NTNU – Trondheim
Norwegian University of
Science and Technology

FieldOpt: Enhanced Software Framework for Petroleum Field Optimization

Development of Software Support System for
the Integration of Oil Production Problems
with Optimization Methodology

Einar Johan Moen Baumann

Master of Science in Engineering and ICT
Submission date: June 2015
Supervisor: Jon Kleppe, IPT
Co-supervisor: Mathias Bellout, IPT

Norwegian University of Science and Technology
Department of Petroleum Engineering and Applied Geophysics

Abstract

This thesis concerns the development of FieldOpt, a software framework that aims at being a common platform for MSc. and Ph.D. students of optimization theory and petroleum engineering to conduct research. With FieldOpt, we seek to simplify and speed up the process of developing and applying new optimization methodologies to interesting petroleum cases. We achieve this by structuring the source code in a way that facilitates modifications and additions through the use of abstract classes and clearly defined interfaces, and by implementing process-level parallelization.

We present some background on topics relevant to our work. We start with a general description of optimization algorithms, in particular pattern search algorithms, focusing on the compass search algorithm that we implemented in FieldOpt. Other key topics are reservoir simulation; parallel computing, focusing on process-level parallelization in distributed systems; and software architecture, which is crucial to the implementation of any software framework. We then go on to describe *how* FieldOpt was implemented, using class- and sequence diagrams to describe the structure of the code and the sequence of instructions during execution. We also present a case study examining the correctness and performance of our implementation under various conditions.

The optimization interface we created is simple and well defined. It should be flexible enough to facilitate implementation of most pattern search algorithms, but it will likely need some modifications in the future. We find that our implementation exhibits exceptional performance for large problems and that the performance scales very well when a large number of processors are available. When run in parallel, our implementation reaches a solution more than 90% faster than when run in serial for large problems. However, we also identify some issues linked to our implementation of workload scheduling, caused by irregular behavior in the reservoir simulator. These irregularities may cause the program to have a very long execution time for some problems, and a systematic handling of them should be implemented.

Sammendrag

Denne oppgaven tar for seg utviklingen av FieldOpt, et programvarerammeverk som tar sikte på å være en felles plattform for MSc. og Ph.D. studenter innen optimaliseringsteori og petroleumsteknologi til å drive forskning. Med FieldOpt forsøker vi å forenkle og effektivisere prosessen med å utvikle og anvende nye optimaliseringsmetoder på interessante petroleumsproblemer. Vi oppnår dette ved å strukturere kildekoden på en måte som muliggjør modifikasjoner og utvidelser ved bruk av abstrakte klasser og klart definerte grensesnitt, samt ved å implementere parallellisering på prosessnivå.

Vi presenterer litt bakgrunnsinformasjon om emner som er relevante for arbeidet. Vi starter med en generell beskrivelse av optimaliseringsalgoritmer, spesielt *pattern search* algoritmer, med fokus på *compass search*-algoritmen vi implementerte i FieldOpt. Andre sentrale temaer er reservoarsimulering; parallelle beregninger, med fokus på parallellisering på prosessnivå i distribuerte systemer; og programvarearkitektur, som er kritisk for ethvert programvarerammeverk. Videre beskriver vi hvordan FieldOpt ble implementert ved hjelp av klasse- og sekvensdiagrammer som beskriver strukturen i koden og instruksjonssekvenser under kjøring. Vi presenterer også en casestudie der vi undersøker korrektheten og ytelsen til implementeringen vår.

Optimaliseringsgrensesnittet vi laget er enkelt og tydelig definert. Det bør være fleksibel nok til å tilrettelegge for implementering av de fleste *pattern search*-algoritmer, men det vil trolig kreve endringer i fremtiden. Vi ser at implementeringen vår har svært god ytelse for store problemer, og at ytelsen skalerer godt når et stort antall prosessorer er tilgjengelig. Når programmet kjøres i parallell, finner implementeringen en løsning mer enn 90% raskere enn når det kjøres serielt for store problemer. Men vi ser også at vår implementering av “arbeidsfordeling” har noen problemer, forårsaket av uregelmessig oppførsel i reservoarsimulatoren. Disse uregelmessighetene kan føre til at programmet får veldig lang kjøretid for enkelte problemer, og en systematisk håndtering av dette bør implementeres.

Preface

This thesis is written as part of the Master's degree in Engineering and ICT with specialization in Integrated Operations in the Petroleum Industry, at the Department of Petroleum Engineering & Applied Geophysics at the Norwegian University of Science and Technology, NTNU. It was written during the spring semester of 2015 under the supervision of Prof. Jon Kleppe and Postdoc Mathias Bellout.

We assume the reader of this report is familiar with reservoir simulation and object-oriented programming and related terminology.

The software developed during this project builds upon and significantly enhances ResOpt, an optimization software package created by Alexander Juell during his Postdoc at the IO Center at NTNU. The work in this project has both enhanced the functionality of and added new features to ResOpt. The code has been restructured to increase efficiency and allow for extensive parallelization; the optimizer interface was completely rewritten, and a pattern search algorithm was added. The main result of this work is that the code is now better organized, more robust and has greater extension and scaling capabilities. Besides an expanded documentation, these changes and additions are the key features we rely on to make the further planned development of FieldOpt more efficient.

We call the new version of the software package FieldOpt. FieldOpt currently comprises more than sixteen thousand lines of C++ source code, of which around four thousand are written from the ground up; the rest originate from ResOpt's source code. The source code for FieldOpt is available in its entirety on GitHub (`https://github.com/iocenter/FieldOpt`).

Acknowledgments

I would like to thank my supervisor professor Jon Kleppe for valuable conversations and advice, and my co-supervisor Mathias Bellout for his help and advice on almost every aspect of this thesis – without his expertise in integrating optimization theory with petroleum field development problems I would not have been able to complete this project.

I would also like to thank Alexander Juell: without his original ResOpt code FieldOpt would probably not have been created at all.

Lastly, thanks to SINTEF Applied Mathematics for making their MRST reservoir simulator available for free and open-source. It made the development of FieldOpt easier and its open-source nature is likely to prove very useful in the future development.

Contents

List of Figures

Chapter 1

Introduction

This thesis concerns the development of FieldOpt, a software framework that aims at being a common platform for MSc. and Ph.D. students to conduct research. FieldOpt seeks to promote research by efficiently integrating methodology from two important academic fields: optimization theory and petroleum engineering. The main idea behind FieldOpt is to consolidate and make user-friendly much of the groundwork necessary to conduct optimization on a variety of petroleum problems. By laying this foundation, FieldOpt seeks to simplify and speed up the process of developing and applying new optimization methodologies, as well as making it easier to test new techniques on interesting petroleum cases. In particular, FieldOpt will serve as an integration framework that will both include and facilitate the application of various optimization methods to field development problems, e.g., well placement or production strategy.

FieldOpt as a Software Framework When we refer to FieldOpt as a software framework, we mean that it is not only intended to be modified and extended by its users, but also that it should be able to solve a set of problems "out-of-the-box". One of the main concerns when developing FieldOpt has been to lower the threshold for expanding its functionality. E.g., users wishing to add a new optimizer can do so without having to understand the reservoir model or the simulator interface. Currently, FieldOpt can apply one optimization to one reservoir model that can be evaluated using one reservoir simulator.

ResOpt FieldOpt is based on ResOpt, developed by Alexander Juell. ResOpt is a tool for field development optimization, providing a set of optimizers and simulator interfaces to solve a range of problems. Our main issue with ResOpt besides its lack of extensive documentation is the very tight coupling between many parts of the code, which makes the code difficult to modify. The tight coupling is a significant issue when our goal is to develop a framework for customized research. We, therefore, decided only to keep the model (the code describing the "problem") and simulator interface from ResOpt, clean up the code and improve the documentation.

Scope of Work ResOpt is designed to solve a whole range of problems that include pipes, compressors, and pressure boosters coupled to the reservoir model in a tightly connected system. From this starting point, for our first steps in the development of FieldOpt, we have chosen a fundamental scope of work where we focus primarily on improving the core functionality of the software. Roughly, this core functionality consists of the main coupling between optimization algorithm and production case, and the efficient operation of this joint system. We believe that focusing on consolidating and further improving this core functionality, e.g., through greater modularity, is the best way to facilitate future development of the software package. Once properly deployed, the strong core will also make FieldOpt more robust to a broad range of users. Within the scope of this work, the type of problems we chiefly deal with involve procedures to find improved well locations and production strategy while making extensive use of reservoir simulation models for cost function evaluations. However, as FieldOpt is made to be easily extensible, we plan for a wider range of problems, including pipe network and facilities, to be implemented in the future.

Parallelization Solving optimization problems that deal with petroleum cases typically involves computationally demanding reservoir simulations. Moreover, since these type of problems often include a relatively large number of variables (in the hundreds), we have in our development of FieldOpt put significant effort on achieving good performance through parallelization. ResOpt utilizes thread-level parallelization to exploit multiple local processing cores, which is sufficient if the best available hardware is a workstation. However, this setup cannot exploit the much larger computational power available in distributed hardware architectures, such as computer clusters and supercomputers. For this reason, in FieldOpt we use process level parallelization with cross-process communication based on message passing in FieldOpt's parallel "runner", which lets us exploit both distributed and localized hardware architectures.

Software Architecture If a software framework is to be easily modifiable, it needs to have a good software architecture. Moreover, a good architecture makes implementing parallelism, especially when utilizing message passing for communication, a much more straightforward task. Because of its importance, software architecture is discussed quite extensively in this report.

Structure of this Document The next chapter presents background topics such as optimization algorithms, reservoir simulation, parallel computing and software architecture. Chapter 3 presents how we implemented the various features of FieldOpt, and we discuss which and how different software libraries were used. We also include diagrams that describe FieldOpt's architecture, interfaces and program flow. In Chapter 4 we introduce a case study examining FieldOpt's performance. We look at how well the program can utilize large amounts of available computational power, as well as how well the implemented optimization algorithm handles various problem sizes. Chapter 5 summarizes the work in this project, and the results we have obtained. In this chapter, we also discuss further work and highlight different facets of FieldOpt that need further improvement and extension.

Chapter 2

Background

The work done in this project combines aspects of optimization, reservoir simulation, software development and parallel computing. In this chapter, we provide brief introductions to those parts of these topics that are relevant to our work. The aim of this background chapter is to help clarify why we decided on implementing the software the way we did. We start with a description of optimization algorithms in general, and in particular pattern search methods. We then give a short description of reservoir simulators, which are, in our current setup, the main engine for evaluating cost function values. We then provide a thorough introduction to parallel computing, and finally, we discuss software architecture.

2.1 Terminology

To avoid misunderstandings, we here define some common terms used in this thesis:

CPU: Central Processing Unit, contains one or more processors.

Processor: A physical processing unit or *core*, part of the CPU.

Process: An instance of a program.

Driver File: Used to deliver a set of configurations to FieldOpt, ResOpt and various reservoir simulators.

Node: A hardware unit in a computer cluster or supercomputer, containing memory modules and one or more CPUs.

Parallel Computation: Computations running in parallel either by use of multiple processes or multiple threads.

Framework: Software which provides rudimentary functionality that is intended to be modified and expanded by its users.

Move: In pattern search optimization, a “move” refers to a coordinate generated by the algorithm that is some distance away from some origin point.

Model: A complete description of a problem, including the reservoir model, as well as the definition of possible variables and constraints associated with that problem.

Case: A variation of a model. It is used both in terms of a particular problem size (a model with a specific number of variables) and as a perturbation of a model (a specific set of variable values).

Batch: The set of coordinates generated by an optimizer in one iteration.

2.2 Problem Formulation and Optimization Algorithms

Given FieldOpt is intended as a framework for petroleum field optimization, with support for a wide range of algorithms, we start with a brief introduction of the general problem formulation used in this work. Mathematically, we define optimization as the minimization or maximization of a function subject to constraints on its variables [1]. We write this as

$$\min_{x\in\mathbb{R}^n} f(x) \quad \text{subject to} \quad \begin{cases} c_i(x) = 0, & i \in \mathcal{E} \\ c_i(x) \geq 0, & i \in \mathcal{I} \end{cases} \qquad (2.1)$$

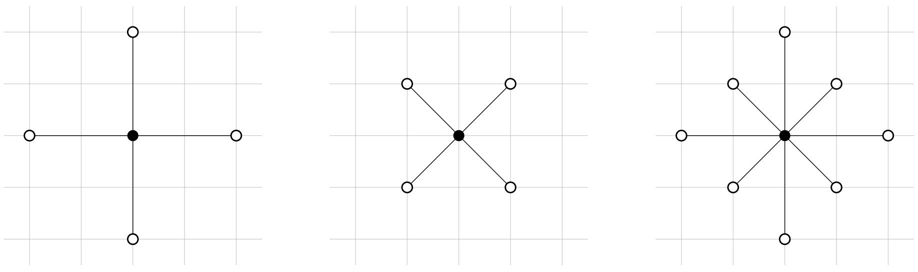
where $\mathcal{E}$ and $\mathcal{I}$ are sets of indexes for equality and and inequality constraints, respectively. Currently, however, our problems only involve field development topics such as well placement or production strategy. Therefore, we may state the problem in (2.1) in a more specific form [2]:

$$\min_{\mathbf{x}\in\mathbb{Z}^{n_1},\mathbf{u}\in\mathbb{R}^{n_2}} -f(\mathbf{x},\mathbf{u}) \quad \text{subject to} \quad \begin{cases} \mathbf{x}_d \leq \mathbf{x} \leq \mathbf{x}_u, \\ \mathbf{u}_d \leq \mathbf{u} \leq \mathbf{u}_u, \end{cases} \qquad (2.2)$$

Here $\mathbf{x}$ denotes discrete well placement variables while $\mathbf{u}$ refers to continuous well control variables. This is the standard formulation we use in the current implementation of FieldOpt. However, other types of formulations, e.g., using real variables to describe well locations, are entirely possible.

Furthermore, in this project, we limit our work to bound constraints only, i.e. sets of values describing a bound that any valid coordinate must be inside. We restrict ourselves to bound constraints because they have been sufficient for the problems we have considered. Moreover, for the simple test case we have implemented, we use only one type of variables, $\mathbf{x}$, since the algorithms we present handle integer and real variables in the same manner.

The remainder of this section is restricted to a particular class of derivative-free optimization algorithms, namely, pattern search methods, which are a subclass of direct search methods.



(a) Compass. **(b)** Factorial. **(c)** Composite.

Figure 2.1: Examples of patterns applied to two-dimensional objective functions. Adapted from [5].

