NTNU – Trondheim
Norwegian University of
Science and Technology

A Parallel Multiscale Mixed Finite-Element Method for the Matlab Reservoir Simulation Toolbox

Anders Hoff

Master of Science in Physics and Mathematics
Submission date: June 2012
Supervisor: Helge Holden, MATH
Co-supervisor: Bård Skaflestad, SINTEF ICT

Norwegian University of Science and Technology
Department of Mathematical Sciences

+++Mr. Jelly! Mr. Jelly!+++

+++Error At Address: 14, Treacle Mine Road, Ankh-Morpork+++

+++MELON MELON MELON+++

+++Divide By Cucumber Error. Please Reinstall Universe And Reboot+++

+++Whoops! Here Comes The Cheese!+++

+++Oneoneoneoneoneoneone+++[1]

[1] A list of not so helpful error messages uttered by 'HEX', the great thinking-engine. HEX is featured in the Discworld universe, created by Sir Terry Pratchett.

Problem Statement

Multiscale simulation is a promising approach to facilitate direct simulation of large and complex grid-models for highly heterogeneous petroleum reservoirs. One such method, the Multiscale Mixed Finite-Element (MsMFE) method, has been developed by SINTEF and implemented in the Matlab Reservoir Simulation Toolbox (MRST) [3]. The MsMFE method has an inherent parallelism in the computation of basis functions (and correction functions) that has so far not been demonstrated in actual computations. In the thesis, the student will develop a prototype parallel implementation in MRST using the MATLAB Parallel Computing toolbox [2]. The performance and scalability of the code should be tested for different configurations.

Abstract

We start by giving a brief introduction to reservoirs and reservoir modelling at different scales. We introduce a mathematical model for the two-phase flow, before we look at numerical discretizations. In particular we look at the Multiscale Mixed Finite-Element (MsMFE) Method from the MATLAB Reservoir Simulation Toolbox (MRST) [3], developed by SINTEF. Next we introduce a mimetic method, which is used for solving the local flow problems required to construct the basis functions used in the MsMFE method. After we have given a short introduction to parallel computing, and some common terms, we introduce a parallel MsMFE method. The method makes use of the MATLAB Parallel Computing Toolbox [2], and it lets us calculate the inner products, as well as construct the basis functions of the MsMFE Method, in parallel. The new method makes use of a structure for storing the inner products that proves to be facillitate faster construction of the required basis functions than the regular structure used in MRST. We conclude that the new functions performs quite well, and consequently that the MsMFE method is well suited for parallelization. Additinally, we conclude that the Parallel Computing Toolbox works well for this task. We note that, for larger problems, the parallel MsMFE method displays a near linear speedup for up to twelve MATLAB workers; provided the blocks are not too large. The new parallel prototype is released as a module for MRST, under the GNU General Public License (GPL)[2]. The module can be downloaded from `http://master.andershoff.net`.

[2] `http://www.gnu.org/licenses/gpl.html`

Sammendrag

Vi starter med å gi en kort introduksjon til reservoarer og reservoarmodellering ved ulike målestokker. Videre introduserer vi en matematisk modell for to-faseflyt, før vi ser på noen numeriske diskretiseringsmetoder. Hovedsaklig ser vi vi på en Multiscale Mixed Finite-Element (MsMFE) metode som brukes i MATLAB Reservoir Simulation Toolbox (MRST) [3]. Videre introduserer vi en mimetisk metode, som brukes til å løse de lokale flytproblemene som behøves for å bygge basisfunksjonene som brukes i MsMFE metoden. Etter dette gir vi en liten introduksjon til parallelle beregninger, og noen sentrale begreper, før vi introduserer en parallell MsMFE metode. Den nye metoden benytter seg av funksjonalitet fra MATLAB Parallel Computing Toolbox [2], og den lar oss beregne indreprodukter, og de nødvendige basisfunksjonene parallelt. I den nye metoden benytter vi en struktur for å lagre indreproduktene som avviker fra den tradisjonelle strukturen i MRST. Denne nye strukturen viser seg å legge til rette for noe raskere konstruksjon av basisfunksjoner. Vi konkluderer med at den nye prototypen har god ytelse, og dermed at at MsMFE-metoden egner seg godt til parallellisering. I tillegg konkluderer vi med at Parallel Computing Toolbox fungerer bra til denne oppgaven. For tilstrekkelig store systemer har de parallelle funksjonene nær lineær speedup, for opp til tolv kjerner; så lenge blokkene ikke er for store. Prototypen kan lastes ned som en modul for MRST, under GNU General Public License (GPL)[3], fra `http://master.andershoff.net`.

[3]`http://www.gnu.org/licenses/gpl.html`

Preface

This paper was written for my master's thesis in Industrial Mathematics at the Norwegian University of Science and Technology (NTNU). Sections 2 and 3 originate from my term project 'An Introduction to Reservoir Simulation and the Multiscale Mixed Finite Element-Method', but some changes have been made; in particular Section 3.4 has been added. In short, the purpose of this thesis has been to develop a prototype parallel Multiscale Mixed Finite-Element (MsMFE) Method for reservoir flow simulations, using the MATLAB Reservoir Simulation Toolbox, and the MATLAB Parallel Computing Toolbox. This paper is a documentation of the relevant background, of the implementation, and of the final results.

Working on this thesis, and writing this paper, has provided me with a great deal of knowledge when it comes to reservoirs and reservoir simulation. Additionally it has given me great insights into to the functionality of the MATLAB Parallel Computing Toolbox, as well as new insights into programming. It has been both challenging, and, at times, tiring. I do not know quite how many times I have redesigned the provided code, but I believe that the final version turned out rather promising.

I would like to thank my advisors, Prof. Knut-Andreas Lie[4], Dr. Bård Skaflestad[4] and Prof. Helge Holden[5]. All of them have provided me with valuable insights and suggestions throughout the writing process. I would also like to thank the people in the support team associated with the 'Kongull' computing cluster for their help with testing my code, and for lightning fast replies to my queries. The same goes for Edric Ellis and Konrad Malkowski on the mathworks forums[6]. Finally, I would like to thank my friend, André Ingvaldsen, and my Nespresso machine.

Anders Hoff
June 12, 2012
Trondheim

[4]SINTEF ICT, Oslo.
[5]Department of Mathematical Sciences, NTNU, Trondheim.
[6]`http://www.mathworks.se/matlabcentral/answers/`

Contents

Problem Statement iii

Abstract v

Sammendrag vii

Preface ix

1 Introduction 1

2 Describing Reservoirs 3

2.1 Pore Scale Model 5

2.2 Core Scale Model 5

2.3 Geological Model 6

2.4 Simulation Model 7

2.5 Mathematical Model 7

2.5.1 Two-phase Flow 8

3 Discretization Methods 11

3.1 Mixed Finite-Element Method 12

3.2 Schur Complement Reduction 15

3.3 Multiscale Mixed Finite-Element Method 17

3.4 Mimetic Discretization Methods 19

3.4.1 Inner Products 21

4 Parallel Computing 25

4.1 Amdahl Versus Gustafson 25

5 Matlab Parallel Computing Toolbox 29

5.1 Setting up the Environment 29

5.2 The Parallel For Loop 30

5.3 The spmd Construct 31

5.4 Scheduling Jobs 36

5.5 Performance 40

5.6 Some Issues 40

6 A Parallel Multiscale Mixed Finite-Element Method **43**
6.1 Some Notes on the Implementation 44
6.1.1 Initializing and Broadcasting the Problem 44
6.1.2 A New Inner Product Structure 45
6.1.3 Distributing the Inner Products 48
6.1.4 Building the Basis 49
6.2 Getting the Code 49

7 Results and Discussion **53**
7.1 Test Platform 53
7.2 Testing Speedup on a Cartesian Grid 53
7.3 Testing Speedup on a More Realistic Geometry 56
7.4 Testing the Inner Product Structure 57
7.5 Future Work 61

8 Concluding Remarks **65**

9 References **67**

Appendices **69**

A A Working Example **69**

1 Introduction

Recent advances in parallel computing have made sure that most computers today have multi-core processors. This development comes, largely, as a reaction to the fact that the tremendous increase in processing speed of the monolithic single-core processor is beginning to come to a halt. The main reason for the halting processing speed is the increasingly high heat production from circuits with higher clock speeds [19]. Hence manufacturers have started putting several processors on the same integrated circuits [20]. Typically each processor has a fairly small and fast cache, slower caches are shared by several processors.

When developing serial programs in the 'single-core era', an increase in processing speed would, more or less, automatically mean an increase in the speed of the serial program. If a program was not fast enough, a developer could simply wait for the next generation of microprocessors, and the program would speed up with it [20]. Taking advantage of advances in parallel computing requires some more work. This task is now in the hands of the developers, who need to adjust their algorithms—and their programs—to utilize the new opportunities offered by parallel environments.

A field of research that is only becoming more relevant is the area of subsurface fluid flow in highly heterogeneous porous media. Both for simulating hydrocarbon reservoirs, storage of carbon dioxide and utilization of groundwater. Representing reservoirs, and simulating the fluid flow in a reservoir, is an immensely complex engineering task. The main reason being that the geometry and the key properties of a reservoir vary over a multitude of scales. That is, there are large variations in the compositions of the involved fluids, the rock types, and the geometric compositions. We have to weigh the possible loss of detail, when using a coarser discretization, against the challenge of acquiring and storing the information needed for a finer discrete representation.

Ability to obtain and store increasingly comprehensive models of these reservoirs requires us to adapt the way that these models are used in simulations. The field of reservoir simulations has, unfortunately, not kept up the ability to digitally represent reservoirs [5]. An industry grade reservoir model might have a number of cells between 10 and 100 million, and these numbers are steadily growing. On the other hand, the ability to run simulations on the available models is not increasing fast enough.

Traditionally this gap has been overcome by downscaling the reservoir model into a more manageable simulation with less cells. A promising alternative to

this is Multiscale Mixed Finite-Element (MsMFE) method. This is a method that facilitates direct computation pressure and velocities for large and complex grid-models, for highly heterogeneous reservoirs. The method relies on the numerical construction of generalised basis functions, from solving local flow-problems, that accounts for fine-scale heterogeneity [5]. The construction of these basis functions is an inherently parallel task. I.e. it can be done in arbitrary order. Taking advantage of this parallelism could speed up the method considerably in a multi-core environment.

In this paper we introduce the prototype for a parallel Multiscale Mixed Finite-Element Method that builds on The MATLAB reservoir Toolbox (MRST) [3], and the MATLAB Parallel Computing Toolbox (PCT) [2]. We start by going through the necessary background theory. First we give an introduction to the complex task of representing heterogeneous reservoirs on multiple scales, in Section 2. We also present some common reservoir models in Sections 2.1 to 2.4, before we present a mathematical model for two-phase flow in Section 2.5.1.

Next we introduce the relevant discretization methods in Section 3. First we look at the Mixed Finite-Element Method, and the corresponding hybrid form, in Section 3.1. Furthermore, we introduce the Multiscale Mixed Finite-Element (MsMFE) method in Section 3.3. We go through some common terms related to parallel computing in Section 4, before we introduce the parallel prototype in Section 6.[7]. We present some results in terms of performance in Section 7, along with a discussion.

Finally we remark that the interested read can download the prototyped code as a MRST module named '`xmsmfem`', from `http://master.andershoff.net`.

[7]For completeness, a complete working example is included in Appendix A.

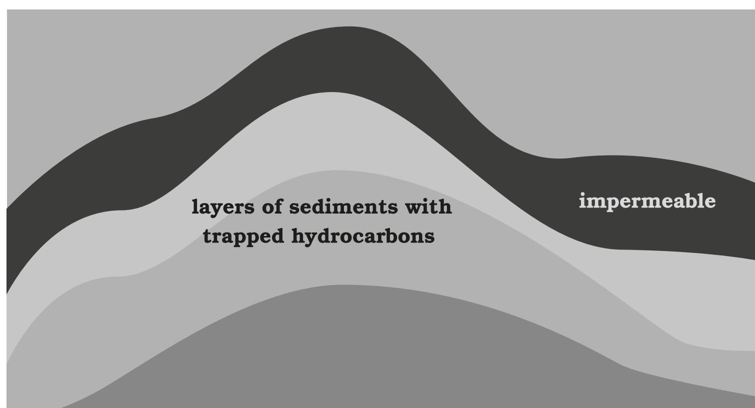


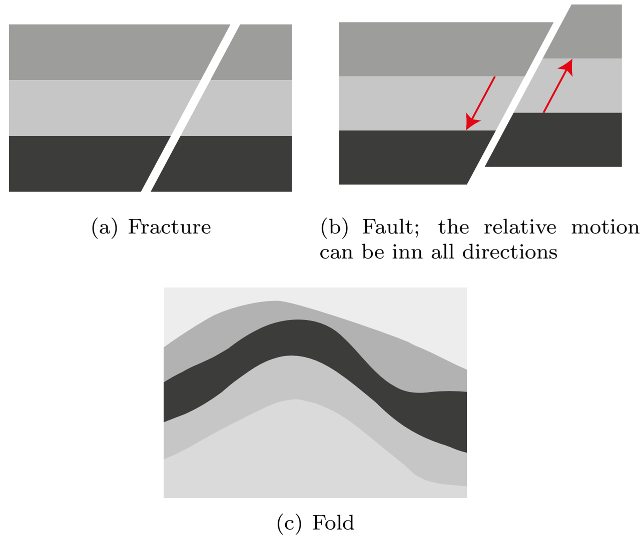
Figure 1: Hydrocarbons are trapped in various layers of porous media, like sandstone, beneath an impermeably layer. This layer can consist of for instance clay or slate.

2 Describing Reservoirs

A petroleum reservoir is a pool of subsurface hydrocarbons trapped in a porous medium. As living organisms died and fell to the bottom of the sea millions of years ago they were trapped in various debris that continuously falls to the bottom. This debris is known as sediments, and is a collection of various minerals, and other naturally occurring substances in the water, as a result of erosion. As more debris and organic material fell to the bottom, the pressure and temperature at the bottom of the stack began to rise. This caused the layers to get gradually more compacted. Eventually the trapped organic material began to boil, which in turn caused the chemical structure to change. The result is a collection of liquid, gaseous and solid hydrocarbon compounds, colloquially known as crude oil.

The sediments typically vary in size between the micrometer scale and up to the centimeter scale. The size and composition of the sediments depend on the surrounding environment. In turn this variation gives rise to layers with different properties as they are compacted. Most importantly we get impermeable layers, such as slate and clay, and highly permeable layers such as sandstone. Eventually the hydrocarbons began to seep upwards through the porous layers, until they escaped through the surface, or they were trapped beneath an impermeable layer, as illustrated in Figure 1.

A reservoir might stretch over as much as several kilometers in the horizontal direction and as little as a few meters in the vertical direction. Within a reservoir



(a) Fracture

(b) Fault; the relative motion can be inn all directions

(c) Fold

Figure 2: Schematic examples of some ways facies can be displaced.

we will observe sedimentary beds consisting of of several laminae (layers of sediment). The thickness and composition of these beds typically vary throughout the reservoir. Variations between a few millimeters and up to tens of meters are not uncommon [7]. These beds are often separated by thin layers with significantly lower permeability.

Depending on the geological activity in the reservoir, facies[8] can be displaced relative to each other by faults, or separated by fractures, as shown in Figures 2(a) and 2(b). Folds, such as in Figure 2(c) are the results of the same forces. These sort of structures can have a large impact on the permeability in the surrounding area.

Representing and modelling a reservoir is a complex task with many physical interactions and forces that span over several orders of magnitude. For this reason there are many different types of models in use. These models act on several different resolutions, or scales as demonstrated in Figure 3. We will look at pore scale and core scale models, on the smaller scale, and simulation and geological models on the larger scale. Afterwards we will define a few important physical quantities that are essential to reservoir studies, before we give a brief introduction to the incompressible two-phase model.

[8] A layer of rock formed under the same sedimentation environment.

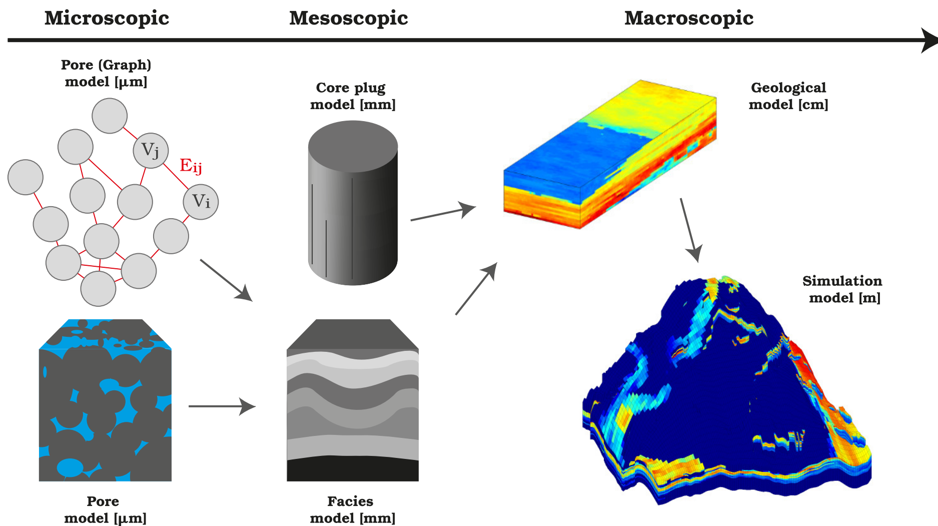


Figure 3: Models at different scales; their typical length scales and how the data is used. The Simulation model was generated using MRST [3], and the model is from the SAIGUP project [1]. *The illustration of the geological model was provided by Knut-Andreas Lie.*

2.1 Pore Scale Model

A pore scale model is usually a graph, (V, E), where V denotes vertices or nodes and $E \in V \times V$, are edges that connect two vertices. This can be seen in Figure 3. Thus we can view the vertices as pores and the edges as the mapping of the physical connections between the pores. From this representation we can iteratively update the model to find the way a fluid percolates (seeps) through the medium by letting the fluid invade neighbouring nodes, depending on various physical properties. On this scale the most important factor is usually capillary forces. But other factors, such as gravity, may be have an impact as well [7]. As we will see next it is impossible to model a real reservoir with this model because we can not rely on knowing the pore structure of even a tiny fraction of the reservoir. Nor can we hope to have the necessary computing resources. However, these models can provide valuable information that can be put into other models.

