. The unique minimiser to this function is the solution to $Ax = b$.

$$f(x) = \frac{1}{2}x^T Ax - x^T b \tag{2.11}$$

The residual at each step k is, as in Equation 2.8:

$$r_k = b - Ax_k \tag{2.12}$$

Which gives us the full conjugent gradient method in Algorithm 2.1. The dominating operations are the matrix-vector products, which in the general case require $O(m)$ operations, where m are the number of non-zero entries in the matrix. When A is sparse, and $m \in O(n)$. Convergence also depends on the condition number κ, a measure on how sensitive the function is to small changes in its input. CG has a time complexity of $O(n\sqrt{\kappa})$ [25].

Algorithm 2.1 Conjugent Gradient

Input: The real, symmetric, positive-definite coefficient matrix A
Input: The constant vector $\mathbf{b}$
Output: The solution vector $\mathbf{x}$

1: $\mathbf{r}_0 \leftarrow \mathbf{b} - A\mathbf{x}_0$
2: $\mathbf{p}_0 \leftarrow \mathbf{r}_0$
3: $k \leftarrow 0$
4: **loop**
5: $\quad \alpha_k \leftarrow \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T A \mathbf{p}_k}$
6: $\quad \mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$
7: $\quad \mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha_k A \mathbf{p}_k$
8: **if** $\mathbf{r}_{k+1}$ is sufficiently small **then**
9: **return** $\mathbf{x}_{k+1}$
10: **end if**
11: $\quad \beta_k \leftarrow \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$
12: $\quad \mathbf{p}_{k+1} \leftarrow \mathbf{r}_k + \beta_k \mathbf{p} k$
13: $\quad k \leftarrow k + 1$
14: **end loop**

2.1.7 Generalised Minimal Residual

The **Generalised Minimal Residual Method (GMRES)** was developed by Saad and Schultz [24] and, like CG, is a Krylov subspace method. GMRES assumes the $m \times m$ matrix A is invertible, and relies on the Arnoldi iteration [3] of Algorithm 2.2 for the generation of a set of orthonormal vectors $q_2, q_3, \ldots, q_n$ that ensures linear independence, which cannot be guaranteed by the vectors $b, Ab, A^2b, \ldots, A^{t-1}b$. q_1 is an arbitrary vector with norm 1.

The vectors $q_1, q_2, \ldots, q_n$ forms an $m \times n$ matrix Q_n. Since the vectors form a basis for the Krylov subspace $\mathcal{K}$, the vector x_n can be expressed as in Equation 2.13, where y_n is some approperiate vector.

$$x_n = Q_n y_n \tag{2.13}$$

Algorithm 2.2 k-th Arnoldi iteration

Input: The coefficient matrix A
Input: Q_{k-1}, H_{k-1}
Output: The matrices Q_k, H_k

1: $q_k \leftarrow Aq_{k-1}$
2: **for** $j \leftarrow 1, 2, \ldots, k-1$ **do**
3: $\quad h_{j,k-1} \leftarrow q_j q_k$
4: $\quad q_k \leftarrow q_k - h_{j,k} - q_j$
5: **end for**
6: $h_{k,k-1} \leftarrow ||q_k||$
7: $q_k \leftarrow \frac{q_k}{h_{k,k-1}}$

The $h_{i,j}$ variables of the Arnoldi iterations are organised in the matrix H_n, where $AQ_n = Q_{n+1}H_n$. By the orthogonality of the columns in Q_n, we have

$$||Ax_n - b|| = ||H_n y_n - \beta e_1|| \tag{2.14}$$

where vector $e_1 = (1, 0, 0, \ldots, 0)^T$ and the default initial residual $\beta = ||b - Ax_0||$. We can compute x_n by minimising the residual $r_n = H_n y_n - \beta e_1$, which gives the method its name. The full procedure is shown in Algorithm 2.3. The matrix-vector product must be computed at every iteration, in addition to $O(nm)$ floating point operations, but the algorithm will converge in $O(m)$ iterations, although the solution is often a good approximation after fewer iterations.

Algorithm 2.3 Generalised Minimal Residual

Input: The coefficient matrix A
Input: The constant vector b
Output: The solution vector x

1: $\beta \leftarrow ||b - Ax_0||$
2: **repeat**
3: $\quad Q_n, H_n \leftarrow$ ARNOLDI(A, Q_{n-1}, H_{n-1})
4: $\quad r_n, y_n \leftarrow$ MINIMISE$(r_n = H_n y_n - \beta e_1)$
5: $\quad x_n \leftarrow Q_n y_n$
6: **until** r_n is sufficiently small

2.1.8 Algebraic Multigrid Method

Algebraic Multigrid Methods (AMGs) [9] are a *family* of multigrid methods that solve a hierarchy of discretisations. They are named **algebraic** because only depend on the *coefficients* of the matrix. Multigrid methods are *optimal* because they can solve systems with N unknowns doing $O(N)$ work, and parts of the work can be efficiently distributed and performed in parallel [35].

Multigrid methods employs two processes, **smoothing** and **course-grid correction**. Smoothing is applying a smoother, typically a simpler iterative method like Incomplete Lower Upper factorisation (ILU) or Gauss-Seidel, and course-grid correction is solving a course-grid system of equations and transfer the result back into the finer grid through interpolation. Alternative names are **relaxation** and **prolongation** for the smoothing and the interpolation, respectively. Falgout [9] is a nice introduction to AMGs, and Algorithm 2.4 is the generalised two-grid method described in the paper.

Algorithm 2.4 Generalised two-grid AMG

Input: The real coefficient matrix A
Input: The constant vector $\mathbf{b}$
Input: the course-to-fine grid prolongation mapping P
Output: The solution vector $\mathbf{x}$
1: Do v_1 smoothing steps on $A\mathbf{x} = \mathbf{b}$
2: $\mathbf{r} = \mathbf{b} - A\mathbf{x} = A\mathbf{b}$
3: Solve $A_c\mathbf{b}_c = P^T\mathbf{r}$
4: Correct $\mathbf{x} \leftarrow \mathbf{x} + P\mathbf{b}_c$
5: Do v_2 smoothing steps on $A\mathbf{x} = \mathbf{b}$

2.2 High-performance computing

This section briefly discusses they key concepts of scalable - and high performance computing. Since it is not an integral part of this thesis, but is an important aspect of OPM and reservoir simulation in general, it is included. For a thorough description of the modeling of high performance and heterogeneous systems, see Meyer's doctoral thesis [20].

2.2.1 Amdahl's Law

Amdahl's Law is a model that describes the maximum achievable improvement to a solution when only a part of the solution is improved. Amdahl's observation [2] was later used to derive the following formula.

Definition 2.17. Amdahl's Law

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}} \tag{2.15}$$

where $r_s + r_p = 1$, r_p is the sequential part of a program, r_p is the parallelisable part of a program and n are the number of processors.

Briefly, the consequence of Amdahl's Law is that achievable improvement from parallelisation is bounded by the sequential portion, which quickly dominates the execution time. This is known as **strong scaling**, i.e. *how quickly can I solve this problem.*

Put into numbers, consider a problem where exactly 50% of the solution can be parallelised. Putting these numbers into Equation 2.15 we get:

$$\frac{1}{0.5 + \frac{0.5}{n}}$$

The equation is similar for a parallel portion of 95%. Solving this for $n = 1, 4, 8, 32, 64, 128, 256$ we get the results in Table 2.3. Regardless of how many processors we add we cannot achieve more than 2x and 20x speedup respectively. Figure 2.7 plots Amdahl's Law for different values, and demonstrates that improvements to parallelism has an upper bound.

2.2.2 Gustafson's Law

Gustafson's Law provides a counterpoint to the pessimistic view of Amdahl's Law by changing the assumption that the *data set* is of constant size. The law was first described by Gustafson and Barsis [11].

Definition 2.18. Gustafson's Law

$$Speedup = n - \alpha \cdot (n - 1) \tag{2.16}$$

Table 2.3: Achievable speedup according to Amdahl's Law

n	50%	95%
1	1.0	1.0
4	1.6	3.48
8	1.78	5.92
32	1.93	12.54
64	1.97	15.42
128	1.98	17.42
256	1.99	18.62

where

n is the number of processors

α is the non-parallelisable portion (p_s in Amdahl's Law).

This observation gives a new dimension to scaling, which is sometimes referred to as **weak scaling**, and briefly turns the question of improvement into *how big a problem can we solve given a set amount of time?*. According to Gustafson, improvement through parallelisation is possible and feasible, given that the problem is scaled accordingly. Figure 2.8 plots the scaling predicted by Gustafson's Law, analoguous to Figure 2.7

2.2.3 Amdahl's Law and Gustafson's Law for multicore systems

Recent research discuss the validity of both Amdahl's Law and Gustafson's Law in modern computing. Hill and Marty [13] suggest that Amdahl's Law also applies for multicore chips. However, Sun and Chen [32] claim this is just a corollary to Amdahl's Law, and propose a memory bounded performance model, and points out that multicore architectures tends to have more (fast) memory with more cores, and claim that by scaling the problem size according

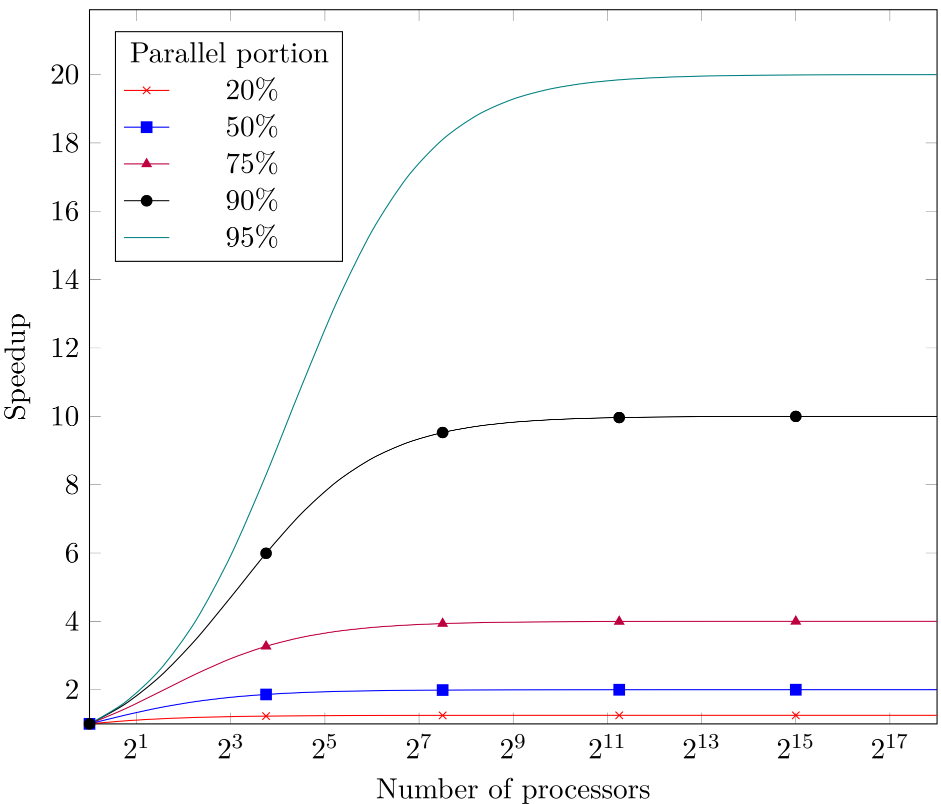

Figure 2.7: Achievable speedup and their bounds for different parallel portions according to Amdahl's Law. The x-axis scale is logarithmic.

to the number of cores we can achieve scaling, in accordance with Gustafson's Law.

2.2.4 MPI

Message Passing Interface (MPI) [18] is a message passing library *specification*, with multiple available implementations, including the popular free[1] MPICH and OpenMPI. It is supported by virtually all high-performance computing systems, and allows application developers to write performant and portable code. MPI-1 was designed for C and Fortran77. MPI-2 [19] extends on this by adding bindings for Fortran90 and C++.

[1] Free as in free licence, not just cost-free.

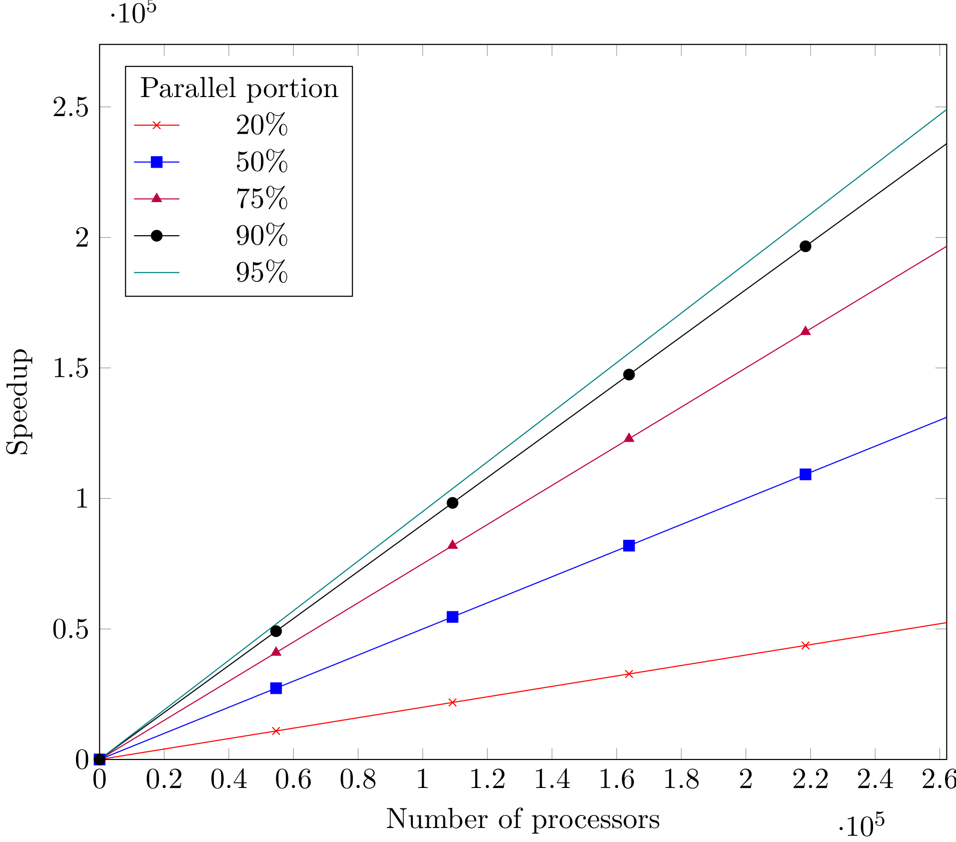

Figure 2.8: Achievable speedup and their bounds for different parallel portions according to Gustafson's Law.

MPI is designed around the concept of a **communicator**, an object that acts as a post office for subsets of the **processes** involved in the full system. This is the only way for processes to communicate - no direct memory sharing is possible and all messages between processes must be sent via a communicator that both processes are a part of. This means all communication and parallelisation is *explicit*. Any MPI process can be a part of arbitrary many communicators. An MPI process is an execution of a full program, i.e. it has a one-to-one mapping with say a unix process, but the processes do not require to be run on the same computer in order to communicate. These are important concepts for PETSc and other libraries that rely on MPI for

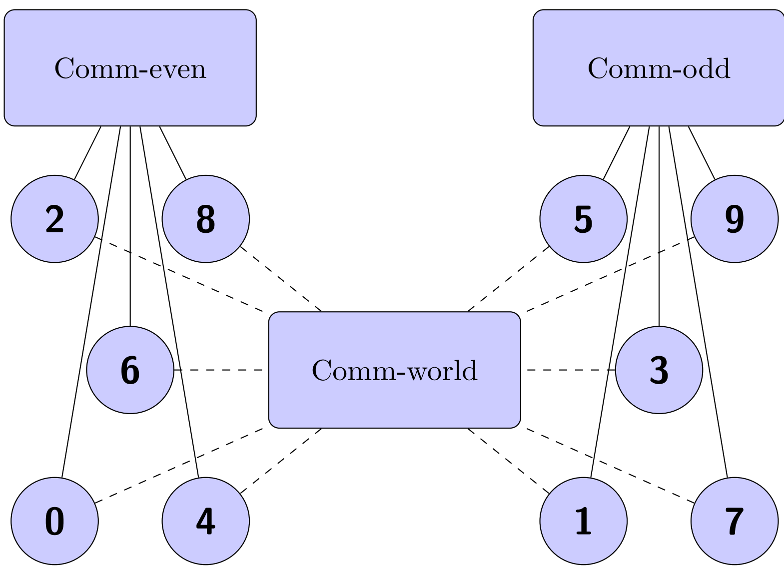

Figure 2.9: An MPI application with three communicators

communication, but it is often sufficient for programs to only use a global communicator and a single computer.

2.3 Numerical software

This section briefly introduces some well established numerical software packages that have been studied, considered or used for this thesis.

2.3.1 BLAS and LAPACK

Basic Linear Algebra Subprograms (BLAS) is a specification originating from a Fortran library [16], with bindings to both C and Fortran, and is the de facto interface for linear algebra libraries. Many vendors offer their own BLAS-compatible library, such as ACML by AMD and MKL by Intel.

Linear Algebra Package (LAPACK) [21] is one of the standard libraries, written in Fortran, that relies on BLAS and provides routines for solving *higher level problems* than BLAS, such as eigenvalue problems and linear least squares.

2.3.2 DUNE

Distributed and Unified Numerics Environment (DUNE) and the DUNE Iterative Template Solver Library (ISTL) [5] is a modular GPLv2 licenced collection of solvers and data structures, developed in C++ by several universities and research institutions, including the universities of Heidelberg, Freiburg and Münster, as well as the International Research Institute of Stavanger. Similarly to the C++ standard template library, DUNE separates data structures from algorithms and relies heavily on generic programming techniques and static polymorphism to achieve high-performant and flexible code.

DUNE is the established numerical engine in the upstream OPM project.