2.2.1 Pattern Search Methods For Optimization

Pattern search methods are a subclass of direct search methods. Hooke and Jeeves [3] defined direct search algorithms as algorithms that perform a

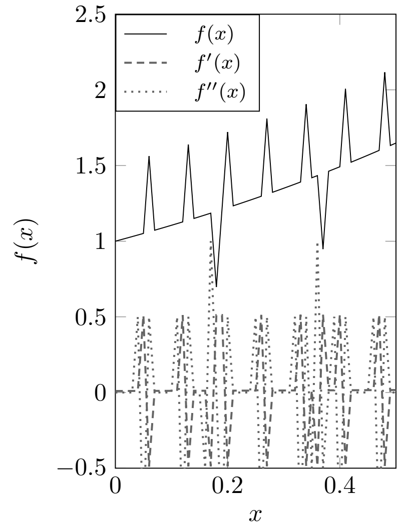
> *... sequential examination of trial solutions involving comparison of each trial solution with the "best" obtained up to that time together with a strategy for determining (as a function of earlier results) what the next trial solution will be.*

This explanation of direct search also applies in a straightforward manner to the case where we have sets of trial solutions, i.e., batches computed in parallel, where we compare the best trial solution from the batch to the best solution up to that time. Another property of direct search methods in general is that they do not compute or explicitly approximate derivatives of the objective function [3, 4].

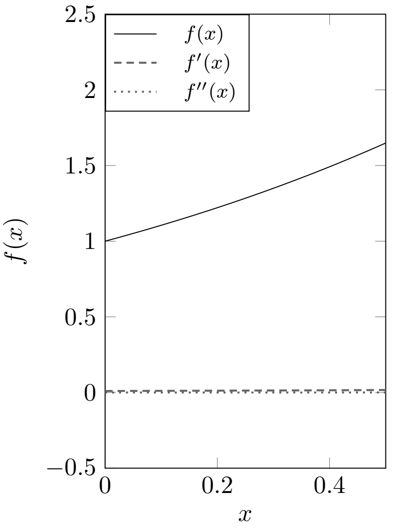
Torczon [4] defines pattern search methods as direct search methods that perform a search using a pattern of points which are independent of the objective function. Many different patterns may be used, and which is more successful depends on the objective function. Examples of three patterns applied to two-dimensional grids are shown in Figure 2.1.

One of the main reasons we chose to focus on pattern search methods is that they are easily parallelized. That an algorithm lends itself to parallelization is an important feature because it means that, even though algorithms requiring fewer function evaluations exist, a parallelized algorithm may reach a solution after a shorter time because we can evaluate several coordinates simultaneously.

Another argument for pattern search methods and derivative-free methods in general is that they are more easily applied to our problems than classical methods using gradients are. Gradients are often problematic when dealing with reservoir models as they often have non-smooth objective functions, usually a result of heterogeneities in the reservoir (e.g. varying permeability, fractures) [2]. This is a significant issue, as noise in the objective will severely affect the approximation of derivatives, as illustrated in Figure 2.2.



(a) Plot of a non-smooth function and its derivatives.



(b) Plot of a smooth function and its derivatives.

Figure 2.2: Plot of a non-smooth function and a smooth function and their corresponding derivatives. The graphs are meant to illustrate how severely numerically calculated derivatives are affected by non-smooth functions.

A final, less important point is that pattern search methods enable us to acquire tentative results before the optimization is completed. We can do this because we always keep the "best" result obtained up to the current time [3].

2.2.2 Compass Search

Compass search is one of the simplest and earliest types of pattern search algorithms. It is often slow to reach an accurate solution, but it converges reasonably fast in the beginning, and it is easy to implement in a computer program [5]. The compass search algorithm may be summarized as follows:

1. Try moves of equal length in each direction along each coordinate axis from the current position.
2. If any of the moves yields a reduction in the objective function value, move to that point. Return to step 1.
3. If none of the moves yield a reduction in the objective value, halve the step length and return to step 1.

More formally, the compass search algorithm starts at an initial position $\mathbf{x}_0$, and moves a distance Δ_0 along each direction $d \in \mathcal{D}$, where $\mathcal{D}$ is the set of positive and negative unit coordinate vectors along all the n dimensions in the objective function:

$$\mathcal{D} = \{e_1, e_2, \ldots, e_n, -e_1, -e_2, \ldots, -e_n\}.$$

This initial step is shown on a two-dimensional objective function in Figure 2.3a. If we get a reduction in the objective value from any of the moves, the move that yielded the lowest objective value is stored as the new "best" position, and further moves are made from this point, as in Figure 2.3b,2.3c. If none of the moves yield a lower objective value, as in Figure 2.3d, the step length Δ is halved, and new moves half the distance from the current best position are evaluated, as shown in Figure 2.3e,2.3f. This process repeats itself until the set tolerance Δ_{tol} is reached. Pseudo-code for this algorithm is shown in Algorithm 1.

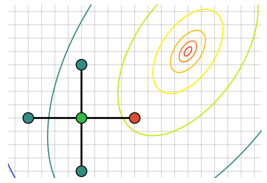
How quickly the algorithm converges, and to which stationary point it converges, depends on the values of x_0 and Δ_0. How exact our solution is depends on the value of Δ_{tol}.

Note that the "best" position may have been found long before the algorithm terminates, but the algorithm will always continue until $\Delta \leq \Delta_{tol}$.

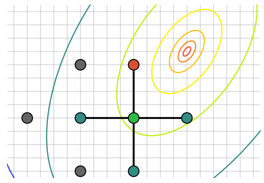
2.2.2.1 Bound Constraints

A variable often has a range that it has to be inside. Such ranges are called bound constraints. In our problems, dealing with reservoir simulation, these may for example state that the set of coordinates describing a well position is not allowed to be outside the reservoir; or that the bottom hole pressure in a well is not allowed to be negative. In equation (2.2) we wrote this as

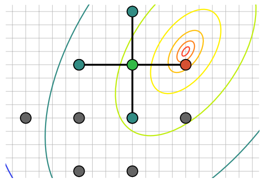
$$\mathbf{x}_d \leq \mathbf{x} \leq \mathbf{x}_u,$$



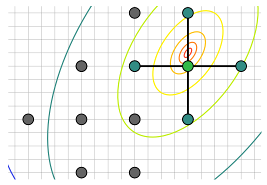
(a) Initial pattern. Found better value east.



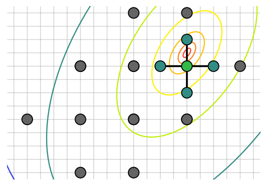
(b) Moved east. Found better value north.



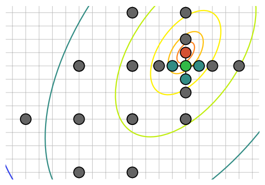
(c) Moved north. Found better value east.



(d) Moved east. Found no better value.



(e) Contracted. Found no better value.



(f) Contracted. Found better value north. Δ_{tol} reached.

Figure 2.3: Iterations using compass search. Green indicates current center; red indicates successful moves; blue indicates unsuccessful moves; gray indicates previously evaluated points. This figure is adapted from [5].

Algorithm 1 Compass Search Algorithm adapted from [5].

```
1:  procedure CompassSearch($f, \mathbf{x}_0, \Delta_{tol}, \Delta_0, \mathcal{D}$)
2:      $f_{best} \leftarrow f(\mathbf{x})$                          ▷ Set initial value for $f_{best}$.
3:      $\mathbf{x} \leftarrow \mathbf{x}_0$                                ▷ Set initial value for $\mathbf{x}$.
4:      $\Delta \leftarrow \Delta_0$                                ▷ Set initial step length.
5:      while $\Delta > \Delta_{tol}$ do  ▷ Iterate while step length is greater than tolerance.
6:          $\mathbf{M} \leftarrow (\mathbf{x} + \Delta \times d_k)$ for all $d_k \in \mathcal{D}$      ▷ Create list $\mathbf{M}$ of moves $\mathbf{x}_k$.
7:          $\mathbf{x}_{min} \leftarrow \text{argmin } f(\mathbf{x}_k)$              ▷ Find best move in iteration
8:          if $f(\mathbf{x}_{min}) < f_{best}$ then                ▷ Found a better position.
9:              $\mathbf{x} \leftarrow \mathbf{x}_{min}$                     ▷ Change the "best" position.
10:             $f_{best} = f(\mathbf{x}_{min})$           ▷ Change the "best" objective value
11:         else                                ▷ Did not find a better position.
12:             $\Delta \leftarrow \frac{1}{2}\Delta$                             ▷ Reduce the step-length.
13:         end if
14:     end while
15: end procedure
```

Algorithm 2 Compass Search Algorithm adapted from [5], with added bound constraint checking. The added part is on lines 7-11.

```
1: procedure CompassSearch($f, \mathbf{x}_0, \Delta_{tol}, \Delta_0, \mathcal{D}, \mathbf{x}_d, \mathbf{x}_u$)
2:     $f_{best} \leftarrow f(\mathbf{x})$
3:     $\mathbf{x} \leftarrow \mathbf{x}_0$
4:     $\Delta \leftarrow \Delta_0$
5:     while $\Delta > \Delta_{tol}$ do
6:         $\mathbf{M} \leftarrow (\mathbf{x} + \Delta \times d_k)$ for all $d_k \in \mathcal{D}$
7:         for all $\mathbf{x}_k \in \mathbf{M}$ do
8:             if $\mathbf{x}_k < \mathbf{x}_d$ or $\mathbf{x}_k > \mathbf{x}_u$ then    ▷ Check if the move is inside bounds.
9:                 Discard $\mathbf{x}_k$                       ▷ Discard moves outside the bounds
10:            end if
11:        end for
12:        $\mathbf{x}_{min} \leftarrow \text{argmin } f(\mathbf{x}_k)$
13:        if $f(\mathbf{x}_{min}) < f_{best}$ then
14:            $\mathbf{x} \leftarrow \mathbf{x}_{min}$
15:            $f_{best} = f(\mathbf{x}_{min})$
16:        else
17:            $\Delta \leftarrow \frac{1}{2}\Delta$
18:        end if
19:    end while
20: end procedure
```

Where $\mathbf{x}_d$ and $\mathbf{x}_u$ are vectors containing values that the corresponding values in $\mathbf{x}$ must be greater than or equal to, or less than or equal to, respectively.

The simplest way to integrate bound constraints on this form into Algorithm 1 is to check whether or not the generated set of moves violates the bound conditions. If a move violates a bound condition, it is discarded before evaluating the objective function. This is shown in Algorithm 2.

2.2.2.2 Bookkeeping

As seen in Figure 2.3, compass search often evaluates the same position more than once. When the evaluation entails performing a costly simulation, this may lead to a lot of wasted time. This issue may be alleviated by using *bookkeeping*, i.e. keeping track of all previously evaluated positions. By doing this, we can check whether we have already evaluated a position or a position very near it, and discard it if it has before we spend time evaluating it again. Algorithm 3 shows how we added bookkeeping to the compass search algorithm shown in Algorithm 2.

When implemented, a bookkeeper will naturally lead to some overhead, especially in terms of memory consumed by the optimizer. However, this is negligible when compared to the cost of evaluating the objective function in our application.

Algorithm 3 Compass Search Algorithm adapted from [5], with added bound constraint checking and bookkeeping. The added parts are on lines 5 and 11-15.

```
 1: procedure CompassSearch($f, \mathbf{x}_0, \Delta_{tol}, \Delta_0, \mathcal{D}, \mathbf{x}_d, \mathbf{x}_u$)
 2:     $f_{best} \leftarrow f(\mathbf{x})$
 3:     $\mathbf{x} \leftarrow \mathbf{x}_0$
 4:     $\Delta \leftarrow \Delta_0$
 5:     $\mathbf{B} \leftarrow \mathbf{x}$                         ▷ Add the initial position to the bookkeeper.
 6:     while $\Delta > \Delta_{tol}$ do
 7:         $\mathbf{M} \leftarrow (\mathbf{x} + \Delta \times d_k)$ for all $d_k \in \mathcal{D}$
 8:         for all $\mathbf{x}_k \in \mathbf{M}$ do
 9:             if $\mathbf{x}_k < \mathbf{x}_d$ or $\mathbf{x}_k > \mathbf{x}_u$ then
10:                 Discard $\mathbf{x}_k$
11:             else if $\mathbf{x}_k \approx \mathbf{x}_b$ for any $\mathbf{x}_b \in \mathbf{B}$ then     ▷ Check Bookkeeper.
12:                 Discard $\mathbf{x}_k$
13:             else
14:                 $\mathbf{B} \leftarrow \mathbf{x}_k$                    ▷ Add move to bookkeeper.
15:             end if
16:         end for
17:         $\mathbf{x}_{min} \leftarrow \text{argmin } f(\mathbf{x}_k)$
18:         if $f(\mathbf{x}_{min}) < f_{best}$ then
19:             $\mathbf{x} \leftarrow \mathbf{x}_{min}$
20:             $f_{best} = f(\mathbf{x}_{min})$
21:         else
22:             $\Delta \leftarrow \frac{1}{2}\Delta$
23:         end if
24:     end while
25: end procedure
```

2.2.3 Parallelization

As previously stated, most pattern search algorithms lend themselves easily to parallelization. This is also true for compass search. One way of parallelizing the compass search algorithm is to evaluate the moves generated in an iteration simultaneously. Given an n-dimensional objective function, each iteration will produce two moves for each of the n dimensions, i.e. $2n$ moves[1]. This means that we can distribute the main workload in the optimization, i.e. the function evaluations, across up to $2n$ processors. An implication of this is that the time it takes to evaluate all the moves in an iteration is independent of the dimensionality of the problem if we have access to a sufficient number of processors.

2.2.4 Other Pattern Search Algorithms

Naturally, compass search is not the only algorithm suitable for our type of problem. Some algorithms are more flexible than compass search, others have a better convergence rate, and others still are more efficiently parallelized. In this subsection, we will discuss two other algorithms: one more flexible than compass search, and one designed specifically to run in parallel.

2.2.4.1 Generating Set Search

Generating Set Search (GSS), as described by Kolda, Lewis and Torczon [5], is really a subclass of pattern search algorithms. It is, however, possible to implement GSS as a single algorithm that takes parameters and subroutines describing the pattern and tactics to be used. This flexibility allows us to refine the pattern and tactics to fit the current problem better.

Note that compass search is part of the GSS subclass, i.e. an implementation of GSS run with a particular set of parameters will behave exactly like compass search, but GSS is significantly more complex to implement.