2.2 Core Scale Model

When wells are drilled in a reservoir, the removed cores, or core, plugs can be extracted and studied to get more information about the reservoir. Commonly,

the core plugs are examined under an electron microscope, with X-ray or with CT-scanners. One can also do flow experiments on them. This can provide useful information, such as relative permeability and capillary pressures. Extrapolating this local information into the geological or simulation model is desirable, but inherently difficult because of the heterogeneous nature of reservoirs [7]. This is naturally also the case with the pore scale model as well as with the models mentioned next.

2.3 Geological Model

A geological model is a discrete geometric representation of a reservoir in the form of a set of grid cells. Each grid cell is assigned a set of properties that are constant over that cell. Most notably, permeability and porosity, as discussed further in Section 2.5. Typical cell sizes range between 10–50 m horizontally, and 10 cm to 10 m vertically [6]. A full geological reservoir model might contain as much as a hundred million cells [5].

Because the geophysical data of a reservoir is not readily available, a range of methods are utilized to acquire as much data as possible. We have already mentioned that core plug models and pore models can be used to estimate permeability and capillary pressures for different rock types. It is also possible to look at other places in the world with similar geological histories; by examining outcrops here one can acquire information about the stratigraphy (composition of the layers) and mechanical deformations. The combination of these methods give a model of the reservoir, which in turn can be used to determine how the seismic surveys should be carried out. Because seismic surveys are time-consuming and expensive, one can not simply investigate the whole reservoir in detail. It is also important to note that seismic surveys have limited accuracy, so you can not expect to see structures thinner than tens of meters. It is possible to get data with somewhat higher accuracy from well-logs. This refers to the process of lowering different types of probes into the well. Here the resolution can be expected to be on the centimeter scale [7]. However, the data is limited to the area in close proximity to well.

In practice a combination of geostatistical techniques are used to estimate characteristics of the field between wells. Preferably as much specific data as possible is incorporated in this process as well as information about the environment that the reservoir has evolved under. Stochastic simulation techniques are utilized to emulate the missing pieces of the puzzle. All in all this makes it possible to provide predictions about various production characteristics with an estimate of the

uncertainty in the model [7].

2.4 Simulation Model

It is easy to think that the geological model is best suited for simulating flow in a reservoir. This is, after all, the most accurate data available. There are, however, a number of reasons why this is not desirable, or even possible, for a full-sized reservoir.

First of all, even if the geological model was entirely accurate for the given resolution, it leaves out potentially important data that could affect the fluid flow on a finer scale. The various media in a reservoir are heterogeneous on all scales, which makes it unrealistic to represent all details about it in discrete form. Next, as much of the geological model is qualified guesswork to begin with, running simulations on the finest scale available would give flow simulations that are more precise, but not necessarily more accurate [7]. In addition, large-scale flow patterns are not necessarily that heavily affected by the flow patterns on the smaller scales. This means that simulations with a lower resolution than the available geological model might yield good enough data for the purpose intended.

Having said this, the main reason why simulations are not run at the finest scale available is that the required computational effort is much too large even for the largest supercomputers available today; both when it comes to memory and processing power. As an example, using the Finite-Element method introduced Section 3.1 to discretize a 10 m $\times$ 1 km $\times$ 1 km reservoir, with a resolution of 1000 $\times$ 10^4 $\times$ 10^4 cells would require around 1 TB simply to store one time step of the solution. One can see why this is impractical at best.

2.5 Mathematical Model

Porosity, commonly denoted ϕ, is a central quantity when modelling a reservoir. This refers to the void volume fraction of a medium; thus we have $0 \leq \phi \leq 1$. Depending on the medium not all of the porosity is connected, so we can distinguish between effective and noneffective porosity [13]. As an example the effective porosity can range between 0.1–0.3.

Next we have the rock compressibility,

$$c_r = \frac{1}{\phi}\frac{\partial \phi}{\partial p}, \tag{1}$$

where p is the pressure in the reservoir. Because of the complicated nature of reservoir mechanics the compressibility of the rock is sometimes neglected or linearised to simplify the calculations.

Porosity in a medium gives rise to the concept of permeability, $\mathbf{K}$. Permeability is a measure of the ability of the medium to let fluids pass through it. In the case of reservoir studies $\mathbf{K}$ is often a tensor. In other words, a matrix where the diagonal terms represent flow in the corresponding spatial direction as a result of a pressure change in the same direction. Furthermore, the off-diagonal terms represent change in flow perpendicular to the change in pressure [6].

Permeability will often vary over several orders of magnitude within the same reservoir, and even over short distances. This will, for instance, be the case in an intersection between two facies with different physical properties, or in a fault or fracture. Variations between 1 mD and 10 D are not unusual [6]. The unit here is millidarcy or darcy, after Henry Darcy; 1 D $\approx 10^{-12}$ m^2. A medium is said to be isotropic when the permeability can be written as a scalar function. Conversely, a medium where this is not the case is said to be anisotropic. The permeability is in many ways the most important quantity in reservoir modelling because this is the quantity that has the strongest influence on the fluid flow.

2.5.1 Two-phase Flow

Mathematically the flow of fluids in a reservoir is governed by the well-known continuity equations; one for each fluid's phases, on the form

$$\frac{\partial}{\partial t}\left(\phi \rho_i s_i\right) + \nabla \cdot \left(\rho_i \mathbf{v}_i\right) = q_i. \tag{2}$$

Here ρ_i is the density of the fluid phase i, the velocity is denoted $\mathbf{v}_i$, and the saturations are written as s_i. The saturation refers to the volume fractions occupied by each phase; thus we need $\sum_{i=1}^{n} s_i = 1$. This is often referred to as the closure relation [7]. On the right-hand side we have the source term, q_i, which models injectors and wells. Commonly three phases are considered; aqueous, liquid and vapour. Also it is common to group components together into the 'component groups' water, oil and gas.

Following this we have the empirical relation known as Darcy's law

$$\mathbf{v}_i = -\frac{\mathbf{K} k_{ri}}{\mu_i}\left(\nabla p_i - \rho_i \mathbf{G}\right), \tag{3}$$

which describes the relation between the phase velocity $\mathbf{v}_i$ and the phase pressure p_i. As previously mentioned $\mathbf{K}$ is the permeability (tensor) from the geological model; μ_i denotes the viscosity and $k_{ri} = k_{ri}(s_1, \ldots, s_n)$ is the reduced permeability of phase i due to the presence of the other phases. So, even though we know that the different phases are immiscible, we assume that they can be present in the same space simultaneously and that they affect each other. This gives an effective permeability, $\mathbf{K}_i = \mathbf{K}k_{ri}$, for phase i. Lastly, $\mathbf{G}$, is the gravitational vector pointing in the downward direction. It is also customary to introduce the phase mobility as

$$\lambda_i = \frac{k_{ri}(s_i)}{\mu_i} \tag{4}$$

Darcy's law and the continuity equations yields one equation for each phase to solve for the pressure, p_i, and for the phase saturation s_i. In addition we have the closure relation. In the case of single-phase flow we can simplify (2) and (3) to the following elliptic equation

$$-\nabla\left(\frac{\mathbf{K}}{\mu}(\nabla p - \rho\mathbf{G})\right) = \frac{q}{\rho}, \tag{5}$$

provided we assume incompressibility.

For multi-phase flow this changes. Now we can only eliminate one of the saturations [7, 6]. Also, we get extra equations describing the capillary pressure functions. That is, the pressure change across the interface between immiscible (unmixable) fluids; expressed as $p_{cij} = p_i - p_j$. Commonly one assumes that the capillary pressure functions only depends on the saturations. Some of these capillary functions can be found from flow simulations on core plugs or from tables for a certain type of rock with known properties.

We now present the system for two-phase flow. We assume incompressible flow and let $i = \{w, o\}$ denote water and oil respectively. Now, letting $p = p_o + p_c$ be the total pressure and similarly, $\mathbf{v} = \mathbf{v}_o + \mathbf{v}_w$, be the total velocity, we can write

$$\nabla \cdot \mathbf{v} = q, \quad \text{with} \quad \mathbf{v} = -\mathbf{K}\left[\lambda\nabla p - (\lambda_w\rho_w + \lambda_o\rho_o)\mathbf{G}\right], \tag{6a}$$

$$\phi\frac{\partial s_w}{\partial t} + \nabla\cdot\left[f_w\left(\mathbf{v} + \mathbf{K}\lambda_o\nabla p_{cow} + \mathbf{K}\lambda_o(\rho_w - \rho_o)\mathbf{G}\right)\right] = \frac{q_w}{\rho_w}. \tag{6b}$$

Here, $\lambda = \lambda_w + \lambda_o$, is known as the total mobility, $f_w = \lambda_w/\lambda$, is the fractional flow of water, p_{cow}, is the capillary pressure and, $q = q_w/\rho_w + q_o/\rho_o$ is the total in- and outflow.

In Equation (6b) we can recognize three terms that each describes a different physical force. First, $f_w\mathbf{v}$, stems from the viscous forces, $f_w\mathbf{K}\lambda_o\nabla p_{cow}$ comes from capillary forces and $f_w\mathbf{K}\lambda_o\left(\rho_w-\rho_o\right)\mathbf{G}$ is due to gravity. Noting that $\mathbf{G}=-g\mathbf{n}_z$ e.g. the gravitational pull in the vertical direction.

How much each of these forces influence the flow in a particular reservoir depends on a number of things [7]. The first obvious factor is the composition and geometry of the reservoir in question. If a reservoir has good communication in the vertical direction gravity will tend to have a larger impact than if this is not the case. Similarly, gravity is more likely to influence the flow if the layers are tilted at an angle to the vertical axis. Systems with mostly gas and oil will also tend to be more influenced by gravity because of larger differences in the density of the involved fluids. Next we note that capillary effects tend to be of more importance in small-scale models, but also in strongly heterogeneous systems. All things considered there are a few general rules, but no absolute ones.

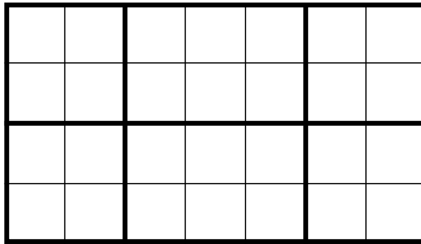


Figure 4: Cartesian grid of 7×4 cells divided into 3×2 blocks.

3 Discretization Methods

As mentioned earlier it is not feasible to run flow simulations on geological models because they require too much computational effort. Because of this, a number of so-called upscaling techniques have been developed in order to be able to run simulations on coarser grids. That is, grids that model the same reservoir, but with a considerably lower resolution. Hence, requiring less computing power. We often say that the fine grid consists of cells, whereas the coarse grid consists of blocks. This is illustrated in Figure 4, and we will continue to use this distinction in the rest of this paper.

Upscaling refers to the process of transferring the physical properties from all the cells in a block into corresponding effective properties that are constant within that block. Upscaling is one of the oldest methods to transfer data from geological models to simulation models, hence this topic is widely described. Even so, except from in a few special cases, there exists no rigorous answers as to how the modelling of the PDEs at the various scales correspond to to each other. Because of this it is difficult to know beforehand how a given upscaling will behave for a particular case.

A different approach to handling the multiscaled nature of porous media flow, without directly solving the PDEs on the finest scale, is the Multiscale Finite-Element (MsFE) Method. This method was introduced by Hou and Wu in [16] in 1997. The method is described, among many places, in [14, 5, 17]. The idea behind the MsFEM is to '... capture the multiscale structure of the solution via localized basis functions.' These basis functions are said to '... contain essential multiscale information embedded in the solution ...' [14].

We start by introducing a Mixed Finite-Element Method (MFEM) in Section 3.1 before we extend it to the Multiscale Mixed-Finite Element (MsMFE) method in Section 3.3. Finally we give a short introduction to mimetic methods in Section 3.4.

3.1 Mixed Finite-Element Method

The term 'mixed method' applies to Finite-Element Methods that have more than one approximation space [11]. We start by letting $\Omega \in \mathbb{R}^3$ define a domain with faces $\partial\Omega$, and face normal, $\mathbf{n}$. Then we restate Equation (5) for the case with zero gravity

$$\mathbf{v} = -\mathbf{K}\nabla p, \quad x \in \Omega, \tag{7a}$$

$$\nabla \cdot \mathbf{v} = f, \quad x \in \Omega, \tag{7b}$$

$$\mathbf{v} \cdot \mathbf{n} = 0, \quad x \in \partial\Omega. \tag{7c}$$

As usual $\mathbf{v}$ refers to the velocity and p is the pressure; f denotes the source term. We require that $\int_\Omega f \, \mathrm{d}x = 0$ because we have Neumann boundary condition on the entire border [9]. In addition we need to normalize the pressure because otherwise the pressure is only defined up to an arbitrary constant. This is done by requiring $\int_\Omega p \, \mathrm{d}x = 0$ [5]. If we had imposed a Robin (or Dirichlet) boundary condition on the whole, or a part of, $\partial\Omega$, this extra condition would not be needed.

Next we introduce the partition, $\mathcal{T}$, of Ω into arbitrary polyhedral cells T with faces ∂T. The normals of these cells are denoted $\mathbf{n}^T$ and interfaces, $\gamma_{ij} = \partial T_i \cap \partial T_j$. Here it is useful to notice the distinction between half-faces and distinct faces. That is, we refer to half-faces in a cell, but when two half-faces of cells i and j coincide we refer to this as the (global) face, or interface, γ_{ij}. We will make a note when this distinction becomes important.

We also need the following three function spaces [5]

$$H^{\mathrm{div}}(T) = \left\{ \mathbf{v} \in L^2(T)^d : \nabla \cdot \mathbf{v} \in L^2(T) \right\},$$
$$H_0^{\mathrm{div}}(\mathcal{T}) = \left\{ \mathbf{v} \in H^{\mathrm{div}}(\cup_{T\in\mathcal{T}} T) : \mathbf{v} \cdot \mathbf{n} = 0 \in \partial\Omega \right\},$$
$$H_0^{\mathrm{div}}(\Omega) = H_0^{\mathrm{div}}(\mathcal{T}) \cap H^{\mathrm{div}}(\Omega).$$

Here $L^2(T)$ refers to all square-integrable functions in cell T. That is, all functions, y, such that $\int_T |y(x)|^2 \, \mathrm{d}x < \infty$.

In addition we mention the following identity

$$\int_\Omega [\mathbf{a} \cdot \nabla b + b \nabla \cdot \mathbf{a}] \, \mathrm{d}V = \oint_{\partial\Omega} b \, \mathbf{a} \cdot \mathbf{n} \, \mathrm{d}S, \tag{8}$$

where $\int_\Omega \mathrm{d}V$ denotes the volume integral over Ω; e.g. in three dimensions we could also write $\iiint_\Omega \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z$. This identity can be obtained by applying Gauss' theo-

rem[9] to the product of a scalar function, b, and a vector function, $\mathbf{a}$.

Now we multiply Darcy's law (7a) with an arbitrary vector function, $\mathbf{u} \in H_0^{\text{div}}(\Omega)$, and integrate over Ω to obtain

$$\int_\Omega \mathbf{u} \cdot \mathbf{K}^{-1}\mathbf{v}\,\mathrm{d}V = -\int_\Omega \mathbf{u} \cdot \nabla p\,\mathrm{d}V \overset{(8)}{=} -\underbrace{\int_{\partial\Omega} p\mathbf{u} \cdot \mathbf{n}\,\mathrm{d}S}_{0} + \int_\Omega p\nabla \cdot \mathbf{u}\,\mathrm{d}V.$$

Note that the surface integral is zero because of the border condition (7c). Similarly, we multiply (7b) with an arbitrary scalar function, $q \in L^2(\Omega)$, and integrate over Ω to get

$$\int_\Omega q\nabla \cdot \mathbf{v}\,\mathrm{d}V = \int_\Omega qf\,\mathrm{d}V.$$

We can now state the mixed formulation of Equations (7a) to (7c) so that we look for $(p, \mathbf{v}) \in L^2(\Omega) \times H_0^{\text{div}}(\Omega)$ such that

$$\begin{aligned} b(\mathbf{u},\mathbf{v}) - c(\mathbf{u},p) &= 0, && \forall \mathbf{u} \in H_0^{\text{div}}(\Omega), \\ c(\mathbf{v},q) &= (f,q), && \forall q \in L^2(\Omega). \end{aligned} \tag{10}$$

With the bilinear forms

$$\begin{aligned} b(\cdot,\cdot) : H_0^{\text{div}}(\mathcal{T}) \times H_0^{\text{div}}(\mathcal{T}) \to \mathbb{R}, &\quad b(\mathbf{u},\mathbf{v}) = \sum_{T\in\mathcal{T}} \int_T \mathbf{u} \cdot \mathbf{K}^{-1}\mathbf{v}\,\mathrm{d}V, \\ c(\cdot,\cdot) : H_0^{\text{div}}(\mathcal{T}) \times L^2(\Omega) \to \mathbb{R}, &\quad c(\mathbf{v},p) = \sum_{T\in\mathcal{T}} \int_T p\nabla \cdot \mathbf{v}\,\mathrm{d}V, \\ (\cdot,\cdot) : L^2(\Omega) \times L^2(\Omega) \to \mathbb{R}, &\quad (p,q) = \int_\Omega pq\,\mathrm{d}V. \end{aligned}$$

To discretize this problem we need to replace $L^2(\Omega)$ and $H_0^{\text{div}}(\Omega)$ by finite-dimensional subspaces U and V, respectively. We will use the Raviart-Thomas finite elements as they are given in [9]. First we have

$$U = \operatorname{span}\{\chi_m : T_m \in \mathcal{T}\}, \quad \chi_m(x) = \begin{cases} 1, & \text{if } x \in T_m, \\ 0, & \text{otherwise.} \end{cases}$$

[9]Also known as the divergence theorem. It takes the form

$$\int_\Omega (\nabla \cdot \mathbf{F})\,\mathrm{d}V = \oint_{\partial\Omega} (\mathbf{F} \cdot \mathbf{n})\,\mathrm{d}S \tag{9}$$

where $\mathbf{n}$ denotes the unit normal out of the domain and $\mathbf{F}$ is a continuously differentiable vector field.