2.3.3 Hypre

Hypre is a library developed at Lawrence Livermore National Laboratory which provides several high-performance solutions for linear systems. The main strength is the multigrid preconditioners for structured and unstructred problems [10].

2.3.4 Fluent

The Fluent software by ANSYS is a commercially avilable software for modelling flow, turbulance and similar pyhsics. While the licencing model of Fluent is not compatible with OPM, they have since the 15.0 release support for GPU accelerated numerical computation [22].

2.4 PETSc

Portable, Extensible Toolkit for Scientific Computation (PETSc) is a suite of data structures and routines for scientific computing [4] developed by Argonne National Laboratory. It is designed for large-scale scientific computation, written mainly in C with some code in Fortran and C++, uses MPI for distributed computing and message passing and performs basic numerical computation with BLAS and LAPACK. It is broken up into into several major components, summarised in Table 2.4, all with multiple underlying implementations with different characteristics. The same program can thus

Table 2.4: Major components in PETSc, e.g. `Mat` has several different matrix representations.

`Vec`	Vectors and vector operations
`Mat`	Matrices and matrix operations
`PC`	Preconditioner algorithms and configuration
`KSP`	Krylov subspace methods
`SNES`	Non-linear equations solvers and configuration
`TS`	Time-stepping components

solve a linear system using both CG and GMRES, determined by command-line options.

This architecture also enables extensions and plugins, as well as third-party sub engines. PETSc currently has experimental GPU acceleration support, which is achieved through the ViennaCL library for OpenCL and CUDA routines for CUDA. PETSc also supports the Hypre package, UMFPACK[2], MUMPS[3], SuperLU[4] and more.

Run time configuration is done by storing options in a global **options database**, which is made available to all processes. PETSc code can then opt to not specify what implementation to use for some object, and have PETSc query the database to determine what to use. The options database can be populated both by user run-time options (typically the command line, but also environment variables and configuration files), calls to `PetscOptionsSetValue` and PETSc defaults. This allows for last-minute configuration of memory layout and algorithm choice, depending on what works best for the data set at hand, and what hardware is available at the machine running an instance of the program.

MPI plays a major role in the design of PETSc, and all objects must be connected to some MPI communicator, even when PETSc is compiled

[2]http://faculty.cse.tamu.edu/davis/suitesparse.html
[3]http://mumps-solver.org/
[4]http://crd-legacy.lbl.gov/~xiaoye/SuperLU/

without MPI support, in which case the communicator is a trival handle, and single-processes executions. This however, mean that the same program can be used for single-processor, multi-process and distributed execution environments, which makes code written in PETSc very flexible and portable.

The design is object oriented, meaning that the major components of Table 2.4 are the **classes** that can be instantiated. The class of an object determines what **messages**, implemented as C function calls, it can receive. This ensures a uniform interface and hidden implementation, which helps PETSc achieve polymorphism. Objects can only interact with eachother if they belong to the same communicator.

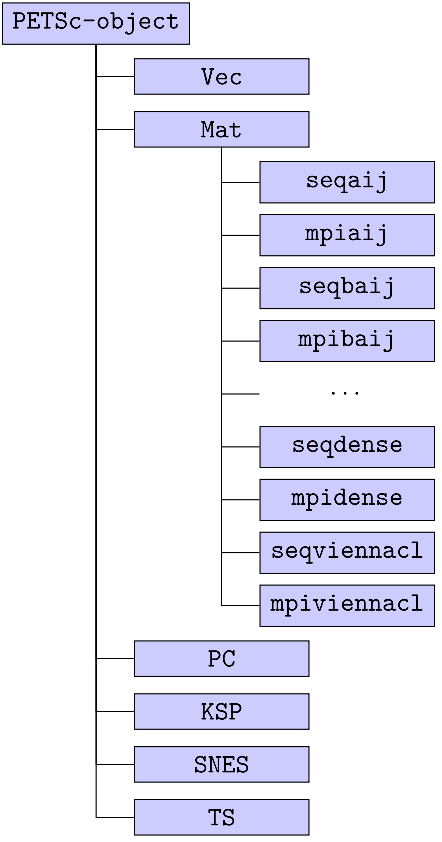

Figure 2.10: PETSc classes and their is-a relations.

Finally, PETSc provide excellent debugging and profiling tools, such

as run-time measurement on matrix insertion efficiency, allocations needed, wasted space as well as error tracing, performance logging, event logging, distributed GDB integration and more.

2.5 C++

C++ is a powerful and standardised [14] multi-vendor programming language that is mostly source-compatible with C [28] and has a strong focus on low-cost abstractions. C++, along with C and Fortran, make up the three most popular languages for performant scientific computing. This section describes som of the core features that are important for this work. Some familiarity with C-like programming languages is required to understand this section. For a thorough introduction and description of the language, please refer to the books discussed in Section 2.6. This section relies on basic understanding of C-like syntax and basic programming concepts.

The C programming languages deals primarily with raw types of data, namely primary types such as `char`, `int`, and `double`, the aggregates `struct` and `union`, and pointers. The term **object** has traditionally been used for an arbitrary block of data, including integers, but has additional meanings in certain C++ contexts. The term object throughout this thesis generally refer to this exact meaning, not a strict object-oriented programming definition, although the object-oriented meaning can arguably apply in all both at the conceptual level. `union` will not be discussed as it is a rarely used feature in C++ and does not appear in the thesis work at all.

2.5.1 Complex objects

Primary types are largely the same in C++ as in C, but C++ extends `struct` behaviour and introduces the new keyword `class`. Both enable the design of complex objects, which has **access rules**, implicitly called routines and (optionally) member variables and functions, sometimes called methods.

The C++ compiler enforces complex type member access. This enables class designers to *hide* data and functionality from the user and expose it to the compiler. An example can be seen in Listing 2.1 where an instantiation

Listing 2.1 Class member accessability

```
class A {
    public:
        int read() const { return this->x; }
    private:
        int x = 10;
};

int main() {
    A a;
    std::cout << a.x << std::endl; // illegal
    std::cout << a.read() << std::endl; // legal
}
```

can be allocated in an activation record, but disallow direct member access. The types of memory access are:

Public All code with a reference to the object can access public members.

Protected No code outside the scope of the class itself or its children can access protected members.

Private No code outside the scope of the class itself can access private members.

While the `struct` is a complex type, C++ preserves the notion of the *C struct* with the definition of **Plain Old Data (POD)**[14, §9 class]. Briefly, a POD is guaranteed to have the same memory layout as if written in C, which requires it to only consist of primary types and other PODs, have no protected or private members, only trivial constructors and no explicit destructor. The `struct` and `class` keywords are equivalent with the exception that members of a `struct` are public by default, and members of a `class` are private by default.

2.5.2 Constructor and destructor

C++ introduces the concept of the **constructor**. The constructor is a special function that *initialises* an object. Primary types, such as `int` and `double`,

are for performance reasons and C compability *not* initialised, i.e. unless a value has explicitly been assigned to a variable, its value is unspecified. Complex objects, such as classes and structs, will always be properly initialised by either a compiler-generated constructor or an explicitly defined one. The constructor will initialise, and potentially recursively construct, the members of a complex type in the order they were defined.

The **destructor** in C++ is a procedure performed at the end of the object's lifetime, and can be considered the *reverse constructor*. The destructor is defined by the standard to always run whenever an object is about to go out of scope, even in the case of exceptions [14, §15.2], and is a clean and powerful way of *cleaning up* after objects. All classes have destructors, but it is usually implicitly implemented by the compiler. Explicit destructors are still sometimes needed, particularly when dealing with pointers and other explicitly managed resources. For primary types the destructor can be considered a no-op during the tear-down of the activation record.

Analogous to constructors, destructors are *recursive* in terms of classes, so that every child will be fully destructed before the parent's destructor is called, and members are destroyed in the reverse order they were defined. Accessing a destroyed object is undefined behaviour.

These features give rise to the concept of **object lifetimes**. An object is considered to be alive when its constructor has finished executing, and before its destructor is invoked. Both constructors and destructors can perform arbitrary computation, which includes the acquisition or release of non-trivial resources such as file handles and heap objects.

2.5.3 The `this` pointer

Within the scope of a class member function, there is a special variable called the `this` pointer [14, §9.3.2 class.this], which has the value of the adress of object on which the function is called.

When a function is declared a member of a class, it is actually the *function* being made aware of the object, not the object being made aware of the function. The compiler implicitly passes the object adress as an

Listing 2.2 Inheritance

```
class A {
    public:
        int read() const { return x; }

    private:
        int x = 10;
};

class B {
    public:
        /* expose and call A (parent's) read */
        int  read() const { return parent.read(); }

    private:
        A parent;
};

class C : public A {};
```

argument to the function, effectively transforming the call `foo.method()` into `method(&foo)`.

2.5.4 Inheritance

Class inheritance is specifying a special relationship between two distinct classes, as summarised in Listing 2.2, where `class C` automatically does what `class B` does manually - make `class A` a hidden member, but expose its operations. Inheritance is subject to accessors analogous to members.

Public inheritance in C++ establishes an is-a relationship between two classes, called the **parent** and the **child**, as well as automatically adding every public visible symbol of the parent class to the child class. In C++, a public-inherited child class is a **subtype** of its parent. Briefly, anything that is public in the parent class is public in the child class in the case of public inheritance. Child classes can provide their own implementation to parent methods if the parent method has been marked as **virtual**, but this C++ feature has not been utilised in the thesis work and will not be discussed further.

When a child classed is used in a function, the compiler will first look after the function that takes the child `class C` as an argument. If it cannot be found, it will look for the symbol which takes its immediate parent `A` as an argument, effectivly casting from `C` to `A`. Going with the example in Listing 2.2, the compiler can translate `c.read()` to `read(&c)`, a symbol which will not be found, before it tries `read((A*)&c)`. Two submitted proposals for C++17, N4164 [33] by Sutter and N4174 [30] by Strostroup, put forward a unified call syntax where this relationship is explicit, i.e. the function `seek(file, bytes, options)` can be called as `file->seek(bytes, options)`. Strostroup also argues that the reverse should also be valid.

While not enforced by the language, it is considered bad design if a child class, or subtype, does not match the properties of the parent. This was formulated by Liskov and Wing in 1994 [17], and is known as the **Liskov substitution principle**.

Definition 2.19. Liskov substitution principle Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be provable for objects y of type S, where S is a subtype of T.

Private inheritance does not establish the is-a relationship between the parent and the child; instead, the parent class is merely an implementation detail of the child class and the inheritance is used as an implementation technique for code reuse and modularity. An example of this is discussed in Section 4.5. It does not automatically get the parent class interface, but can **internally** use the parent's public members, as well as implcitly using the parent's storage. This is equivalent of using the parent class as a member variable, but using inheritance can provide some syntactical shortcuts. The child class can expose individual parent features through the `using` keyword. See Listing 2.3 for an example.

2.5.5 Overloading

C++ supports overloading, i.e. using the same symbol name for different symbols. In C++, the function name, arity and argument types are all parts of the symbol name, meaning `foo(int)`, `foo(int, int)` and `foo(double)` are three different functions.

Listing 2.3 Private inheritance

```
class A {
    public:
        void foo();
        void bar();
};

class B : private A {
    public:
        using A::foo;
};

int main() {
    B b;
    b.foo(); // ok
    b.bar(); // illegal
}
```

Overloading does not only apply to functions, but also class methods and operators. This is a very useful feature when writing (statically) polymorphic code and in the design of good interfaces.

2.5.6 Template programming

C++ offers a very powerful [34] compile time sub language called templates. Templated functions and classes are parsed and represented internally in the compiler, but not fully completed until they are instantiated with a specific type. The representation is then copied and the template *placeholders* are substituted with the actual type. This allows for very flexible, type safe code at the cost of more expensive compilations.

The C++ standard library is packed with templated code. An example is the popular vector container which is templated on both a value type and its resource allocator. `std::vector<int>` is a different type than `std::vector<char>`, but their interfaces are identical and they perform as if they were implemented by hand as `vector_int` and `vector_char`.

Templates and template signatures are sometimes time consuming to write, but are rewarding in terms of extra compile time safety. The compiler can in a

Listing 2.4 Template programming

```
template< typename T, typename U >
T max( T x, U y ) {
    // assumes T > U is defined
    if( x > y ) return x;
    return y;
}

int find_max( int* arr, int size ) {
    int current = numeric_min();
    for( int i = 0; i < size; ++i ) {
        // calls max< int, int >
        current = max(current, arr[i]);
    }

    return current;
}
```

lot of cases infer what specific template to use at the call site from arguments, see listing 2.4 for an example.

C++11 introduced **variadic templates**, templates that take a variable number of arguments. This is similar to C's `va_arg`, except types are expanded and instantiated compile time. One key application is recursively expanding and generating functions, classes and invocations - such as the solve function described in Section 4.8.

2.5.7 Move semantics

Temporary values in C++ before C++11 were unmodifiable, but the introduction of **move semantics** and **rvalues**, syntactically denoted `T&&`, changed this. Move semantics allow some optimisations that would otherwise trigger deep copies right before the destruction of the original. Since temporary value references are now distinguishable from *regular* references, classes can provide different constructors and overloads, which can help implement clear transfer of resource **ownership**.

Consider the code in Listing 2.5. Until C++11, the *contents* of the

Listing 2.5 move without copy

```
1 std::vector allocate_large_vector();
2
3 int main() {
4     std::vector x = allocate_large_vector();
5 }
```

`std::vector` would have to be copied into a temporary[5] variable, and the original, inside the `allocate_large_vector` function would be destroyed by going out of scope. Values given to `return` are implicitly rvalues as they can no longer be used anyway, so the underlying pointers of the `std::vector` containers are simply copied, without having to copy whatever they point to. This saves a potentially expensive copy operation, even though the program *behaves* as if the `std::vector` was copied.

2.5.8 Resource Acquisition is Initialisation

Resource Acquisition Is Initialisation (RAII) is an idiom, not a feature, but relies on the capabilities of the constructor and the destructor, and is fundamental to the design of the library. Using RAII, resources are acquired during object construction, the beginning of its lifetime, and released when the object lifetime ends and its destructor is called, as opposed to use temporarily uninitialised variables, init functions etc. This idiom simplifies resource management, provides exception safety and simplifies reasoning about programs.

The simplest example of RAII is the management of heap-allocated memory, see Listing 2.6. However, RAII works for resources in general, including file descriptors, database handles and third party initialsation and cleanup routines.

2.6 Further reading

Kreyszig [15, parts B and E] is an easy introduction to the fundamentals of linear algebra and numerical methods through software; Butenko [7] is

[5] We assume that return-value optimisation does not apply.

Listing 2.6 RAII managing a heap-allocated array

```
class vec {
    public:
        vec(int s) :
            storage( new int[ sz ] ),
            size( sz )
        { /* acquire resources */ }

        ~vec() {
            /* release resources */
            delete[] this->storage;
        }

    private:
        int* storage;
        int size;
};
```

another nice alternative. Press [23] provides an introduction to numerical analysis and a lot of algorithms with code examples.

Stroustrup [31] is a thorough and beginner-friendly introduction to C++ programming. For an introduction to effective generic programming techinuqes, see Alexandrescu [1]. For a K&R style language reference, see Strostroup [29].

For an introduction to profiling tools such as gprof and valgrind, GPUs, and optimising for GPUs, see Stinessen [27] and Skolmedal [26].

Upscaling

This Chapter introduces some of the terms and concepts that comes across in reservoir simulation, reservoir engineering and exploration, and describes the main aspects of upscaling and the *upscale* family of programs in OPM. Section 3.1 introduces some basic concepts of reservoir engineering and Darcy's Law. Section 3.2 explains the basics grids required to understand upscaling, while Section 3.3 describes the basics of upscaling. Section 3.4 describes the *upscale* program design and characteristics. Finally, Section 3.5 discusses the scalability of the *upscale* programs.

Algorithm 3.1 The general OPM *upscale* design.

Input: Model and rock descriptions
Output: Upscaled model
1: parse model
2: parse functions
3: tesselate the grid
4: upscale

3.1 Reservoir engineering, permeability and Darcy's Law

Hydrocarbons such as petroleum and natural gas are typically found in subsurface **reservoirs**, pools of porous or fractured rock formations trapped under overlying rocks with low **permeability** that form a **seal**. Permeability is the measure of the ability a porous medium has to allow fluid to pass

Figure 3.1: A porous rock with clearly visible perforations. Used with permission from Jonathan Zander.

through it. The SI unit is m^2, however the practical unit darcy, $d = 10^{-12}m^2$, is often used. A **porous medium**, sometimes referred to as a porous material, is a material that consists of pores or voids. Typically these pores are filled with a fluid, which in the case of the oil reservoir would be hydrocarbons.

Hydrocarbons and other fluids can flow through porous media. The **superficial flow velocity**, the hypothetical flow velocity if the fluid in question was the only one present in a given area, is described by **Darcy's Law**.

Definition 3.1. Darcy's Law

$$v = \frac{\kappa}{\mu}\frac{\Delta P}{\Delta x} \tag{3.1}$$

v is the superficial fluid flow velocity.

κ is the permeability of the medium.