2.2.4.2 Asynchronous Parallel Pattern Search

Asynchronous Parallel Pattern Search (APPS) was described by Hough, Kolda, and Torczon [6]. It is, as indicated by the name, a pattern search method specifically designed to be executed in parallel while incorporating asynchronous features. Asynchrony is useful because the number of available processors may not be an integer fraction of the batch size, and/or because the execution time for the function evaluations may not be constant. If either of these is the case, and our algorithm is synchronous, a potentially large fraction of the available processors may have to wait until the current batch of evaluations is completed until they are given more work.

The APPS algorithm solves these issues by making each process responsible for its search direction. The processes communicate their current best points

[1]When bound constraints are enforced, or bookkeeping is used, we will often get fewer valid/new moves.

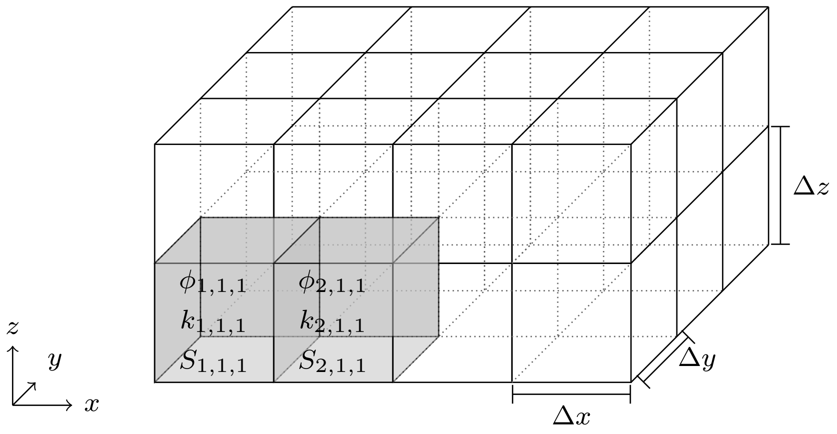


Figure 2.4: Illustration of a $4 \times 3 \times 2$ reservoir model.

with each other and make decisions on which direction to move from what they currently know. The processes are also able to detect on their own whether they have converged, and send a terminate signal to the other processes if they have.

While the APPS algorithm is likely too different from other pattern search algorithms to be implemented in FieldOpt, elements from it are likely to prove very useful as future additions to the part of FieldOpt controlling the general program flow.

2.3 Reservoir Simulation

In FieldOpt, reservoir simulators play the role of objective function evaluators. The reservoir model is simulated with a set of properties, and the resulting value of, for instance, the cumulative oil production is taken as the value of the objective function. In this section, we present a very brief overview of the basics of reservoir simulation.

A reservoir simulator utilizes a number of differential equations to describe the flow of fluids. As solving these equations analytically is only feasible for very simple systems [7], it is necessary to create discrete reservoir models so that the flow equations can be solved numerically.

The reservoir is discretized by dividing it into blocks, each associated with a set of properties (e.g. porosity, permeability, saturations) approximating the properties in the corresponding area of the real reservoir. This is illustrated in Figure 2.4. This discrete model allows us to calculate the flow within the model using a set of easily solved equations, each of which describe the state of each block, including the flow into- or out of it, at some point in time.

2.3.1 About the MRST Reservoir Simulator

The MRST (MATLAB Reservoir Simulation Toolbox) is developed by SINTEF Applied Mathematics. It provides multiple solvers suitable for a range of systems. It is currently the only reservoir simulator supported by FieldOpt. MRST is mainly intended as a toolbox for rapid prototyping and demonstration of new simulation methods and modeling concepts on unstructured grids [8]. As indicated by the acronym, it is developed in the MATLAB programming language.

2.4 Parallel Computing

This section describes the most common parallel computer architectures, focusing on the ones available for us to use at NTNU. The descriptions are somewhat simplified in that cache memory and network topologies are ignored. We do this largely because they are not critical to the development of FieldOpt. The cache-intensive operations are performed by simulators over which we have little or no control, and FieldOpt's communication is not extensive enough to benefit significantly from specific topologies.

Parallel computing may be implemented at different levels in the system, the four main levels being [9]

1. *Data-level parallelism.* When we simultaneously operate on multiple bits of data. An example is when different processors perform some mathematical operation on different parts of a vector. This form of parallelism is often a component in other forms of parallelization.

2. *Instruction-level parallelism.* When a processor simultaneously executes more than one instruction. One method of achieving this is through instruction pipelining, where the multiply and add operations in $\vec{a} = \vec{b} + \gamma\vec{c}$ may be performed simultaneously. Instruction pipelines are normally present in modern processors, and compilers automatically optimize code for to take advantage of them.

3. *Thread-level parallelism.* A thread is a portion of a program that may or may not run on a separate processor in a shared memory system. It shares the resources of its parent process.

4. *Process-level parallelization.* A process is a program that is running on the computer. It has its own set of resources. When utilizing process-level parallelization, the different instances may all reside on a single machine, or spread out across a network, and normally share information using message passing.

The two first forms of parallelism will not be discussed further, as FieldOpt does not use data-level parallelism[2], and instruction-level parallelism is taken care of automatically by the hardware and compiler.

[2]Although data-level parallelism is not explicitly implemented in FieldOpt, it is likely used extensively by the reservoirs simulator it utilizes.

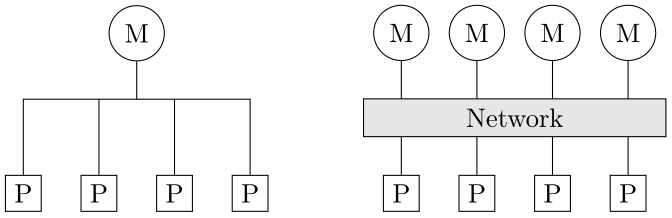


Figure 2.5: Examples of two shared-memory multiprocessor architectures. On the left, the processors are connected to the shared memory using a system bus. On the right, the processors are connected to the shared memory using a network. The illustration is adapted from [9].

2.4.1 Parallel Computer Hardware Architecture

When implementing a parallel computer program, we need to take into consideration the architecture of the hardware on which the program will be executed. The hardware architecture is important because the different architectures require different forms of communication between the parallel parts of the program. The two primary types of architectures that are interesting in our case are *shared-memory* and *distributed-memory*.

2.4.1.1 Shared-Memory Multiprocessor Architectures

In a shared-memory multiprocessor system, multiple processors share the same memory and can use it for communication [9]. ThereYay are different ways of implementing a shared-memory architecture. The processors may be connected to the memory using a system bus, or through a network; both implementations are illustrated in Figure 2.5. The former implementation is most common today and is found in computers with a multi-core CPU or multiple CPU's.

The processors in a shared-memory architecture are most easily exploited using thread-level parallelism, through implementations like POSIX and OpenMP. Using thread parallelization is convenient as we do not have to worry about how to get data from one thread to another: they can all read the same information stored in the local memory. Thread-level parallelism often also gives us a performance boost compared to process-level parallelism, as we avoid the latency and bandwidth issues associated with the use of message passing for communication.

2.4.1.2 Distributed-Memory Multiprocessor Architectures

In a distributed-memory multiprocessor, each processor is associated with a memory module (Figure 2.6), and the processors are connected to each other through some network [9]. The distributed-memory architecture is commonly used in larger computers, e.g. supercomputers.

As the processors are unable to access each others' memory, communication, i.e. data transfer, is achieved by utilizing a message passing library, for example

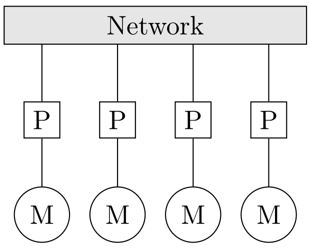


Figure 2.6: Example of a distributed-memory multiprocessor architecture. Each processing unit has its own memory, and the processors are interconnected through a network. The illustration is adapted from [9].

the MPI (Message Passing Interface) library. We use the library to spawn multiple instances of the program that can communicate with each other by sending and receiving messages. The performance depends on the network topology and its inherent latency and bandwidth, as well as the size and frequency of data transfers.

2.4.2 Computer Clusters

Computer clusters consist of multiple computers connected to each other through a network, usually a local area network (LAN). Computer clusters normally consist of multiple compute-nodes, as well as (at least) one login node and shared memory nodes.

Modern computer clusters often combine the shared-memory and distributed-memory architectures. For example, the Kongull computer cluster at NTNU consist of one login node, four I/O (shared memory) nodes ad 108 compute-nodes [10]. An illustration of its architecture is shown in Figure 2.7. Each of the compute-nodes has an internal shared-memory architecture with private memory. The nodes are connected to each other through a LAN, i.e. distributed-memory architecture, and they all have access to the same information stored in the shared memory nodes.

2.4.3 Parallel Programming Paradigms

As mentioned in the previous section, different hardware architectures require us to use different forms of communication in our parallel programs. These forms of communication are implemented in different programming paradigms. Here we will discuss the two most commonly used paradigms: thread based implemented using OpenMP, and message passing implemented using MPI.

To illustrate how to implement multi-threading and message passing, we will implement a program that calculates the sum of all elements in a vector containing all integers from 0 to 99. A serial implementation of the program is shown in Listing 2.1. Besides the `main`-function, it contains one function to initialize the vector and one function to calculate the sum of the vector.

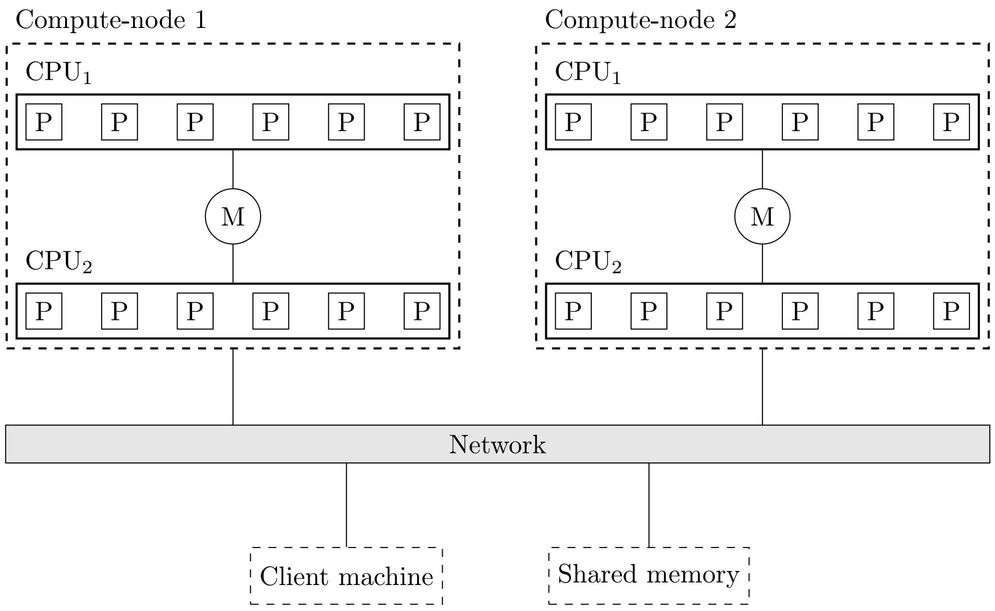


Figure 2.7: Kongull's hardware architecture. Multiple nodes are connected to each other through a network. Each node contains two CPUs with six processor cores sharing the same memory. In addition to the computation nodes, there's also one login/client node and shared memory/storage.

Listing 2.1: Serial implementation of a program that generates a vector containing the numbers from 0 to 99 and calculates the sum of the numbers.

```cpp
int calculateSum(vector<int> numbers) {
    int sum = 0;
    for (int i = 0; i < numbers.size(); ++i) {
        sum += numbers[i];
    }
    return sum;
}

vector<int> initializeNumbers(int size, int start) {
    vector<int> numbers(size);
    for (int i = 0; i < size; ++i) {
        numbers[i] = start + i;
    }
    return numbers;
}

int main(int argc, char **argv) {
    int size = 100;
    vector<int> numbers = initializeNumbers(size, 0);
    int sum = calculateSum(numbers);
    cout << sum << endl;
    return 0;
}
```

Parallelization will be shown using OpenMP and MPI, but with a larger focus on MPI as this is what we have implemented in FieldOpt. We include OpenMP because it is a simple way to provide further parallelization in more computationally demanding optimization algorithms that may be implemented in the future.

2.4.3.1 OpenMP

Thread based parallelization is the easiest way to implement parallelization on shared-memory systems. One way of implementing it is by use of OpenMP, available for the C, C++ and Fortran programming languages [9]. If we want to, for example, parallelize a for-loop, this is easily achieved using a *pragma*, i.e. a compiler statement. A parallel version of the `calculateSum`-function in Listing 2.1 using OpenMP is shown in Listing 2.2.

Note that OpenMP has to be supported by the compiler to work (and most modern compilers do support it), but the program will still compile (without parallelization) if it is not supported.

The main reason parallelizing the `for`-loop with OpenMP is as easy as it is, is that all the threads have access to the same vector-variable stored in the memory. OpenMP simply distributes intervals of the for-loop to the threads using some scheduling system (which we may select manually).

2.4.3.2 MPI

On distributed-memory architectures, parallelism is usually implemented using process-level parallelism and message passing through the MPI library. Information

Listing 2.2: Parallel implementation of the program in Listing 2.1, using OpenMP. The only change from the serial version is the addition of a pragma-statement in the `calculateSum`-function, the remaining code is identical.

```
int calculateSum(vector<int> numbers) {
    int sum = 0;
    #pragma omp parallel for
    for (int i = 0; i < numbers.size(); ++i) {
        sum += numbers[i];
    }
    return sum;
}
```

is transferred between processes using point-to-point or collective communication. The following are the most important communication functions found in MPI:

Send Send data from one process to another process.

Receive Receive data from another process. When the program reaches a call to a Receive function, it will wait until it receives something, i.e. the function is "blocking".

Broadcast Send the same data data from one process to all other processes.

Scatter Send different parts of a data array from one process to all other processes.

Gather/Allgather Inverse of scatter. Receive parts of an array from other processes and assemble it into one array. Allgather will assemble the full array on all processes.

Reduce/Allreduce Similar to Gather/Allgather, this function will collect the elements of an array on other processes and perform some operation on them, e.g. calculate the sum or average of the numbers in an array spread across several processes.

The MPI library assigns each instance of the program a unique *rank* (an ID number) and provides a function to access the number of instances of the program. We use this information when we call the Send/Receive functions, and to determine which process should perform which task. The library also provides functions for configuring the network topology, i.e. how the processes are "connected" to each other.

Listing 2.3 shows how the `main`-function in Listing 2.1 is modified to implement MPI-parallelization across four processes. The `initializeNumbers` and `calculateSum` functions used are identical to the serial implementation, but the `main`-function is modified to ensure that each process only initializes one-fourth of the vector before calculating the sum of that part. The `MPI_Allreduce`-function is then used to sum all the part-sums, giving the total sum on all the processes.