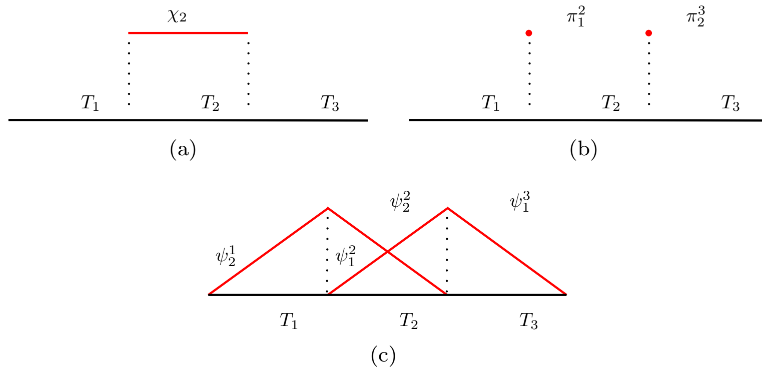


Figure 5: Schematic illustration of all element functions 'related' to the two interfaces γ_{12} and γ_{23} of the three cells T_1, T_2 and T_3 in one dimension.

This corresponds to cell-wise constant functions as shown in Figure 5(a).

Next,

$$V = \text{span}\left\{\psi_i^m : T_m \in \mathcal{T}, \nabla \cdot \psi_i^m \in U, \psi_i^m \in \mathcal{P}_0\left(\gamma_i^j\right)\right\},$$

as illustrated in Figure 5(c) [17]. We can now state this as a system of linear equations

$$\begin{bmatrix} \mathbf{B} & \mathbf{C}^T \\ \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ -\mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{f} \end{bmatrix}, \tag{11}$$

with

$$\mathbf{B} = \left[b\left(\psi_i^m, \psi_j^n\right)\right] \quad \text{and} \quad \mathbf{C} = \left[c\left(\psi_i^m, \chi_n\right)\right].$$

Additionally, $\mathbf{v}$, corresponds to the vector of velocities at each half-face, $\mathbf{p}$ is the pressure in each cell, and $\mathbf{f}$ is the source term in each cell. The problem with the system given in Equation (11) is that this system is generally indefinite[10] [10]; not positive (or negative) definite. This makes the system more difficult to solve because it requires more advanced linear solvers. This problem is often circumvented by transforming the mixed formulation in (10), and consequently in (11), into the so-called mixed hybrid form. The idea behind the hybrid form is to introduce an extra set of unknowns in the form of Lagrangian multipliers [5]. These Lagrangian multipliers will correspond to face pressures at γ_{ij}. Restating the mixed formulation we now look for $(\mathbf{v}, p, \pi) \in H_0^{\text{div}}(\mathcal{T}) \times L^2(\Omega) \times H^{\frac{1}{2}}(\partial\mathcal{T} \setminus \partial\Omega)$

[10] A matrix, $\mathbf{A}$, is indefinite if there exists non-zero vectors, $\mathbf{x}$ and $\mathbf{y}$ st. $\mathbf{x}^T\mathbf{A}\mathbf{x} < 0 < \mathbf{y}^T\mathbf{A}\mathbf{y}$.

such that

$$\begin{aligned} b(\mathbf{u},\mathbf{v}) - c(\mathbf{u},p) + d(\mathbf{u},\pi) &= 0, & \forall \mathbf{u} \in H_0^{\mathrm{div}}(\mathcal{T}), \\ c(\mathbf{v},q) &= (f,q), & \forall q \in L^2(\Omega). \\ d(\mathbf{v},\mu) &= 0, & \forall \mu \in H^{\frac{1}{2}}(\partial\mathcal{T} \setminus \partial\Omega). \end{aligned} \tag{12}$$

Here we have introduced the additional bilinear form

$$d(\cdot,\cdot) : H_0^{\mathrm{div}}(\mathcal{T}) \times H^{\frac{1}{2}}(\partial\mathcal{T}) \to \mathbb{R}, \quad d(v,\pi) = \sum_{T\in\mathcal{T}} \int_T \pi\mathbf{v}\cdot\mathbf{n}_T \,\mathrm{d}s.$$

We note that we are no longer looking for $\mathbf{v} \in H_0^{\mathrm{div}}(\Omega)$, but $\mathbf{v} \in H_0^{\mathrm{div}}(\mathcal{T})$. Which means that we do not enforce continuous flux with $\mathbf{v}$, but by the newly introduced π instead.

The discretization can now be achieved by introducing the finite-dimensional space to replace $H^{\frac{1}{2}}(\partial\mathcal{T} \setminus \partial\Omega)$,

$$\Pi = \mathrm{span}\left\{\pi_j^i : |\gamma_j^i| > 0\right\}, \quad \pi_j^i(x) = \begin{cases} 1, & \text{if } x \in \gamma_j^i, \\ 0, & \text{otherwise.} \end{cases}$$

This corresponds to functions that are constant on each interface, as shown in Figure 5(b). This way we can write

$$\mathbf{D} = \left[d\left(\psi_k^m, \pi_j^i\right)\right],$$

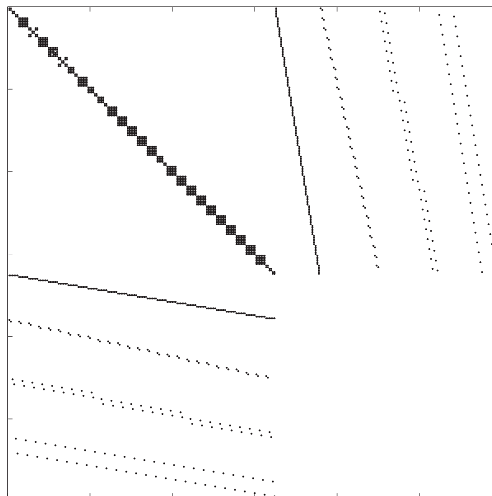
so that we get the following system

$$\begin{bmatrix} \mathbf{B} & \mathbf{C}^T & \mathbf{D}^T \\ \mathbf{C} & 0 & 0 \\ \mathbf{D} & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ -\mathbf{p} \\ \boldsymbol{\pi} \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{f} \\ 0 \end{bmatrix}. \tag{13}$$

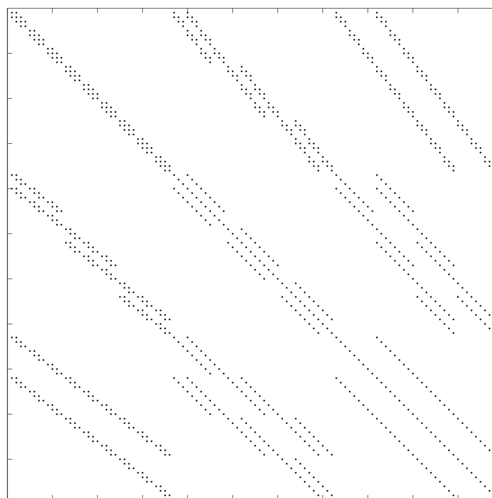
Here $\boldsymbol{\pi}$ is a vector of the lagrangian multipliers, π. An example of the matrix from Equation (13) is given in Figure 6(a) [5].

3.2 Schur Complement Reduction

We will now describe the Schur complement reduction as it is presented in [21]. The first thing to note about Equation (13) is that $\mathbf{B}$ is block-diagonal because all basis functions in a cell only have support within that cell. Hence the inverse of



(a) The resulting system matrix as given in Equation (13)



(b) The reduced system matrix, **S**, as given in Equation (15)

Figure 6: Resulting system matrix and corresponding Schur complement reduced matrix when solving Equation (5) on a unit cube with $3 \times 3 \times 3$ cells with the MFEM.

$\mathbf{B}$, is readily available. We start by expanding (13), and note that

$$\begin{aligned} \mathbf{Bv} - \mathbf{C}^T\mathbf{p} + \mathbf{D}^T\boldsymbol{\pi} &= 0, \quad \Leftrightarrow \quad \mathbf{v} = \mathbf{B}^{-1}\left(\mathbf{C}^T\mathbf{p} - \mathbf{D}^T\boldsymbol{\pi}\right), \\ \mathbf{Cv} &= \mathbf{f}, \\ \mathbf{Dv} &= 0. \end{aligned}$$

We now insert $\mathbf{v}$ into the two remaining equations to get

$$\begin{aligned} \mathbf{Cv} &= \mathbf{CB}^{-1}\left(\mathbf{C}^T\mathbf{p} - \mathbf{D}^T\boldsymbol{\pi}\right) \\ &= \underbrace{\mathbf{CB}^{-1}\mathbf{C}^T}_{\mathbf{E}}\mathbf{p} - \underbrace{\mathbf{CB}^{-1}\mathbf{D}^T}_{\mathbf{F}^T}\boldsymbol{\pi} = \mathbf{f}. \end{aligned}$$

Similarly,

$$\begin{aligned} \mathbf{Dv} &= \mathbf{DB}^{-1}\left(\mathbf{C}^T\mathbf{p} - \mathbf{D}^T\boldsymbol{\pi}\right) \\ &= \underbrace{\mathbf{DB}^{-1}\mathbf{C}^T}_{\mathbf{F}}\mathbf{p} - \mathbf{DB}^{-1}\mathbf{D}^T\boldsymbol{\pi} = 0, \end{aligned}$$

Which can be restated as

$$\begin{bmatrix} \mathbf{E} & \mathbf{F}^T \\ \mathbf{F} & \mathbf{DB}^{-1}\mathbf{D}^T \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ -\boldsymbol{\pi} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ 0 \end{bmatrix}. \tag{14}$$

This process can be repeated once more to achieve a system that can be solved solely for $\boldsymbol{\pi}$,

$$\mathbf{S}\boldsymbol{\pi} = \mathbf{FE}^{-1}\mathbf{f}, \quad \text{with} \quad \mathbf{S} = \mathbf{DB}^{-1}\mathbf{D}^T - \mathbf{FE}^{-1}\mathbf{F}^T. \tag{15}$$

$\mathbf{S}$ is called the Schur complement with respect to $\boldsymbol{\pi}$. From [21, Preposition 14.1] we know that $\mathbf{S}$ is symmetric positive definite (SPD), provided that the matrix in (14) is SPD as well.

Having achieved this we can obtain $\mathbf{p}$ and $\mathbf{v}$ by back-substitution, i.e. $\mathbf{p} = \mathbf{E}^{-1}\left(\mathbf{f} + \mathbf{F}^T\boldsymbol{\pi}\right)$, and $\mathbf{v} = \mathbf{B}^{-1}\left(\mathbf{C}^T\mathbf{p} - \mathbf{D}^T\boldsymbol{\pi}\right)$ as above.

3.3 Multiscale Mixed Finite-Element Method

In order to describe the MsMFE method we look at a grid with a partition, $\mathcal{T} = (T)$, of arbitrary polyhedral cells, as before. However, now we let $\mathcal{B} = \{B\}$ denote coarse grid blocks, B, so that B is a connected union of fine cells, T, in the underlying

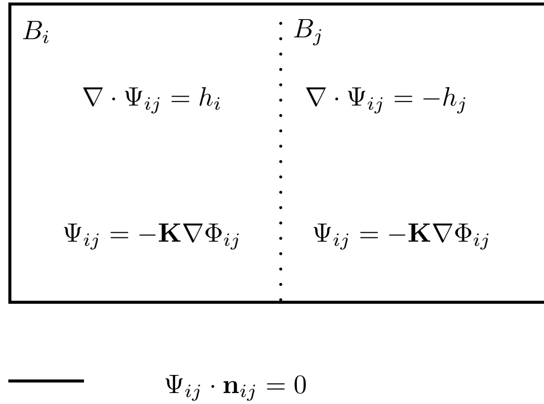


Figure 7: Local problem for the interface $\Gamma_{ij} = \partial B_i \cap \partial B_j$ between blocks B_i and B_j. Note that $\mathbf{n}_{ij}$ is the normal vector pointing out of the domain $\overline{\Omega}_{ij} = B_i \cup \Gamma_{ij} \cup B_j$ on $\partial\Omega_{ij}$.

(fine) grid. We also introduce $\Gamma_{ij} = \partial B_i \cap \partial B_j$, that is, the interface between the coarse blocks B_i and B_j.

Now we introduce basis functions Ψ_{ij} for each interface, Γ_{ij}. In addition we have the Darcy equation $\Psi_{ij} = -\mathbf{K}\nabla\Phi_{ij}$. Each function Φ_{ij} has support in $\overline{\Omega}_{ij} = B_i \cup \Gamma_{ij} \cup B_j$. This is illustrated in Figure 7.

Now we can use the functions Ψ_{ij} and Φ_{ij} as basis for the velocity and the pressure, respectively. Since we are now on the coarse grid, the idea is that these basis functions will contain information about local variations in the permeability, $\mathbf{K}$.

To obtain the actual basis functions numerically we solve the elliptic problem

$$\Psi_{ij} \cdot \mathbf{n}_{ij} = 0 \quad \text{on} \quad \overline{\Omega}_{ij}, \quad \nabla \cdot \Psi_{ij} = \begin{cases} h_i\left(x\right), & \text{for } x \in B_i, \\ -h_j\left(x\right), & \text{for } x \in B_j. \end{cases} \tag{16}$$

Here the source terms are defined as follows;

$$h_i\left(x\right) = \frac{w_i\left(x\right)}{\int_{B_i} w_i\left(x\right)\, \mathrm{d}V}, \quad w_i = \begin{cases} f, & \int_{B_i} f\, \mathrm{d}V \neq 0, \\ \operatorname{trace}\left(\mathbf{K}\right), & \text{otherwise.} \end{cases}$$

This definition will drive force unit flow over the interface Γ_{ij} [17, 5]. We now let $V^{\mathrm{ms}} = \operatorname{span}\{\Psi_{ij}\}$. In the case of diagonal permeability tensor when the grid is composed of triangles, tetrahedrons or rectangular parallelepipeds, this reduces to the first-order Raviart-Thomas basis (RT0) when $\mathbf{K}$ and h_i are constant [5].

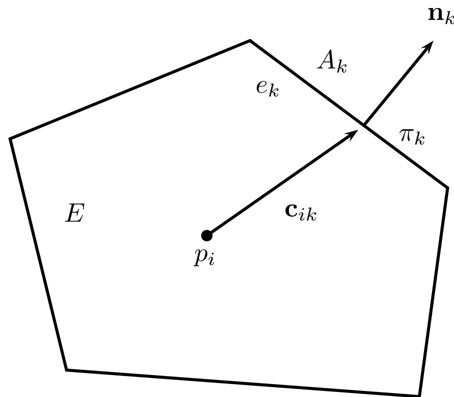


Figure 8: An arbitrary cell, E, with surface $\partial E = e_1 \cup \cdots \cup e_{k_E}$, where, e_{k_E}, denotes the number of faces of the current cell. p_i is pressure at the cell centroid, $\mathbf{c}_{ik}$ is the vector pointing from the centroid to face k, A_k is the area of face k, π_k is the corresponding face pressure, and $\mathbf{n}_k$ is the normal vector.

Now, we quickly realize that the main computational effort in solving Equations (7a) to (7c) using the MsMFE method is spent constructing the basis functions of V^{ms}. The real saving in computational effort of the MsMFE method comes when we solve two-phase problems over time. Here we need to solve the set of equations multiple times, but it has been shown that it is not necessary to recompute the multiscale basis for each step [5].

Additionally we see that there is an inherent parallelism here; each basis function in V^{ms} can be computed independently of the others. Provided all information about the problem is available, the basis functions can be computed in an arbitrary order. This makes the construction of the basis 'embarrassingly parallel'.

3.4 Mimetic Discretization Methods

The default discretization method used in MRST for solving Equation (16) is a mimetic method [18]. We recall the resulting system of linear equations from the Mixed Finite-Element Method, as given in Equation (13). Now, for each cell, E_i, we have the following sizes: $\mathbf{u}_i$, a vector of outward fluxes from each face of the cell; p_i, the pressure at the cell centroid; and, $\boldsymbol{\pi}_i$, a vector of face pressures. Carefully examining the system in Equation (13), we see that for each cell we can we can write [18]

$$\mathbf{u}_i = \mathbf{T}_i \left(\mathbf{e}_i p_i - \boldsymbol{\pi}_i \right), \quad \mathbf{e}_i = (1, \ldots, 1)^T.$$

That is, for each cell, the centroid pressure, the face pressures, and the fluxes, are related through a matrix, $\mathbf{T}_i$, of one-sided transmissibilites. So, for a system of n cells, we can write $\mathbf{B}$ as a block diagonal matrix;

$$\mathbf{B} = \begin{bmatrix} \mathbf{T}_1 & & \\ & \ddots & \\ & & \mathbf{T}_n \end{bmatrix}.$$

Mimetic methods are also defined in this fashion. Here we use either a local inner product, $\mathbf{M}$, or the inverse local inner product, $\mathbf{T} = \mathbf{M}^{-1}$, as above. So, if we drop all subscripts, we have

$$\mathbf{Mu} = \mathbf{e}p - \boldsymbol{\pi}, \quad \mathbf{e} = (1, \ldots, 1)^T. \quad \mathbf{u} = \mathbf{T}\left(\mathbf{e}p - \boldsymbol{\pi}\right), \tag{17}$$

for each cell. In addition, mimetic methods are constructed in a fashion such that $\mathbf{M}$ is Symmetric Positive Definite. They are also required to be exact for linear pressure fields [18].

If we let, $p = \mathbf{x} \cdot \mathbf{a} + b$, be a linear vector field with arbitrary vector, $\mathbf{a}$, and scalar, b, we see that we can write the Darcy velocity as

$$\mathbf{v} = -\mathbf{Ka}.$$

Additionally, we denote the area-weighted normal vector from face e_k, as $\mathbf{n}_k$; and the vector from the cell centroid to face e_k, as $\mathbf{c}_{ik}$. This is illustrated in Figure 8. Furthermore, we now introduce the matrices, $\mathbf{C}$, where each row corresponds to $\mathbf{c}_{ik}^T$; and $\mathbf{N}$, where each row corresponds to $\mathbf{n}_k^T$.