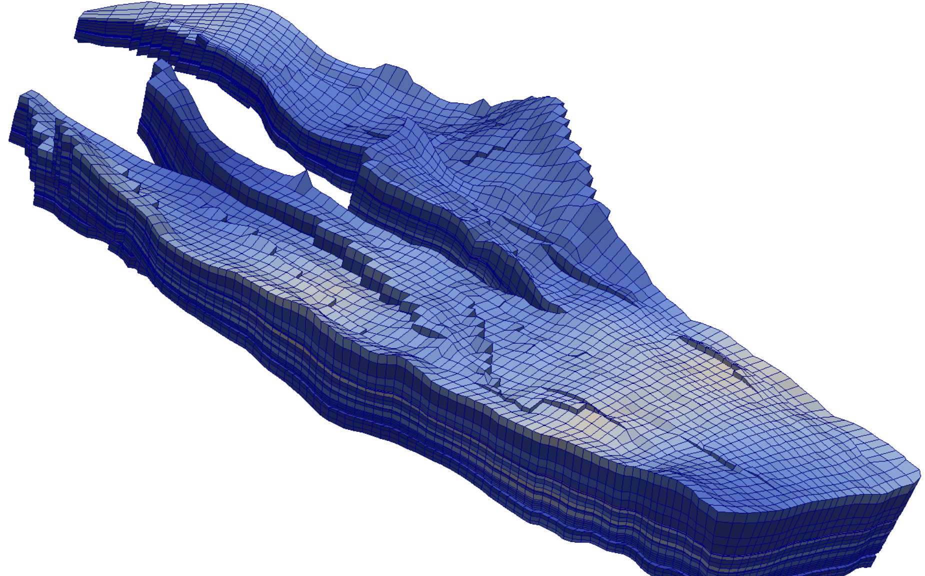

Figure 3.2: The tesselated Norne reservoir. Used with permission from the OPM Initiative.

μ is the dynamic viscosity of the fluid.

ΔP is the applied pressure difference.

Δx is the thickness of the porous medium bed.

A **fault** is a fracture or discontinuity of rock with relative displacement of the rocks on the opposite sides of the fracture. **Flux** is the *rate of flow* per unit area.

3.2 Grid

A grid is a discretisation of a model, meaning turning a geological model of a field into a discrete system on which we can solve equations. A tesselated representation of the Norne reservoir can be seen in Figure 3.2

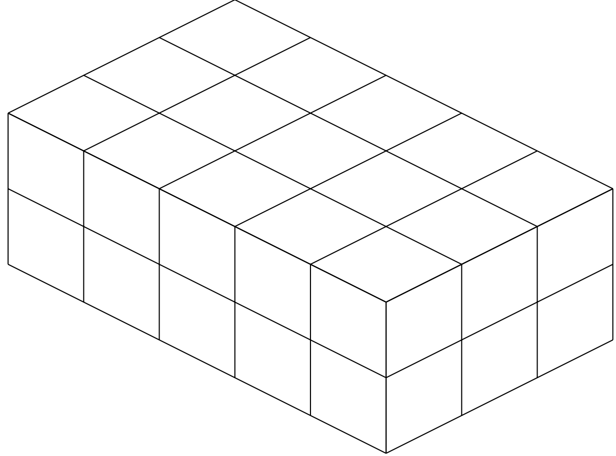

Figure 3.3: A simple 3D Cartesian grid (dimensions 5 x 3 x 2)

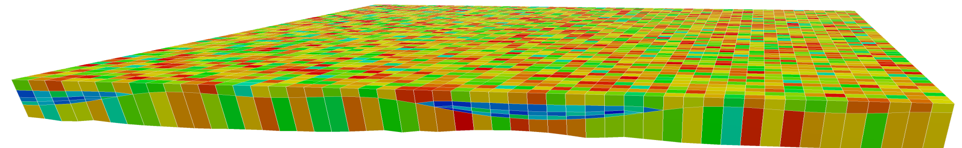

Figure 3.4: A 3D hexahedral grid.

The simplest 3D grid is the **cartesian grid**, where the identically shaped cells can simply be identified with their (x, y, z) index values, as seen in Figure 3.3.

The simple cartesian grid is rarely sufficient for a good capture of the geometry of a reservoir. OPM has support for several different grid types, but the most common are **hexahedral grid** (or **corner-point grid**), which allows a more precise description of faults, and the **unstructured grid**. The hexahedral cell is defined by the positions of its eight corners and bilinear planes as its geometry, as shown in Figure 3.4. The pillars of the corner-point have some lexicographic ordering.

The basic coordinate system of the corner-point grod can still be unsuitable for the irregularities of a reservoir. In these cases, truncating or distorting the grid can be done, but other solutions include **multiple-domain grids**,

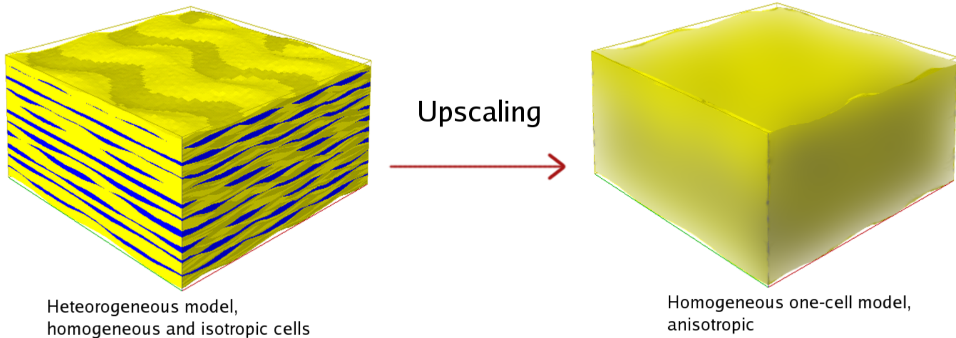

Figure 3.5: Used with permission from Statoil.

where an additional parameter to the (x, y, z) indexing is added to specify the local grid system, local grid refinements or the unstructured grid.

An **unstructured grid** is a space tesselated by simple shapes. Unlike the discussed grids in a regular coordinate system, unstructured grids requires tracking of connectivity, as in a graph.

3.3 Basics of upscaling

Upscaling is an averaging procedure where we coarsen the characteristics of a finer scale model into a coarser model. Coarser in this sense means with larger, less precise grid cells. 3D geological models may contain very detailed descriptions of the reservoir, and while it would be nice to preserve these details in all simulations, the computational requirements can become impractical. The coarser, upscaled model is often sufficient for reservoir engineering, and can greatly reduce the total computational cost.

Simplified, we calculate the flow velocity through a cell with Equation 3.1, Darcy's Law. We ignore viscosity for now, as it is a constant contribution for a single fluid, and the thickness of the bed can be considered constant as well. This reduces upscaling to the approximation of Equation 3.2, a linear function of permeability and the pressure difference.

$$v = \kappa \Delta P \tag{3.2}$$

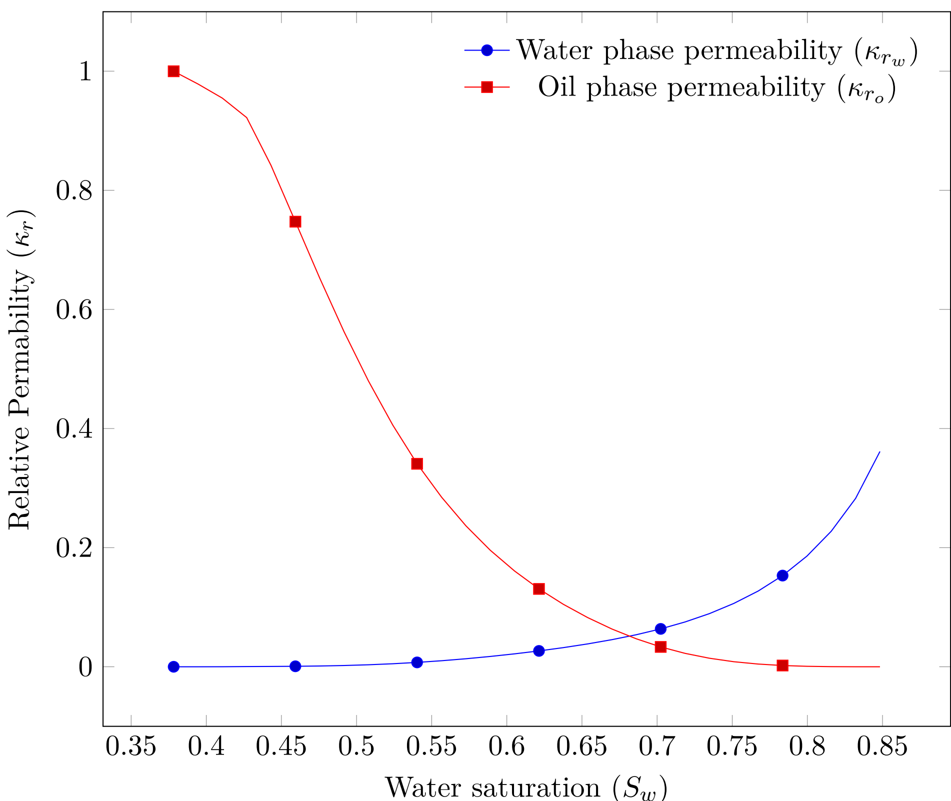

Figure 3.6: Upscaled relative permeability of an example grid.

κ is as before a property of the rock. However, the assumption for the *upscale_relperm* there is not only water, but also oil present. We introduce the functions k_{r_w} and k_{r_o} to correct for this, as oil flows better when most of the rock content is oil, and vice versa for water. The term $k_r\kappa$ is called **phase permeability** and depends on the water saturation of the cell, as seen in Figure 3.6.

$$v = k_r \kappa \Delta P \tag{3.3}$$

3.4 Program design

The family of upscaling programs in OPM follow the same design, depicted in Figure 3.1. Exactly what the **upscale** step entails depends on the considered property of that particular run or configuration. The case I have studied the most is *upscale_relperm* program which performs upscaling as discussed in the previous section. Algorithm 3.2 fleshes out the upscaling step from Algorithm 3.1.

Algorithm 3.2 The general OPM *upscale_relperm* design

Input: Model and rock descriptions
Output: Upscaled model
1: parse model
2: parse functions
3: tesselate the grid
4: upscale capillary
5: upscale single-phase permeability
6: **for all** upscaling points **do**
7:     upscale saturation point
8: **end for**

The *upscale* programs use DUNE as the third party numerical engine for most of its work, which is solving the linear system that arises from some equation, typically Darcy's Law, such as the solver I have studied which is described in this section. A significant portion of the program is spent solving these linear systems: Table 3.1 shows the timings for the different parts of an unmodified *upscale_relperm* with DUNE as its backend. The measurements was performed on my workstation and development machine, the specification of which can be found in Table 4.1. The unmodified program uses FastAMG as preconditioner and CG as the Krylov subspace algorithm.

The *upscale* programs use the *IncompFlowSolverHybrid* solver in *opm-porsol*, which uses Darcy's Law to model mixed formulation incompressible flow. The discretisation procedure produces the linear system of Equation 3.4, where v are the interface fluxes of each cell and p are the cell interface

Table 3.1: Execution time breakdown for an unmodified *upscale_relperm*.

Program part	Time (seconds)	Portion (%)
Parsing grid	16.5	$3.46 \cdot 10^{-2}$
Tesselation	5.1	$1.07 \cdot 10^{-2}$
Min/max capillary pressure	0.06	$1.26 \cdot 10^{-4}$
Upscaling capillary pressure	42.9	$9 \cdot 10^{-2}$
Upscaling	4702.39	98.6
Total	4766.95	100

pressures.

$$\begin{bmatrix} B & C & D \\ C^T & 0 & 0 \\ D^T & 0 & 0 \end{bmatrix} \begin{bmatrix} v \\ -p \\ \pi \end{bmatrix} = \begin{bmatrix} f \\ g \\ h \end{bmatrix} \tag{3.4}$$

By performing Schur complement analysis [37] Equation 3.4 is transformed to Equation 3.5.

$$\begin{bmatrix} B & C & D \\ 0 & -L & -F \\ 0 & 0 & S \end{bmatrix} \begin{bmatrix} v \\ -p \\ \pi \end{bmatrix} = \begin{bmatrix} f \\ \hat{g} \\ r \end{bmatrix} \tag{3.5}$$

where

$$\begin{aligned} L &= C^T B^{-1} C \\ F &= C^T B^{-1} D \\ S &= D^T B^{-1} D - F^T L^{-1} F \\ \hat{g} &= g - C^T B^{-1} f \\ r &= D^T B^{-1} f + F^T L^{-1} \hat{g} - h \end{aligned}$$

A linear solver package, e.g. DUNE, solves the subsystem

$$S\pi = r \tag{3.6}$$

By using the computed solution to Equation 3.6, the simpler systems of Equation 3.7 and Equation 3.8 yield the cell fluxes and interface pressures through a back substitution process.

$$Lp = \hat{g} + F\pi \tag{3.7}$$

$$Bv = f + Cp - D\pi \tag{3.8}$$

3.5 Scalability

Achieving scalability is not the main topic of this thesis, however, it is an interesting and useful property for reservoir software to have, and is desirable in order to simulate large reservoirs within reasonable time. This section briefly discusses some design choices and its consequences for parallelism in the *upscale* family of programs.

The *upscale* programs all use a very simple parallelisation scheme. The upscaling of the the single phase permeability is only done on the master node, and the saturation points are distributed among the available MPI processes. Interpreting this in the context of Amdahl's Law (Equation 2.15), the maximum achievable speedup is bounded by how much computing the saturation points makes up of the full program. While this portion is quite large ($> 98\%$, see Table 3.1), Amdahl's Law limits speedup to 71x *with the assumption that the parallel portion can be perfectly broken down.* This assumption, however, does not hold with the current implementation.

On the other hand, interpreting the scalability of this design choice in context of Gustafson's Law, as demonstrated in Equation 3.9, we can compute S_p, the number of saturation points, within a time limit. Gustafson's Law, which fixes time spent instead, provides a better tool for modelling the scalability in a **task**-oriented manner, which applies to the *upscale* design. A

task is an uninterruptable computation that runs linearly from start to finish, and the upscaling of each saturation point is an example of such a task. It is worth noting that performing the task-internal computation in parallel is a possibility, but would in terms of scaling reduce the wallclock time of each task, without changing their externally sequential behaviour.

$$\begin{aligned} n &= S_p \\ Speedup &= n - \alpha \cdot (n-1) \quad \text{(Gustafson's Law)} \end{aligned} \tag{3.9}$$

A noteworthy property of upscaling is that the problem size has its natural limits. It is bounded both by the the practical size of the surveyed data, i.e. how big the reservoir actually is, and the limits of significance for the results, i.e. properties of rock far away from the reservoir will not have any impact on the reservoir itself. This means we cannot keep on increasing the size of the reservoir to preserve scaling. However, since the limiting factor in scalability is the number of saturation points to be upscaled, and more processors would allow us to upscale more saturation points within a given time, as modelled by Equation 3.9. Increasing the number of saturation points will yield a more fine-grained function, as show by the two graphs in Figure 3.7. This may or may not be interesting beyond a certain point.

The parallelisation scheme is static, i.e. there is no work stealing, load balancing or scheduling. Assuming no overhead and perfect parallelisation, execution time is dominated by the slowest saturation point. Figure 3.8 demonstrates a parallel run of the *upscale_relperm*. The linear solver is called within each instance of *perm* and *satnum*.

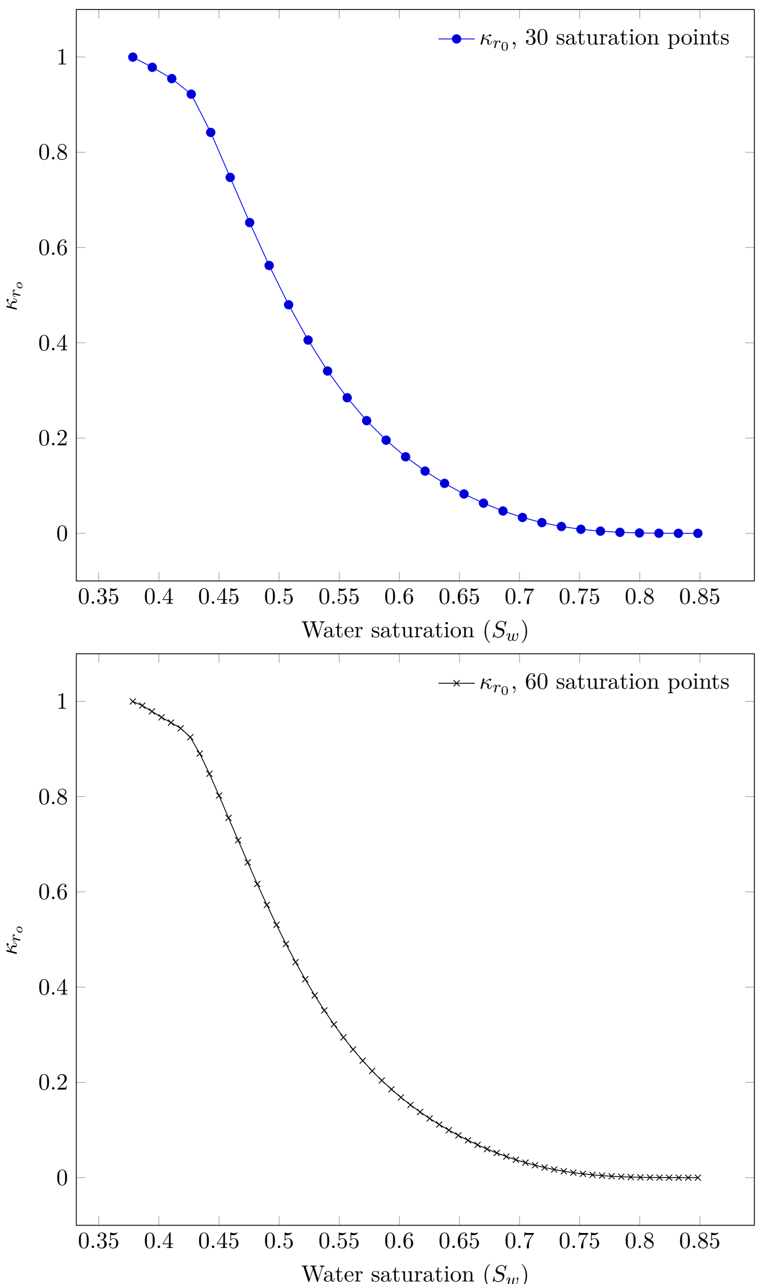

Figure 3.7: Upscaled relative permeability with 30 and 60 saturation points.