Listing 2.3: Parallel implementation of the program in Listing 2.1, using MPI. The initializeNumbers and calculateSum functions are identical to the serial implementation. Only the required parts of the vector is initialized by each process before the sum of that part is calculated. Finally, the total sum is calculated and distributed to all processes using MPI_Allreduce.

```
int main(int argc, char **argv)
{
    int rank, worldSize;
    MPI_Init(&argc, &argv);  // Initialize MPI.
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);  // Get the total number of
     processes.
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Get the rank of this process.
    int size = 100;

    vector<int> part = initializeNumbers(size/worldSize, size/worldSize*rank);
       // Initialize part of the vector.
    int partSum = calculateSum(part);  // Calculate the part-sum.
    int sum;
    MPI_Allreduce(&partSum, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);  //
     Calculate the complete sum.
    cout << sum << endl;
    MPI_Finalize();
    return 0;
}
```

2.4.4 Performance Metrics

To quantify how good the implementation of a parallel program is, we need some standard metrics. The two primary performance metrics for parallel software systems are *speedup* and *efficiency*. The two are closely related to each other and require us to measure the serial execution time when only one processor is available, and the parallel execution time when p processors are available [11].

2.4.4.1 Speedup

Speedup is a measure of the reduction in execution time when the number of processors available is increased from one to p. It is calculated as the ratio between the serial execution time T_s and the parallel execution time on p processors T_p:

$$S_p = \frac{T_s}{T_p} \qquad (2.3)$$

The speedup typically increases up to some limit with the number of processors available for solving the problem. The speedup flattens out when the amount of work to be performed is too small to keep the available processors busy. If the amount of data which needs to be transferred between the processors is large, or if data has to be sent many times during the execution, the speedup may also be reduced due to network transfer times and latency.

The *ideal* speedup is $S_p = p$, i.e. the execution time is reduced linearly with increasing p. If a program displays perfect speedup, increasing p from one to two will halve the execution time, increasing it to four will reduce it to one fourth, and so on.

2.4.4.2 Efficiency

Efficiency is a measure of how well the available processors are utilized on average. It is defined as the ratio between the speedup S_p and the number of available processors p:

$$\eta_p = \frac{S_p}{p} \qquad (2.4)$$

The ideal efficiency is $\eta_p = 1$ for any p.

2.4.4.3 Amdahl's Law for Multiprocessor Systems

Amdahl's law for multiprocessor systems allows us to calculate the speedup we can expect to get from a program running on p processors, given that the fraction of work performed by the program that lends itself to parallelization, f, is known [9]. Amdahl's law for speedup is given as

$$S_p = \frac{1}{(1-f) + f/p} \qquad (2.5)$$

From equation (2.5) and Figure 2.8 we see that if we are to get a significant speedup as p grows large, we must have

$$1 - f \ll f/p, \qquad (2.6)$$

i.e. nearly all the work must be parallelizable.

In our case, the parallelizable work f is the evaluation of the batches of cases produced by the optimization algorithm. The serial work $1 - f$ is the work performed by the optimizer itself. Thus we will get a larger speedup for optimization algorithms which do not perform much work themselves, and which produces large batches of cases to be evaluated in each iteration.

2.4.5 Communication Overhead

In all distributed parallel systems, there will be communication delays between the running processes. The two primary measures of this delay are *latency* and *bandwidth*. The latency of a system is the time it takes for one bit to travel from the source to its destination. The bandwidth of a system is the amount of information that can be transferred per time unit after the first bit has arrived [12].

Typically, for a node in a computer cluster with an internal shared-memory architecture, the latency will be very low, and the bandwidth will be very high. However, when we utilize more than one node, information has to be transferred over the local network. The latency will be significantly higher and the bandwidth considerably lower than for internal communication in the node.

This overhead may become a significant performance bottleneck for programs that need to transfer large amounts of data between the processes if the bandwidth is low, or for programs that need to transfer information very frequently if the latency is high.

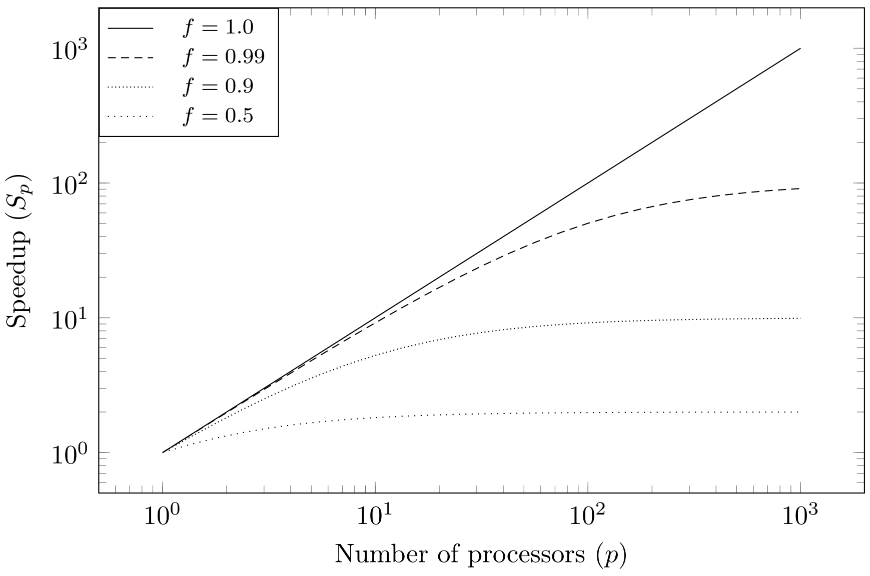


Figure 2.8: Amdahl's law.

2.5 Software Architecture

One of the most widely used textbooks on the topic [13] uses the following definition of software architecture:

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

For larger systems such as FieldOpt, which are intended to be used for a long time and to be modified by several different people, the architecture is crucial. The system's structure has to facilitate maintenance (e.g. bug-fixing and upgrading of libraries) and expansion (e.g. adding new optimizers and simulator interfaces). The architecture is also important for the initial implementation of features. For example, implementing MPI parallelization is made much easier by having low coupling in the portion of the program concerned with the control flow of the program.

A useful way to categorize the properties a system should have, i.e. be built to support, is through quality attributes, defined by [13] as

... a measurable or testable property of a of a system that is used to indicate how well the system satisfies the needs of its stakeholders.

The quality attributes we deemed to be most important for FieldOpt are:

Modifiability: How easily modifications can be made to the system.

Performance: The system's ability to make timing requirements.

Scalability: How well the system is able to utilize added resources.

Testability: How easily the system can be made to demonstrate its faults.

Quality attributes are implemented through the use of architectural tactics and patterns. Architectural patterns are collections of architectural decisions that are often found in practice, and tactics are the building blocks of patterns. Many tactics may simultaneously work to fulfill multiple quality attributes.

Some examples of tactics to increase the modifiability of a system are to reduce the size of modules, increase the cohesion in the modules, and reduce the coupling in the modules. Together, these tactics let us more easily change the functionality in one part of the system without having to make modifications to many other parts of the system, thus saving time.

The testability of a system may be increased by, perhaps most importantly, limiting the complexity of the modules in a system so that we can thoroughly test them independently to ensure their correctness. Note that the tactics employed to increase the modifiability of a system often has reduced module complexity as a side-effect, so designing the system to satisfy the first attribute often satisfies the other.

The performance of a program like FieldOpt, where most of the work is actually performed by an external program (a reservoir simulator) which we have little or no control over, can primarily be increased by executing the external program in a more efficient manner, for example by introducing concurrency. This brings us to the last quality attribute: scalability. We can increase the scalability of our program by enabling it to exploit a larger amount of resources, i.e. design it so that it can utilize a large number compute-nodes and communicate efficiently between them.

Chapter 3

Implementation

In this chapter, we describe how we implemented the various features in FieldOpt. We briefly describe the overall architecture of the program and the libraries we decided to use before we go more in depth on the optimizer and simulator interfaces and the implementation of the parallel runner and broker. We also include a brief discussion on the use of driver files and how FieldOpt handles them.

In this chapter and the next, class names, attributes and methods/functions are typeset using a `mono-spaced` font.

On the Importance of a Good Software Implementation To the user, FieldOpt may be perceived as a black box, taking in a driver file and giving out a solution and some logs (Figure 3.1). However, how quickly the program can give you a solution, how well it utilizes the available resources, and for how long it will be able to do its job depends on *how* it is implemented.

If the program's source code lacks documentation, uses "strange" libraries, or lacks a good structure, it will be difficult to maintain and extend it in the future. If a new developer tries to add new functionality to the program but is unable to understand some critical part of the existing code, he may not be able to do his job. If the code is very tightly coupled, he may have to change many parts of the source code to fix one bug. If the code uses a library that is no longer maintained, it may become incompatible with other more updated libraries or in the worst-case scenario unavailable.

Today, how quickly a computationally heavy program such as FieldOpt can do its job largely depends on how well it can utilize a larger number of processors. A program is much easier to parallelize if the different functionalities in its source code are isolated, i.e. the critical parts of the program have as few responsibilities as possible. If this is not the case, distributing the workload efficiently will be difficult or impossible.

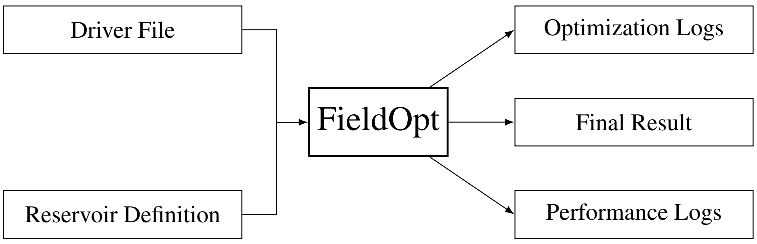


Figure 3.1: How FieldOpt operates as seen from the outside.

3.1 Frameworks and Libraries

Like most other non-trivial programs, FieldOpt is implemented using a set of frameworks and libraries to ease the implementation of some basic tasks. In this section, we give a short description of the frameworks and libraries we've used.

3.1.1 Qt Framework

Qt is a cross-platform application and UI framework for C++ developed by The Qt Company [14]. We decided to use the Qt Framework while developing FieldOpt mainly because ResOpt was developed using Qt: using Qt in FieldOpt eased the reuse of ResOpt code.

The Qt Framework also provides features that proved very useful when developing the new system, such as the *Signals and Slots* event handling system, which we most widely employ for handling exceptions. It also provides wrappers for common C++ structures and data types, such as strings and vectors, which make them easier to work with.

Another advantage of using Qt is the build system: *qmake*. Qmake helps with locating libraries and generating build scripts for various platforms. When using the *Qt Creator* IDE (the official Qt IDE) for development, using qmake is much more convenient than using the alternatives (e.g. *CMake*), because qmake takes semi-automatically generated Qt Creator project files as input for creating build scripts.

3.1.2 Boost

Boost provides a set of free libraries for C++ complementing the standard libraries. Many of Boost's features have been incorporated into the C++11 standard [15]. FieldOpt primarily uses Boost's MPI library, which provides wrappers for the most important functionalities in MPI and makes it easier to work with.

3.1.3 Open MPI

Open MPI is an open source implementation of Message Passing Interface (MPI). It is maintained by several academic, research and industry partners [16].

3.2 Diagrams

In the following sections we will make use of a few standard types of UML (Unified Modeling Language) diagrams. Here, we describe these briefly. Slightly more extensive descriptions are given in the appendices.

3.2.1 Class Diagrams

Class diagrams describe the properties of a class. We generally show simplified diagrams, only including the interesting methods, attributes, and inheritances. A brief explanation of the notation we use is given in Appendix A.1. Also, note that in our diagrams, when a class implements an abstract class, it implicitly implements *all* of its virtual functions.

3.2.2 Sequence Diagrams

Sequence diagrams show the sequence of instructions executed in a particular scenario and the objects involved in the execution. A brief explanation of how they should be interpreted is given in Appendix A.2.

3.3 Architecture

The main concerns when creating FieldOpt were to establish an architecture that better facilitates performance and scalability when running the program, as well as increasing the modifiability and testability of the code. This section gives an overview of the architectural tactics employed while porting the ResOpt code and implementing the original FieldOpt code.

The primary architectural tactics used to increase modifiability and testability are increased cohesion and reduced coupling. These two are closely related, and collectively they state that each module and submodule should have very specific responsibilities and depend on as few other modules as possible. An example of how these tactics have impacted the structure of FieldOpts code include the complete separation of the code into a *library*, intended to be a stable core, which includes the model description, optimizers and simulator interfaces; and an *application* which utilizes the library to perform the actual tasks. These separate modules also use different external libraries: the FieldOpt library only depends on the Qt library, while the much smaller application additionally depends on Boost and Open MPI. These relations are illustrated in Figure 3.2. Another example of how the reduced coupling tactic is employed is the use of abstract classes such as `Optimizer` and `Simulator` to limit the amount of changes required in the application code when the library is changed. A simplified overview of the relations between the most important classes in the application- and library layers are given in Figure 3.3.

The same tactics used to increase modifiability and testability also help with better utilizing the available resources in parallel systems, as the different types

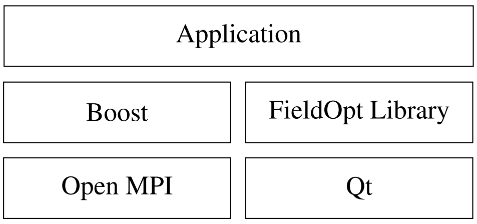


Figure 3.2: An overview of FieldOpts architecture and the libraries it utilizes. A component utilizes all components below it, but none above it or beside it.