Having done this, we can write the flux from face e_k as $u_k = -\mathbf{n}_k^T\mathbf{Ka}$. Which, for the whole cell, can be restated as

$$\mathbf{u} = \mathbf{NKa}, \tag{18}$$

The pressure drop from the centroid to face e_k, can be expressed as $p_i - \pi_k = \mathbf{c}_{ik} \cdot \mathbf{a}$. For the whole cell this gives us

$$\mathbf{e}p - \boldsymbol{\pi}_k = \mathbf{Ca}. \tag{19}$$

Now we can combine the first part of Equation (17) with Equation (19) to get

$$\mathbf{Mu} = \mathbf{MNKa} = \mathbf{e}p - \boldsymbol{\pi} = \mathbf{Ca}.$$

Which means that we get the following condition for the inner products

$$\mathbf{MNK} = \mathbf{C}. \tag{20}$$

We get another condition from the second part of Equation (17) by noting that

$$\mathbf{u} = \mathbf{NKa} = \mathbf{T}\left(\mathbf{e}p - \boldsymbol{\pi}\right) = \mathbf{TCa},$$

which means that we must have

$$\mathbf{NK} = \mathbf{TC}. \tag{21}$$

With a suiting inner product one can now construct the global (stiffness) matrix, $\mathbf{B}$ (Equation (13)), as a diagonal block matrix where each block corresponds to the inner product of a cell. Now, if we build the remaining elements of Equation (13), as described in Section 3.1, we can solve the system for for the face pressures, π_k, using the Schur Complement Method (Section 3.2), before we substitute back to get the pressure and the velocity.

When dealing with reservoir simulations it is customary to consider inner products of fluxes, rather than velocities, and the relation between the two is trivial. If we let $\mathbf{A}$ be a diagonal matrix where the elements are the areas of each face, a_k, the relation is [18]

$$\mathbf{M}_{\text{flux}} = \mathbf{A}^{-1}\mathbf{M}_{\text{vel}}\mathbf{A}^{-1},$$

3.4.1 Inner Products

We will now present a class of valid inner products as given in [18]. First we assume that we have $d = \{2, 3\}$, and that the spatial variables are $x_1, \ldots, x_d$. We continue by introducing the following identity

$$\oint_{\partial E} (x_j \nabla x_i) \cdot \mathbf{n}\, \mathrm{d}S = \int_E \nabla x_j \cdot \nabla x_i \, \mathrm{d}V. \tag{22}$$

This is a version of the divergence theorem (Equation (9)) applied to the vector field, $x_j \nabla x_i$. As noted in [12], we can now introduce the two $k_E \times d$ matrices

$$\mathbf{N}_{k,i} = \nabla x_i \cdot \mathbf{n}_k \quad \text{and} \quad \mathbf{C}_{k,i} = \oint_{e_k} x_i \, \mathrm{d}S. \tag{23}$$

The first matrix is the face normals, as before, but the second matrix consists of the centroids of E assuming, for simplicity, that the cell centroid is positioned at the origin.

Now, by combining Equations (22) and (23) we get the following identity

$$\mathbf{C}^T \mathbf{N} = |E| \mathbf{I}; \tag{24}$$

$|E|$ denotes the volume of cell E.

And by multiplying Equation (20) by $\mathbf{K}^{-1}\mathbf{C}^T\mathbf{N}$, and using Equation (24), we can write

$$\mathbf{MN} = \frac{1}{|E|} \mathbf{C}\mathbf{K}^{-1}\mathbf{C}^T\mathbf{N}.$$

By introducing a symmetric positive definite matrix, $\mathbf{M}'$, such that $\mathbf{M}'\mathbf{N} = \mathbf{0}$ we see that valid inner products $\mathbf{M}$ are on the form [18]

$$\mathbf{M} = \frac{1}{|E|} \mathbf{C}\mathbf{K}^{-1}\mathbf{C}^T + \mathbf{M}' = \mathbf{M}^* + \mathbf{M}'. \tag{25}$$

If we let, $\mathbf{Q}_\mathbf{N}^\perp$, denote an orthonormal basis to the nullspace of N^T, and, $\mathbf{S}$, be any symmetric positive-definite matrix, we can write write $\mathbf{M}' = \mathbf{Q}_\mathbf{N}^\perp \mathbf{S} {\mathbf{Q}_\mathbf{N}^\perp}^T$.

It is easy to see that $\mathbf{M}$ is symmetric. Furthermore, the following argument, as provided in [12, 18], assures us that $\mathbf{M}$ is actually SPD. First we introduce an arbitrary vector, $\mathbf{z} = \mathbf{N}\mathbf{z}^* + \mathbf{z}' \neq \mathbf{0}$, such that $\mathbf{N}^T\mathbf{z}' = \mathbf{0}$. Recalling that we need $\mathbf{z}^T\mathbf{M}\mathbf{z} > 0$ for $\mathbf{M}$ to be PD, we start by assuming $\mathbf{z}' = \mathbf{0}$. That is

$$\mathbf{z}^T\mathbf{M}\mathbf{z} = \mathbf{z}^*\mathbf{N}^T\mathbf{M}^*\mathbf{N}\mathbf{z}^* + \mathbf{z}^*\mathbf{N}^T \underbrace{\mathbf{M}'\mathbf{N}}_{=\mathbf{0}} \mathbf{z}^* > 0.$$

The last part holds because we know from Equation (24) that $\mathbf{C}^T\mathbf{N}$ is of full rank. Conversely, if $\mathbf{z}' \neq \mathbf{0}$, we see that we have

$$\mathbf{z}^T\mathbf{M}\mathbf{z} = \underbrace{\mathbf{z}^T\mathbf{M}^*\mathbf{z}}_{\geq \mathbf{0}} + \underbrace{\mathbf{z}'^T\mathbf{M}'\mathbf{z}'}_{> \mathbf{0}} > 0.$$

An argument similar to this holds for the inverse inner product given as

$$\mathbf{T} = \frac{1}{|E|}\mathbf{N}\mathbf{K}\mathbf{N}^T + \mathbf{Q}_{\mathbf{C}}^{\perp}\mathbf{S}\mathbf{Q}_{\mathbf{C}}^{\perp T}. \tag{26}$$

Where $\mathbf{Q}_{\mathbf{C}}^{\perp}$ is a orthonormal basis to the nullspace $\mathbf{C}^T$.

In the parallel implementation, that we describe in Section 6, we will be using the ip_simple inner product, as given in [18]. Which we restate here, without proof:

$$\begin{aligned} \mathbf{Q} &= \mathrm{orth}(\mathbf{A}^{-1}\mathbf{N}), \\ \mathbf{M} &= \tfrac{1}{|E|}\mathbf{C}\mathbf{K}^{-1}\mathbf{C}^T + \tfrac{d|E|}{6\,\mathrm{tr}(\mathbf{K})}\mathbf{A}^{-1}(\mathbf{I} - \mathbf{Q}\mathbf{Q}^T)\mathbf{A}^{-1}. \end{aligned} \tag{27}$$

Here d refers to the dimension of the problem; either 2 or 3. In particular, we will be using the approximate inverse

$$\begin{aligned} \mathbf{Q} &= \mathrm{orth}(\mathbf{A}\mathbf{C}), \\ \mathbf{T} &= \tfrac{1}{|E|}\left[\mathbf{N}\mathbf{K}\mathbf{N}^T + \tfrac{6}{d}\,\mathrm{tr}(\mathbf{K})\left(\mathbf{I} - \mathbf{Q}\mathbf{Q}^T\right)\mathbf{A}\right]. \end{aligned} \tag{28}$$

4 Parallel Computing

For a long time the continual improvement of microprocessor design has made computers faster. Effectively meaning that programs written for these systems would become gradually faster along with the developing hardware. This increase in processing speed is often referred to as 'Moore's Law'. More precisely, Moore's Law says that the number of transistors on an integrated circuit doubles approximately every two years. This, however, is beginning to change. The main reason is the growing amount of heat produced from these circuits when the clock speed is increased [19]. As a result the manufacturers of microprocessors have started putting several processors on single integrated circuits [20]. This change requires developers to change the way they write code. Simply running programs on systems with several processing cores is not likely yield better performance, that is, unless the code is specifically written to take advantage of the parallel processing capabilities.

Along with the desire for code that makes greater use of parallel hardware comes the corresponding requirement that the algorithms can be parallelized. For instance if we wish to sort a sequence of numbers, we can not simply distribute the numbers across the available processors and then sort each 'sublist'. If we put these sorted sublists together again we will quickly see that the list is in fact not sorted at all. Hence, some form of communication between the processors is needed to get the correct result. On the other hand, some problems do not require any communication to take place. An example is numerically finding the value of $\int_a^b f(x)\,\mathrm{d}x$. Here we can simply have each each core calculate the value of subsections of the interval $[a, b]$. But also here we need to sum the partial sums of the integral when all processes have finished calculating their part. Problems where there is no communication needed between the processing cores are often said to be 'embarrassingly parallel'. We note, however, that at the very least, some of the execution time is spent preparing for the parallel operations to be executed.

4.1 Amdahl Versus Gustafson

A central term when when discussing parallel software is 'speedup'. Speedup is commonly denoted, S_p. This refers to the ratio between the execution time of the sequential (serial) algorithm, T_1, and the execution time of the parallel algorithm, T_p, using p processors. That is, we have

$$S_p = \frac{T_1}{T_p}.$$

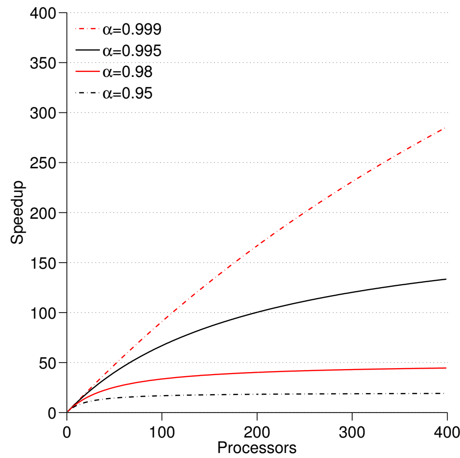


Figure 9: Speedups for programs with different serial fractions as given by Amdahl's law (29).

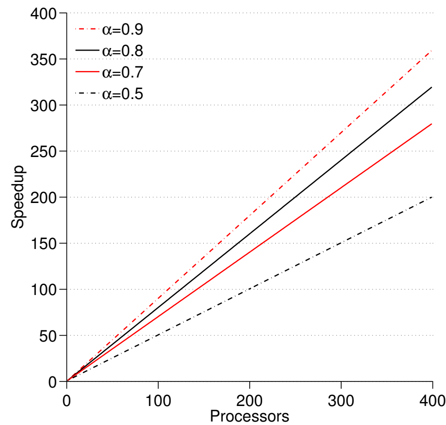


Figure 10: Speedups for programs with different serial fractions as given by Gustafson's law (30).

So, if the execution time when using two processors is half of the execution time when using only one processor, we say that we get a speedup of two. When $S_p = p$, we say that the speedup is ideal. In rare cases one can observe so-called 'super linear' speedup, that is, $S_p > p$. The most common cause of this is that when a problem is split into smaller parts, more of the subproblems can be stored in faster cache on the processor. Thus speeding up the calculations relative to solving on one processor.

Most of the time, however, we can not hope to be that lucky. This brings us to another useful metric; namely 'parallel efficiency',

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p},$$

which tells us something about how much of the parallel potential of the hardware we are utilizing.

In 1976 Gene Amdahl published his paper 'Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities' [8], which gave rise to what today is known as 'Amdahl's law'. Essentially, Amdahl's law states that the highest possible speedup achieved by parallelizing a program, is limited by the serial section of the program. This has later been described mathematically, and is presented, among many places in [20]. If we assume that the parallel fraction of a program is α, that the serial section is $1 - \alpha$, and that we use p processors, we get the following expression for the speedup:

$$S_p = \frac{\text{execution time on one processor}}{\text{parallell execution time}} = \frac{1}{(1-\alpha) + \frac{\alpha}{p}}. \tag{29}$$

We see that when $p \to \infty$ we can write the maximum available speedup for a program with a certain serial fraction as

$$S_\infty = \frac{1}{1-\alpha}.$$

Now, if we have a program which is 99% parallelizable, we see that S_∞ is 100. This drops alarmingly quick for larger serial sections. For instance, it is halved for a program which is 98% parallelizable. A plot of S_p for programs with different values of α is given in Figure 9. Having said this, one could quickly start to wonder why we would be very interested in parallelizing code at all. However, there are a few more things to consider.

In 1988 John L. Gustafson presented an article called 'Reevaluating Amdahl's

Law' [15], where he introduces an idea that has since come to be known as 'Gustafson's law'. Gustafson's law suggests that larger data sets can be more efficiently parallelized. Now, we introduce the idea that on a parallel computer, the time spent in the serial section of the program is $1 - \bar{\alpha}$, whereas the time spent in the parallel section is $\bar{\alpha}$. The serial execution time would then be $(1 - \bar{\alpha}) + p\bar{\alpha}$. This yields a speedup

$$\bar{S}_p = \frac{1 - \bar{\alpha} + p\bar{\alpha}}{(1 - \bar{\alpha}) + \bar{\alpha}} = 1 - \bar{\alpha} + p\bar{\alpha}. \tag{30}$$

This is a linear function, but we have plotted $\bar{S}_p$ for a few different values of $\bar{\alpha}$ in Figure 10 nonetheless. If we compare this with the speedups from Amdahl's law in Figure 9 this looks rather more promising. We also note that if the parallel section, $\bar{\alpha}$, approaches 1, as we could hope for when the problem size is increased, the speedup approaches p.

What we can conclude from this is that there are several things to think about when writing parallel code. Simply having the most powerful computers is not enough if too much of the code needs to be run serially. However, we can hope to solve problems in greater detail by increasing the size of the calculation and at the same time using more computing power. As Gustafson states:

> One does not take a fixed-sized problem and run it on various numbers of processors except when doing academic research; in practice, the problem size scales with the number of processors. [15]

5 Matlab Parallel Computing Toolbox

The Matlab Parallel Computing Toolbox (PCT) is a comprehensive parallel extension to Matlab. It allows for up to twelve local parallel processes, or Matlab workers, on a multi-core desktop. If accompanied by the Matlab Distributed Computing Server software you can also run parallel jobs across a cluster of computers [2]. PCT has several parallel constructs that makes it possible to run jobs in batches, or to run interactive parallel code. There are two different constructs that can be used within Matlab scripts, as well as within batch jobs, namely the `parfor` loop, and the `spmd` construct. Here `parfor`, naturally, is short for 'parallel for loop', whereas `spmd` is short for Single Program Multiple Data. The `parfor` loop is in many ways similar to the `parallel for` directive used in OpenMP. Whereas the latter is a common programming model used in, among others, the Message Passing Interface (MPI) for C and Fortran.

We now give an introduction to the Parallel Computing Toolbox, some examples of use, and some issues to be aware of. The performance of the PCT is likely to depend heavily upon the hardware, and on the algorithms in question. For this reason, we present some performance benchmarks, but we note that these are meant as examples of behaviour, and of what we can hope to achieve, rather than objective measurements of performance. We have focused on features of PCT that are important for the development of the parallel Multiscale Mixed Finite-Element Method that we describe in 6, although we briefly describe some other functionality as well. The interested reader is advised to review the more comprehensive guide to PCT, which can be found in [2]. The code in the following sections has been tested in Matlab 2011b, with Parallel Computing Toolbox version 5.2.

5.1 Setting up the Environment

In the rest of this Section we will look at some examples of how the Parallel Computing Toolbox can be used on a multi-core computer; not on a distributed cluster of computers. The programming techniques are similar, but distributed systems carry with them some extra considerations especially with regards to distribution of data, to reduce communication overhead. The contents of the following sections are based on [2], on the documentation in Matlab, and on our experiences when developing the code presented in Section 6.

Both the `parfor` loop, as described in Section 5.2, and the `spmd` construct, as described in Section 5.3, require a Matlab pool to be running. However, we

Listing 1: Simple example of how the `parfor` loop is used.

```
A = rand(n,1)
parfor k = 1:n
  A(k) = A(k) + 1;    % allowed
end

parfor k = 1:n
  % not allowed as it will yield unpredictable results
  A(k) = A(k+1) + 1;
end
```

note that a `parfor` loop can be written such that if there are no active workers it will simply execute as an ordinary `for`-loop. The `spmd` construct will also execute the block in serial if there is no MATLAB pool running, but this might result in unexpected results if care has not been taken to handle this case. In particular for more complicated code. Starting a matlabpool is done with the `matlabpool` command. This will start up as many as twelve local workers, depending on the architecture. The number of workers can also be overridden when invoking the command. PCT also allows for scheduling jobs to run in parallel, this is described further in Section 5.4.

When starting any number of workers these workers will be separate MATLAB processes in addition to the main, or host, process. They require an amount of memory similar to the host process. In addition there is no way to directly share memory between these processes, meaning that if the whole of an array is to be used on several of the processes, it will be copied in memory to all relevant workers. In the remainder of the text we will refer to an ordinary MATLAB instance as the 'host', and MATLAB instances started via the `matlabppol` command, as 'workers'.

Finally, we mention that PCT allows for a so-called parallel mode, or `pmode`. This gives you access to a terminal interface for each worker. This will not be discussed any further in this paper.

5.2 The Parallel For Loop

The simplest construct is the `parfor` loop. This behaves much like an ordinary `for` loop, but there is one important restriction: when accessing different indices of a variable in a `parfor` loop, one iteration of the loop can not depend on other iterations of the same loop. A variable that has this property is called a sliced variable. This is illustrated in Listing 1.

Listing 2: Classification of variables in a `parfor` loop.

```
a = 999;
c = pi;
z = 0;
r = rand(1,10);  % sliced variable

parfor i = 1:10  % i is a loop variable
  a = i;         % a is a temporary variable
  z = x+i;       % z is a reduction variable
  b(i) = r(i)    % b is a sliced variable
  if i <= c      % c is a broadcast variable
    d = 2*a;     % d is a temporary variable
  end
end

disp(a)          % will display 999
```

There are also a few other considerations to make when using a `parfor` loop. MATLAB will automatically understand if you are using a variable as a reduction variable, provided that you remember to initiate the variable outside of the loop. Similarly, MATLAB will understand if you use a variable inside the loop as a temporary variable, and the last value assigned to temporary variable will not be accessible after the loop has finished execution. If a variable is given a value before the `parfor` loop, and used as a constant within the loop, the variable is known as a broadcast variable. An illustration of these distinctions is given in Listing 2.