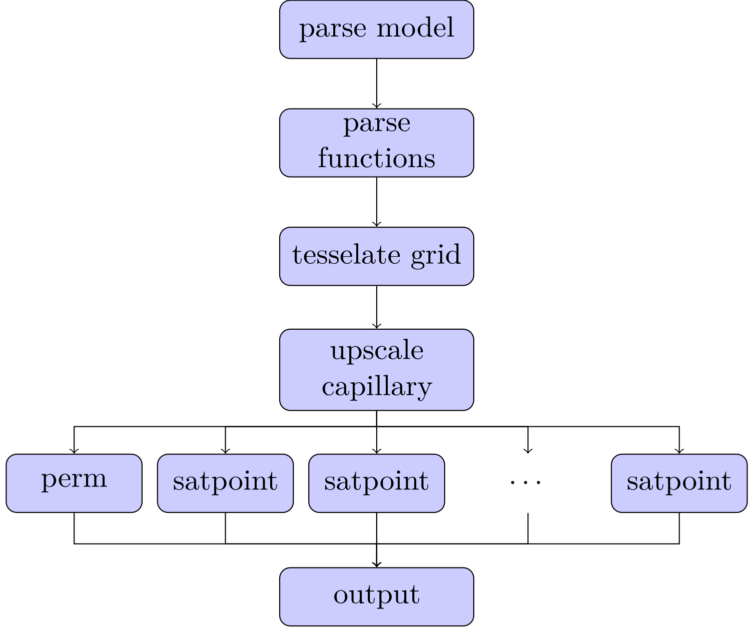

Figure 3.8: The OPM *upscale_relperm* parallel design.

Chapter 4

Integrating PETSc with OPM

This Chapter describes the main work of this thesis, which is the design and implementation of the integration layer so that PETSc could be used with ease in OPM programs. The motivation for this is access to PETSc's algorithms, third party support and excellent profiling - and debugging tools. While performance always is of interest, improving overall performance is not the goal, but instead a potential benefit. Section 4.1 briefly describes how the thesis work was done, while Section 4.2 discusses the motivations for additional code on top of PETSc, instead of just porting applications using no abstraction. Section 4.3 describes the use of types, and Section 4.4 extends on the idea of a higher level interface to PETSc features. Sections 4.5, 4.6, 4.7, and 4.8 describe the interfaces and implementations of the individual components of the thesis work. Finally, Section 4.9 describes the necessary steps to port *upscale_relperm* to the PETSc backend, and Section 4.10 describes bugs found and fixed during development and testing.

4.1 On development

An iterative refinement model was employed for the thesis work, where a component, say `Vector`, was the focus of the design. An initial design was developed, tested, before the need of a redesign was evaluted. Testing includes profiling and performance evaluation as well as experimenting with the quirks, edge cases and general behaviour of the component in question. More than three different `Vector` approaches were tested before the final design was settled. To test the finalised design, the *upscale_relperm* was ported to get a

Table 4.1: Workstation hardware configuration.

System	
CPU	Intel Core i7 950 (Bloomfield)
Frequency	3.07 GHz
Cores	4 (8 with HT)
Memory	12 GB
Instruction set	x86_64
GPU	Nvidia GeForce GTX980
Video driver	nvidia-346.47
OS	Ubuntu 14.04
GCC	4.8.2

feeling of how the design worked in practice. The resulting interfaces were submitted as a working draft on a standard interface for linear algebra and numerical computation in OPM.

Development and initial profiling was done on my HPC-lab workstation, specifications listed in Table 4.1.

4.2 Motivation for a library

Adding support for a new third-party mathematical engine into an existing, well-established project can require substantial effort. Different conventions, slightly different semantics, different interface guidlines and different types all contribute to friction between established code and code yet to be written. This chapter discusses the design choices, techniques and the resulting PETSc support library developed during this thesis.

Consider the code examples in Listing A.11. This is solving a very simple system in PETSc. While still easy to understand, this style of code is alien to a modern C++ codebase. Listing A.10 is equivalent code written with the OPM PETSc library.

The goal of this is to make interacting with PETSc *easier*, so that code can be developed faster, simpler, and, ideally, with fewer bugs.

A remark on philosophy: the goal was not to support all of PETSc's features, but rather provide easier access to commonly used structures and functions. In light of this, I have only implemented a small subset of the features for both containers and solvers, and extended this subset when needed. This way, less initial code must be developed and tested, at the cost of burdening future maintenance with adding more features as needed. However, this removes a burden of implementing features *that will never be used*, and keeps the library as slim and simple as possible.

4.3 Types and containers

As discussed in Section 2.1.1, the primary tools for linear algebra and numerical computation are matrices and vectors. One of the established third-party numerics engine in OPM, DUNE, has a variety of matrix and vector types. Table 4.2 lists the matrix and vector types available in DUNE 2.3. `BCRSMatrix` is by far the most used matrix class.

Table 4.2: Containers in DUNE 2.3

Matrix	
`BCRSMatrix< >`	Block compressed-row storage
`BDMatrix< >`	Block-diagonal
`BTDMatrix< >`	Block-tridiagonal
`Matrix< >`	Dense matrix

Vector	
`BlockVector< >`	Vector of blocks
`VariableBlockVector< >`	Vector of differently sized blocks

PETSc offers its own vector and matrix types. However, contrary to DUNE, PETSc only provides a polymorphic handle to *all* types, and determines the actual implementation at runtime. This design choice, coupled with PETSc being written in C, means there is no lifetime management of objects, and

the programmer is responsible for managing and releasing resources. This applies to all PETSc types, and is not limited to matrices and vectors. This conflicts with the lifetime-managed idioms of modern C++, which is applied by DUNE and OPM, and in order for PETSc to integrate well with OPM and be easier to use, and a set of smart pointers areound the PETSc handles are developed. These provides automatic memory management at zero additional runtime cost through RAII, as described in Section 2.5.8.

The polymorphic property of PETSc containers has been carried over to the library. Consequently, there are only two containers in the library; `Vector` and `Matrix`. This was a conscious design choice in order to map closely onto PETSc definitions and to support configuration via the options database (described in Section 2.4). Extensions, such as strict single-type containers, are discussed in Section 6.3.

4.4 High-level interface

PETSc is written in C, and can be called without modification or special Foreign Function Interface (FFI) *bindings* from a C++ program. However, due to some design choices in PETSc and some limitations in the C language, the interface is largely alien in a modern C++ codebase. Since there is no need for language bindings per se, my support library consists mostly of a higher-level reformulated interface to PETSc functionality.

When designing the library I put an emphasis on static properties and ease of use. The main solution for this was encouraging only expressing data transformation in terms of algebraic operations, and not through direct memory manipulation, i.e. on a higher level. This brings three major benefits.

Brevity Reducing the amount of code needed to utilise PETSc solvers in either a new program or when porting a program to the new numerical back-end. Listing A.10 demonstrates how to solve a linear system using my support library, and Listing A.11 shows the equivalent code in unmodified PETSc.

Code clarity The second is code clarity, and is related to brevity. With algorithms easier to read and implementation complexity abstracted

away, it is easier to focus on the problem at hand. OPM does not attempt to be a linear solver, but a simulator, so the full-scale flexilibity and configurability of a linear solver package is not needed.

Scalability When the programmer cannot directly modify or even read memory, the synchronisation task is much less daunting, because we now can assume more about memory accesses, thread safety and data dependencies.

In fact, due to PETSc's MPI-oriented nature, there are rarely guarantees that the cell you want to read or perform some low-level operation on it actually is available to your process without (expensive) communication. Considering that direct memory access are rarely ever *needed*, i.e. there are usually other and better ways of expressing the algorithm, this is a beneficial tradeoff.

The interfaces are discussed in more detail in the following sections.

4.5 Common and mixins

The core of all types introduced in the library is the uptr - a slightly modified `std::unique_ptr`. The full implementation can be seen in Listing A.1. `uptr` privately inherits from `std::unique_ptr` to relinquish the is-a relationship and avoid the potentially wrong overload being called in the case where a function takes an arbitrary `std::unique_ptr< T >&&`. The `typedef`s are there to easier be able to access the raw, underlying types from classes using this mixin.

The `uptr` modifies `std::unique_ptr` by removing some operations; the assignment operator (`=`), the dereference operators (`*` and `->`), the methods `release`, `reset`, `swap`, `get`, and `get_deleter`. These operations are all pointer specific, and the idea is for the `uptr` class to be the only one actually dealing with a pointer - the classes using the mixin and their resulting interface should not expose this implementation detail. Additionally, this ensures the the responsilibties for resource management is left fully to the mixin.

As previously discussed, the aim has been to simplify PETSc use. However, for some problems the programmer must do fine-grained configuration in order to achieve performance or to use some feature not supported by the OPM library. In these situations, an implicit conversion to the underlying PETSc type is supported, `operator pointer()`. This allows `uptr` derived objects to be used as drop-in replacements for PETSc handles in raw PETSc functions.

The class is templated on the arbitrary type `T`, but the mixin itself is defined in terms of `T`. This is done as an effort to make code more understandable, as PETSc exposes the types `Vec`, `Mat` etc., which are pointers to some opaque structures `_p_Vec` and `_p_Mat`. By defining `uptr< T* >` instead of `uptr< T >`, it is sufficient for classes using the mixin to specify the PETSc handle, not whatever the opaque structure is actually called, as seen in Listing A.3. The `uptr` requires that `struct deleter` is defined for the child class, which it in turn passes to `std::unique_ptr`. By not passing this explicitly as in `std::unique_ptr` we keep declarations slightly simpler, at the cost of implicitly passed information. `deleter` should call the approperiate Destroy function for the PETSc object in question. Using this techinque, PETSc's explicit cleanup functions are called implicitly when an object goes out of scope, which is what a modern C++ programmer would expect.

Besides modularising and reducing code duplication, the mixin provides some level of provability to classes using it. Since all ownership and resource management are managed in `uptr`, any class derived from this, as long as they do not specifically override behaviour, will also have sound and verified resource management, resulting in more confidence in code.

Implementing resource management using the `uptr` class does not contribute any overhead in the resulting program. Resources would have to be released regardless of mechanism, and the compiler will simply inject `deleter`'s destructor whenever the object, and consequently `uptr`, goes out of scope, and is no different from the programmer calling e.g. `VecDestroy` manually. Additionally, since `uptr` has no member variables it does not increase the size of the object [14, §5.10 expr.eq, §5.3.3 expr.sizeof, §9 class, §9.2 class members].

4.6 Vector

The `Vector` class provides automatic storage and operations for PETSc's `Vec` class. The full interface with brief descriptions can be found in Table 4.3.

No default constructor is offered by `Vector`, which means that the library actively *prevents* (i.e the compiler will refuse the program) classes of errors and the need for checks that arises from performing operations that would otherwise be undefined. The compiler enforces that `Vector`s *must* have a size, that they are initialised properly and perform undefined actions such as adding two vectors where one is of size 0. This also means that classes that has `Vector` as a member variable must ensure it is initialised properly during its own constructor to not be ill-formed. When programming with C and PETSc directly, this is something the programmer must manually keep track of and ensure, so the burden on the programmer is strictly less than without the library.

Resource management for `Vector` is provided by `uptr`. This means that a `deleter` struct must be defined, and is as in Listing A.4. This, unlike `uptr`, uses the underlying symbol `_p_Vec`. This is an arbitrary choice, but I consider `deleter` a very simple implementation detail, meaning the only time it should be exposed is to a library developer, who is expected to have a firm grasp of the concept. Additionally, implementing it differently requires either as many lines of delegating code as the `struct` itself, or the use of a new C++11 feature seen in Listing A.5 called `decltype`. While elegant, it is slightly exotic and has not seen that much use yet, so I concluded that using `_p_Vec` would be simpler.

In the code examined during the thesis work there has been little manipulation of vectors - in fact, the most advanced use has been writing the constant vectors and reading the solution vectors, and consequently few operations have been implemented in `Vector`. Regardless, none of these features include direct element access or reading memory. Instead, vectors are treated as complete algebraic objects. `Vector` contents may possibly be distributed across several MPI processes, but support for this is only experimental.

The `Vector` class uses a value-oriented design. Considering the code in

Table 4.3: `Vector` interface. `const` qualifiers omitted.

Constructors	
`Vector(Vec)`	Takes ownership of PETSc handle
`Vector(Vector&)`	Copy constructor
`Vector(Vector&&)`	Move constructor
`Vector(size_type n)`	n-sized vector
`Vector(size_type n, scalar x)`	n-sized x-element vector
`Vector(std::vector&)`	Copy of a `std::vector`
`Vector(std::vector&, std::vector&)`	`std::vector` copy at specific indices
Queries	
`size_type size()`	Query vector size
Modifiers	
`void assign(scalar)`	Assign a single value to all entries
Operators	
`Vector& =(Vector&)`	Copy assignment
`Vector& =(Vector&&)`	Move assignment
`Vector& += scalar`	Add a constant to all entries
`Vector& -= scalar`	Remove a constant to all entries
`Vector& *= scalar`	Scale all entries
`Vector& /= scalar`	Scale all entries
`Vector +(Vector, scalar)`	Copy-then-add
`Vector -(Vector, scalar)`	Copy-then-subtract
`Vector *(Vector, scalar)`	Copy-then-scale
`Vector /(Vector, scalar)`	Copy-then-scale
`Vector +(Vector, Vector)`	Vector addition
`Vector -(Vector, Vector)`	Vector subtraction
`scalar *(Vector, Vector)`	Dot product
Functions	
`scalar dot(Vector&, Vector&)`	Dot product
`scalar sum(Vector&)`	Sum
`scalar max(Vector&)`	Max
`scalar min(Vector&)`	Min

Listing A.6, a function from the library, where PETSc uses pointers and error codes, and the library uses a return value. All features are implemented this way, including matrices and solvers, which enables programmers to write simpler code, with less aliasing. Additionally, since functions have return values, there is no need for temporary variables that pollute the namespace in order to read function output.

Most of the library consists of trivial calls to some corresponding PETSc function, as seen in Listing A.6. While this might seem like unecessary to, in the worst case, requires an extra activation record and return-value copy, all which add up in a large program, there are some compelling arguments for still doing it:

Memory access Most PETSc functions would immediately dereference the PETSc object, which causes memory access assuming the object has been evicted from the cache, which quickly dominates the overhead of an extra activation record.

Non-trivial work Most functions perform non-trivial work which, which makes the extra time spent handling activation records insignificant.

Programmer efficiency A small rise in programmer efficiency can be worth a slight increase in runtime cost - otherwise, slower and higher level languages such as Perl would not see much use.

Optimisation Through inlining, return-value optimisations and link-time optimisations the overhead can even disappear completely, meaning we get a nicer interface for free.

4.7 Matrix

The `Matrix` class uses `uptr` to wrap around PETSc's `Mat` handle, and is designed similarly to `Vector`. In fact, most of the considerations and decisions discussed in Section 4.6 retarding value orientation, optimisation and management also applies to `Matrix`. The interface is summarised in Table 4.4.

Matrices are often built in steps. In the case of the *upscaler* programs, they are constructed by renumbering the grid, which is then traversed and

Table 4.4: `Matrix` interface. `const` qualifiers omitted

Constructors	
`Matrix(Mat)`	Takes ownership of PETSc handle
`Matrix(Matrix&)`	Copy constructor
`Matrix(Matrix&&)`	Move constructor
Queries	
`size_type rows()`	Query number of rows
`size_type cols()`	Query number of columns
Operators	
`Matrix& =(Matrix&)`	Copy assignment
`Matrix& =(Matrix&&)`	Move assignment
`Matrix& *= scalar`	$A = cA$
`Matrix& /= scalar`	$A = \frac{1}{c}A$
`Matrix& += Matrix`	$A = A + B$
`Matrix& -= Matrix`	$A = A - B$
`Matrix& *= Matrix`	$A = AB$
`Matrix& /= Matrix`	$A = \frac{A}{B}$
`Matrix *(Matrix, scalar)`	$B = cA$
`Matrix /(Matrix, scalar)`	$B = \frac{1}{c}A$
`Matrix +(Matrix, Matrix)`	$C = A + B$
`Matrix -(Matrix, Matrix)`	$C = A - B$
`Matrix *(Matrix, Matrix)`	$C = AB$
`Matrix /(Matrix, Matrix)`	$C = \frac{A}{B}$
`Vector *(Matrix, Vector)`	$b = Ax$
Functions	
`Matrix multiply(Matrix&, Matrix&)`	Matrix multiplication
`Vector multiply(Matrix&, Vector&)`	Matrix-vector multiplication
`Matrix transpose(Matrix)`	Matrix transpose
`Matrix hermetian_transpose(Matrix)`	Matrix hermetian transpose

constructs the non-zero pattern, which results in a phase of the program where a matrix is partially built and partially populated, but not complete and does not support matrix operations. This is controlled via run-time checks, which will most likely abort your program if you violate this, but a better design would be to statically disallow invalid operations. By making an incomplete *building* matrix a type distinct from a complete *built* matrix, the type system ensures some invalid states cannot be reached, which reduces the need for run-time checks, and allows code to convey more information about the execution of the program.

The implementation uses a family of builder classes. A family was used because there are different strategies for handling multiple assignments in the same cell. The current supported operations are insertion, which means the last value v that was given to the cell ij is the resulting value of a_{ij}, and accumulation, which means a_{ij} is the *sum* of all values v_k. The builders can also act as a staging ground for partially-built matrices.

The `Matrix` class does in fact not have any way of directly constructing a `Matrix`, except through copying, which ensures the only way to obtain a `Matrix` is through the builders, which guarantees a valid state, or other matrices, which obviously are valid. This is not too strange to raw PETSc, as the functions `MatAssemblyBegin` and `MatAssemblyEnd` mark the end of the building phase and the beginning of the completed matrix phase.

Using explicit building phase objects does not introduce additional overhead, since PETSc requires a call to `MatrixAssembly`. The builders do not introduce extra members, object storage is still trivial, and the conversion from a builder to a completed object is a pointer copy, which can be elided, plus the required `MatrixAssembly` call. As seen in Table 4.5, this conversion is handled by the commit functions, which can use either a copy or a move, depending on the builder's reference type.