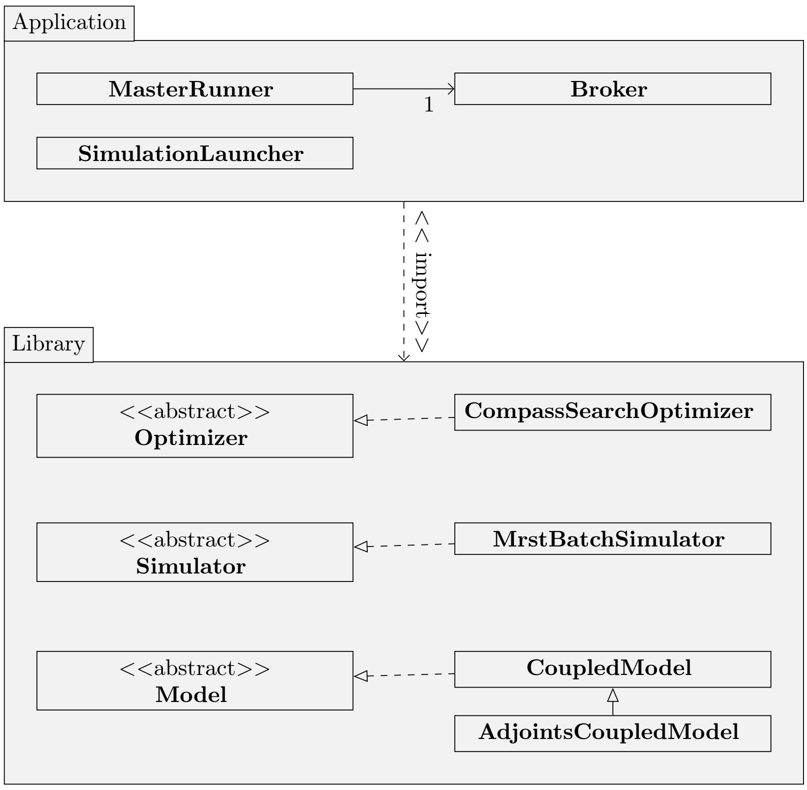


Figure 3.3: A class diagram showing the most important classes in FieldOpt and their relations.

of MPI processes can run more independently. The processes responsible for simulation do not know what the process in charge of the general program does and vice versa, and their only common modules are the objects used to describe the data flow between them. This independence between the parts means that the workload is easily distributed across a network, and changes to one will not affect the other as long as the changes do not affect the communication.

We use *abstract classes* as interfaces for models, simulators, and optimizers. An abstract class defines some common attributes and methods for a particular subtype of classes, but it has some *purely virtual methods*[1] which it defines the signature for, but not the implementation. Abstract classes cannot be instantiated on their own – only classes implementing their virtual methods can be instantiated. Abstract classes let us specify the critical methods and attributes common to all simulator and optimizer variations in one place, and define the methods and attributes specific to an optimization algorithm or simulator interface in a separate class. Abstract classes also free FieldOpts application layer from having to worry which optimizer or simulator is being used after instantiating it, as the common methods are sufficient to utilize the optimizers and simulators.

3.4 Model

The `Model`-class and its subclasses contain all the information needed by FieldOpt to solve the problem specified by the user. It contains the path to the file specifying the reservoir; attributes describing the variables in the model; and settings defining which simulator and optimizer to use and with which settings. The `Model` class contains methods to validate itself; methods to update its various attributes after a simulation has been executed; and methods to apply perturbations to its variables.

`Model` is implemented as an abstract class, so we need to have more specific implementations of it. ResOpt defined several different variations, e.g. coupled model, decoupled model, but FieldOpt currently only uses one: `AdjointsCoupledModel`, a subclass of `CoupledModel` created specifically to be used with the MRST reservoir simulator. The inheritance is shown in the class diagram in Figure 3.4.

Note that, as the models are more or less copied from ResOpt, they contain a lot of code for setting and updating pipe systems. These methods and attributes are not shown in the class diagrams and are not discussed here as this functionality is not used in FieldOpt. They also contain some optimization related methods that are not used, because this work is now handled by optimizers (as it should be).

[1]They can also have virtual methods that are implemented but can be overridden by subclasses.

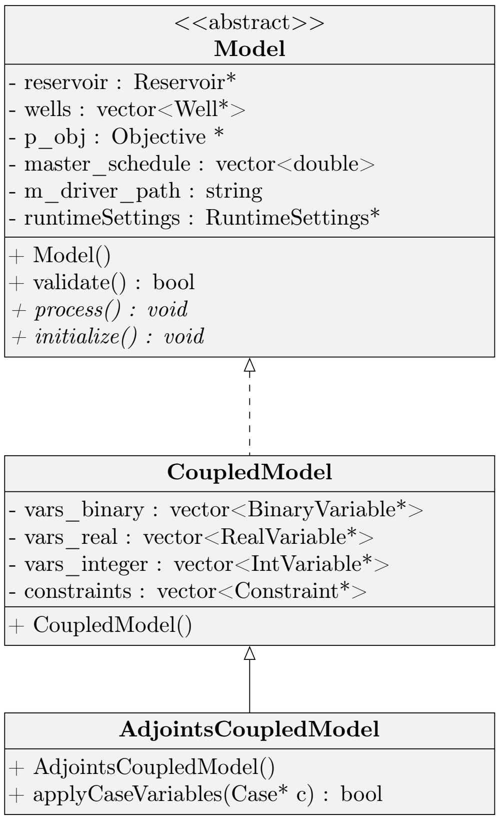

<<abstract>>
Model
- reservoir : Reservoir*
- wells : vector<Well*>
- p_obj : Objective *
- master_schedule : vector<double>
- m_driver_path : string
- runtimeSettings : RuntimeSettings*
+ Model()
+ validate() : bool
+ process() : void
+ initialize() : void
CoupledModel
- vars_binary : vector<BinaryVariable*>
- vars_real : vector<RealVariable*>
- vars_integer : vector<IntVariable*>
- constraints : vector<Constraint*>
+ CoupledModel()
AdjointsCoupledModel
+ AdjointsCoupledModel()
+ applyCaseVariables(Case* c) : bool

Figure 3.4: Class diagram showing the abstract `Model` class and the subclasses that are used by FieldOpt.

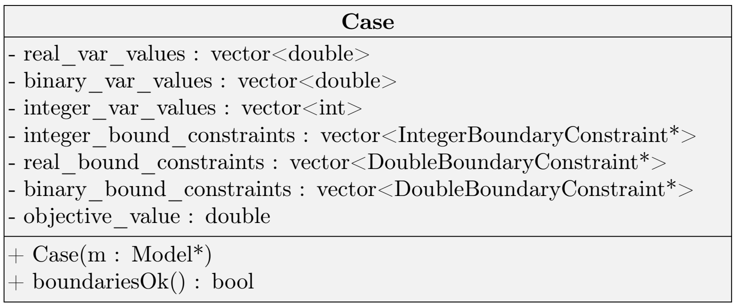

Case
- real_var_values : vector<double>
- binary_var_values : vector<double>
- integer_var_values : vector<int>
- integer_bound_constraints : vector<IntegerBoundaryConstraint*>
- real_bound_constraints : vector<DoubleBoundaryConstraint*>
- binary_bound_constraints : vector<DoubleBoundaryConstraint*>
- objective_value : double
+ Case(m : Model*)
+ boundariesOk() : bool

Figure 3.5: Class diagram showing the implementation of the `Case` class.

3.5 Case

The `Case` class contains a complete set of variable values from a perturbed model. It also contains the bound constraints specified for the problem, as well as a method to verify itself (i.e. to make sure that the variable values do not violate their constraints). A class diagram is shown in Figure 3.5. The `Case` class is used to describe perturbations by most parts of FieldOpt except for the simulator for three main reasons:

- the `Case` class is much more primitive than the `Model` class: it uses vectors of primitive types instead of vectors of custom objects, which generally makes it easier to work with.
- we need to generate many perturbations of each model that all need to be kept in the memory. Because `Case` objects are much smaller than `Model` objects, it is practical, in terms of memory usage, to use `Case` objects that may be applied to a single `Model` object to update the variable values before it is evaluated by a simulator.
- having optimizers only use `Case` objects means that if someone wishes to implement a new optimizer, he or she need only learn how the relatively simple `Case` object works.

Note that the `Case`-class in FieldOpt contains a number of attributes and methods which are not used. These have been kept from ResOpt, but will likely be removed in the future.

3.6 Simulator Interface

The reservoir simulator interface has not been changed from ResOpt, only some internal code cleanup has been performed. The abstract `Simulator` class and its `MrstBatchSimulator` implementation is shown in Figure 3.6.

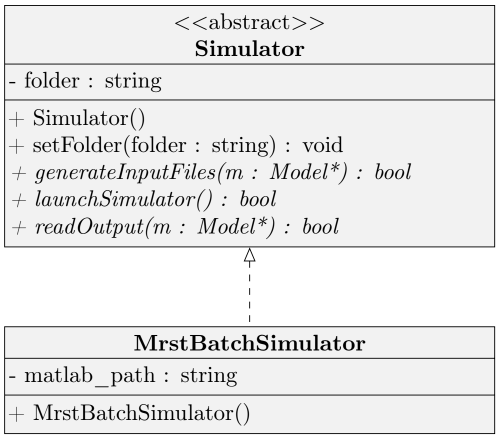

<<abstract>>
Simulator
- folder : string
+ Simulator()
+ setFolder(folder : string) : void
+ generateInputFiles(m : Model*) : bool
+ launchSimulator() : bool
+ readOutput(m : Model*) : bool
MrstBatchSimulator
- matlab_path : string
+ MrstBatchSimulator()

Figure 3.6: Class diagram showing the `Simulator` interface and the `MrstBatch-Simulator` implementation.

Currently, only an interface for an old version of MRST (2013a) has been implemented. Among the interfaces available in ResOpt, MRST was chosen because it is the easiest one to get working on NTNU's Kongull computer cluster.

After instantiating a specific simulator, performing a simulation using any simulator and acquiring the results requires four methods to be called:

1. `setFolder(folderPath)`: Set the work directory for the simulator. The driver files for the simulator will be created here, and the simulator will write its output here.
2. `generateInputFiles(model)`: Generate the driver files required by the simulator to evaluate the `Model` object provided as parameter.
3. `launchSimulator()`: Execute the simulation.
4. `readOutput(model)`: Read the output from the simulation and store it in the provided `Model` object.

3.6.1 MRST Interface

The MRST interface is not an interface in the same sense as the other interfaces discussed in this chapter. It is a connection between FieldOpt and the external simulator. It generates Matlab scripts and executes them using Matlab.

The implementation in FieldOpt has not changed much from ResOpt. Only two changes other than general cleanup have been made:

- The Matlab scripts were updated to make them work with the most recent versions of Matlab.

- POSIX is now used to execute Matlab instead of QProcess from the Qt framework. This was necessary as QProcess refuses to play nice with MPI, but it means that FieldOpt cannot be executed on recent versions of the Windows operating system.

3.7 Optimizer Interface

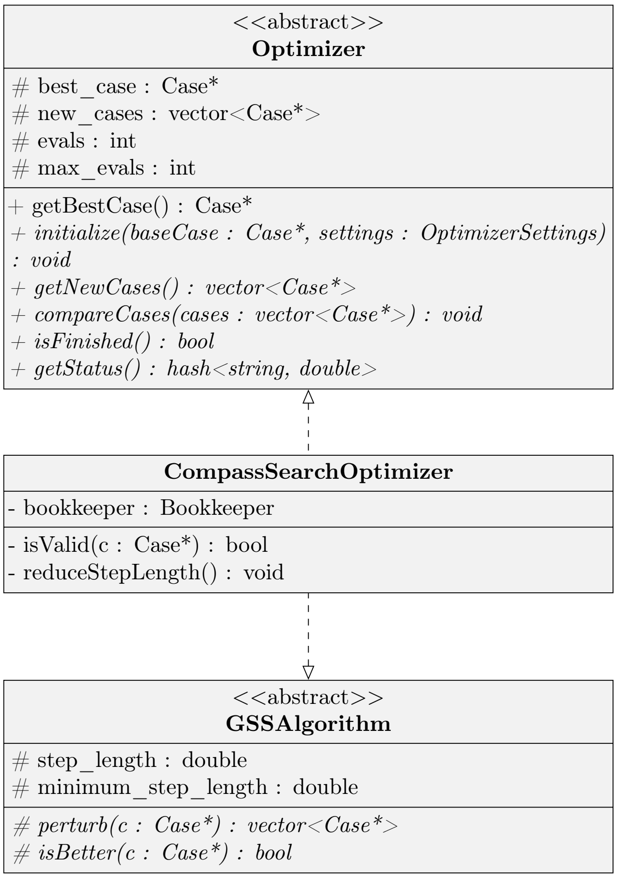
Beside the MPI parallelization, the specification of the optimizer interface was the area where the most effort was made when we created FieldOpt. We strove to create an interface that is as simple as possible (i.e. requires few external function calls to operate) while at the same time allowing a large set of algorithms to be implemented using it. The resulting abstract `Optimizer` class is shown in Figure 3.7. The interface is designed to fit most pattern search algorithms, although there are certainly some exceptions (e.g. APPS [6] which requires the algorithm to perform tasks that in FieldOpt are delegated to the application layer).

The `Optimizer` class has four attributes which are available to all classes implementing it:

- `best_case`: Holds the best case found at any time.
- `new_cases`: Holds the most recently generated set of perturbations.
- `evals`: The number of perturbations that have been generated and returned to the application layer. When a case has been returned, it is considered to have been evaluated and counts towards the maximum number of function evaluations allowed.
- `max_evals`: The maximum number of function evaluations allowed. This may be specified by the user in the driver file.

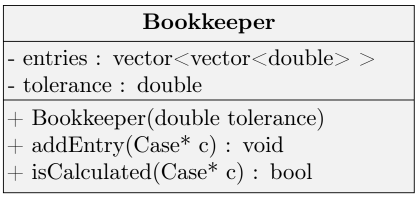
It also provides a method which returns the current best case, `getBestCase()`, and specifies five virtual methods which must be implemented by all optimizers:

- `initialize(baseCase, optimizerSettings)`: Initialize the optimizer by setting the base-case and various settings, such as pattern parameters, maximum number of evaluations etc.
- `getNewCases()`: Get a new set of perturbations.
- `compareCases(cases)`: Process a set of evaluated perturbations. Update the best case and adjust pattern parameters if necessary.
- `isFinished()`: Check if the optimizer is finished; i.e. if any of the termination conditions has been met (minimum step length, maximum number of allowed evaluations, etc.).
- `getStatus()`: Get the current status of the optimizer. This may be used to by the application layer to monitor the overall progress of the optimizer.



<<abstract>>
Optimizer
# best_case : Case*
# new_cases : vector<Case*>
# evals : int
# max_evals : int
+ getBestCase() : Case*
+ initialize(baseCase : Case*, settings : OptimizerSettings) : void
+ getNewCases() : vector<Case*>
+ compareCases(cases : vector<Case*>) : void
+ isFinished() : bool
+ getStatus() : hash<string, double>
CompassSearchOptimizer
- bookkeeper : Bookkeeper
- isValid(c : Case*) : bool
- reduceStepLength() : void
<<abstract>>
GSSAlgorithm
# step_length : double
# minimum_step_length : double
# perturb(c : Case*) : vector<Case*>
# isBetter(c : Case*) : bool

Figure 3.7: Class diagram showing the abstract `Optimizer` and `GSSAlgorithm` classes and the `CompassSearchOptimizer` class.



```
Bookkeeper
- entries : vector<vector<double> >
- tolerance : double
+ Bookkeeper(double tolerance)
+ addEntry(Case* c) : void
+ isCalculated(Case* c) : bool
```

Figure 3.8: Class diagram showing the implementation of our bookkeeper.