We can see how this construct could be used to speed up `for` loops by simply changing them into `parfor` loops, or at least without having to do any major changes to the existing code. The `parfor` loop is also useful if one wants to do things like Monte Carlo simulations where a great amount of data is needed.

5.3 The spmd Construct

The `spmd` construct is considerably more powerful than the `parfor` loop. `spmd` is short for Single Program Multiple Data, which hints that construct allows us to run the same section of code in parallel, with different input data. Unlike the `parfor` loop the `spmd` construct does not recognize reduction variables, but it has a number of features that are not supported in `parfor` loops.

To access data across workers, when using the `spmd` construct, one can use either composite or distributed variables. An example of how composite variables

Listing 3: Example of how the `spmd` construct is used.

```
spmd(p)                % p is number of wanted workers
  b = labindex;        % assign thread number to b (composite)
  A = 10*b+rand();     % assign value to A (composite)
end

for i = 1:p
  disp(A{i})           % access values assigned to A (composite)
end
```

are used is given in Listing 3. A composite variable that has the same name and content on all workers is known as a replicated array. On the other hand, if the content of the array varies across the workers, the array is said to be variant. It is not necessary to create a composite array on all workers, thus if an array only exists on some of the workers, the array is called a private array.

Combined with `labBroadcast`, composite variables provide a convenient way of distributing replicated variables. This is illustrated in Listing 4. Distributing a variable in this way is considerably more memory efficient than simply introducing a variable on the host and then naively using it on the workers. This naive solution seems to work well, and requires less code, but yields unpredictable results for larger variables. We illustrate this in Listing 5. The naive solution causes spikes in memory that reach around the double of the amount of memory that is required to actually store the broadcast variable on all workers[11]. Using `labBroadcast`, on the other hand, causes a considerably smaller spike in memory usage, if at all.

Naturally it is not always enough to simply broadcast variables to all workers. PCT also provides functions for sending and receiving data on each worker. Namely `labSend` and `labReceive`. The usage is quite intuitive, as we see in Listing 6. When sending and receiving messages between parallel workers like this, it is essential to note that calls to `labReceive` will block until a corresponding call is made to `labSend`. Sending and receiving in this fashion does not cause the previously mentioned spikes in memory usage.

Another useful feature is the ability to partition arrays across workers with the `codistributed` and `distributed` commands. By default the partitioning will happen column-wise for a two-dimensional array, but this can easily be manipulated with ordinary MATLAB commands, or by using other distribution constructs.

[11] Most likely these spikes are because of send and receive buffers on the host/workers. These spikes are evident when using `parfor` loops as well.

Listing 4: Distributing variables using `Composite`.

```
% initiate variable, c1, on worker 1,
% and distribute to the others

c1 = Composite(); c1(:) = cell(1,matlabpool('size'));

spmd
  if labindex == 1
    c1 = rand(10,1);
  end
  c1 = labBroadcast(1,c1);
  % use c1
end

% initiate variable, c2, on the host,
% and distribute to the workers

c2 = Composite(); c2(:) = cell(1,matlabpool('size'));
c2{1} = rand(10,1);

spmd
  c2 = labBroadcast(1,c2);
  % use c2 for something
end
```

Listing 5: Example of spike in memory when 'distributing' a large variable to workers naively, compared to using `labBroadcast`. Notice the comparably high usage of memory required by part 2.

```
% assuming there are no active workers

N = 2^27; % 2^27 doubles requires 1 GB of memory.

% part 1
matlabpool('open',4);
c1 = Composite(); c1(:) = cell(1,4);
spmd % highest amount of memory used inside block: 4.5 GB
  if labindex ==  1
    c1 = rand(1,N);
  end
  c1 = labBroadcast(1,c1);
  x1 = sum(c1(:)); % do something with c1
end
% memory used by MATLAB here: 4.5 GB

% part 2
matlabpool('close'); matlabpool('open',4);
c1 = rand(1,N);
spmd % highest amount of memory used inside block: 11 GB
  x2 = sum(c1(:)); % do something with c1
end

% memory used by MATLAB here: 5.5 GB
% +1 GB because c1 is initialized on host
```

Listing 6: Using `labSend` and `labReceive`.

```
% contents (and size) of A will be different on each worker

getArrForWorker = @(w) rand(1,w);
spmd
  if labindex == 1
    A = getArrForWorker(1);
    for lab = 2:numlabs
      % may return before corresponding call to labReceive
      labSend(getArrForWorker(lab),lab); % send to lab
    end
  else
    % blocking function
    A = labReceive(1); % receive from worker 1
  end
  % use A on workers
end
```

Listing 7: Distributing and accessing distributed arrays.

```
% two blocks that result in the same distributed array B
% with the same local array C
% note that there might be differences in efficiency
% depending on the systems and operations involved

A = magic(10);
B = distributed(A);
spmd
  C = getLocalPart(B);
end

% gives same result as the section above
spmd
  A = magic(10);
  B = codistributed(A);
  C = getLocalPart(B);
end
```

Listing 8: Reconstructing a distributed array.

```
% reconstructing the distributed array A
% cell arrays behave in a similar fashion

A = distributed.rand(2^10,1);

% assign entire A to B, on host
B = gather(A);

% assign entire A to composite variable, B, on worker i
spmd, B = gather(A,i); end

% assign entire A to composite variable, B, on workers
spmd, B = gather(A); end
```

The difference between the two commands is a subtle one, but essentially using `codistributed` within a `spmd` block gives the same result as using `distributed` outside a `spmd` block [2]. When an array is distributed in this fashion, only a part of the array is stored in local memory on each worker. For this reason, the PCT framework provides useful commands to get the part of a distributed array associated with each lab. An example is given in Listing 7. Similarly, one can reconstruct a distributed array, either on the host, or on any worker, as we show in Listing 8. Note that when indexing distributed arrays one always uses the global indices. To get the global indices for a distributed array on a worker one can use the `globalIndices` command. We also note that iterating over a distributed array within an `spmd`-block is likely to be slow, and as such, in most cases it will be faster to iterate over the local part. We illustrate this difference in Listing 9.

5.4 Scheduling Jobs

With PCT one can run jobs in parallel through a scheduler, as well as with the already discussed parallel blocks. For instance, it is possible to schedule a number of completely serial jobs to run in parallel on the available workers. When running scheduled jobs there is some overhead associated with starting the necessary workers. This will be particularly noticeable when the jobs does not have long execution time. Jobs can naturally be parallel, as well as serial in nature; they can contain `parfor` loops, `spmd` blocks, or they can be so-called parallel-jobs that have an environment for communication between the workers.

The simplest available form of a parallel job is provided by the `dfeval` function. This lets you evaluate a function handle in parallel, and provide different input for each worker. The number of executed workers depends on the number of input arguments to the function handle.

A more flexible way of scheduling jobs is provided by the `createJob` and the `createParallelJob` commands. The difference between the two commands is that the latter allows for communication between the workers, whereas the former does not. For this reason parallel jobs are synchronized, whilst non-parallel jobs will run as soon as the scheduler has ‘room’ for the job. To give an example of how a parallel job is executed we provide the code in Listing 10.

Listing 9: Iterating over local parts of a distributed array.

```
% (very) slow

B = distributed(rand(2^15,1));
tic
spmd
  indices = globalIndices(B,1);
  for i = indices
    B(i,1) = 1;
  end
end
toc

% the following is more than 200 times faster
% on our test machine

B = distributed(rand(2^15,1));
tic
spmd
  dist = getCodistributor(B);
  lpB = getLocalPart(B);
  for i = 1:length(lpB)
    lpB(i,1) = 1;
  end
% rebuild the codistributed array.
% the 'noCommunication' flag has an impact
% on performance for larger matrices.
B = codistributed.build(lpB,dist,'noCommunication');
end
toc
```

Listing 10: Scheduling a parallel job.

```
function res = testPar()
  h = @myfunk;
  sched = findResource('scheduler', 'type', 'local');
  pjob = createParallelJob(sched);
  set(pjob, 'FileDependencies', {'tmp.m'});
  set(pjob, 'MaximumNumberOfWorkers', 4);
  set(pjob, 'MinimumNumberOfWorkers', 4);
  t = createTask(pjob, h, 1, {});
  submit(pjob);
  waitForState(pjob);
  res = getAllOutputArguments(pjob);

  destroy(pjob);

  function res = myfunk(varargin)
    snd = [2 3 4 1]; % i sends to snd(i)
    rcv = [4 1 2 3]; % i receives from rcv(i)
    labSend(labindex*10,snd(labindex));
    A = labReceive(rcv(labindex));
    res = A;
    end
  end
```

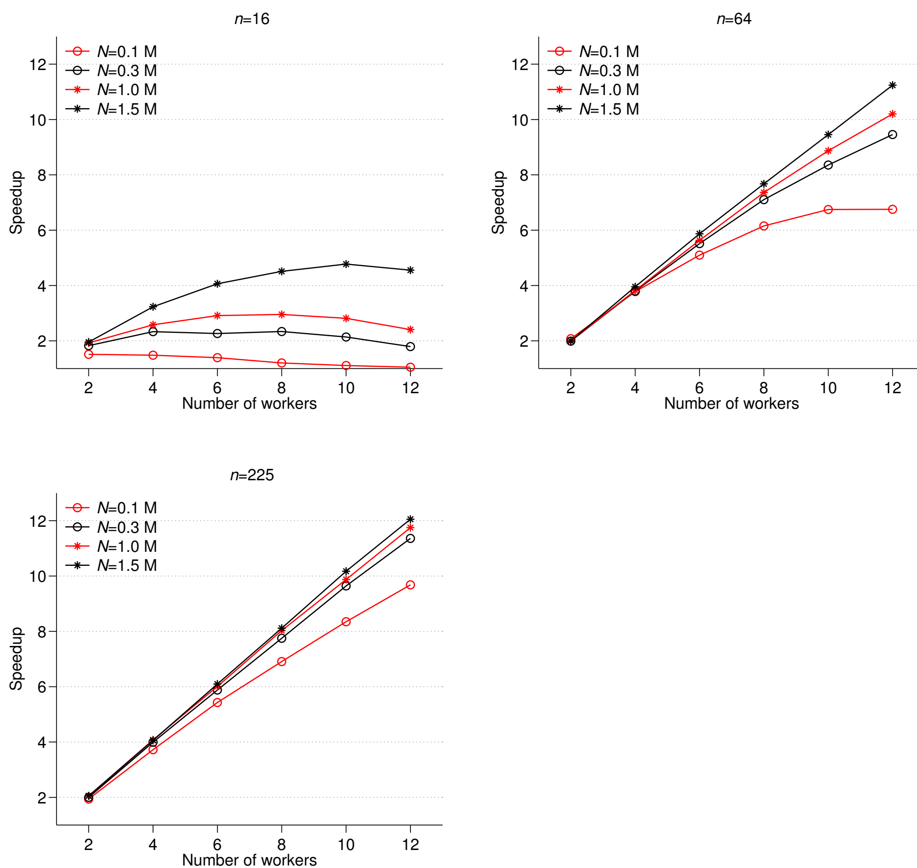


Figure 11: Speedup when using PCT to solve sparse $n \times n$ systems.

5.5 Performance

Because MATLAB is a high-level programming environment it is not surprising if it sometimes does not give as good a performance as more low-level, compiled, programming languages such as C/C++ and Fortran. The same caveat is naturally also an issue when using PCT. The parallel environment provided by PCT provides easy-to-use functions for distributing jobs, running blocks of code in parallel, as well as communication between workers. This ease of use might be at the expense of performance, in terms of execution speed. On the other hand, programming within MATLAB and PCT means that many things are more easily available; for instance plotting, profiling and debugging.

To get some idea of how well PCT performs we have solved N systems of equations, $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} \in \mathbb{R}^{n \times n}$, and $\mathbf{b} = (1, \ldots, 1)^T$. The systems have been solved for $\mathbf{x}$ using `mldivide`. We dividing the N systems across p workers. We have used matrices, A, that are sparse, symmetric, and block tri-diagonal, with a block size of $\sqrt{n} \times \sqrt{n}$. The number of non-zero elements is $\mathcal{O}(5n)$, and the bandwidth is n. The resulting speedups, for $p = \{1, 2, 4, \ldots 10, 12\}$ workers, when varying problem sizes, $n = \{16, 64, 225\}$, and number of equation systems, $N = \{10^5, 3 \cdot 10^5, 5 \cdot 10^5, 10^6, 1.5 \cdot 10^6\}$, are given in Figure 11. We see that we get quite good speedup as long as the amount of work is large enough. That is, we need the the work to be load balanced, and the overhead, associated with setting up the parallel workers, to be small compared to the time spent solving the problems.

5.6 Some Issues

When working with PCT we have experienced two issues worth mentioning. Both are related to MATLAB's implicit parallelism. That is, they arise because MATLAB has parallel implementations of many built-in functions. Among others we mention matrix-matrix multiplication (`mult`), and Gauss elimination (`mldivide`). Most of these functions utilizes Basic Linear Algebra Subroutines (BLAS).

Utilization of multi-core environments is now default behaviour in MATLAB, but all workers are still single-threaded. This means that performing the exact same actions, involving implicitly parallel operations, is likely to have different run times on workers compared to in the 'regular' MATLAB environment. Hence, when parallelizing code that already makes good use of implicit parallelism it will be more difficult, or even impossible, to achieve any significant speedup. It is possible to run all code in MATLAB in single threaded mode by starting MATLAB with the flag `-singleCompThread`.

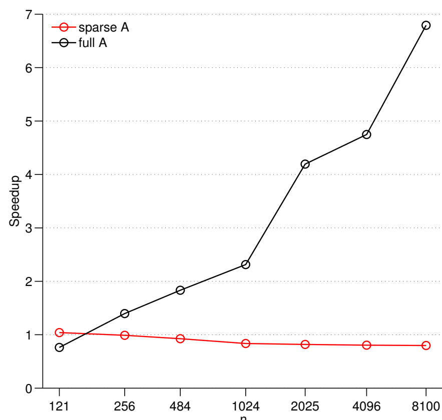


Figure 12: Implicit speedup for MATLAB's `mldivide` when solving $Ax = b$ for x for both full and sparse A. A is of size $n \times n$. The test was performed on a system with 12 CPUs.

In order to test the efficiency of the implicit parallelism of `mldivide` we have solved two different systems of equations. Both on a MATLAB worker, and on the host. One system is a full, random matrix, $\mathbf{A}$, and the other is a sparse, symmetric, block tri-diagonal matrix $\mathbf{A}_s$. In both systems $\mathbf{b}$ is a vector of ones. The measured speedups for various matrix sizes, $n \times n$, are given in Figure 12. We see that the implicit speedup is quite good for the full system, whereas we get a speedup of less than 1 for the sparse system. We know that the algorithm used by `mldivide` depends on the properties of the systems involved. For this reason it is not surprising that there is a difference in performance. The fact that there is such a large difference is however useful knowledge[12].

The other issue we feel is worth mentioning, is that one might experience numerical deviations between identical operations performed on a worker and in the regular MATLAB environment. This is in no way unexpected behaviour, as floating point operations on computers are not generally commutative. However it might be useful to be aware if this behaviour when comparing results. We illustrate this behaviour by solving the system $\mathbf{Ax} = \mathbf{b}$ for random matrix, $\mathbf{A}$, with, $\mathbf{b}$, as a vector of ones. We do this for various sizes of $\mathbf{A}$, both on a worker, and in the

[12]The low speedup associated with the implicit parallelism, when solving even fairly large, sparse systems, means that we can hope for a considerable gain in speedup when using PCT in the parallel Multiscale Mixed Finite-Element Method that we introduce in Section 6.

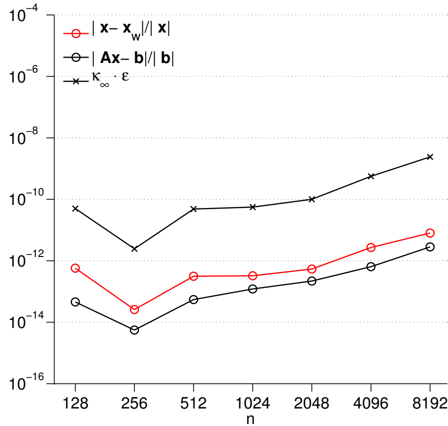


Figure 13: Resulting relative residuals, $\mathbf{Ax} - \mathbf{b}$ when the `mldivide` function is used to solve a random system of equations of size, n, as well as the relative deviation introduced to the answer as a result of floating point commutativity. $\mathbf{x}$ is the answer obtained in the regular MATLAB environment, and $\mathbf{x}_w$ is obtained on a worker. The norm in question is the infinity norm, $||\cdot||_\infty$. κ_∞ refers to the condition number of $\mathbf{A}$, and ε is the floating point relative accuracy of MATLAB. We note that the relative residuals are proportional to κ_∞, as they should be.

regular MATLAB environment, using `mldivide`.

Now, we compare the relative magnitude of the residuals, $\mathbf{r} = \mathbf{Ax} - \mathbf{b}$, as well as the relative difference between the two obtained unknown vectors; $\mathbf{x}$, from the regular MATLAB environment, and $\mathbf{x}_w$, as obtained on the worker. The results are given in Figure 13. We observe that there are deviations between the serial and the implicitly parallel results. We can also note that the relative deviations are smaller than the relative magnitude of the residuals.

One possible way of avoiding having to deal with this discrepancy is to run MATLAB with the `-singleCompthread` flag, as indicated above. Another option is to run the implicitly parallel code inside an `spmd` block, and on one worker only. Since the workers are single-threaded this will keep multi-threaded BLAS from being invoked.