4.8 Solver

Similar to `Vector` and `Matrix`, the `Solver` class also relies on `uptr` for its resource management, but unlike `Vector` and `Matrix`, `Solver`'s resources is mostly configuration related, and not actual data.

Table 4.5: `Builder` interface for inserter. `const` qualifiers omitted. The interface is identical for the accumulator, with `add` substituted for `insert`.

Constructors	
`Builder(size_type, size_type)`	$m \times n$ matrix
`Builder(size_type, size_type, std::vector&)`	$m \times n$ matrix with preallocation hints
`Builder(size_type, size_type, std::vector&, std::vector&)`	$m \times n$ matrix with preallocation hints
Operations	
`insert(size_type, size_type, scalar)`	Insert at a_{ij}
`insert(std::vector&, std::vector&, std::vector&)`	Insert CSR-formatted submatrix
`insert_row(size_type, std::vector&, size_type)`	Insert a (partial) row
`insert_row(size_type, std::vector&, std::vector&)`	Insert a non-continuous row
Modifiers	
`Matrix commit(Builder&)`	Finalise the matrix and create a copy. The builder can still be used.
`Matrix commit(Builder&&)`	Finalise the matrix. The builder is left in an unspecified state and can no longer be used.

Table 4.6: `Solver` interface. `const` qualifiers omitted.

Constructors	
`Solver()`	Default constructor
Configuration	
`set(Linear_tolerance&)`	Set linear tolerances, max iterations etc
`set(Nonlinear_tolerance&)`	Set non-linear tolerances
`set(Matrix&)`	Set operator for the preconditioner
`set(Pc_type)`	Set preconditioning algorithm
`set(Ksp_type)`	Set Krylov subspace method
`set(Snes_type)`	Set non-linear solve
Operators	
`()(Matrix&, Vector&)`	Solve $Ax = b$.
Functions	
`template< Args... > Vector solve(Matrix&, Vector&, Args && ...)`	Solve $Ax = b$ with arbitrary configuration

PETSc uses inheritance as a code structure and interface tool. This is most clear in the case of its solver objects, as the linear (Krylov subspace) solvers *must* have an internal preconditioner (`PC`) object, and the non-linear (`SNES`) solvers has an internal `KSP` object. Analogously this applies also for `TS`. This is accomplished through explicit inheritance, as C has no syntactic support for it. As in C++, this implies that constructing a `KSP` object also implies constructing a `PC` object, and supports configuring `PC` through the `KSP` handle. This relationship has *partially* been carried over to the library, where the only implemented solver handle is `SNES`, even though the use case studied has only been relying on `KSP`. The reasoning is that there is no significant overhead when using `SNES` [6] and we get support for non-linearity *for free*, i.e. without the need for extra code. Because features were developed as needed, only linear solvers are implemented and tested properly.

There are two ways of using the solver: as a function and as a context object. For plenty use cases the simple expression `x = solve(A, b)` is sufficient, e.g.

there is no need for caching, re-using the preconditioned solution or overriding options from the options database. For more complicated systems, the solver can be more finely tuned and stored, at the cost of more and potentially less obvious code.

Both when used as a function and stored as an object, the solver uses overloading *type resolution* to determine what aspect is being configured. All configuration is done through the `set` member functions that are distinguished through their argument type, which was done for two reasons:

Safety and declarativity The compiler will ensure you pass a supported algorithm and that the configuration option is sane. Due to time constraint and a small scope, this has not been fully done so some run-time configuration issues can still occur, even though the framework is in place.

Arbitrary and unordered configuration By using the same basic symbol (`set`), combined with C++11 variadic templates, arbitrary signatures for one-shot configuration is supported.

The second point is accomplished through the function `solve` in Table 4.6. It uses C++ compile time recursive calls to transform a single explicit `solve` call to repeated applications of `solver.set`, followed by `operator()`, as demonstrated in Listing A.7. The `y = solve` line is transformed into code using the object.

4.9 Porting *upscale_relperm* to use PETSc

As little code as possible was modified when ported from the DUNE backend to the PETSc backend. Still, some assumptions of the *IncompFlowSolverHybrid* code do no longer apply, and had to be changed. Listing A.8 is the original call to `upscale` which *delegates* to a series of other functions, about 280 lines of code, that performs the required setup and calls to DUNE, while Listing A.9 provides the same functionality via PETSc and the library. The actual DUNE code is too verbose to print here, however, the `solveLinearSystem` functions

are printed in Listing A.12. The other significant difference is the matrix population in Listing 4.1.

As demonstrated, adapting code to use PETSc instead of DUNE will, unless there is some dependency to a very DUNE specific feature, be simple and easy and be mostly simple textual substitution. The entire porting took approximately 10 minutes, removed approximately 400 lines (from an 1800 line source file) of code and supports more algorithms, more run-time configurations, and includes experimental GPU acceleration.

4.10 Bugs

During development I discovered two critical bugs in upstream packages. One was when experimenting with GPU accelerated solvers, which exposed a memory leak in PETSc's ViennaCL container. The second was the *upscale_relperm* program not sending all the data to the master node. Both patches have been merged by the respective projects.

Listing 4.1 The port of matrix population in *IncompFlowSolverHybrid*. `diff` syntax.

```
@@ -1836,7 +1522,7 @@ namespace Opm {
     // equation of the form: a*x = a*p where 'p' is
     // the known pressure value (i.e., condval[r]).
     //
-    S_   [ii][ii] = S(r,r);
+    B.add( ii, ii, S( r, r ) ); // S_[ii][ii] = S(r,r);
     rhs_[ii]     = S(r,r) * condval[r];
     continue;
 case Periodic:
@@ -1859,13 +1545,13 @@ namespace Opm {
         const double a = S(r,r), b = a * condval[r];

         // Equation (1)
-        S_   [         ii][           ii] += a;
-        S_   [         ii][ppartner[r]] -= a;
+        B.add( ii, ii, a );
+        B.add( ii, ppartner[ r ], -a );
         rhs_[         ii]                 += b;

         // Equation (2)
-        S_   [ppartner[r]][           ii] -= a;
-        S_   [ppartner[r]][ppartner[r]] += a;
+        B.add( ppartner[ r ], ii, -a );
+        B.add( ppartner[ r ], ppartner[ r ], a );
         rhs_[ppartner[r]]                 -= b;
     }

@@ -1892,7 +1578,7 @@ namespace Opm {
                 jj = ppartner[c];
             }
         }
-        S_[ii][jj] += S(r,c);
+        B.add( ii, jj, S( r, c ) );
     }
     break;
 }
```

Listing 4.2 PETSc-ViennaCL patch. Git patch syntax.

```
The sub containers in the ViennaCL matrix were not free'd
due to inversed logic. Since delete on a nullptr won't do
anything we can unconditionally call delete on the
member pointers without checking for nullptr.
---
 src/mat/impls/aij/seq/seqviennacl/aijviennacl.cxx | 10
    ++++++----
 1 file changed, 6 insertions(+), 4 deletions(-)

diff --git
   a/src/mat/impls/aij/seq/seqviennacl/aijviennacl.cxx
   b/src/mat/impls/aij/seq/seqviennacl/aijviennacl.cxx
index d2af327..da1e348 100644
--- a/src/mat/impls/aij/seq/seqviennacl/aijviennacl.cxx
+++ b/src/mat/impls/aij/seq/seqviennacl/aijviennacl.cxx
@@ -357,10 +357,12 @@ PetscErrorCode
   MatDestroy_SeqAIJViennaCL(Mat A)

   PetscFunctionBegin;
   try {
-    if (!viennaclcontainer->tempvec)        delete
   viennaclcontainer->tempvec;
-    if (!viennaclcontainer->mat)            delete
   viennaclcontainer->mat;
-    if (!viennaclcontainer->compressed_mat) delete
   viennaclcontainer->compressed_mat;
-    delete viennaclcontainer;
+    if(viennaclcontainer) {
+        delete viennaclcontainer->tempvec;
+        delete viennaclcontainer->mat;
+        delete viennaclcontainer->compressed_mat;
+        delete viennaclcontainer;
+    }
     A->valid_GPU_matrix = PETSC_VIENNACL_UNALLOCATED;
   } catch(std::exception const & ex) {
     SETERRQ1(PETSC_COMM_SELF,PETSC_ERR_LIB,"ViennaCL
        error: %s", ex.what());
```

Listing 4.3 *upscale_relperm* patch. Git patch syntax.

```
In the case of both phases being upscaled, only one of
the tensors were sent, i.e. the send-buffer was 3 doubles
short. This causes program output to always be wrong in
the MPI version.
---
 examples/upscale_relperm.cpp | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/examples/upscale_relperm.cpp
    b/examples/upscale_relperm.cpp
index 1650df6..b2381bd 100644
--- a/examples/upscale_relperm.cpp
+++ b/examples/upscale_relperm.cpp
@@ -1631,7 +1631,7 @@ try
      for (int voigtIdx=0; voigtIdx < tensorElementCount;
          ++voigtIdx) {
          sendbuffer[2+tensorElementCount+voigtIdx] =
             Phase2Perm[idx][voigtIdx];
      }
-     MPI_Send(sendbuffer, 2+tensorElementCount,
    MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
+     MPI_Send(sendbuffer, 2+2*tensorElementCount,
    MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
 }
 else {
     double sendbuffer[2+tensorElementCount];
```

Chapter 5

Results and Measurements

This Chapter presents and discusses timing measurements and result differences between the DUNE- and PETSc backends.

5.1 Configuration

The tests were run on a the testing machine (Table 5.1). A Sandy Brige platform was chosen instead of a Haswell platform because of availability on the lab when I needed to run the measurements. PETSc was compiled from source, git revision `b0e188dedeffe8f0b01186133554504cb3c623ea`, with the makefile generated by the following configuration string:

```
./configure --with-fc=0 --with-blas-lapack-dir=/usr/
--with-open-mpi=yes --with-mpi-dir=/usr --prefix=/usr/local/
--with-opencl=yes --with-viennacl=yes --with-clanguage=c++
--with-hypre=yes --download-hypre=yes --with-debugging=0
COPTFLAGS="-O3 -march=native -mtune=native"
CXXOPTFLAGS="-O3 -std=c++11 -march=native -mtune=native"
```

All DUNE packages were the upstream master branched cloned 23-05-2015 and configured as the default release build `-DCMAKE_BUILD_TYPE=RELEASE`, which passes the compiler flags `CXX_FLAGS = -std=c++11 -O3 -DNDEBUG -std=c++11`, and all OPM packages, release `2015.04`, were configured with `-DCMAKE_BUILD_TYPE=RELEASE DUSE_MPI=ON`. The runs were orchestrated by

Table 5.1: Testing system hardware configuration.

System	
CPU	Intel Core i7 3930K (Sandy Bridge)
Frequency	3.20 GHz
Cores	6 (12 with HT)
Memory	32 GB
Instruction set	x86_64
GPU	AMD Radeon HD 7950 (Tahiti)
Video driver	fglrx-15.2
OS	Ubuntu 14.04
GCC	4.8.2

the program B.3. All algorithms are implementations by the PETSc team with the exception of *hypre*, which is **BoomerAMG** provided by the third party **Hypre** library.

The input data was a 46MB representative, typically sized heterolithic reservoir rock model.

5.2 Differences

The uperscaler performs an *approximation*, so small differences in output between numerical backends and algorithms are likely, but since configurations always approximate the same ideal solution, the expected *differences* between configurations are ideally small. Differences can stem from many factors, but the main two are floating point rounding errors and difference in numerical stability.

The upscaler outputs a report as in Figure 5.1, which forms an 8×30 matrix. I wrote the small program B.1 that reads the output of two runs, forms two matrices and calculates the differences between all entries $C = A - B$, so that $c_{ij} = |a_{ij} - b_{ij}|$. The first matrix A is always the arbitrarily chosen DUNE-driven ILU/BiCGStab solver, since the exact solution is unavailable.

```
######################################################################
# Results from upscaling relative permeability.
#
#          (MPI-version)
# Finished: Fri May 29 15:41:07 2015
# Hostname: hpc-lab-ruben
#
# Eclipse file: /home/jorgekva/Files/model.grdecl
#        cells: 444606
#  Pore volume: 0.0274506
#       volume: 0.10065
#     Porosity: 0.272733
#
# Stone 1: /home/jorgekva/Files/rock1.txt (41 points)
# Stone 2: /home/jorgekva/Files/rock2.txt (41 points)
# Stone 3: /home/jorgekva/Files/rock3.txt (41 points)
# Stone 4: /home/jorgekva/Files/rock4.txt (41 points)
#         jFunctionCurve: 4
#
# Timings:   Tesselation: 5.60263 secs
#              Upscaling: 374.077 secs (wallclock time)
#                         187.931 secs pr. saturation point
#              MPI-nodes: 12
#                Speedup: 10.2737, efficiency: 0.85614
#
# Options used:
#     Boundary conditions: Fixed (no-flow)
#                  points: 30
#         maxPermContrast: 1e7
#                 minPerm: 1e-12
#                 minPoro: 0.0001
#          surfaceTension: 11 dynes/cm
#                 gravity: 0
#
# Single phase permeability
#  |Kxx  Kxy  Kxz| = 30.603  0  0
#  |Kyx  Kyy  Kyz| = 0  32.3877  0
#  |Kzx  Kzy  Kzz| = 0  0  1.22139
#
######################################################################
#  Pc (Pa)        Sw          Krwxx        Krwyy        Krwzz        Kroxx        Kroyy        Krozz
   3.674e+05      0.3783      0.000        0.000        0.000        0.9998       0.9998       0.9999
   7.529e+04      0.3945      3.612e-06    9.626e-06    1.382e-06    0.9784       0.9926       0.6228
   3.880e+04      0.4107      7.757e-05    0.0001511    3.977e-05    0.9547       0.9822       0.3198
       9790.      0.4269      0.0003704    0.0004275    0.0008340    0.9220       0.9532       0.001182
       5016.      0.4431      0.0005488    0.0005411    0.005178     0.8417       0.8712       0.001106
       3524.      0.4593      0.0008153    0.0007778    0.01438      0.7473       0.7742       0.001089
       2675.      0.4755      0.001306     0.001236     0.02760      0.6525       0.6765       0.001046
       2094.      0.4917      0.002120     0.002005     0.04266      0.5624       0.5834       0.001035
       1659.      0.5079      0.003355     0.003179     0.05737      0.4799       0.4980       0.0009892
       1310.      0.5241      0.005076     0.004820     0.07006      0.4058       0.4211       0.0009223
       1022.      0.5403      0.007333     0.006980     0.08049      0.3408       0.3536       0.0008375
       775.0      0.5566      0.01014      0.009674     0.08904      0.2847       0.2952       0.0007107
       558.5      0.5728      0.01348      0.01290      0.09624      0.2366       0.2452       0.0005748
       363.8      0.5890      0.01734      0.01664      0.1025       0.1957       0.2026       0.0004142
       184.4      0.6052      0.02172      0.02090      0.1083       0.1607       0.1663       0.0002702
       14.83      0.6214      0.02663      0.02571      0.1137       0.1307       0.1351       0.0001609
      -149.7      0.6376      0.03214      0.03113      0.1190       0.1049       0.1084       9.213e-05
      -314.0      0.6538      0.03839      0.03730      0.1243       0.08260      0.08522      5.329e-05
      -482.6      0.6700      0.04554      0.04439      0.1299       0.06333      0.06529      3.716e-05
      -660.3      0.6862      0.05383      0.05265      0.1358       0.04693      0.04836      3.239e-05
      -852.5      0.7024      0.06360      0.06241      0.1422       0.03334      0.03434      3.085e-05
      -1066.      0.7186      0.07523      0.07408      0.1494       0.02253      0.02320      2.969e-05
      -1308.      0.7348      0.08923      0.08817      0.1576       0.01435      0.01478      2.978e-05
      -1592.      0.7510      0.1063       0.1054       0.1669       0.008497     0.008756     2.944e-05
      -1937.      0.7673      0.1272       0.1265       0.1777       0.004560     0.004704     2.892e-05
      -2375.      0.7835      0.1531       0.1528       0.1904       0.002123     0.002194     2.810e-05
      -2970.      0.7997      0.1858       0.1860       0.2055       0.0007952    0.0008245    2.636e-05
      -3880.      0.8159      0.2276       0.2284       0.2239       0.0002122    0.0002204    2.189e-05
      -5674.      0.8321      0.2827       0.2844       0.2469       3.044e-05    3.133e-05    1.049e-05
  -1.177e+06      0.8483      0.3615       0.3630       0.3114       0.000        0.000        0.000
```

Figure 5.1: Output from *upscale_relperm* with 12 MPI processes driven by PETSc.

Table 5.2: Difference between PETSc driven solvers and the DUNE ILU/BiCGStab base.

PC/KSP	Sum	Max
gamg/bcgsl	$1.890\,00 \times 10^{-6}$	$1.000\,00 \times 10^{-6}$
gamg/cg	$1.030\,20 \times 10^{-4}$	$1.000\,00 \times 10^{-4}$
gamg/gmres	$4.229\,00 \times 10^{-5}$	$1.000\,00 \times 10^{-5}$
hypre/bcgsl	$3.626\,00 \times 10^{-5}$	$6.000\,00 \times 10^{-6}$
hypre/cg	$1.518\,00 \times 10^{-5}$	$3.100\,00 \times 10^{-6}$
hypre/gmres	$4.080\,40 \times 10^{-4}$	$1.000\,00 \times 10^{-4}$
ilu/bcgsl	$2.772\,52 \times 10^{-3}$	$3.000\,00 \times 10^{-4}$
ilu/cg	$5.500\,00 \times 10^{-6}$	$2.000\,00 \times 10^{-6}$
jacobi/bcgsl	$3.321\,43 \times 10^{-2}$	$3.900\,00 \times 10^{-3}$
jacobi/cg	$0.000\,00$	$0.000\,00$

This leads to several measurements of difference:

Sum $s = \sum c_{ij}$

Max $max(c)$, the maximum element in the difference matrix.