Compass search is the only algorithm implemented thus far in FieldOpt. In addition to the abstract `Optimizer` class, it also implements the abstract `GSSAlgorithm` class, which specifies a few attributes and methods that are common to all generating set search algorithms. The implementation utilizes a bookkeeper to avoid dealing out cases for evaluation which have already been evaluated, and uses `Case`'s built-in ability to check whether all variables are within their defined bound constraints to avoid dealing out invalid cases.

3.7.1 Bookkeeper

The `Bookkeeper` class allows optimization algorithms to submit `Case` objects to it, and check whether a `Case` object has already been added to the bookkeeper. A class diagram of the implemented `Bookkeeper` class is shown in Figure 3.8.

When a case is added to the bookkeeper (using `addEntry(case)`), it is converted to a vector containing all of its variables, describing a point in n-dimensional space, which is stored in the `entries` vector. When an optimizer attempts to check if a case has already been calculated (using `isCalculated(case)`), the bookkeeper vectorizes the variables in the case and calculates the distance d between the input case and each of the other cases in the `entries` vector using the expression

$$d = \sqrt{\sum_{i=1}^{n} \left[\left(C_i^{new} - C_i^{old} \right)^2 \right]} \quad \ldots\ldots\ldots\ldots \quad (3.1)$$

where C_i^{new} represents variable i in the "new" case, and C_i^{old} represents variable i in one of the cases stored in `entries`. If the distance between the new case and any of the old cases is less than the value specified in `tolerance`, the case is said to have been previously calculated, and the method returns `true`; otherwise it is a new case and the method returns `false`.

3.8 Parallelization

All parallel aspects in FieldOpt are implemented in the application layer using MPI. To facilitate this, we use three main classes: `MasterRunner` which takes care of the program flow in general, calls the optimizer, distributes perturbations and prints results; `SimulationLauncher` which receives perturbations from `MasterRunner`, applies them to the `Model`, performs the simulation and returns the results; and a `Broker`, part of the `MasterRunner`, which is responsible for effectively distributing the perturbations among the processes running instances of `SimulationLauncher`.

Overall, the application layer in FieldOpt has a “master/slave” architecture. The master process, running an instance of `MasterRunner`, takes care of all program flow, and controls all slave processes running `SimulationLauncher` through MPI messages.

As mentioned earlier, the `MasterRunner` and `SimulationLauncher` classes are very loosely coupled. Their only common classes are the so-called transfer objects `Perturbation` and `Result`, which are used to transfer new perturbations and simulation results.

A flowchart showing the overall flow of FieldOpt on process-level is shown in Figure 3.9. We give more detailed descriptions of the program flow in in the processes running `MasterRunner` and `SimulationLauncher` in the following subsections.

3.8.1 MasterRunner

When FieldOpt is executed, a single instance of the `MasterRunner` class is created in the root process, i.e. the MPI process with rank zero. This object uses an optimizer to get new perturbations, assigns unique IDs to them, and passes them on to the `Broker`. It then orders the `Broker` to have the perturbations evaluated. When the `Broker` has terminated, the `MasterRunner` retrieves the results from it and resets it before submitting the results to the optimizer. This process is repeated until the optimizer reports that it has reached some termination condition. The process is illustrated in the sequence diagram in Figure 3.10.

3.8.1.1 Broker

The `MasterRunner` keeps an instance of the `Broker` class. It is responsible for cross-process communication for the `MasterRunner`. It sends `Perturbation`-objects and receives `Result`-objects. The broker keeps track of all perturbations in the current iteration and notes which have been evaluated. It also tracks which processes are currently free (i.e. not currently performing a simulation). It uses this information to ensure that as long as there is a sufficient number of not-evaluated perturbations available, all processes are kept busy. By doing this asynchronously, the broker mitigates the issue of some simulations taking a longer time to evaluate than others.

3.8.2 SimulationLauncher

Besides receiving perturbations sent from the root process and returning results to it, the `SimulationLauncher` class only has one responsibility: launching simulations. After it receives a new perturbation, it applies it to the `Model` before simulating it. It then extracts the objective value from the simulation results and

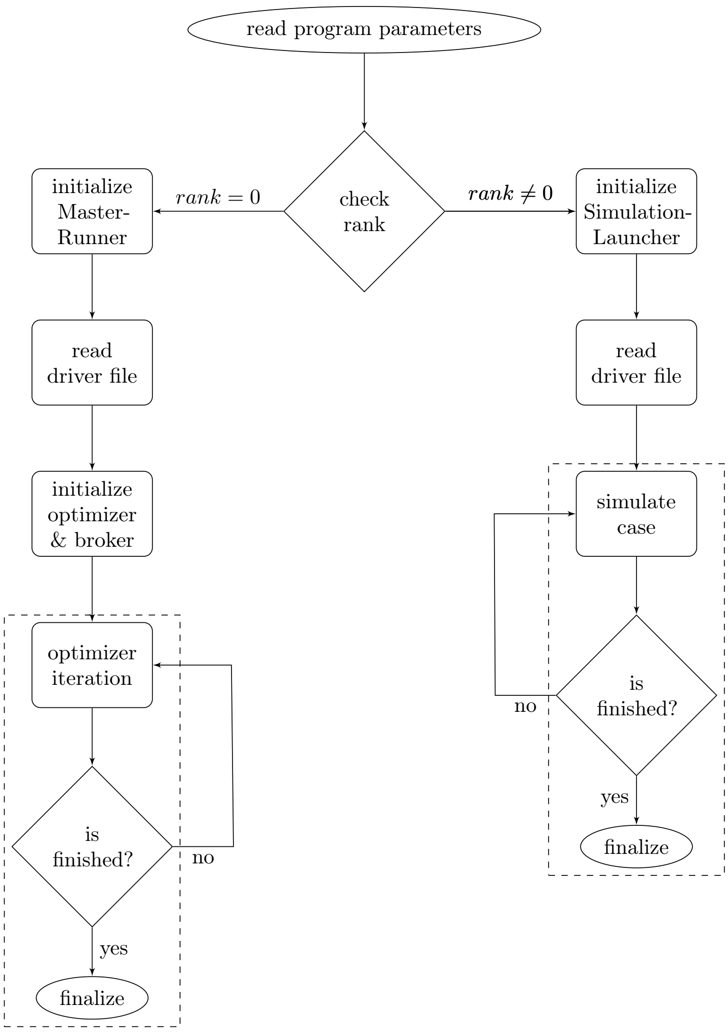


Figure 3.9: Flowchart of the program flow on process-level. “Rank” is the unique number MPI assigns to each process.

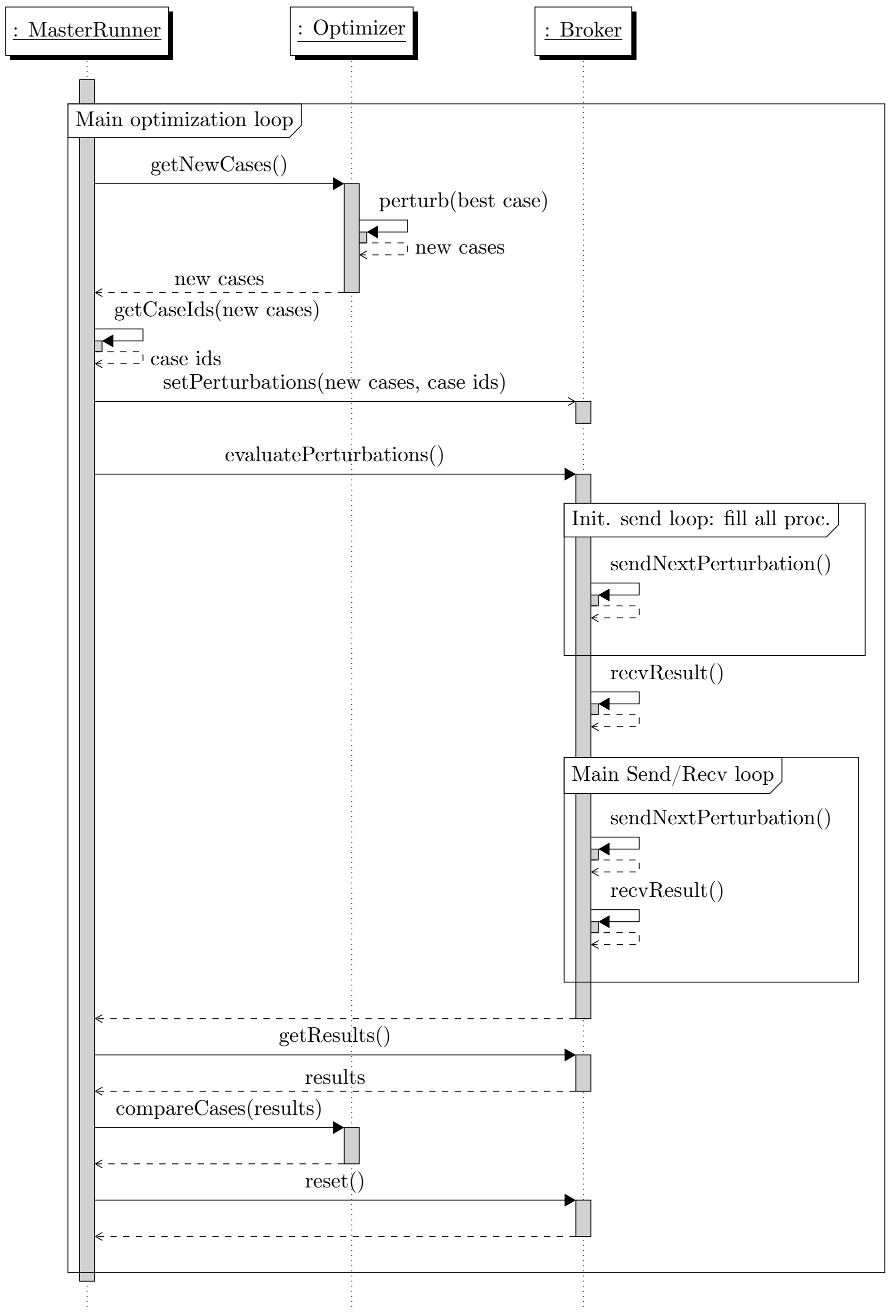


Figure 3.10: Sequence diagram showing the program flow as seen from the root process. This is a detailed view of the framed portion on the left side of Figure 3.9.

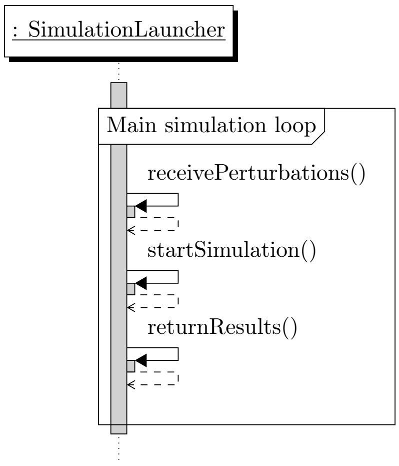


Figure 3.11: Sequence diagram showing the program flow as seen from a processes responsible for simulations. This is a detailed view of the framed portion on the right side of Figure 3.9.

returns it to the `MasterRunner`. A sequence diagram illustrating the process is shown in Figure 3.11.

3.8.3 Communication

The communication between processes in FieldOpt's application layer is performed using primitive `MPI_Send/Recv` function calls, facilitated by two different transfer objects: `Perturbation`, containing all variable values in a perturbation; and `Result`, containing the value of the objective function after the perturbed model has been evaluated. Both objects have an ID attribute, and `Perturbation` and `Result` objects corresponding to the same perturbation will have the same ID. Class diagrams of the objects are shown in Figure 3.12. Note that these objects cannot be sent directly using MPI, so they are disassembled and sent as primitive arrays. When they are received, the arrays are reassembled into the proper type of object. This decomposition is illustrated in Figure 3.13.

The overall communication in the application layer is shown using a sequence diagram in Figure 3.14. This diagram is a combination of Figures 3.10 and 3.11 with the interactions between them added. Essentially, the procedure of one optimizer iteration as seen from the root process is as follows:

1. The `MasterRunner` retrieves a set of perturbed cases from an optimizer, generates `Perturbation`-objects with assigned IDs, submits them to the `Broker`, and asks it to have them evaluated.

2. Given p available processes, the broker sends out $p-1$ perturbations, thus putting all `SimulationLauncher` instances to work. When a perturbation is

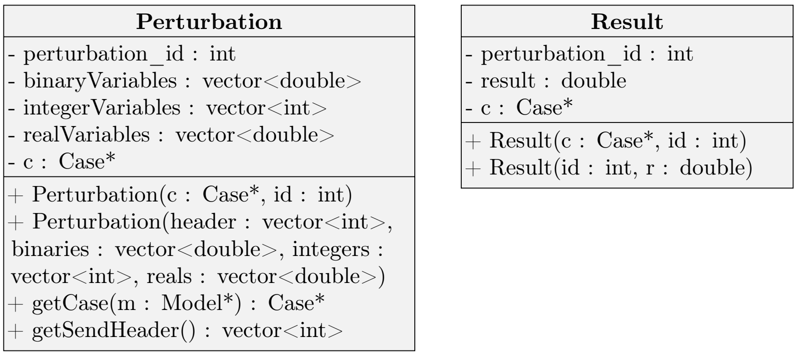

Perturbation
- perturbation_id : int
- binaryVariables : vector<double>
- integerVariables : vector<int>
- realVariables : vector<double>
- c : Case*
+ Perturbation(c : Case*, id : int)
+ Perturbation(header : vector<int>, binaries : vector<double>, integers : vector<int>, reals : vector<double>)
+ getCase(m : Model*) : Case*
+ getSendHeader() : vector<int>
Result
- perturbation_id : int
- result : double
- c : Case*
+ Result(c : Case*, id : int)
+ Result(id : int, r : double)

Figure 3.12: Class diagrams showing the implementation of the transfer objects.

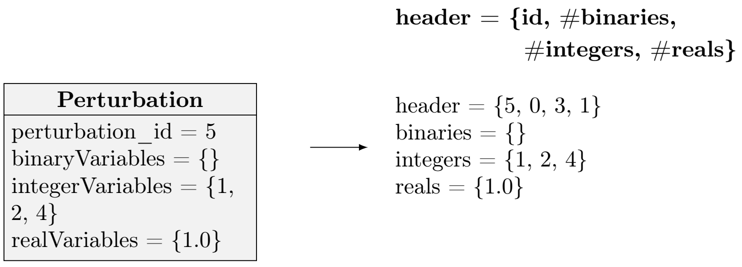


Figure 3.13: Illustration of how the attributes in a `Perturbation` object are decomposed into four primitive vectors, the first one describing the object as a whole and the rest containing variable values.

sent to a process, that process is noted as being "busy".