6 A Parallel Multiscale Mixed Finite-Element Method

As we note in Section 3.3, the construction of the basis functions in Equation (16) in the Multiscale Mixed Finite-Element Method is a so-called embarrassingly parallel task. That is, there is no communication needed between the workers that are involved in constructing the required basis functions for a system. The same is the case for calculating inner products, **T** of each cell, or the corresponding inverse inner products **M** (See Equation (13)). Now, with the MATLAB Reservoir Simulation Toolbox [3] as a starting point, along with the Parallel Computing Toolbox as described in Section 5, we have developed a series of functions that let us take advantage of parallel computing environments with up to twelve workers.

We have implemented code that together performs the following tasks:

- initiate workers
- initiate variables that define the problem,
- distribute variables that define the problem,
- calculate mimetic inner products in parallel,
- distribute the inner products,
- construct basis functions in parallel.

For brevity we will refer to this new set of functions as `xmsmfem`. The functions that make up `xmsmfem` utilizes functionality and data structures from the MATLAB Reservoir Simulation Toolbox to the largest extent possible. That is, we make use of the grid structure, rock structure, fluid structure, partitioning structure, boundary condition and source structure, and coarse grid structure from MRST.

In addition, a new structure, for storing the inner products, has been introduced (See Section 6.1.2). It should be quite possible to understand the main features of `xmsmfem` without being intimately familiar with MRST, however, some basic knowledge is recommended. In particular about the previously mentioned structures. We recommend [18], or `http://www.sintef.no/Projectweb/MRST/Tutorials/`, for some examples and tutorials on MRST.

First we will discuss the key elements of `xmsmfem`. Here we also present some pseudocode that highlight the key elements of the implementation. If you wish to obtain the code you should see Section 6.2. The interested reader can find a complete working example in Appendix A. We present the results, in terms of

performance, in Section 7, along with a discussion. A brief conclusion is given in Section 8.

6.1 Some Notes on the Implementation

We have made an attempt to use as much of the built-in functionality in the Parallel Computing Toolbox as is possible. When considering the operation of computing inner products, and then constructing basis functions for each coarse face in light of the discussion of PCT in Section 5, it would seem ideal to simply use `parfor` loops for all these operations. This way one could simply iterate over all cells, when computing inner products. And similarly, one could iterate over all faces when building the basis functions. This method could be very sparse in the way of code, and thus, easy to understand. After some testing, however, we settled on using the `spmd` construct in all functions. This provides somewhat more control, and is not much more complicated. Additionally, using `spmd` for everything, makes the structure of the code more similar across the various functions of `xmsmfem`.

6.1.1 Initializing and Broadcasting the Problem

The typical flow of a program using `xmsmfem` is given in Listing 11. We note that indexing in the pseudocode is done mimicking the way indexing is done in MATLAB. In particular, if we write `A[i]`, then `i` can be either a single index, or a range of indices. We start by initializing the number of wanted workers. Then we initialize the variables that define the problem. This involves geometry and permeability, as well as boundary conditions and well structures, and is done on worker 1 only. We have chosen to do the initialization of the problem on only one worker before we broadcast the variables. This simplifies things if, for instance, one is generating a random permeability field, or reading a geometry from disk.

MATLAB workers are completely separate processes. Hence they have completely separate blocks of memory as well. Even if they are running on the same machine as the host. For this reason one would benefit memory-wise from having only the parts of each variable that will used, stored on the relevant worker. Particularly since the memory footprint of each MATLAB process, whether host or worker, is fairly large. However, we have decided to distribute the structures to all workers, for simplicity. This way we can use the available functions in MRST on the workers easily. The main structures that are distributed are the following:

- fine geometry structure,

- coarse geometry structure,
- partitioning,
- rock structure,
- fluid structure,
- boundary conditions and source structures.

One reason in particular that made us decide to abandon the `parfor` loop is this decision to broadcast the key structures. Composite variables are well suited for this, as we illustrate in Listing 5, however, `parfor` loops do not accept composite variables. In addition, we note that `parfor` loops suffer from the same spikes in memory usage, as illustrated in Listing 5, when broadcasting variables in the naive way.[13]

6.1.2 A New Inner Product Structure

First, we note that we have chosen to implement the Multiscale Mixed Finite-Element Method with the ip_simple inner product [18]. Allowing only one inner product simplifies the code a little, whilst at the same time demonstrating the ideas. Extending the code to utilize other local flow problem solvers, when constructing the basis functions, should not be too extensive a task. However, it is outside the scope of this project.

In MRST the inner products are stored in the sparse block diagonal matrix `BI`, where each block matrix contains the inner product associated with one cell. Remembering the formulation of the mimetic method from Equation (17), and assuming we have n cells, we can write this as

$$\mathtt{BI} = \begin{bmatrix} \mathbf{T}_1 & & \\ & \ddots & \\ & & \mathbf{T}_n \end{bmatrix}.$$

We illustrate the process of constructing `BI` in Listing 12. One possible way of using `BI` when constructing basis functions in parallel is to simply distribute the matrix to all workers; much in the same way as the ‘problem variables’ are distributed. If one wishes to avoid the overhead of storing the inner products of all cells on

[13]When the amount of required memory got larger than the available memory on ‘Kongull’ because of the memory spikes, the compute node would become entirely unresponsive. And a complete manual restart was necessary.

Listing 11: Pseudocode that describes the typical main program of `xmsmfem`. Note that some details have been omitted for simplicity. A complete working example of `xmsmfem` can be seen in Appendix A.

```
function main():
  initiateWorkers(procs) % start procs workers

  parallel:
    if worker == 1:
      n     := [n1,n2,n3] % number of cells
      N     := [N1,N2,N3] % number of blocks
      G     := loadGeometry(n)
      rock  := loadPermeability(N)
      p     := partitionCoarseGrid(G,N)
      CG    := generateCoarseGeometry(G,p)
      fluid := initiateFluid(G)
      bc    := setBoundaryConditions(G)
      mob   := calculateMobility(G,rock,fluid)
    end if

    % The broadcasted variables can be used on
    % workers as normal, local, variables.
    % They are, however, still listed in the following
    % function calls

    broadcastVariables(G,rock,p,CG,fluid,bc)
  end parallel

  % XBI is a distributed array

  XBI := computeMimeticIP(G,rock)

  % XBI[w] represents inner products needed on worker, w.
  %   C[w] contains indices of cells in worker, w.
  %   F[w] contains indices of faces on worker, w.
  % Hence if C[w][i] == k we know that the inner product of
  % the global cell with index k is stored in XBI[w][i].
  % Similarly, if F[w][f] == j we know that the basis
  % function of global coarse face j will be
  % computed on worker w.

  [XBI C F] := distributeIP(XBI,G,p,CG,bc)

  XCS := buildBasis(G,CG,p,rock,mob,bc,XBI,C,F)

end function
```

Listing 12: Pseudocode that illustrates the computation, and storage, of inner products in MRST

```
function computeMimeticIP(G,rock)
  for i in [1,...,numberOfCells]:
    f := faceIndicesOfCell(i,G)
    BI[f][f] = ip(i,G,rock)
  end for

  return BI
end function
```

Listing 13: Pseudocode that describes the parallel computation if inner products.

```
function computeMimeticIP(G,rock):
  parallel:
    % Cells are distributed block-wise, and in such a fashion
    % that the number of cells on each worker is as similar
    % as possible.
    c := cellsOnWorker(worker,G)
    for i in c:
      XBI[i] := ip(i,G,rock)
    end for
  end parallel

  return XBI
end function
```

all workers, however, this is not an option. Additionally, distributing the relevant parts of `BI` to each worker is quite possible, but somewhat complex. Storing parts of `BI` on each worker will also cause us to perform a lot of indexing into sparse matrices.

Another concern is that, if we wish to compute inner products in parallel, neither composite variables nor distributed variables seem that well suited to let us store the resulting inner products directly into a sparse matrix. One would need to store the inner products one by one, before constructing the final matrix serially when all inner products had been computed. For these reasons we started looking for a different solution.

Having already experimented with storing each inner product separately, before reconstructing `BI`, the answer came quite readily: we decided to store all the inner products individually, in a distributed cell array. That is, we spread the cells across the available number of workers, compute the inner products in parallel, and store them in a distributed cell array. Assuming we have n cells and p workers, we assign the first n/p workers to worker 1, the next n/p cells to worker 2, and so forth. If n does not divide into p we distribute the cells so that the number of cells on each worker is as even as possible. We refer to the new structure as `XBI` and, as an illustration, we can write

$$\texttt{XBI} = [\mathbf{T}_1, \dots, \mathbf{T}_n] \, .$$

We show the pseudocode for calculating the inner products using the new cell structure in Listing 13. We also note that when storing the inner products in this way, in addition to making it easier to distribute and store them, it is also quite easy to reconstruct the matrix `BI` because of the way distributed arrays can be reconstructed, as we show in Listing 8.

6.1.3 Distributing the Inner Products

Next, we need to distribute the inner products. Before we can do that, however, we need to decide which workers will compute which basis functions. We have chosen to simply distribute blocks of faces with consecutive indices, whilst trying to keep the number of faces on each worker as even as possible. Thus, if we have 10 faces and four workers, we will have the following distribution: worker 1 will have faces $\{1, 2, 3\}$, worker 2 will have faces $\{4, 5, 6\}$, worker 3 will have faces $\{7, 8\}$, and worker will have faces$\{9, 10\}$.

With the newly introduced inner product structure, and using the `spmd` construct, it proves quite straightforward to distribute only the inner products that

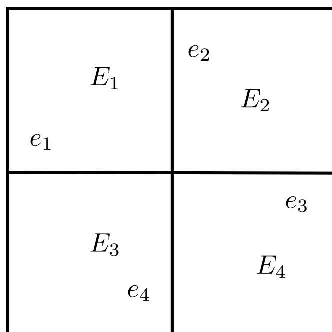


Figure 14: Four blocks, $E_1, \ldots E_4$, and global (internal) interfaces, $e_1, \ldots, e_4$.

will be used on each worker to that particular worker. There is some redundancy involved in this, however. Some inner products will need to be stored on several workers (in most cases). The easiest way of visualizing this redundancy is by considering Figure 14. Assuming that we calculate the basis functions for the four internal faces, e_i, concurrently on four workers, we see that we will need all inner products associated with two of the blocks, on each worker. This redundancy is addressed further in Section 7.4.

We distribute the inner products, using the PCT functions `labSend` and `labReceive`, in a similar fashion to the example given in Listing 6. The pseudocode can be seen Listing 14.

6.1.4 Building the Basis

Once the distribution of coarse faces has been decided, and the corresponding inner products have been distributed, we are ready to compute the basis functions. The pseudocode for this can be seen in Listing 15. We note that this function takes the arrays `C` and `F`, containing the information about the distribution, as well as `XBI`, which contains the inner products. Finally, the resulting system of basis functions is gathered to worker 1 and returned as the structure `XCS`. Now `XCS` can be used to solve the actual coarse system with the MsMFE method.

6.2 Getting the Code

`xmsmfem` is constructed as a module for MRST, and can be downloaded from `http://master.andershoff.net`. As with MRST, the code is released under the terms

Listing 14: Pseudocode that describes the distribution of inner products. Note that this function also decides which basis functions will be computed on which workers. That is, the coarse faces are distributed among the workers in this function.

```
function distributeIP(XBI,G,p,CG,bc):
  parallel:

    % Faces are distributed block-wise and such that
    % the number of faces on all workers is as similar
    % as possible.

    F[worker] := coarseFacesOnWorker(worker,G,CG,bc)
    b         := blocksConnectedToCoarseFace(f,CG)
    C[worker] := cellsInBlock(b,p)

    % F[w] contains indices of global coarse
    % faces in worker, w
    % C[w] contains indices of cells on worker, w

    BI := gather(XBI,1) % gather XBI to worker 1
    if worker == 1:
      for w in [1,...,numberOfWorkers]:
        sendTo(w,BI[C[w]])
      end for
    end if
    XBI[worker] := receiveFrom(1)
  end parallel

  return [XBI,C,F]
end function
```

Listing 15: Pseudocode that describes the construction of basis functions. Note that some details have been omitted for simplicity.

```
function buildBasis(G,CG,p,rock,mob,bc,XBI,C,F):
  parallel:
    for f in F[worker]:
      b := blocksConnectedToCoarseFace(f,CG)
      c := cellsInBlock(b,p)
      i := localIndicesOfIPs(c,C[worker])
      % corresponds to local hybrid system
      [sBI,sC,sD,sF,sG,sH] := buildLocalSystem(...
        XBI[worker][i],G,CG,bc,mob,rock)
      [v,p] := solveLocalSystem(sBI,sC,sD,sF,sG,sH)
      CS[f] := assignBasisFunction(v,p)
    end for
  end parallel

  % gather CS in worker 1
  XCS = gather(CS,1)
  return XCS
end function
```

of the GNU General Public License (GPL)[14]. To use the code, first make sure you have MRST installed. Then you can copy the folder containing the `xmsmfem` module into `MRSTROOT/modules/xmsmfem`. After starting MRST, `xmsmfem` can be initialized in MATLAB by running `mrstModule add xmsmfem`.

[14] `http://www.gnu.org/licenses/gpl.html`

7 Results and Discussion

To test the performance of `xmsmfem` we will look at two different geometries. We have measured the execution time of constructing basis functions using different numbers of workers, cells and block sizes. We decided to use the execution times associated with both distributing the problem variables, calculating and distributing the inner products, as well as constructing the basis functions. This is the time we refer to as the 'total execution time'. There is, however, also a considerable amount of execution time associated with both initiating the problem and constructing the geometry structure, as well as solving the actual coarse system. Since the parallelization of these operations have not been the focus of this paper, we do not consider these execution times.

7.1 Test Platform

The measurements of speedups, and other performance have been done on the computing cluster 'Kongull'. This system has 93 computing nodes; 44 of which has 48 GiB RAM, and the remaining 49 nodes have 24 GiB. Each node has two AMD 2431 Istanbul processors with six cores each, as well as 149 GiB 15000 RPM SAS hard drives.

The system in question runs MATLAB 2011b, with Parallel Computing Toolbox version 5.2, and MATLAB Reservoir Simulation Toolbox 2011b. This is also the configuration used for testing all the provided code. More details about the configuration of the system can be found here: `http://docs.notur.no/Members/hrn/kongull.hpc.ntnu.no/kongull-hardware-1/`.

7.2 Testing Speedup on a Cartesian Grid

We start by considering a quite artificial geometry, and some slightly extreme block sizes. We partition a perfectly cubic block into $144 \times 144 \times 144$ cells (just below 3 million cells). Furthermore, we divide the domain into equally sized, square blocks, consisting of n cells. We also generate a random, full, symmetric permeability tensor for each cell. Now, for $n = \{6^3, 8^3, 9^3, 12^3, 18^3, 24^3\}$, the resulting runtimes and speedups can be seen in Figure 15. We note that the speedups are compared to the runtime of `xmsmfem` on one worker, not the serial version that can be found in MRST.

First we emphasize that there are no effects from implicit speedups in this test. All operations that have potential implicit speedup have been run on workers, so

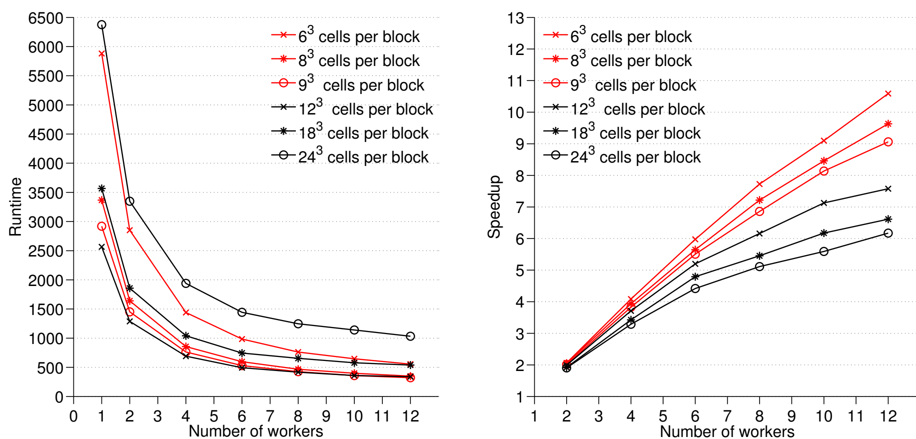


Figure 15: Speedups and runtimes on a cubic Cartesian grid.

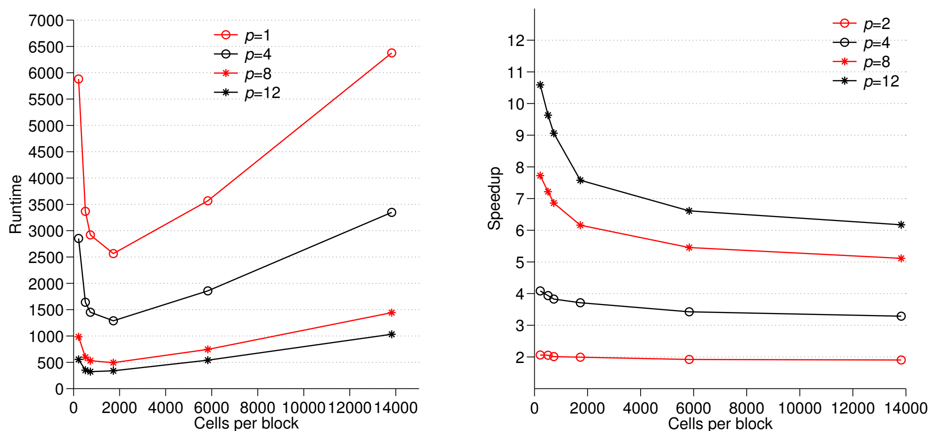


Figure 16: Execution times and corresponding speedups, for different combinations of block sizes and workers.