Average $\frac{\sum c_{ij}}{6 \cdot 30}$, as the non-constant output of the upscaler is in the six rightmost columns.

Median The median of all *nonzero* elements.

Relative $r = \frac{c_{ij}}{b_{ij}}$, where ij is index of $max(c)$.

Differences between PETSc driven runs are compiled in Table 5.2 and Table 5.3, all which are sufficiently small (less than 1%) to consider the upscaling correct. Jacobi preconditioning and the conjugate gradient method has output identical to the reference output. All DUNE driven runs have identical output.

Table 5.3: Difference between PETSc driven solvers and the DUNE ILU/BiCGStab base.

PC/KSP	Average	Median	Relative
gamg/bcgsl	1.05000×10^{-8}	3.00000×10^{-8}	8.66551×10^{-4}
gamg/cg	5.72333×10^{-7}	1.00000×10^{-7}	7.03235×10^{-4}
gamg/gmres	2.34944×10^{-7}	7.00000×10^{-7}	1.03896×10^{-4}
hypre/bcgsl	2.01444×10^{-7}	1.00000×10^{-6}	6.44815×10^{-3}
hypre/cg	8.43333×10^{-8}	3.00000×10^{-7}	3.10652×10^{-3}
hypre/gmres	2.26689×10^{-6}	5.00000×10^{-6}	5.10986×10^{-4}
ilu/bcgsl	1.54029×10^{-5}	1.38700×10^{-5}	9.37207×10^{-4}
ilu/cg	3.05556×10^{-8}	2.00000×10^{-7}	1.73461×10^{-3}
jacobi/bcgsl	1.84524×10^{-4}	2.40300×10^{-4}	6.30150×10^{-3}
jacobi/cg	0.00000	0.00000	0.00000

5.3 Computation time

The immediate observation from elapsed computation time is the *massive* difference between algorithms, which by far has the most impact. In fact, some combinations of preconditioner and algorithm did *not* finish in reasonable time, which is the arbitrarily chosen cutoff at 900 minutes (15 hours) of execution time. As seen in the tables and figures of this section, many configurations are able to solve the problem in *minutes*, meaning 15 hours is simply unreasonable.

The timing measurements are simple executions of the *upscale_relperm* program, driven by program B.3, which executes some configurations and records the program output. The *upscale_relperm* reports the time it spends on upscaling, which is dominated by the numerical engine. This is a macro measurement, i.e. full program and not on a function- or component basis, so execution times have been rounded to the closest second, and multiple identical runs have not been done. This means numbers only *indicate* performance differences, and are not comprehensive descriptions of performance differences between algorithms and models. In fact, it measures the impact of change numerical backend on *total* program execution time, not the isolated execution

time of the solvers.

The processor in the testing system supports **hyper threading**, a proprietary Intel technology for simultaneous multithreading, which maps two logical cores per phyiscal core, which can improve execution time for some workloads. The testing system has 6 physical cores, and to investigate the effect of hyper threading, which also demonstrates some utilisation properties of the code (memory stalls etc.), the programs were also run with twice as many processes as cores. With DUNE's AMG/CG feature using more processes than cores actually slowed down the program, but this was only observed in this case.

Wallclock The **wallclock** time is the elapsed in seconds from when tesselation is done until the master node has received all computed values from all processes.

Upscaling The **upscaling** T_U is the *accumulated* average time T_p spent in the upscalign phase by each process p, so that $T_U = \frac{1}{i} \sum T_p$, where i is the number of upscaling points ($i = 30$ in my runs).

MPI The number of **MPI nodes** used in the run.

As seen in the data, the *combination* of preconditioner and Krylov subspace algorithm is paramount to program performance, regardless of backend. There is some performance to gain from using more processes than phyiscal cores, but this benefit is in no way linear. This is unsurprising since work is statically distributed between processes, which means that the process that gets assigned more saturation points (or just the slowest point in the case of one saturation point per process) dominates execution time.

For reference numbers, I performed single-core runs of the fastest DUNE configuration, the fastest PETSc configuration, as well as a single-core run of the PETSc configuration with GPU acceleration. As seen in Table 5.6 there was for this problem no improvement in running the solver with GPU acceleration.

Table 5.4: Time elapsed for PETSc driven runs. Lower is better.

PC/KSP	Wallclock (sec)	Upscaling (sec)	MPI nodes
hypre/gmres	374	188	12
hypre/cg	375	190	12
hypre/bcgsl	427	214	12
hypre/gmres	544	121	6
hypre/cg	582	118	6
hypre/bcgsl	638	129	6
ilu/cg	1592	752	12
ilu/cg	1756	381	6
gamg/cg	2063	1271	12
gamg/cg	2333	709	6
gamg/bcgsl	3031	1487	12
gamg/bcgsl	3280	753	6
jacobi/bcgsl	3635	1842	12
jacobi/cg	4011	2343	12
jacobi/bcgsl	4109	1002	6
jacobi/cg	4440	1272	6
gamg/gmres	4735	2351	12
ilu/bcgsl	4875	2316	12
ilu/bcgsl	5315	1113	6
gamg/gmres	5738	1215	6
ilu/gmres	-	-	12
ilu/gmres	-	-	6
jacobi/gmres	-	-	12
jacobi/gmres	-	-	6

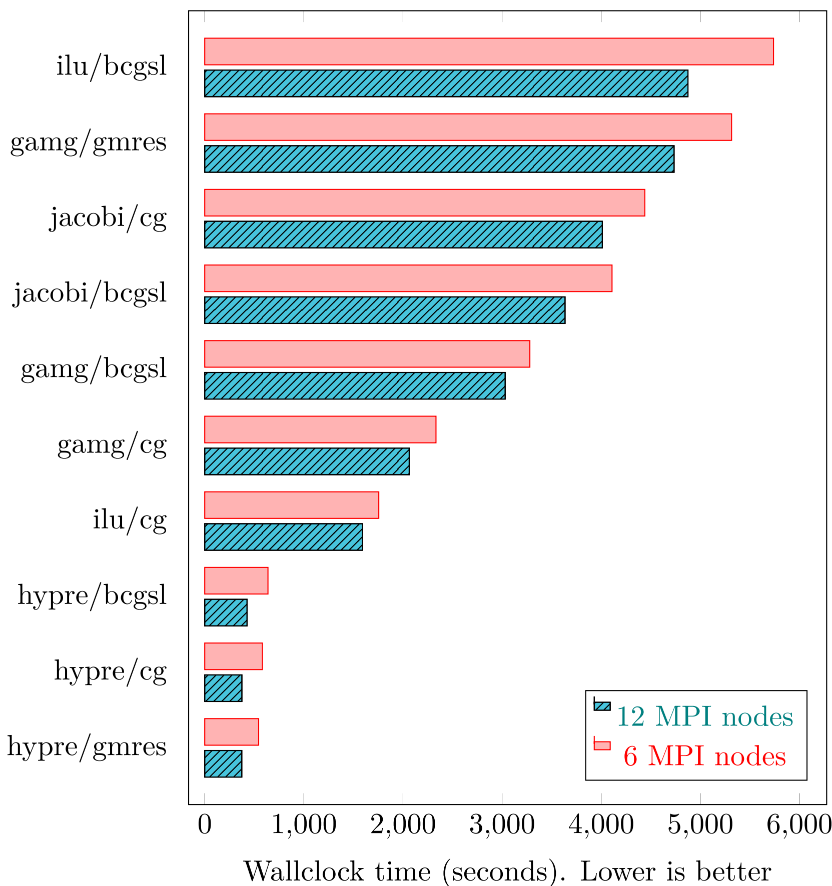

Figure 5.2: Time elapsed for PETSc driven runs.

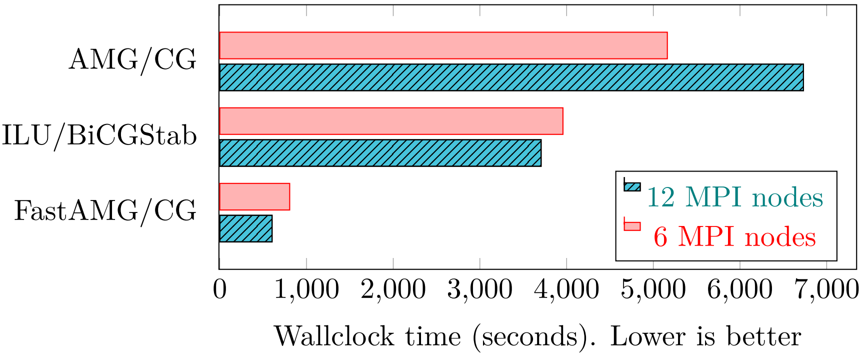

Figure 5.3: Time elapsed for DUNE driven runs.

Table 5.5: Time elapsed for DUNE driven runs. Lower is better.

PC/KSP	**Wallclock (sec)**	**Upscaling (sec)**	**MPI nodes**
FastAMG/CG	603	313	12
FastAMG/CG	806	170	6
ILU/BiCGStab	3706	1807	12
ILU/BiCGStab	3958	921	6
AMG/CG	5162	2028	6
AMG/CG	6732	3320	12
KAMG/CG	-	-	12
KAMG/CG	-	-	6

Table 5.6: Time elapsed for single process runs. Lower is better.

PC/KSP	**Wallclock (sec)**	**Upscaling (sec)**
FastAMG/CG	3019	101
hypre/gmres (CPU)	2275	76
hypre/gmres (GPU)	2390	80

As a final remark, considering scalability in the fastest case (hypre/gmres), we can see an approximate 6x speedup with 12 processes and an approximate 4.1x speedup with 6 processes compared to the single process run. The processor still only had 6 phyiscal cores, but with the aid of hyper threading it was able to achieve linear speedup due to the highly parallel nature of the problem given a model of this size.

Chapter 6
Conclusions and Future Work

The work for this thesis has been very practical in nature, with some clear contributions to the OPM project. This final Chapter briefly concludes and reflects upon the contributions and results, and suggestions for future work.

6.1 Contributions

The contributions of this thesis are:

- Free-licenced code that enables OPM to support a vast array of algorithms and data structures with a safe and easy-to-use interface to enhance and further develop reservoir simulator tools.
- A proposal for the design of a generalised interface of linear algebra and numerical methods, including access to vectors, matrices and solvers, which can abstract several underlying implementations.
- A demonstration of achievable performance with PETSc on a real-world data set on a regular workstation system.
- Experimental support for GPU accelerated numerical solvers through the PETSc library and its plugins.

6.2 Conclusions

By using BoomerAMG we managed to calculate the relative permeability of the model approximately 48% faster than the upstream program that uses DUNE, with a much simpler code. While the solution is simple, it relies on an extensive support library that retwrites parts of PETSc's interface. However, some of the complexity of the linear solver library is now moved from application code into a library, which means that development and porting of applications is easier and faster.

Additionally, the integration of PETSc with OPM includes many debugging and profiling utilities, which could be used to further enhance, improve and develop new solutions, solvers and simulators.

Our results demonstrates that there are multiple benefits to have several solvers available, as their performance and correctness vary between problems. It indicates that Hypre's BoomerAMG is well suited for the *upscale* programs, producing results within reasonable difference from the reference DUNE driven solver in less time. The experiments also show that for the *upscale* programs, GPU accelerated linear solvers are unlikely to provide any speedup for upscaling.

6.3 Future Work

This section briefly presents some suggestions for future work on the PETSc integration with OPM and for generalised OPM numerics:

Strict container types Introduce more and stronger vector types that encode assumptions into data structures. Like `std::vector` is templated on an inner type, `Vector` can be transformed to `Vector<double>` etc. Additionally, this can be used to enforce certain properties regarding sparsity patterns which algorithms can exploit.

Builders Develop and refine more builders, both for matrices and vectors.

Higher order functions Develop support for higher order functions on all containers, as well as support option types.

MPI/parallel aware interfaces Refine or define interfaces that support parallel aware code, such as first class support for internally parallel containers and an elegent interface for distribution population of containers.

Flow simulator Port the flow simulator and other OPM applications to use the new backend.

GPU experiments Perform more experiments with GPU accelerated numerics, using different data sets and different algorithms.

References

[1] Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (New York, NY, USA, 1967), AFIPS '67 (Spring), ACM, pp. 483–485.

[3] Arnoldi, W. E. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Q. Appl. Math 9*, 17 (1951), 17–29.

[4] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F., and Zhang, H. PETSc Web page. http://www.mcs.anl.gov/petsc, 2014.

[5] Blatt, M., and Bastian, P. The iterative solver template library. In *Applied Parallel Computing. State of the Art in Scientific Computing* (2007), B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, Eds., vol. 4699 of *Lecture Notes in Computer Science*, Springer, pp. 666–675.

[6] Brown, J., and Knepley, M. G. PETSc tutorial at SUNY Buffalo. http://www.mcs.anl.gov/petsc/documentation/tutorials/BuffaloTutorial.pdf, 2014.

[7] Butenko, S., and Pardalos, P. M. *Numerical methods and optimization. An introduction.* Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series. Boca Raton, FL, 2014.

[8] Byun, J.-H., Lin, R., Yelick, K. A., and Demmel, J. Autotuning sparse matrix-vector multiplication for multicore. Tech. Rep. UCB/EECS-2012-215, EECS Department, University of California, Berkeley, Nov 2012.

[9] Falgout, R. D. An introduction to algebraic multigrid. *Computing in Science and Engg. 8*, 6 (Nov. 2006), 24–33.

[10] Falgout, R. D., and Yang, U. M. hypre: a library of high performance preconditioners. In *Preconditioners," Lecture Notes in Computer Science* (2002), pp. 632–641.

[11] Gustafson, J. L. Reevaluating Amdahl's Law. *Communications of the ACM 31* (1988), 532–533.

[12] Hestenes, M. R., and Stiefel, E. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards 49*, 6 (Dec. 1952), 409–436.

[13] Hill, M. D., and Marty, M. R. Amdahl's law in the multicore era. *IEEE COMPUTER* (2008).

[14] ISO/EIC JTC1/SC22/WG21. C++11 standard. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, May 2014. [Online; accessed 08-May-2015].

[15] Kreyszig, E. *Advanced engineering mathematics.* John Wiley, Dec. 2006.

[16] Lawson, C. L., Hanson, R. J., Krogh, F. T., and Kincaid, D. R. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw. 5*, 3 (Sept. 1979), 324–325.

[17] Liskov, B. H., and Wing, J. M. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16* (1994), 1811–1841.

[18] Message Passing Interface Forum. Mpi: A message-passing interface standard. Specification, Message Passing Interface Forum, Knoxville, TN, USA, 1994.

[19] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification, Message Passing Interface Forum, Sept. 2009.

[20] Meyer, J. C. *Performance Modeling of Heterogeneous Systems.* PhD thesis, Norwegian University of Science and Technology, Trondheim, Nov. 2012.

[21] Netlib. Netlib LAPACK Web page. http://www.netlib.org/lapack/.

[22] NVIDIA. *Accelerating ANSYS Fluent 15.0 using NVIDIA GPUs.* NVIDIA, June 2014.

[23] Press, W. H., Teukolsky, S. A., and Vett, W. T. *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, 2007.

[24] Saad, Y., and Schultz, M. H. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput. 7*, 3 (July 1986), 856–869.

[25] Shewchuk, J. R. An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[26] Skolmedal, A. B. Profiling, optimization and parallelization of seismic inversion code. Master's thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, 2011.

[27] Stinessen, B. O. Profiling, optimization and parallelization of seismic inversion code. Master's thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, 2011.

[28] Stroustrup, B. C++ faq. http://web.archive.org/web/20080617183013/ http://www.research.att.com/~bs/bs_faq.html#C-is-subset, Feb. 2008. [Online; accessed 08-May-2015].

[29] Stroustrup, B. *The C++ Programming Language*, 4th ed. Addison-Wesley Professional, 2013.

[30] Stroustrup, B. Call syntax: x.f(y) vs. f(x,y). https://isocpp.org/files/papers/N4174.pdf, Oct. 2014.

[31] Stroustrup, B. *Programming: Principles and Practice Using C++ (2nd Edition)*, 2nd ed. Addison-Wesley Professional, 2014.

[32] Sun, X.-H., and Chen, Y. Reevaluating Amdahl's Law in the Multicore Era. *J. Parallel Distrib. Comput. 70*, 2 (Feb. 2010), 183–188.

[33] SUTTER, H. Unified call syntax. https://isocpp.org/files/papers/N4165.pdf, Oct. 2014.

[34] VELDHUIZEN, T. L. C++ templates are turing complete. Tech. rep., Indiana University, 2003.

[35] YANG, U. M. Parallel algebraic multigrid methods—high performance preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer, 2006, pp. 209–236.

[36] YOUNG, D. M. *Iterative Solution of Large Linear Systems*. Dover Books on Mathematics Series. Dover Publications, 2003.

[37] ZHANG, F. *The Schur Complement and Its Applications*, vol. 4 of *Numerical Methods and Algorithms*. Springer, 2005.

Appendix A

Code snippets

This appendix contains various relevant code snippets from the library or the port of *upscale_relperm*. Full source code for the PETSc integration or *upscale_relperm* is either bundled with this thesis or can be found via my forks and pull requests at https://github.com/jorgekva.