3. The `Broker` waits until it has received the first result. When the result arrives, the process that sent it is marked as "free". The received result is reassembled into a `Result`-object and stored.

4. The broker now enters a loop which alternates between sending a perturbation to a free process (and marking it as busy), and receiving a result from a process (and marking it as free). This continues until all perturbations have been sent for evaluation.

5. When all processes have been marked as "free", i.e. when all results have been received by the `Broker`, control is given back to the `MasterRunner`.

6. When the `MasterRunner` asks for the results, the `Broker` assembles the `Perturbation`, `Result` and `Model` objects into a vector of `Case` objects which is then passed on to the optimizer.

As seen from the `SimulationLauncher`, the procedure is significantly simpler:

1. The `SimulationLauncher` waits to receive a perturbation.

2. When a perturbation is received, the `Perturbation` object is reassembled and a `Case` object is created from it before applying the `Case` to the `Model`.

3. A simulator is launched to evaluate the `Model`.

4. The objective value is extracted from the simulation results and a `Result` object is created. This object is sent back to the root process.

5. Return to step 1.

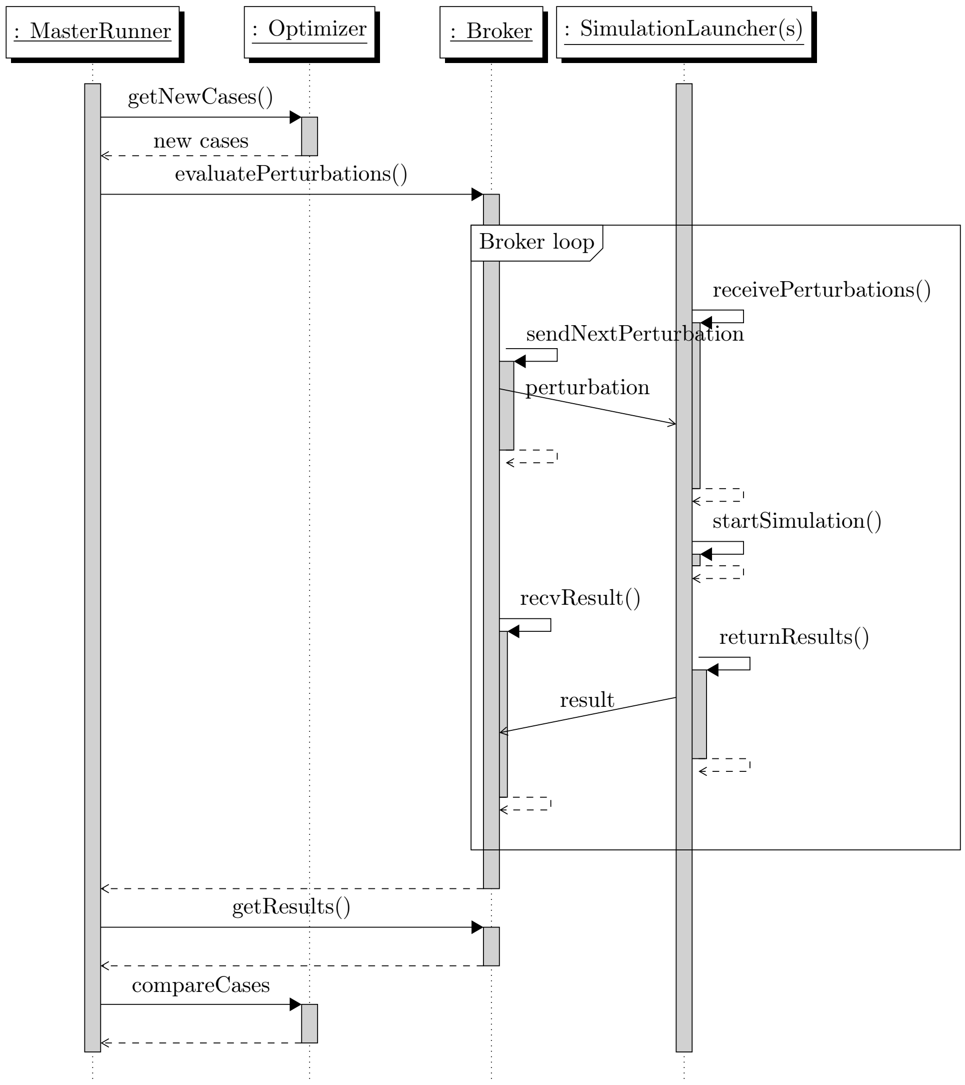


Figure 3.14: Sequence diagram showing an overview of the communication in the application layer. This is a simplified combination of Figures 3.10 and 3.11.

3.9 Driver Files

FieldOpt uses the same driver files as ResOpt, with some added (optional) properties for configuring the new type of optimizers. The driver files are, however, handled slightly differently. Whereas ResOpt created instances of `Optimizer` and `Simulator` in addition to `Model` when reading the driver file, FieldOpt creates the `Model` object and settings objects describing the optimizer and simulator settings. These settings objects are attached to the `Model` object and handled in the application layer. Decoupling the `Optimizer` and `Simulator` classes from the `DriverReader` class was done as part of a larger strategy to reduce the size and complexity of the `DriverReader` class after migrating it from ResOpt. We felt that doing this was necessary because it is one of the most complex classes, with very tight coupling to almost every part of ResOpt, which is not desirable.

3.10 User Interface

FieldOpt's user interface is console-based and very simple. The program is started by executing one of three command variations:

```
    mpirun -n 6 Console /path/to/driver/file.dat
    mpirun -n 6 Console /path/to/driver/file.dat -verbose
    mpirun -n 6 Console /path/to/driver/file.dat /path/to/mrst/
```

"Console" is the (temporary) name of the application calling the FieldOpt library. All these three variations will execute FieldOpt with 6 MPI processes. The first example executes FieldOpt in "silent mode" (which hides all printing from the library layer except for severe errors) using the MRST-path specified in the driver file; the second does the same as the first, but prints *all* warnings and debug-related messages from the library layer; the third does the same as the first, but ignores the MRST path specified in the driver file and instead uses the one provided as a parameter. The third mode is included because when a program is executed on Kongull, it is necessary to give file paths that include a system variable. Achieving this is significantly easier when the shell decodes the variable before it reaches FieldOpt than if FieldOpt is to read and decode the path including the variable from the driver file.

Chapter 4

Case Study

To test FieldOpt, we used one of the example models provided with ResOpt. The reservoir model used is described in the next section. We created several variations of the driver file for the problem, where the bottom hole pressure was allowed to vary with various numbers of time steps. The cases were run on NTNU's Kongull computer cluster. The test runs allowed us to verify that our implementation of the compass search algorithm is correct, and also, more interestingly, to measure the performance of FieldOpt under various conditions.

4.1 Model

The model consists of a $20 \times 10 \times 6$ reservoir grid with one producing well. The grid blocks have dimensions $60' \times 60' \times 3'$; the porosity is 30%; the permeabilities are $k_x = 50, k_y = 50, k_z = 25$ mD; the initial oil saturation is 85% and homogeneous; the well is placed in a corner and penetrates the 5 topmost layers. The model is illustrated in Figure 4.1.

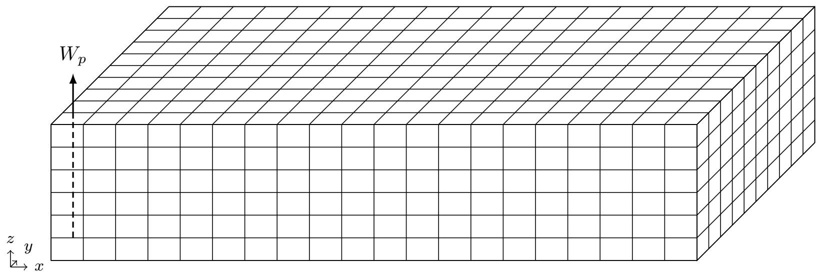


Figure 4.1: Reservoir model used in this case study. It consists of $20 \times 10 \times 6$ blocks; it has one production well

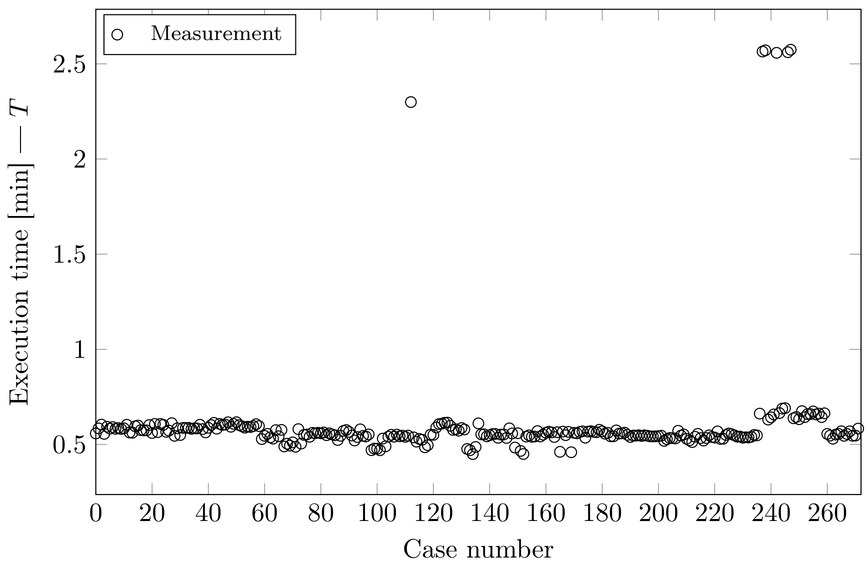


Figure 4.2: Plot showing the execution time for the reservoir simulator for the 272 different cases evaluated for the case with 12 variables, run on 26 processors.

4.2 Simulator Execution Time

As we see in Figure 4.2, the execution times for the simulations are quite similar. The vast majority of the 272 cases have execution times of around 0.5 minutes, but there are a few (6) cases with execution times of up to 2.5 minutes, i.e. approximately five times longer. It is unclear what causes this. Likely, it is due to some artifact in simulating the model with those specific sets of variables, e.g., convergence issues with the solver that may be related to how the numerical problem is set up for that particular configuration. Another possible source, at broker-level, might be some unknown bug in the load balancing system, causing it to hand multiple simulations to the same processor. In any case, this type of situations is one of the main reasons why we implemented the broker. In our current implementation, the other processors will continue to simulate other cases while the simulations with longer execution times are running.

4.3 Optimizer Convergence

Figure 4.3 illustrates the convergence rate of the compass search optimizer implementation. We see that the end result has already been found after iteration 3, but the algorithm has to continue until some termination condition has been reached. In this case, it reaches the minimum step length after iteration 9.

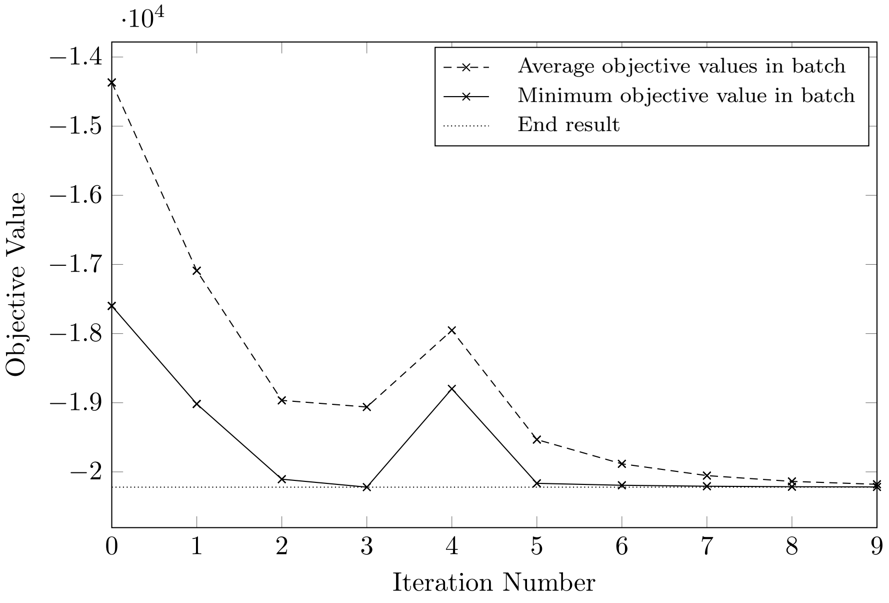


Figure 4.3: Batch objective values showing the convergence of the optimizer. A batch is one set of cases generated by the optimizer, i.e. the perturbations generated in one iteration.

4.4 Performance

We measured the performance of FieldOpt by executing the same model with 2, 4, 8, 12 and 24 variables, on between 2 and 48 processors. Runs were not made using a single process, as FieldOpt requires at least two processes to function (one for the `MasterRunner` and at least one for `SimulationLauncher`s). The serial execution time used when calculating speedup and performance is therefore from runs with two processes. This is valid because FieldOpt operates serially when run with only two processes: the root process and the other process never perform work at the same time.

We did not measure the serial execution time for the case with 24 variables, as this would likely have taken around 30 hours. Instead, we use relationships deduced from the measured times to approximate an execution time for this case.

4.4.1 Execution Time

The *execution time* of a program is the simplest measure of performance, and it is a critical component of most other performance measures. The execution time of a program is the time measured from the moment the program is executed to the moment it terminates. The term is also used for subsections of the program, for example the time it takes to perform a simulation or a batch of simulations. In this section we look at FieldOpt's execution times for problems with varying numbers of variables and available processors to determine how the program scales with these parameters.

4.4.1.1 Dependence on Problem Size Under Serial Execution

In Figure 4.4 we examine how the execution times increase with an increasing number of variables when FieldOpt is run in serial (i.e. on two processors). The times were measured for cases with 2, 4, 8 and 12 variables. We see that the serial execution time T_s increases approximately with the square of the number of variables v:

$$T_s(v) = T_s(2) \times (v-1)^2, \qquad (4.1)$$

where $T_s(2)$ is the serial execution time for the case with two variables.