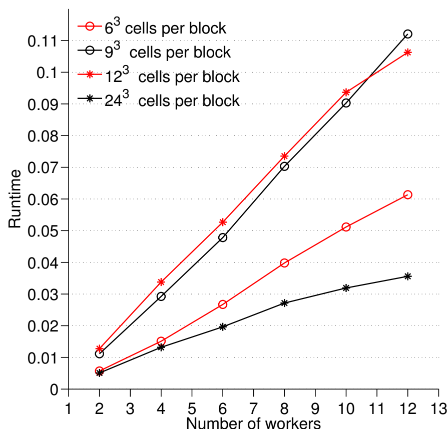


Figure 17: Amount of time associated with communications (distributing problem variables and inner products) relative to the total execution time.

they have been run with a single thread (See Section 5.6). The first thing we note is that all configurations have a significant speedup, ranging from just above 6 to just below 11, for twelve workers. The best speedup is for the smallest blocks, and the speedup decreases steadily with increasing block size. If we examine the execution times, however, the behaviour is somewhat surprising. In particular, the two overall highest runtimes belong to the smallest, and the largest block configurations respectively. This is displayed more clearly in Figure 16. Here we also see that there is a notable drop in execution time for block sizes around 1500 to 2500, before they steadily climb again.

Furthermore, we plot the fractions of the runtimes associated with communication in Figure 17. Here we also notice that the largest and the smallest block configuration stands out. They have quite small fractions for twelve workers, whereas the middle configurations are considerable larger. For 12 workers, the communication fraction is close to 0.1, for the medium-sized block configurations, around 0.03 for the largest block, and 0.06 for the smallest block configuration. If we, on the other hand compare the time spent computing and distributing the inner products, relative to the total execution time, the result is quite different. We see this in Figure 18. This is a suggestion that for the very large block configurations the computation of basis functions dominate the execution time.

For all but the largest block configurations we consider these results to be quite good. The perhaps most interesting result is that the speedup decreases with increasing block size. That is, when we solve fewer, larger, systems. We have been

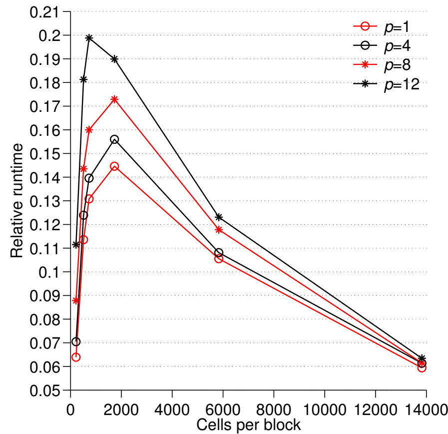


Figure 18: Time spent computing and distributing inner products relative to total execution time.

unable to find a satisfactory explanation for this behaviour. However, we speculate that it could be due to cache effects, since `mldivide` is called on larger systems.

7.3 Testing Speedup on a More Realistic Geometry

Now, we look at a slightly more realistic geometry. We have generated a mesh using the `makeModel3` function from MRST. This function lets us generate geometries of arbitrary resolution. An example of a partitioned geometry resulting from this function can be seen in Figure 19. We note that the geometry has two faults, and a layered structure. We have used a randomly distributed log-normal permeability distribution. All off-diagonal elements of the permeability tensor are zero. In all test cases we have used a resolution of $x \times 200 \times 15$ cells, where $x = \{400, 600, 800, 1000\}$. This corresponds to around $1.0 \cdot 10^6$, $1.5 \cdot 10^6$, $2.0 \cdot 10^6$ and $2.4 \cdot 10^6$ cells. We have measured the runtimes for coarse partitions that correspond to, approximately, 260, 500 and 900 cells per block. The results, using `mldivide` as the main solver in the Schur Complement Reduction for solving the local flow problems, can be seen in Figure 20.

We have also run the exact same tests using the AGMG solver [4] in the Schur Complement Reduction. The AGMG solver was called with a tolerance of 10^{11} and a maximum of 1000 iterations. The results are given in Figure 21.

All in all the results in Figures 20 and 21 are quite similar to each other. We also recognize the same decrease in speedup for increasing block sizes as we saw

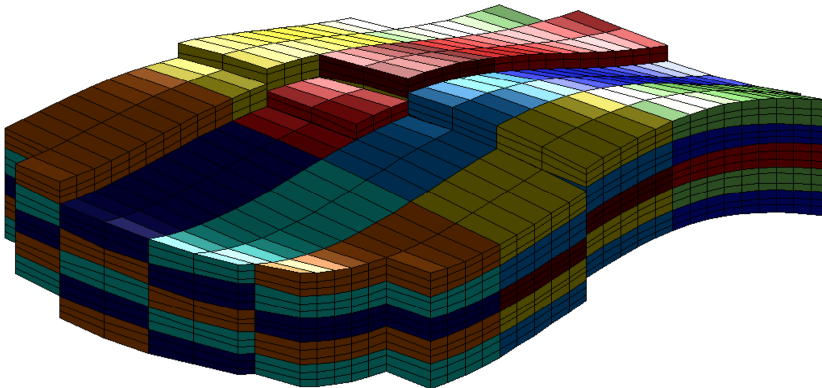


Figure 19: Geometry generated using `makeModel3` as demonstrated in Appendix A. The structure is partitioned into $30 \times 10 \times 15$ cells, grouped into $3 \times 4 \times 5$ blocks.

in Section 7.2. We consider these results to be quite good; in particular, the two smaller block configurations.

7.4 Testing the Inner Product Structure

In Section 6.1.2 we described a new structure for storing and distributing the inner products; the pseudocode of which can be seen in Listing 13. Now, with the geometry described in Section 7.3 in mind, we compare the execution time of running `xmsmfem` using one worker, for the largest and the smallest block sizes considered in Section 7.3, with the corresponding runtime of the serial MsMFE method implemented in MRST. We compare the runtime of computing inner products, and of constructing basis functions separately. Additionally, we highlight that there is little communication overhead associated with running `xmsmfem` on one worker. The results of this comparison can be seen in Figure 22.

We see that, for this case, the new structure facilitates considerably faster construction of basis functions for larger number of cells. In particular for smaller blocks. For the smallest block configuration, `xmsmfem` is three times as fast, and for the largest block configuration, `xmsmfem` is twice as fast. We only display the run times of computing the inner products for the smaller block configuration. Because there is no distribution of inner products taking place, the execution times are mostly independent of block size. We also note that the serial computation of inner products in `xmsmfem` is nearly as fast as the regular function in MRST. The

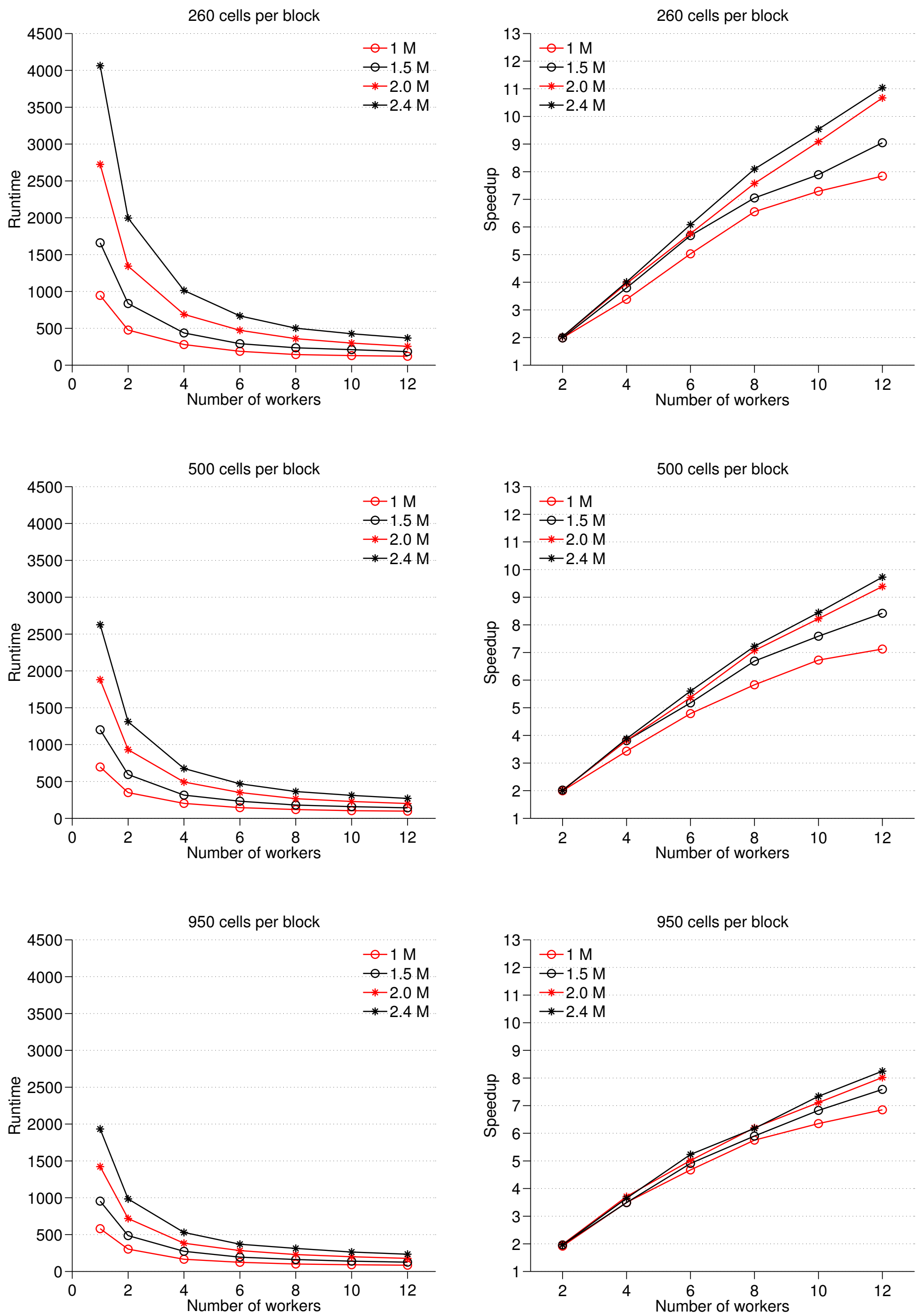


Figure 20: Runtimes and speedups using `mldivide`.

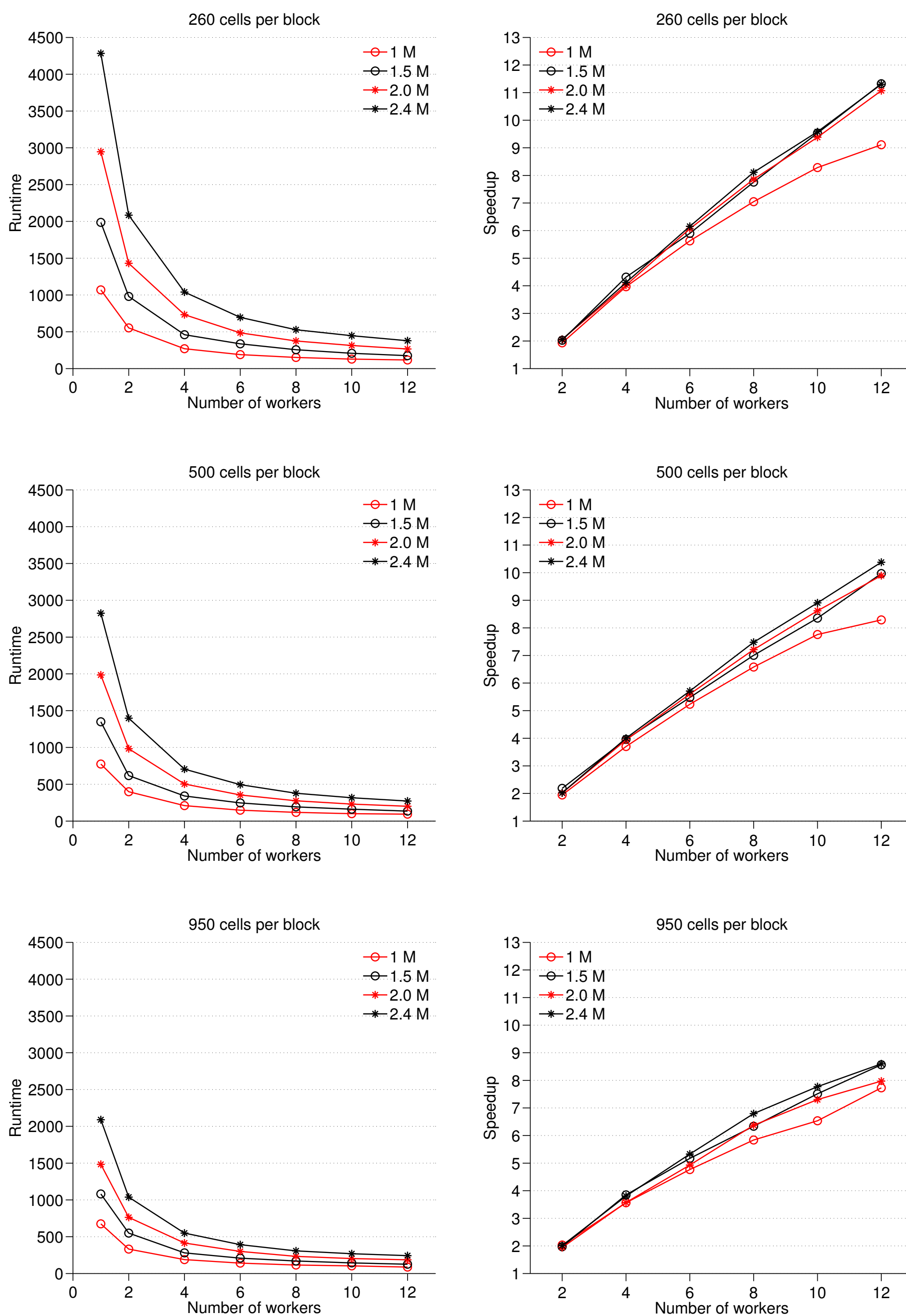


Figure 21: Runtimes and speedups using the AGMG solver [4].

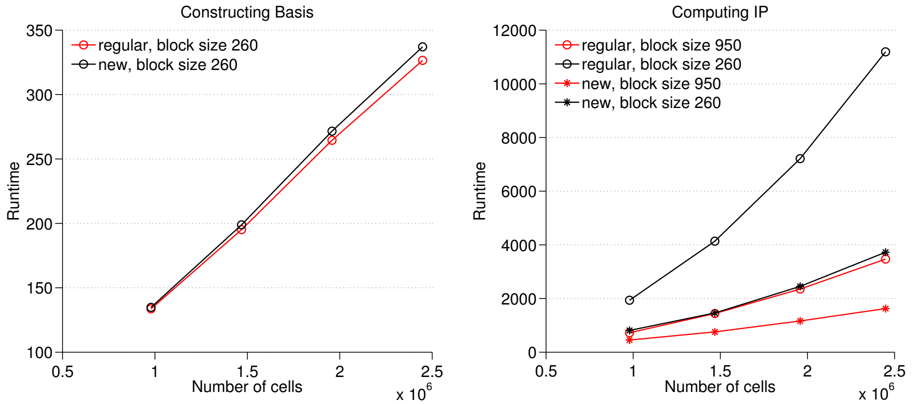


Figure 22: Comparison of execution times in regular MRST, and when using the new cell structure from Section 6.1.2.

main reason for the little difference that we observe is probably that calling the functions of `xmsmfem` on one worker will still result some function calls that will only yield an improved runtime when we have more workers available. I.e. parallel overhead.

As for the reason for the considerable speedup when using the new structure, we believe this is due to the fact that we now avoid getting elements from a large sparse matrix by using (sparse) indexing in MATLAB. As a test we have timed the process of extracting and concatenating $n = \{10, 20, \ldots, 100\}$ block matrices of size 6×6 from a block diagonal matrix of 1000×1000 blocks. For comparison we repeated the same process with a 1×1000 cell array that contains matrices of size 6×6. The resulting 'speedup' is given in Figure 23. We see that there is a considerable difference in the performance of these two operations, which probably accounts for most of the difference when comparing `xmsmfem` with MRST.

The final issue related to the cell structure that warrants an examination is redundancy in storage. As explained in Section 6.1.2 there will be instances of the same inner products stored on several of the workers. We have counted the total number of stored inner products for the example in Section 7.2 in Figure 24. Similarly, the total number of stored inner products for the example in Section 7.3 is given in Figure 25.

For both cases there is a considerable amount of redundancy. However, we still believe that the overhead of transmitting inner products back and forth would slow everything down way too much for this to be an option. An alternative is to

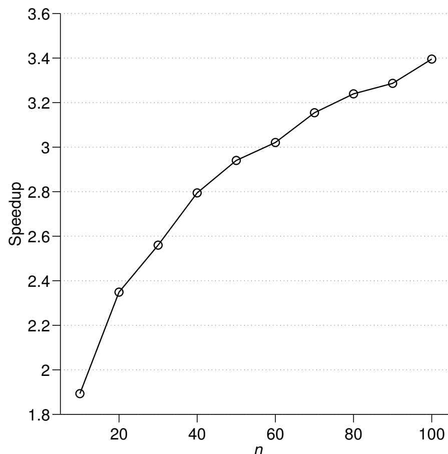


Figure 23: Plot of $S_c = T_c/T_s$. Here, T_c, refers to the time it took to extract n block matrices of size 6 × 6 from a sparse, block diagonal, matrix, **A**, of size 6000 × 6000. Similarly, T_c refers to the time it took to extract n matrices of size 6 × 6 from a cell array with 1000 × 1 matrices. This is nearly analogous to the new inner product structure.

examine the possibility of distributing faces among workers in a way that minimizes the amount of overlap in the necessary inner product storage.

7.5 Future Work

Since the results of this prototype must be considered quite promising the perhaps most obvious 'next' step is an implementation of a similar algorithm in a more low-level programming language. For instance using `MPI` and `C++`. `MPI` has the necessary constructs for this job. Most notably: it is based on the 'single program multiple data' philosophy, and it supports transmitting data between the labs. It would be convenient to make use of the `C++` version of MRST, which is currently under development. This way one could make use of the geometry, fluid, and rock structures, as well as the other useful parts of MRST that we have utilized in `xmsmfem`. It is at least not unreasonable to hope that a `C++` implementation—making use of linear algebra libraries—could perform better than `xmsmfem`, whilst at the same time require less memory. In particular, because one could avoid the fairly large memory footprint that MATLAB has for each worker.