Listing A.1 Definition of the `uptr` resource manager.

```
template< typename T >
class uptr< T* > :
        private std::unique_ptr< T, deleter< T > > {

    private:
        typedef T* pointer;
        typedef std::unique_ptr< T, deleter< T > > base;

    public:
        uptr< T* >(uptr< T* >&&);
        uptr< T* >(T*);

        operator pointer() const;

    protected:
        inline pointer ptr() const;

        void swap(uptr&);
};
```

Listing A.2 Vector object used as `Vec` handle in a PETSc function.

```
Vector v = get_vector();
/* some operations */
VecConjugate(v);
/* v is now conjugated */
```

Listing A.3 Creating a managed resource from a PETSc handle.

```
// uptr< T* >
class Vector : public uptr< Vec > {};
// uptr< T >
class Vector : public uptr< _p_Vec > {};
```

Listing A.4 Deleter for a managed resource.

```
template<>
struct deleter< _p_Vec >
{ void operator()(Vec x) { VecDestroy(&x); } };
```

Listing A.5 Deleter for a managed resource with C++11 `decltype`.

```
template<>
struct deleter< decltype(*Vec) >
{ void operator()(Vec x) { VecDestroy(&x); } };
```

Listing A.6 Value oriented `dot`

```
Vector::scalar dot(const Vector& lhs, const Vector& rhs)
{
    Vector::scalar x;
    VecDot(lhs, rhs, &x);
    return x;
}
```

Listing A.7 Variadic arguments to `solve`

```
/* no explicit configuration */
auto x = solve(A, b);

/* argument order (after b) does not matter */
auto y = solve(A, b, Ksp_type( "cg" ), Pc_type( "ilu" ));
auto z = solve(A, b, Pc_type( "ilu" ), Ksp_type( "cg" ));

```

```
8  /* using object handle */
9  Solver s;
10 s    .set(Ksp_type( "cg" ))
11      .set(Pc_type( "ilu" ));
12 auto v = s( A, b );
```

Listing A.8 Solving with DUNE. Algorithm setup (approx. 300 lines) are omitted.

```
1  {
2  assembleDynamic(r, sat, bc, src);
3  switch (linsolver_type) {
4  case 0: // ILU0 preconditioned CG
5    solveLinearSystem(residual_tolerance,
        linsolver_verbosity, linsolver_maxit);
6      break;
7  case 1: // AMG preconditioned CG
8      solveLinearSystemAMG(residual_tolerance,
          linsolver_verbosity,
9      linsolver_maxit, prolongate_factor, same_matrix,
          smooth_steps);
10     break;
11
12 case 2: // KAMG preconditioned CG
13     solveLinearSystemKAMG(residual_tolerance,
          linsolver_verbosity,
14     linsolver_maxit, prolongate_factor,
          same_matrix,smooth_steps);
15     break;
16 case 3: // CG preconditioned with AMG that uses less
      memory badwidth
17 #if defined(HAS_DUNE_FAST_AMG) ||
      DUNE_VERSION_NEWER(DUNE_ISTL, 2, 3)
18     solveLinearSystemFastAMG(residual_tolerance,
          linsolver_verbosity,
19     linsolver_maxit, prolongate_factor,
          same_matrix,smooth_steps);
20 #else
21 if(linsolver_verbosity)
22    std::cerr<<"Fast AMG is not available; falling back
         to CG preconditioned with the normal
         one."<<std::endl;
```

```
23 solveLinearSystemAMG(residual_tolerance,
       linsolver_verbosity, linsolver_maxit,
24 prolongate_factor, same_matrix, smooth_steps);
25 #endif
26         break;
27 default:
28         std::cerr << "Unknown linsolver_type: " <<
               linsolver_type << '\n';
29         throw std::runtime_error("Unknown linsolver_type");
30         computePressureAndFluxes(r, sat);
31 }
```

Listing A.9 Solving with PETSc.

```
1
2  {
3  auto A = assembleDynamic(r, sat, bc, src);
4
5  Petsc::Solver::Linear_tolerance ltol;
6  ltol.relative_tolerance = residual_tolerance;
7  ltol.absolute_tolerance = 1e-05;
8  ltol.maximum_iterations = linsolver_maxit > 0 ?
       linsolver_maxit : A.rows();
9
10
11 Petsc::Vector b(this->rhs_);
12
13 auto x = Petsc::solve(A, b, ltol);
14
15 /*
16  * computePressureAndFluxes expects a std::vector, not a
17  * Petsc::Vector
18  */
19 Petsc::Vector::scalar* raw_arr;
20 VecGetArray(x, &raw_arr);
21 soln_.resize(x.size());
22 soln_.insert(soln_.begin(), raw_arr, raw_arr + x.size());
23 VecRestoreArray(x, &raw_arr);
24
25 computePressureAndFluxes(r, sat);
26 }
```

Listing A.10 Solving a linear system with OPM-PETSc

```
using namespace Opm::Petsc;

Matrix create_matrix() {
    Matrix::Builder::Bnserter builder(2, 2);
    builder.insert(0, 0, 2);
    builder.insert(1, 1, 2);
    return commit(builder);
}
Vector create_vector() {
    std::vector< Vector::scalar > vec(2);
    vec[ 0 ] = 2; vec[ 1 ] = 1;
    return vec; // implicit conversion to Petsc::Vector
}
void solve() {
    auto A = create_matrix();
    auto b = create_vector();
    // [ 2, 0 ] [ x1 ] = [ 2 ]
    // [ 0, 2 ] [ x2 ] = [ 1 ]
    auto x = solve(A, b);
}
```

Listing A.11 Solving a linear system with PETSc.

```
Mat create_matrix() {
    Mat mat;
    MatCreate(PETSC_COMM_WORLD, &mat);
    MatSetFromOptions(mat);

    MatSetSizes(mat, PETSC_DECIDE, PETSC_DECIDE, 2, 2);
    MatSetUp(mat);

    MatSetValues(Mat, 1, { 0 }, 1, { 0 }, { 2 },
        INSERT_VALUES);
    MatSetValues(Mat, 1, { 1 }, 1, { 1 }, { 2 },
        INSERT_VALUES);
    MatAssemblyBegin(mat, MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(mat, MAT_FINAL_ASSEMBLY);

    return mat;
}
```

```
16 Vec create_vector() {
17     Vec vec;
18     VecCreate(PETSC_COMM_WORLD, &vec);
19     VecSetFromOptions(vec);
20 
21     VecSetSizes(vec, PETSC_DECIDE, 2);
22     VecSetValues(vec, 2, { 0, 1 }, { 2, 1 },
           INSERT_VALUES);
23     VecAssemblyBegin(vec);
24     VecAssemblyEnd(vec);
25 
26     return vec;
27 }
28 void solve() {
29     Mat A = create_matrix();
30     Vec b = create_matrix();
31 
32     KSP ksp;
33     KSPCreate(PETSC_COMM_WORLD, &ksp);
34     KSPSetFromOptions(ksp);
35     KSPSetOperators(ksp, A, A);
36 
37     Vec x;
38     VecDuplicate(vec, &x);
39     KSPSolve(ksp, b, x); // solution is now in x
40 
41     KSPDestroy(&ksp);
42     MatDestroy(&mat);
43     VecDestroy(&b);
44     VecDestroy(&x);
45 }
```

Listing A.12 DUNE code removed from *upscale_relperm*

```
1386 // ----------------------------------------------------
1387 void solveLinearSystem(double residual_tolerance, int
         verbosity_level, int maxit)
1388 // ----------------------------------------------------
1389 {
1390     // Adapted from DuMux...
1391     Scalar residTol = residual_tolerance;
1392 
```

```
1393        typedef Dune::BCRSMatrix <MatrixBlockType>
               Matrix;
1394        typedef Dune::BlockVector<VectorBlockType>
               Vector;
1395        typedef Dune::MatrixAdapter<Matrix,Vector,Vector>
               Adapter;
1396
1397        // Regularize the matrix (only for pure Neumann
               problems...)
1398        if (do_regularization_) {
1399            S_[0][0] *= 2;
1400        }
1401        Adapter opS(S_);
1402
1403        // Construct preconditioner.
1404        Dune::SeqILU0<Matrix,Vector,Vector> precond(S_, 1.0);
1405
1406        // Construct solver for system of linear equations.
1407        Dune::CGSolver<Vector> linsolve(opS, precond,
               residTol,
1408                                          (maxit>0)?maxit:S_.N(),
                                                 verbosity_level);
1409
1410        Dune::InverseOperatorResult result;
1411        soln_ = 0.0;
1412
1413        // Solve system of linear equations to recover
1414        // face/contact pressure values (soln_).
1415        linsolve.apply(soln_, rhs_, result);
1416        if (!result.converged) {
1417            OPM_THROW(std::runtime_error, "Linear solver
                   failed to converge in " << result.iterations
                   << " iterations.\n"
1418                  << "Residual reduction achieved is " <<
                         result.reduction << '\n');
1419        }
1420    }
1421
1422
1423
1424    // ------------------ AMG typedefs --------------------
```

```
1425
1426 // Representation types for linear system.
1427 typedef Dune::BCRSMatrix <MatrixBlockType>        Matrix;
1428 typedef Dune::BlockVector<VectorBlockType>        Vector;
1429 typedef Dune::MatrixAdapter<Matrix,Vector,Vector>
        Operator;
1430
1431 // AMG specific types.
1432 // Old:   FIRST_DIAGONAL 1, SYMMETRIC 1, SMOOTHER_ILU 1,
        ANISOTROPIC_3D 0
1433 // SPE10: FIRST_DIAGONAL 0, SYMMETRIC 1, SMOOTHER_ILU 0,
        ANISOTROPIC_3D 1
1434 #ifndef FIRST_DIAGONAL
1435 #define FIRST_DIAGONAL 1
1436 #endif
1437 #ifndef SYMMETRIC
1438 #define SYMMETRIC 1
1439 #endif
1440 #ifndef SMOOTHER_ILU
1441 #define SMOOTHER_ILU 1
1442 #endif
1443 #ifndef SMOOTHER_BGS
1444 #define SMOOTHER_BGS 0
1445 #endif
1446 #ifndef ANISOTROPIC_3D
1447 #define ANISOTROPIC_3D 0
1448 #endif
1449
1450 #if FIRST_DIAGONAL
1451 typedef Dune::Amg::FirstDiagonal CouplingMetric;
1452 #else
1453 typedef Dune::Amg::RowSum        CouplingMetric;
1454 #endif
1455
1456 #if SYMMETRIC
1457 typedef Dune::Amg::SymmetricCriterion< Matrix,
        CouplingMetric > CriterionBase;
1458 #else
1459 typedef Dune::Amg::UnSymmetricCriterion< Matrix,
        CouplingMetric > CriterionBase;
1460 #endif
```

```
1461
1462 #if SMOOTHER_BGS
1463 typedef Dune::SeqOverlappingSchwarz<Matrix, Vector,
        Dune::MultiplicativeSchwarzMode> Smoother;
1464 #else
1465 #if SMOOTHER_ILU
1466 typedef Dune::SeqILU0<Matrix,Vector,Vector>
        Smoother;
1467 #else
1468 typedef Dune::SeqSSOR<Matrix,Vector,Vector>
        Smoother;
1469 #endif
1470 #endif
1471 typedef Dune::Amg::CoarsenCriterion<CriterionBase>
        Criterion;
1472
1473
1474 // --------- storing the AMG operator and preconditioner
        --------
1475 boost::scoped_ptr<Operator> opS_;
1476 typedef Dune::Preconditioner< Vector, Vector>
        PrecondBase;
1477 boost::scoped_ptr<PrecondBase> precond_;
1478
1479
1480 // -----------------------------------------------------
1481 void solveLinearSystemAMG(double residual_tolerance, int
        verbosity_level,
1482                           int maxit, double
                                 prolong_factor, bool
                                 same_matrix, int
                                 smooth_steps)
1483 // -----------------------------------------------------
1484 {
1485     typedef Dune::Amg::AMG< Operator, Vector, Smoother,
            Dune::Amg::SequentialInformation >
1486          Precond;
1487
1488     // Adapted from upscaling.cc by Arne Rekdal, 2009
1489     Scalar residTol = residual_tolerance;
1490
```

```
1491      if (!same_matrix) {
1492          // Regularize the matrix (only for pure Neumann
                 problems...)
1493          if (do_regularization_) {
1494              S_[0][0] *= 2;
1495          }
1496          opS_.reset(new Operator(S_));
1497
1498          // Construct preconditioner.
1499          double relax = 1;
1500          typename Precond::SmootherArgs smootherArgs;
1501          smootherArgs.relaxationFactor = relax;
1502  #if SMOOTHER_BGS
1503          smootherArgs.overlap =
                 Precond::SmootherArgs::none;
1504          smootherArgs.onthefly = false;
1505  #endif
1506          Criterion criterion;
1507          criterion.setDebugLevel(verbosity_level);
1508  #if ANISOTROPIC_3D
1509          criterion.setDefaultValuesAnisotropic(3, 2);
1510  #endif
1511          criterion.setProlongationDampingFactor(prolong_factor);
1512          criterion.setBeta(1e-10);
1513          precond_.reset(new Precond(*opS_, criterion,
                 smootherArgs,
1514                         1, smooth_steps, smooth_steps));
1515      }
1516      // Construct solver for system of linear equations.
1517      Dune::CGSolver<Vector> linsolve(*opS_,
             dynamic_cast<Precond&>(*precond_), residTol,
             (maxit>0)?maxit:S_.N(), verbosity_level);
1518
1519      Dune::InverseOperatorResult result;
1520      soln_ = 0.0;
1521      // Adapt initial guess such Dirichlet boundary
             conditions are
1522      // represented, i.e. soln_i=A_{ii}^-1 rhs_i
1523      typedef typename Dune::BCRSMatrix
             <MatrixBlockType>::ConstRowIterator RowIter;
```

```
1524      typedef typename Dune::BCRSMatrix
              <MatrixBlockType>::ConstColIterator ColIter;
1525      for(RowIter ri=S_.begin(); ri!=S_.end(); ++ri){
1526          bool isDirichlet=true;
1527          for(ColIter ci=ri->begin(); ci!=ri->end(); ++ci)
1528              if(ci.index()!=ri.index() && *ci!=0.0)
1529                  isDirichlet=false;
1530          if(isDirichlet)
1531              soln_[ri.index()] = rhs_[ri.index()] /
                      S_[ri.index()][ri.index()];
1532      }
1533      // Solve system of linear equations to recover
1534      // face/contact pressure values (soln_).
1535      linsolve.apply(soln_, rhs_, result);
1536      if (!result.converged) {
1537          OPM_THROW(std::runtime_error, "Linear solver
              failed to converge in " << result.iterations
              << " iterations.\n"
1538                 << "Residual reduction achieved is " <<
                        result.reduction << '\n');
1539      }
1540
1541  }
1542
1543  #if defined(HAS_DUNE_FAST_AMG) ||
          DUNE_VERSION_NEWER(DUNE_ISTL, 2, 3)
1544
1545  // ----------------------------------------------------
1546  void solveLinearSystemFastAMG(double residual_tolerance,
          int verbosity_level,
1547                                int maxit, double
                                        prolong_factor, bool
                                        same_matrix, int
                                        smooth_steps)
1548  // ----------------------------------------------------
1549  {
1550      typedef Dune::Amg::FastAMG<Operator, Vector> Precond;
1551
1552      // Adapted from upscaling.cc by Arne Rekdal, 2009
1553      Scalar residTol = residual_tolerance;
1554
```

```
1555     if (!same_matrix) {
1556         // Regularize the matrix (only for pure Neumann
             problems...)
1557         if (do_regularization_) {
1558             S_[0][0] *= 2;
1559         }
1560         opS_.reset(new Operator(S_));
1561 
1562         // Construct preconditioner.
1563         typedef Dune::Amg::AggregationCriterion<
             Dune::Amg::SymmetricMatrixDependency< Matrix,
             CouplingMetric > > CriterionBase;
1564 
1565         typedef Dune::Amg::CoarsenCriterion<
             CriterionBase > Criterion;
1566         Criterion criterion;
1567         criterion.setDebugLevel(verbosity_level);
1568 #if ANISOTROPIC_3D
1569         criterion.setDefaultValuesAnisotropic(3, 2);
1570 #endif
1571         criterion.setProlongationDampingFactor(prolong_factor);
1572         criterion.setBeta(1e-10);
1573         Dune::Amg::Parameters parms;
1574         parms.setDebugLevel(verbosity_level);
1575         parms.setNoPreSmoothSteps(smooth_steps);
1576         parms.setNoPostSmoothSteps(smooth_steps);
1577         precond_.reset(new Precond(*opS_, criterion,
             parms));
1578     }
1579     // Construct solver for system of linear equations.
1580     Dune::GeneralizedPCGSolver<Vector> linsolve(*opS_,
         dynamic_cast<Precond&>(*precond_), residTol,
         (maxit>0)?maxit:S_.N(), verbosity_level);
1581 
1582     Dune::InverseOperatorResult result;
1583     soln_ = 0.0;
1584 
1585     // Adapt initial guess such Dirichlet boundary
         conditions are
1586     // represented, i.e. soln_i=A_{ii}^-1 rhs_i
```

```
1587      typedef typename Dune::BCRSMatrix
             <MatrixBlockType>::ConstRowIterator RowIter;
1588      typedef typename Dune::BCRSMatrix
             <MatrixBlockType>::ConstColIterator ColIter;
1589      for(RowIter ri=S_.begin(); ri!=S_.end(); ++ri){
1590          bool isDirichlet=true;
1591          for(ColIter ci=ri->begin(); ci!=ri->end(); ++ci)
1592              if(ci.index()!=ri.index() && *ci!=0.0)
1593                  isDirichlet=false;
1594          if(isDirichlet)
1595              soln_[ri.index()] = rhs_[ri.index()] /
                     S_[ri.index()][ri.index()];
1596      }
1597      // Solve system of linear equations to recover
1598      // face/contact pressure values (soln_).
1599      linsolve.apply(soln_, rhs_, result);
1600      if (!result.converged) {
1601          OPM_THROW(std::runtime_error, "Linear solver
                 failed to converge in " << result.iterations
                 << " iterations.\n"
1602                 << "Residual reduction achieved is " <<
                        result.reduction << '\n');
1603      }
1604
1605  }
1606  #endif
1607
1608  // ------------------------------------------------------
1609  void solveLinearSystemKAMG(double residual_tolerance,
         int verbosity_level,
1610                              int maxit, double
                                     prolong_factor, bool
                                     same_matrix, int
                                     smooth_steps)
1611  // ------------------------------------------------------
1612  {
1613
1614      typedef Dune::Amg::KAMG<Operator,Vector, Smoother,
             Dune::Amg::SequentialInformation,
1615                          Dune::CGSolver<Vector> >   Precond;
1616      // Adapted from upscaling.cc by Arne Rekdal, 2009
```

```
1617      Scalar residTol = residual_tolerance;
1618      if (!same_matrix) {
1619          // Regularize the matrix (only for pure Neumann
              problems...)
1620          if (do_regularization_) {
1621               S_[0][0] *= 2;
1622          }
1623          opS_.reset(new Operator(S_));
1624
1625          // Construct preconditioner.
1626          double relax = 1;
1627          typename Precond::SmootherArgs smootherArgs;
1628          smootherArgs.relaxationFactor = relax;
1629 #if SMOOTHER_BGS
1630          smootherArgs.overlap =
              Precond::SmootherArgs::none;
1631          smootherArgs.onthefly = false;
1632 #endif
1633          Criterion criterion;
1634          criterion.setDebugLevel(verbosity_level);
1635 #if ANISOTROPIC_3D
1636          criterion.setDefaultValuesAnisotropic(3, 2);
1637 #endif
1638          criterion.setProlongationDampingFactor(prolong_factor);
1639          criterion.setBeta(1e-10);
1640          precond_.reset(new Precond(*opS_, criterion,
              smootherArgs, 2, smooth_steps, smooth_steps));
1641      }
1642      // Construct solver for system of linear equations.
1643      Dune::CGSolver<Vector> linsolve(*opS_,
          dynamic_cast<Precond&>(*precond_), residTol,
          (maxit>0)?maxit:S_.N(), verbosity_level);
1644
1645      Dune::InverseOperatorResult result;
1646      soln_ = 0.0;
1647      // Adapt initial guess such Dirichlet boundary
          conditions are
1648      // represented, i.e. soln_i=A_{ii}^-1 rhs_i
1649      typedef typename Dune::BCRSMatrix
          <MatrixBlockType>::ConstRowIterator RowIter;
```

```
1650      typedef typename Dune::BCRSMatrix
              <MatrixBlockType>::ConstColIterator ColIter;
1651      for(RowIter ri=S_.begin(); ri!=S_.end(); ++ri){
1652          bool isDirichlet=true;
1653          for(ColIter ci=ri->begin(); ci!=ri->end(); ++ci)
1654              if(ci.index()!=ri.index() && *ci!=0.0)
1655                  isDirichlet=false;
1656          if(isDirichlet)
1657              soln_[ri.index()] = rhs_[ri.index()] /
                      S_[ri.index()][ri.index()];
1658      }
1659      // Solve system of linear equations to recover
1660      // face/contact pressure values (soln_).
1661      linsolve.apply(soln_, rhs_, result);
1662      if (!result.converged) {
1663          OPM_THROW(std::runtime_error, "Linear solver
                 failed to converge in " << result.iterations
                 << " iterations.\n"
1664                  << "Residual reduction achieved is " <<
                         result.reduction << '\n');
1665      }
1666
1667  }
```