Another alternative worth considering, in light of the recent advances in Graphics Processing Unit (GPU) computing, is adapting the code to make use of OpenCL[15]

[15] `http://www.khronos.org/opencl/`

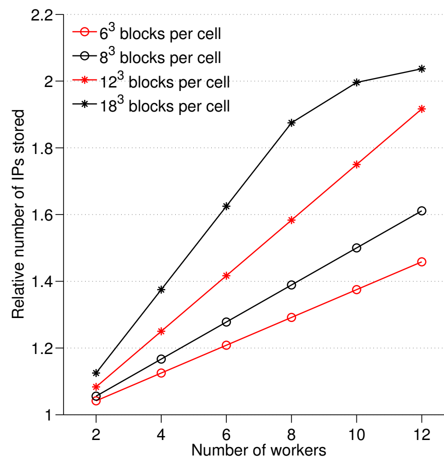


Figure 24: Relative memory usage for a cubic Cartesian grid with 3 million cells. This corresponds to the example given in Section 7.2.

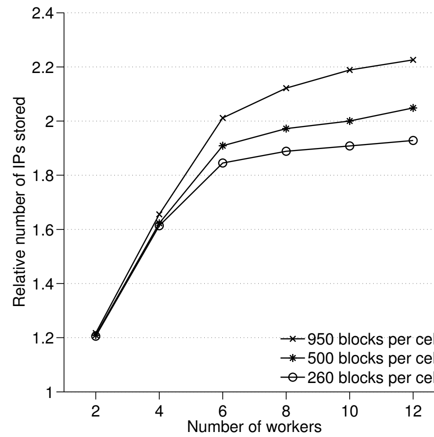


Figure 25: Relative memory usage of the example given in Section 7.3.

or CUDA[16]. CUDA is a parallel computing platform developed by Nvidia. It enables you to write code that executes on CUDA enabled graphics cards, letting you take advantage of the vast potential computing power available on todays GPUs. OpenCL is an open source alternative to CUDA that not only lets you write code for the GPU, but aims to let you write code for heterogeneous computing clusters. Even todays notebooks are available with hundreds of CUDA computing cores, hence there is considerable potential for speeding up code.

One possible model is to construct one massive system of Schur complement reduced systems for each basis function. That is, remembering Equation (15), one could build a system on the following form; assuming we have n basis functions:

$$\overline{\mathbf{S}}\overline{\boldsymbol{\pi}} = \bar{\mathbf{f}} = \begin{bmatrix} \mathbf{S}_1 & & \\ & \ddots & \\ & & \mathbf{S}_n \end{bmatrix} \begin{bmatrix} \boldsymbol{\pi}_1 \\ \vdots \\ \boldsymbol{\pi}_n \end{bmatrix} = \begin{bmatrix} \mathbf{F}_1\mathbf{E}_1^{-1}\mathbf{f}_1 \\ \vdots \\ \mathbf{F}_n\mathbf{E}_n^{-1}\mathbf{f}_n \end{bmatrix}.$$

Here $\mathbf{S}_i$, $\boldsymbol{\pi}_i$, $\mathbf{E}_i$, $\mathbf{F}_i$ and $\mathbf{f}_i$ corresponds to $\mathbf{S}$, $\boldsymbol{\pi}$, $\mathbf{E}$, $\mathbf{F}$ and $\mathbf{f}$ from Equation (15) for a $i = 1, \ldots, n$ local systems. The local parts of this system can be constructed in parallel, the complete system, $\overline{\mathbf{S}}\overline{\boldsymbol{\pi}} = \bar{\mathbf{f}}$, can then be constructed and then solved on a GPU. This system would both be sparse and SPD, and should therefore be easily solvable. Furthermore, we see that the back substitutions that need to be done in order to get the pressures, $\overline{\mathbf{p}}$, and velocities, $\overline{\mathbf{v}}$, could be calculated as a total systems in much the same way. That is

$$\overline{\mathbf{p}} = \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_n \end{bmatrix} = \begin{bmatrix} \mathbf{E}_1^{-1}(\mathbf{f}_1 + \mathbf{F}_1^T\boldsymbol{\pi}_1) \\ \vdots \\ \mathbf{E}_n^{-1}(\mathbf{f}_n + \mathbf{F}_n^T\boldsymbol{\pi}_n) \end{bmatrix},$$

and

$$\overline{\mathbf{v}} = \begin{bmatrix} \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_n \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1^{-1}(\mathbf{C}_1^T\mathbf{p}_1 - \mathbf{D}_1^T\boldsymbol{\pi}_1) \\ \vdots \\ \mathbf{B}_n^{-1}(\mathbf{C}_n^T\mathbf{p}_n - \mathbf{D}_n^T\boldsymbol{\pi}_n) \end{bmatrix},$$

where $\mathbf{B}_i$, $\mathbf{C}_i$ and $\mathbf{D}_i$ corresponds to $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ from Equation (15).

A compromise of the method outlined here is to use the GPU support of PCT to solve $\overline{\mathbf{S}}\overline{\boldsymbol{\pi}} = \bar{\mathbf{f}}$. One could then modify the implemented basis construction of

[16]http://www.nvidia.com/object/cuda_home_new.html

`xmsmfem` to use this method before building the basis functions as before.

8 Concluding Remarks

We have seen that the field of reservoir simulations is a complicated and extensive area of research. With challenges ranging from surveying and gathering information about a reservoir, through to representing and modelling the reservoir using as much as possible of the available data. One of the larger challenges, perhaps surprisingly, is related to the required computational effort resulting from the inherent heterogeneity and size of most reservoirs. This heterogeneity, as we have seen, ranges from the micro scale, with interactions between pores and fluids, and through to the macro scale, with geometric composition such as layers, fractures and faults. Next we have described some aspects of the mathematical model with focus on single-phase flow, and the extension to a two-phase model.

The Multiscale Mixed Finite-Element Method has been introduced as a method to facilitate direct simulation of large and complex grid-models for these highly heterogeneous petroleum reservoirs. We have also seen that this method has an inherent parallelism that could speed up the computations considerably in a parallel computing environment.

the MATLAB Parallel Computing Toolbox has been introduced to remedy this. PCT proves to be a powerful tool for a vast range of parallel computations, and an excellent platform for developing prototypes such as the parallel Multiscale Mixed Finite-Element method (`xmsmfem`) that we introduced in Section 6.

We conclude that the results we have seen are very promising, that the construction of basis functions for the MsMFE method can parallelized readily, as expected, and that MRST and the MATLAB Parallel Computing Toolbox is a useful combination. We have been able to implement a parallel MsMFE method with good, to excellent, speedup for reasonably large systems. The highlight is that we get near linear speedup for larger geometries when using `xmsmfem` on a system with twelve cores. The speedup is highest for small to medium-sized block configurations, but it is considerable for all tested configurations.

9 References

[1] Sensistivity Analysis of the Impact of Geological Uncertainties on Production (SAIGUP) project. http://www.nr.no/pages/sand/area_res_char_saigup, 0ctober 2011.

[2] MATLAB Parallel Computing Toolbox - User's Guide R2011b. http://www.mathworks.com/help/pdf_doc/distcomp/distcomp.pdf, September 2011.

[3] MATLAB Reservoir Simulation Toolbox (MRST). http://www.sintef.no/MRST/, October 2011.

[4] AGgregation-based algebraic MultiGrid. http://homepages.ulb.ac.be/~ynotay/AGMG/, June 2012.

[5] J. Aarnes, S. Krogstad, and K.-A. Lie. Multiscale Mixed/Mimetic Methods on Corner-point Grids. *Computational Geosciences 12, 3, 297-315*, 2008.

[6] J. Aarnes, K.-A. Lie, V. Kippe, and S. Krogstad. Multiscale Methods for Subsurface Flow. *Multiscale Modeling and Simulation in Science. (s. 3-48). Berlin: Springer*, 2009.

[7] J. Aarnes, K.-A. Lie, V. Kippe, and A. B. Rustad. Modelling of Multiscale Structures in Flow Simulations for Petroleum Reservoirs. *Geometrical Modeling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF. (s. 303-356). Berlin: Springer*, 2007.

[8] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[9] T. Arbogast. Implementation of a Locally Conservative Numerical Subgrid Upscaling Scheme for Two-phase Darcy Flow. *Computational Geosciences 6*, pages 453 – 481, 2002.

[10] D. N. Arnold and F. Brezzi. Mixed and Nonconforming Finite Element Methods: Implementation, Postprocessing and Error Estimates. *RAIRO: Modélisation mathématique et analyse numérique, tome 19*, pages 7 – 32, 1985.

[11] S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*. Springer, Chicago, IL, USA, 2007.

[12] F. Brezzi, K. Lipnikov, and V. Simoncini. A Family of Mimetic Finite Difference Methods on Polygonal and Polyhedral Meshes. March 2005.

[13] L. Cosentino. *Integrated Reservoir Studies.* Institut Fracais du Pétrole Publications, Paris, 2001.

[14] Y. Efendiev and T. Y. Hou. *Multiscale Finite Element Methods: Theory and Applications; Electronic Version.* Surveys and Tutorials in the Applied Mathematical Sciences. Springer, Dordrecht, 2008.

[15] J. L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31:532–533, May 1988.

[16] T. Y. Hou and X.-H. Wu. A Multiscale Finite Element Method for Elliptic Problems in Composite Materials and Porous Media. *Journal of Computational Physics*, 134(1):169 – 189, 1997.

[17] V. Kippe, J. Aarnes, and K.-A. Lie. A Comparison of Multiscale Methods for Elliptic Problems in Porous Media Flow. *Computational Geosciences 12, 3, 377-398*, 2008.

[18] K.-A. Lie, S. Krogstad, I. Ligaarden, J. Natvig, H. Nilsen, and B. Skaflestad. Open-source matlab implementation of consistent discretisations on complex grids. *Computational Geosciences*, pages 1–26. 10.1007/s10596-011-9244-4.

[19] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers, Massachusetts, USA, 2008.

[20] P. S. Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann Publishers, Burlington, MA, USA, 2011.

[21] Y. Saad. *Iterative Methods for Sparse Linear Systems; 2nd Ed.* SIAM, Philadelphia, PA, 2003.

Appendices

A A Working Example

We present a working example of `xmsmfem`, as introduced in Section 6, after we look at some of the naming conventions. We use most of the naming conventions, and structures, from MRST. These are more thoroughly described in [18]. In short this means that we use the following variables:

- `G`, structure that contains the geometry;
- `rock`, structure that contains the cell-wise properties of the medium;
- `S`, structure that contains the inner products associated with the geometry;
- `CG`, structure that contains the geometry if the coarse structure, as well as the partitioning;
- `mob`, array that contains the mobility associated with each cell;
- `bc`, structure that contains boundary conditions;
- `src`, structure that contains sources;
- `fluid`, structure that contains the fluid properties of each cell;
- `CS`, structure that contains basis functions and topology matrices of the coarse system;
- `xMs`, structure that contains the solution structure.

As well as a few others.

Additionally, MRST stores the ip_simple inner product in the struct `S`, as the sparse matrix `S.BI`. This corresponds to a block diagonal matrix of $\mathbf{T}$ from Equation (28), for each cell in the structure.[17]. In `xmsmfem` we use a slightly different structure, as described in Section 6.1.2, but essentially the inner products are stored in the variable named `XBI`.

Furthermore, we make quite extensive use of both composite and distributed variables, as described in Section 5. Composite variables are named as they would be in MRST, provided there is a corresponding variable. However, distributed

[17] If the inner products are not inverted the sparse matrix is stored in `S.B`.

variables are prepended with ‘co-’. Local parts of the same distributed array are prepended with ‘lp-’.

A working example can be seen in Listing 16, and a flowchart that describes the example can be seen in Figure 26.

Listing 16: xMain.m

```
% code assumes MRST and xmsmfem is already initiated

% helper function to initiate workers
%   and construct composite variables
% all returned variables are Composite
% the following variables are assigned default values:
%   overlap   = 0;
%   bc        = [];
%   src       = [];
%   facetrans = zeros(0,2);
%   activeBnd = [];
%   weighting = 'perm';
% the remaining variables are empty.
% they need to be initialized as in the example below.
% afterwards xBroadcast needs to be called.
procs = 4 ; % 4 workers
[g,rock,cg,mob,bc,src,overlap,facetrans,weighting,...
 activeBnd] = xInitWorkers(procs);

% disable gravity on all workers
pctRunOnAll gravity off;

% parallel block.
% variables that define the problem are initialized on
% worker 1
% they are then broadcasted using xBroadcast.
spmd
  % initialization of the problem only on worker 1
  if labindex == 1,

    fprintf('initializing on worker 1 ... ');

    % size of fine and coarse geometry
    nx = 100; ny = 50; nz = 15;
```

```
Nx = 10;  Ny = 5;  Nz = 5;

% geometry and rock properties
g = computeGeometry(processGRDECL(makeModel3(...
  [nx, ny, nz])));
K = logNormLayers(g.cartDims, [10, 300, 40, 0.1, 100]);
rock.perm = bsxfun(@times, [1, 100, 0.1], K(:));
rock.perm = convertFrom(rock.perm(g.cells.indexMap, :),...
  milli*darcy);

% coarse partition
p  = partitionUI(g, [Nx, Ny, Nz]);
p  = processPartition(g,p);
cg = generateCoarseGrid(g,p);

% fluid properties
fluid = initSingleFluid('mu' ,1*centi*poise, ...
                        'rho',1000*kilogram/meter^3);

% solution structure
xMs = initState(g, [], 0, [0, 1]);

% get mobility
mu  = fluid.properties(xMs);
kr  = fluid.relperm(ones([g.cells.num,1]),xMs);
mob = kr ./ mu;

% some driving forces.
% sources can be added in a similar fashion here
bc = pside(bc,g,'East',1);
bc = pside(bc,g,'West',0);

fprintf('done\n');

% should you wish to call functions
%   from regular MRST for comparison
%   they can be called on one worker
%   inside spmd blocks such as this,
%   using the same composite variables;
%   eg:
```

```
    %
    %    ss = computeMimeticIP(g,rock);
    %    cs = generateCoarseSystem(g,rock,ss,cg,mob,'bc',bc);
  end

  % distribute Composite variables to *all* workers
  if labindex == 1, tic; end % time on worker 1.
  [g,rock,cg,mob,bc,src,overlap,facetrans,...
   weighting,activeBnd] = xBroadcast(...
   g,rock,cg,mob,bc,src,overlap,facetrans,...
   weighting,activeBnd);
  if labindex == 1,
    fprintf('time xBroadcast:\t%.5f\n',toc);
  end
end

% calculate inner products.
% - XBI -- distributed cell array.
%          XBI{i} contains inner product of cell i.
tic;
XBI = xComputeMimeticIP(g,rock,facetrans);
fprintf('time xComputeMimeticIP:\t%.5f\n',toc);

% distribution of inner products
% - coX     -- distributed cell array.
%                contains inner products
% - coiG    -- distributed cell array.
%                contains global numbering of cells
% - coFaces -- distributed array.
%                contains global numbering of faces
tic;
[coXBI,coiG,coFaces] = xDistributeIP(XBI,g,cg,overlap,...
                                     bc,activeBnd);
fprintf('time xDistributeIP:\t%.5f\n',toc);

% construct basis functions
% - xCS -- composite array
%             non-empty only on worker 1.
%             contains coarse basis functions.
tic;
```

```
XCS = xGenerateCoarseSystem(g,rock,cg,overlap,...
                            bc,src,weighting,mob,...
                            coXBI,coiG,coFaces);
fprintf('time xGenerateCoarseSystem:\t%.5f\n',toc);

% build BI and S from X, and solve coarse system:
spmd
  BI = gather(XBI,1);
  if labindex == 1
    dimProd = double(diff(g.cells.facePos));
    [ind1, ind2] = blockDiagIndex(dimProd, dimProd);
    n = size(g.cells.faces, 1);
    S.BI = sparse(ind1,ind2,vertcat(BI{:}),n,n);
    S.type = 'hybrid'; S.ip = 'ip_simple';

    xMs = solveIncompFlowMS(xMs,g,cg,p,S,XCS,fluid,...
      'bc',bc,'Solver',S.type);
  end
end

% can not plot from workers
% transport xMs and g to host
xxMs = xMs{1}; gg = g{1};
% note that composite variables do not
% support accessing structure fields
% using '.', such as myStruct.field
clf,
  plotCellData(gg, convertTo(xxMs.pressure, barsa()));
  view(3), camproj perspective, axis equal tight off,
  camlight headlight
  cax = caxis; cobar =  colorbar;
```

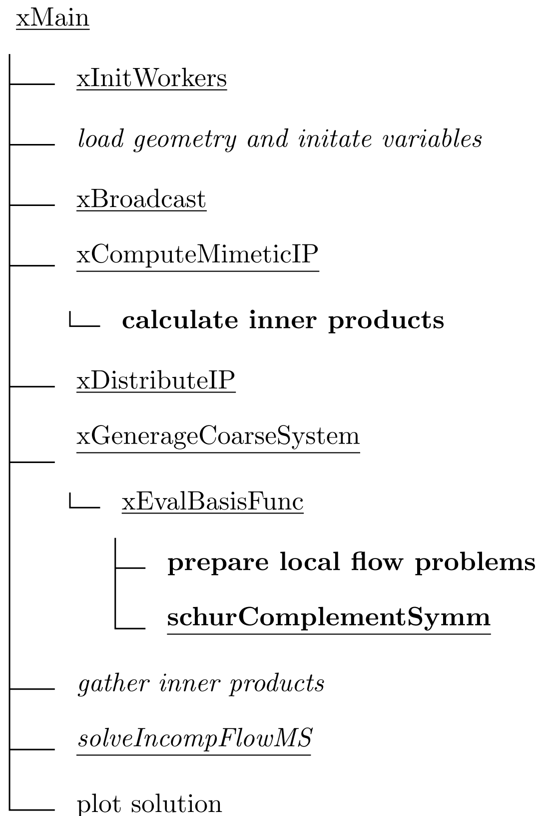


Figure 26: Simplified flowchart that describes the structure of a main script running `xmsmfem`. A full working example can be seen in Appendix A. Execution order is top to bottom. MATLAB functions are underlined. Entries in *italics* are executed on one worker only, whereas functions in **bold** are run (in parallel) inside `spmd` blocks. All other entries are executed from the host.