Appendix B
Programs

This section includes some small utility programs used to compile reports, extract data and preprocess data for this thesis.

Listing B.1 matrix-diff.hs: output differences of *upscale_relperm* runs.

```haskell
import Numeric.LinearAlgebra
import Control.Monad (replicateM)
import Data.Maybe
import Data.List (sort)
import Text.Read (readMaybe)
import Text.Printf

merge :: [a] -> [a] -> [a]
merge xs ys = concatMap (\(x,y) -> [x,y]) $ zip xs ys

mkmatrix :: IO (Matrix Double)
mkmatrix = fmap fromLists . replicateM 30 $ fmap lineToDouble getLine
        where   lineToDouble = mapMaybe (readMaybe . fixDouble ) . words
                fixDouble ('.':[]) = ".0"
                fixDouble (x:xs)    = x : fixDouble xs
                fixDouble []        = []
              -- upscale_relperm does not output a trailing zero, this must be
              -- added for readMaybe to not discard the entry

main :: IO ()
main = do
    mtx1 <- mkmatrix
    mtx2 <- mkmatrix

    let diffmtx = cmap abs $ mtx1 - mtx2
    let nonzeros = 0 : (filter (> 0) . sort . concat $ toLists diffmtx)

    let maxdiff = maximum nonzeros
    let reldiff = maxdiff /  (atIndex mtx2 $ maxIndex diffmtx)
    let sumdiff = sum nonzeros
    let avgdiff = sumdiff / (6 * 30)
    let median = head $ drop (div (length nonzeros) 2) nonzeros

    let rounded = map (printf "%.5e") [sumdiff, maxdiff, avgdiff, median, reldiff]
    putStrLn . concat $ merge ("" : (repeat ",")) rounded
```

Listing B.2 make-result-tables.sh: compile timing tables.

```
#! /bin/bash

if [ $# -lt 1 ]; then
    echo "Need argument 'dune' or 'petsc'"; exit;
fi

basefile=data/baseline-ILU-BiCGStab-out.txt
backend=$1

diffpath=figs/diff-$backend-data.csv
diffdata="Alg,Sum,Max,Average,Median,Relative"

timepath=figs/time-$backend-data.csv
timefull=figs/time-$backend-full.csv
timedata="Alg,Wallclock,Upscaling,MPI"

for node in 6 12; do
    for f in data/$backend-*-$node-out.txt; do
        if [ ! -e $f ]; then continue; fi
        # extract name of pc and ksp
        farray=(${f//-/ })
        pc=${farray[1]}
        ksp=${farray[2]}

        ddata=$(cat <(tail -n 30 $basefile) <(tail -n 30 $f) | ./matrix)
        tdata=$(tail -n 51 $f | head -n 3 | ./extract-timing.pl)
        diffdata="$diffdata\n$pc/$ksp,$ddata"
        timedata="$timedata\n$pc/$ksp$tdata"
    done
done

echo -e $diffdata | sort | uniq | tee $diffpath
echo -e $timedata | sort -g -t, -k2 | tee $timepath $timefull
grep "[MPI|12]$" $timepath | tee figs/time-$backend-12-data.csv
grep "[MPI|6]$" $timepath | tee figs/time-$backend-6-data.csv

for f in data/$backend-*-incomplete.txt; do
    farray=(${f//-/ })
    pc=${farray[1]}
    ksp=${farray[2]}
    nodes=${farray[3]}
    echo "$pc/$ksp, -, -, $nodes" | tee -a $timefull
done
```

Listing B.3 opm-bench.sh: orchestrate timings.

```
#! /bin/bash

outdir=/home/jorgekva/bench-out
runs="6 12"
bin=/usr/local/bin/upscale_relperm

model=/home/jorgekva/Files/model.grdecl
for ext in ‘seq 1 4‘; do model="$model
    /home/jorgekva/Files/rock"$ext".txt"; done

linsolvertypes=‘seq 0 3‘
dunealgs=("ILU.BiCGStab" "AMG.CG" "KAMG.CG" "FastAMG.CG")

if [ $# -lt 1 ]
then
  echo "Error: no arguments given"
  echo "Options are: baseline, dune, compile, petsc"
  exit
fi

if [ $1 = "baseline" ]
then
  for solver in $linsolvertypes; do
      $bin -linsolver_type $solver $model \
      | tee $outdir/baseline-${dunealgs[$solver]}-out.txt
  done
fi

if [ $1 = "dune" ]
then
  for run in $runs; do
    for solver in $linsolvertypes; do
      mpirun -np $run $bin \
      -linsolver_type $solver $model \
      | tee
         $outdir/dune-${dunealgs[$solver]}-$run-out.txt
    done
  done
fi

if [ $1 = "petsc" ]
```

Listing B.4 opm-bench.sh: orchestrate timings.

```
39 if [ $1 = "petsc" ]
40 then
41   #run petsc
42   preconditioners="hypre gamg jacobi ilu"
43   kspalgs="cg gmres bcgsl"
44
45   for run in $runs; do
46     for pc in $preconditioners; do
47     for kspalg in $kspalgs; do
48       mpirun -np $run $bin \
49       -pc_type $pc -ksp_type $kspalg $model \
50       | tee $outdir/petsc-$pc-$kspalg-$run-out.txt
51     done
52     done
53   done
54 fi
55
56 if [ $1 = "mkpetsc" ]
57 then
58   for srcdir in core porsol upscaling; do
59     cd /home/jorgekva/opm-petsc/opm-$srcdir/build
60     make -j6
61     make install
62   done
63 fi
64
65 if [ $1 = "mkdune" ]
66 then
67   for srcdir in core porsol upscaling; do
68     cd /home/jorgekva/opm-$srcdir/build
69     make -j6
70     make install
71   done
72 fi
```

Listing B.5 extract-timing.pl:

```
1  #! /usr/bin/env perl
2
3  use strict;
4  use warnings;
5
6  # assumes to only get the relevant timing lines, i.e.
       tail 51 | head 3
7  while( <> ) {
8      $_ =~ /(\d+(.\d+)?)/;
9      print "," . int( $1 + 0.5 );
10 }
```

Table C.1: Workstation HW config.

System	
CPU	Intel Core i7 950 (Bloomfield)
Frequency	3.07 GHz
Cores	4 (8 with HT)
Memory	12 GB
Instruction set	x86_64
GPU	Nvidia GeForce GTX980
Video driver	nvidia-346.47
OS	Ubuntu 14.04
GCC	4.8.2

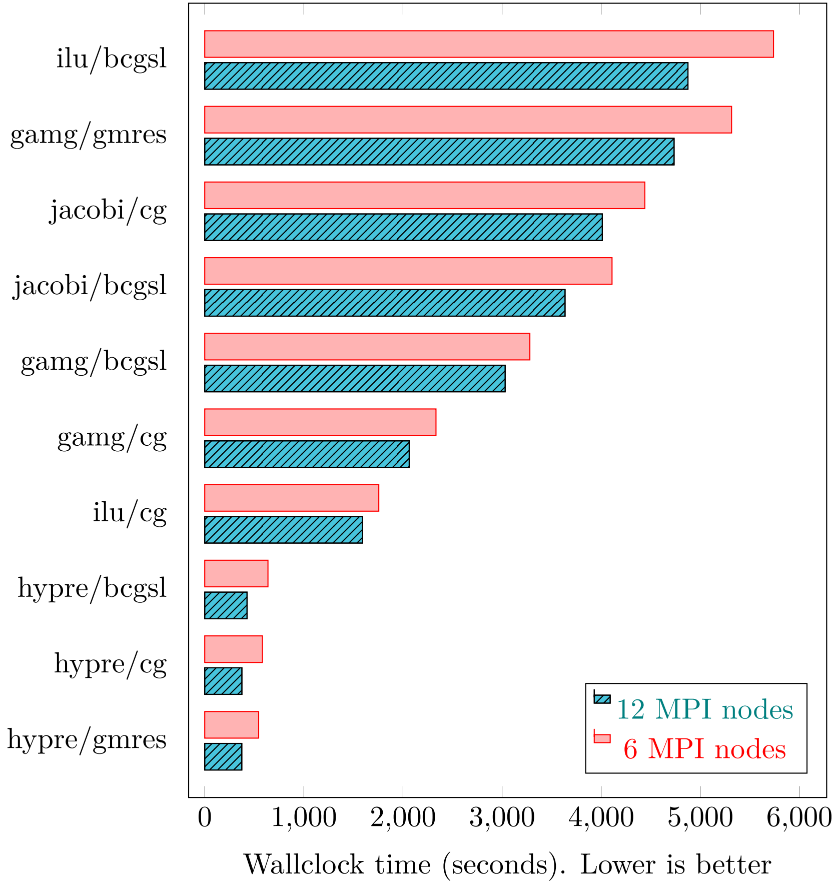

Figure C.1: Time elapsed for PETSc driven runs. Lower is better.

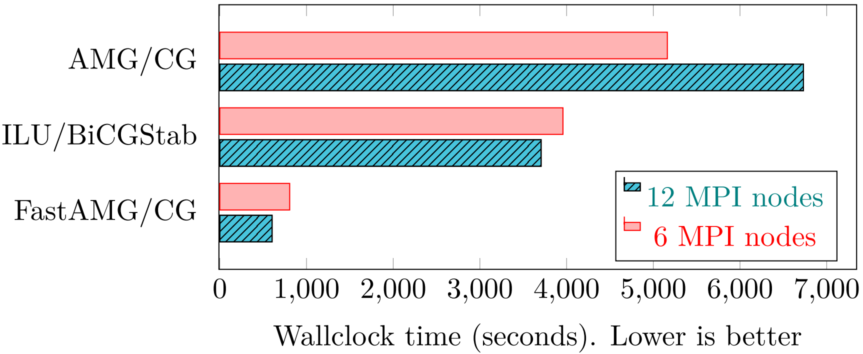

Figure C.2: Time elapsed for DUNE driven runs.

```
######################################################################
# Results from upscaling relative permeability.
#
#          (MPI-version)
# Finished: Fri May 29 15:41:07 2015
# Hostname: hpc-lab-ruben
#
# Eclipse file: /home/jorgekva/Files/model.grdecl
#        cells: 444606
#  Pore volume: 0.0274506
#       volume: 0.10065
#     Porosity: 0.272733
#
# Stone 1: /home/jorgekva/Files/rock1.txt (41 points)
# Stone 2: /home/jorgekva/Files/rock2.txt (41 points)
# Stone 3: /home/jorgekva/Files/rock3.txt (41 points)
# Stone 4: /home/jorgekva/Files/rock4.txt (41 points)
#         jFunctionCurve: 4
#
# Timings:   Tesselation: 5.60263 secs
#              Upscaling: 374.077 secs (wallclock time)
#                         187.931 secs pr. saturation point
#              MPI-nodes: 12
#                Speedup: 10.2737, efficiency: 0.85614
#
# Options used:
#     Boundary conditions: Fixed (no-flow)
#                  points: 30
#         maxPermContrast: 1e7
#                 minPerm: 1e-12
#                 minPoro: 0.0001
#          surfaceTension: 11 dynes/cm
#                 gravity: 0
#
# Single phase permeability
#  |Kxx  Kxy  Kxz| = 30.603  0  0
#  |Kyx  Kyy  Kyz| = 0  32.3877  0
#  |Kzx  Kzy  Kzz| = 0  0  1.22139
#
######################################################################
#  Pc (Pa)        Sw         Krwxx        Krwyy        Krwzz        Kroxx        Kroyy        Krozz
   3.674e+05      0.3783     0.000        0.000        0.000        0.9998       0.9998       0.9999
   7.529e+04      0.3945     3.612e-06    9.626e-06    1.382e-06    0.9784       0.9926       0.6228
   3.880e+04      0.4107     7.757e-05    0.0001511    3.977e-05    0.9547       0.9822       0.3198
       9790.      0.4269     0.0003704    0.0004275    0.0008340    0.9220       0.9532       0.001182
       5016.      0.4431     0.0005488    0.0005411    0.005178     0.8417       0.8712       0.001106
       3524.      0.4593     0.0008153    0.0007778    0.01438      0.7473       0.7742       0.001089
       2675.      0.4755     0.001306     0.001236     0.02760      0.6525       0.6765       0.001046
       2094.      0.4917     0.002120     0.002005     0.04266      0.5624       0.5834       0.001035
       1659.      0.5079     0.003355     0.003179     0.05737      0.4799       0.4980       0.0009892
       1310.      0.5241     0.005076     0.004820     0.07006      0.4058       0.4211       0.0009223
       1022.      0.5403     0.007333     0.006980     0.08049      0.3408       0.3536       0.0008375
       775.0      0.5566     0.01014      0.009674     0.08904      0.2847       0.2952       0.0007107
       558.5      0.5728     0.01348      0.01290      0.09624      0.2366       0.2452       0.0005748
       363.8      0.5890     0.01734      0.01664      0.1025       0.1957       0.2026       0.0004142
       184.4      0.6052     0.02172      0.02090      0.1083       0.1607       0.1663       0.0002702
       14.83      0.6214     0.02663      0.02571      0.1137       0.1307       0.1351       0.0001609
      -149.7      0.6376     0.03214      0.03113      0.1190       0.1049       0.1084       9.213e-05
      -314.0      0.6538     0.03839      0.03730      0.1243       0.08260      0.08522      5.329e-05
      -482.6      0.6700     0.04554      0.04439      0.1299       0.06333      0.06529      3.716e-05
      -660.3      0.6862     0.05383      0.05265      0.1358       0.04693      0.04836      3.239e-05
      -852.5      0.7024     0.06360      0.06241      0.1422       0.03334      0.03434      3.085e-05
      -1066.      0.7186     0.07523      0.07408      0.1494       0.02253      0.02320      2.969e-05
      -1308.      0.7348     0.08923      0.08817      0.1576       0.01435      0.01478      2.978e-05
      -1592.      0.7510     0.1063       0.1054       0.1669       0.008497     0.008756     2.944e-05
      -1937.      0.7673     0.1272       0.1265       0.1777       0.004560     0.004704     2.892e-05
      -2375.      0.7835     0.1531       0.1528       0.1904       0.002123     0.002194     2.810e-05
      -2970.      0.7997     0.1858       0.1860       0.2055       0.0007952    0.0008245    2.636e-05
      -3880.      0.8159     0.2276       0.2284       0.2239       0.0002122    0.0002204    2.189e-05
      -5674.      0.8321     0.2827       0.2844       0.2469       3.044e-05    3.133e-05    1.049e-05
  -1.177e+06      0.8483     0.3615       0.3630       0.3114       0.000        0.000        0.000
```

Figure C.3: Output from *upscale_relperm* with 12 MPI processes driven by PETSc.

Table C.2: Testing system hardware configuration.

System	
CPU	Intel Core i7 3930K (Sandy Bridge)
Frequency	3.20 GHz
Cores	6 (12 with HT)
Memory	32 GB
Instruction set	x86_64
GPU	AMD Radeon HD 7950 (Tahiti)
Video driver	fglrx-15.2
OS	Ubuntu 14.04
GCC	4.8.2