4.4.1.2 Dependence on Number of Available Processors

In Figure 4.5 we look at how the execution time is reduced as more processors are made available when FieldOpt is run repeatedly with the 12 variable-case. Ideally, the parallel execution time T_p as a function of the number of processors p would be $T_p(p) = T_s/p$. We do not quite reach this, as one of the processors (the one running the `MasterRunner`) performs no work most of the time. But, at least asymptotically, we do get close:

$$T_p(p) = \frac{T_s}{p-1} \qquad (4.2)$$

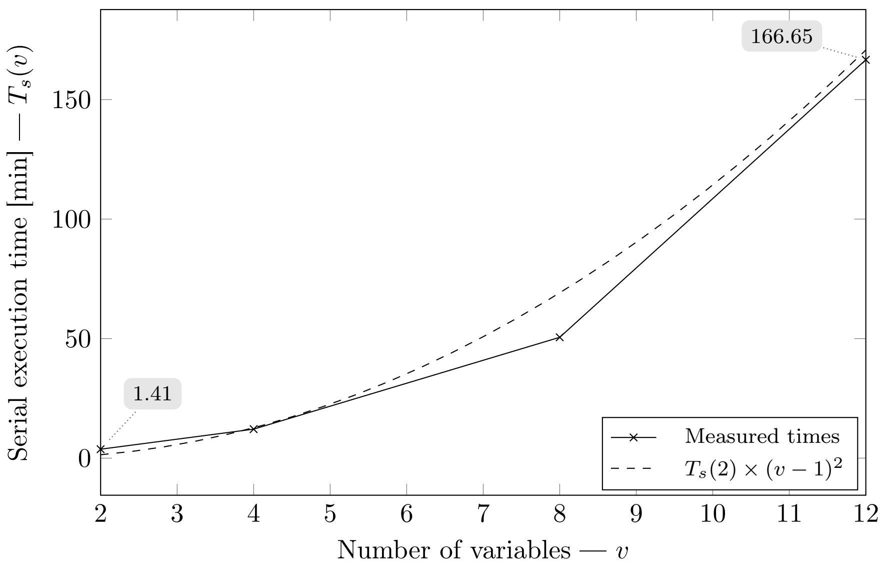


Figure 4.4: Measured execution times for cases with various numbers of variables run with two processors available.

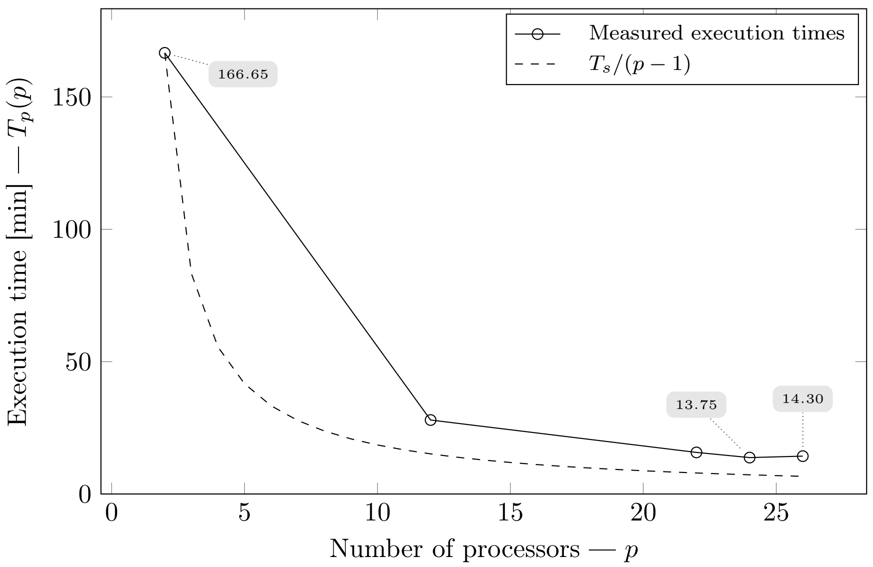


Figure 4.5: Measured execution times for the case with 12 variables for various numbers of available processors.

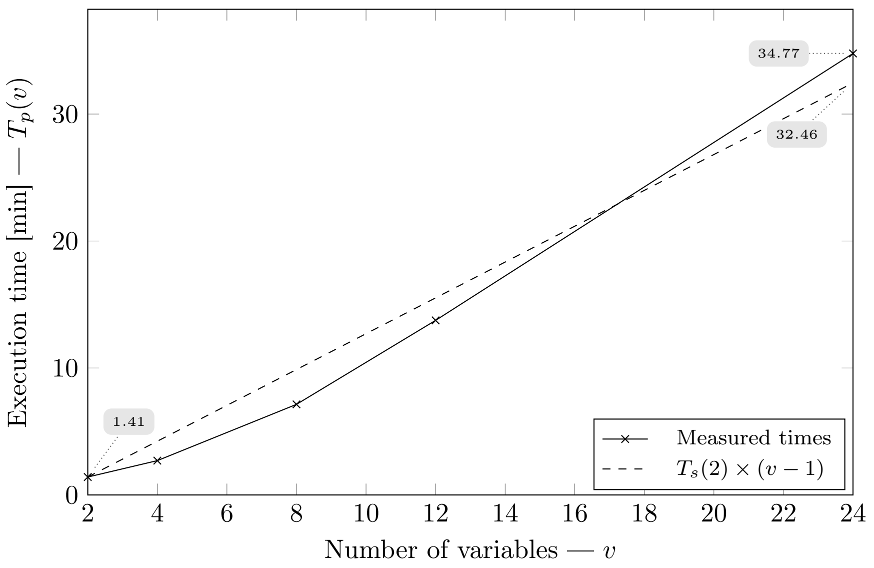


Figure 4.6: Measured execution time versus number of variables when using parallelization. The execution times picked for this plot are the lowest ones for each case, e.g. 4 variables with 9 processors, 8 variables with 18 processors.

4.4.1.3 Dependence on Problem Size Under Parallel Execution

In Figure 4.6 we have selected the cases with the lowest measured execution time for each variable-case, e.g. 4 variables executed on 9 processors and 8 variables executed on 18 processors. This lets us examine the parallel execution time for varying numbers of variables, $T_p(v)$. We see that this relation is approximately

$$T_p(v) = T_s(2) \times (v-1). \qquad (4.3)$$

In other words, the parallelization implemented in FieldOpt has reduced the execution time's dependency on the number of variables from quadratic to linear.

4.4.1.4 Resource Utilization

Figures 4.5, 4.7 and 4.8 collectively show all the execution times we measured. Table 4.1 shows the "best" (i.e. lowest) and "worst" (i.e. highest) execution times for all cases. The worst times for all cases are the times measured using only two processors (for the case with 24 variables, this value was approximated using equation (4.1)).

We see that the best execution times are all achieved when using approximately $p = 2v$ processors. This is because a case with v variables will give at most $2v$ cases to be evaluated per iteration, and to evaluate all of these simultaneously one go, we need $p = 2v + 1$ processors: $2v$ for the simulations and one for the `MasterRunner`.

Table 4.1: Highest execution time T_{worst} measured on two processors (approximated using equation (4.1) for the case with 24 variables); lowest execution times T_{best} measured on p_{best} processors; and the percentage reduction between the two. The execution times are given in number of minutes.

Variables	T_{worst}	T_{best}	Reduction	p_{best}
2	3.73	1.40	62%	24
4	12.12	2.70	78%	9
8	50.55	7.13	86%	18
12	166.65	13.75	92%	24
24	≈ 2000	34.77	98%	44

We see that, overall, we get a good reduction for all problem sizes, but the reduction is significantly larger for the larger problems. This is because the program is able to effectively utilize a larger set of processors when it has more cases to evaluate in each batch.

4.4.1.5 Local Maxima and Minima

From the execution time plots, especially Figure 4.5 and 4.8, we observe an interesting effect: the execution time increases when we add processors in excess of approximately $p = 2v$. In Figure 4.7 we also see that there are several local minima and maxima. Both these phenomena likely have the same cause.

The time it takes to evaluate a batch stays approximately constant at a few different levels when we add processors, but adding more MPI processes also increases FieldOpt's startup time and communication latency. This is because, given constant simulator execution times, we will evaluate the batches in a number of distinct sets. So if we have a batch of 8 cases[1] and we have 4 processors available for evaluation, the batch will be evaluated as two sets of 4 cases. If we increase the number to 5 processors the batch will be evaluated in one set of 5 cases and one of three. Each set will still take the same amount of time to evaluate, but the additional process adds to the initial startup time of FieldOpt, increases the communication latency between the processes, and is not used much of the time. The execution time in this case will not decrease until we add so many processes that each batch can be evaluated as one set. This is illustrated with measured batch execution times in Figure 4.9.

[1]The batch sizes will vary between optimizer iterations because of the bookkeeper and bound conditions. For instance, the batch sizes for a complete run with four varaibles are 8, 7, 4, 3, 2, 4, 4, 4, 4 and 4.

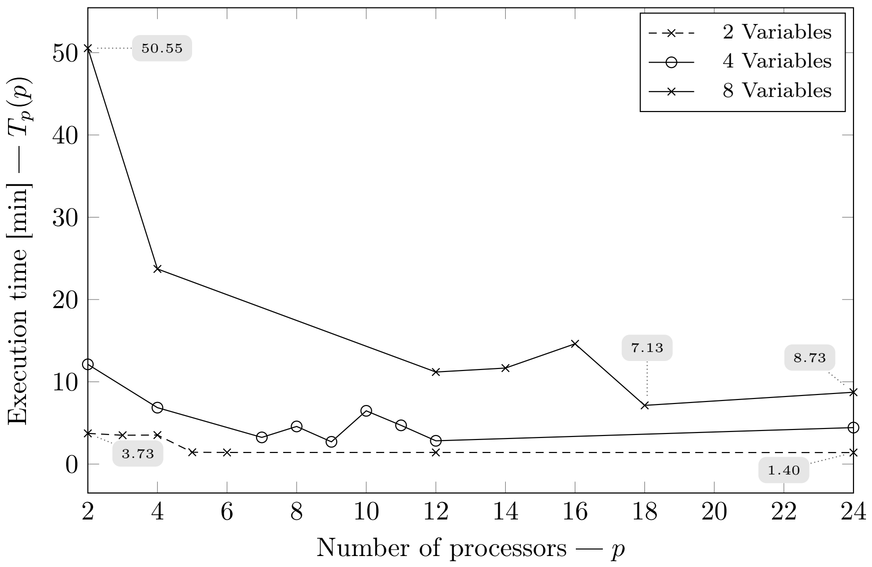


Figure 4.7: Execution time versus number of processors for the cases with 2, 4 and 8 variables.

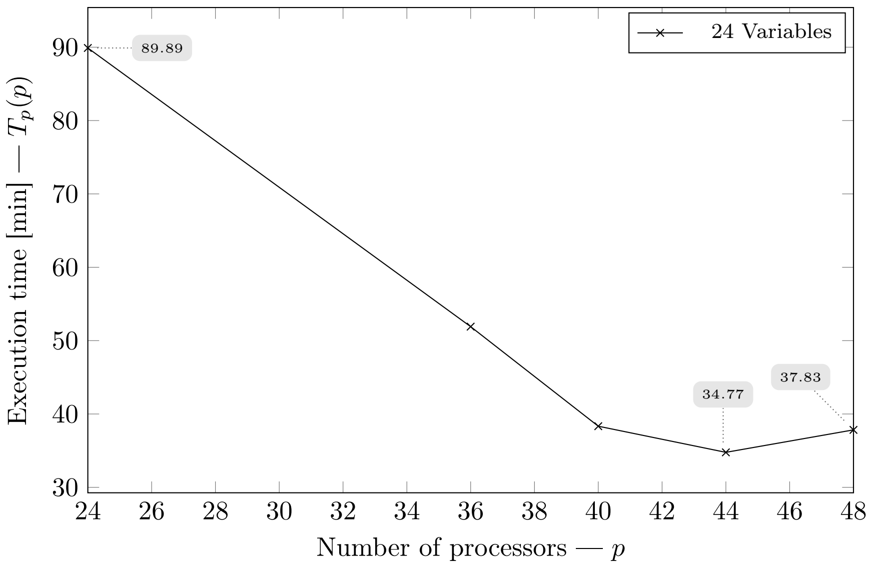


Figure 4.8: Measured execution time versus number of available processors for the case with 24 variables.

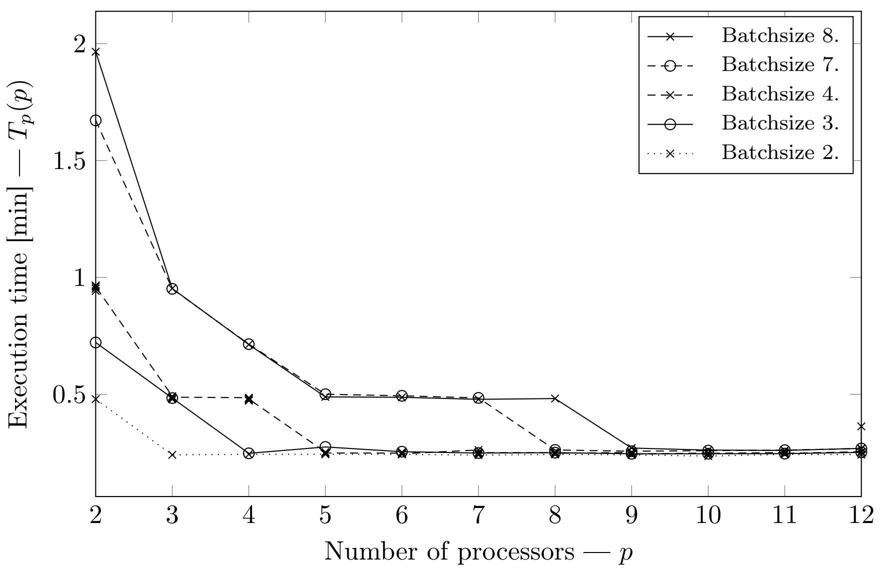


Figure 4.9: Batch execution time versus number of available processors.

4.4.2 Speedup and Efficiency

In Figure 4.10 we see that we achieve a good speedup, especially for the cases with higher numbers of variables. For the case with 12 variables we achieve $S_p > 10$ for $p > 20$, which corresponds to the value predicted using Amdahls Law when assuming that 95% of the work is parallelizable. In reality, the parallelizable work is an even larger portion, but the curve is dragged down by the fact that the root process is doing almost no work most of the time. The impact of this one mostly-idle process is even more apparent in the curves for 2, 4 and 8 variables. However, for larger cases, the impact would likely be negligible and we would come closer to perfect speedup. As we see in Figure 4.11, the estimated speedup for the case with 24 variables lies around optimal speedup.

The *efficiency* plot in Figure 4.12 shows that the parallel efficiency, especially for the smaller problems, rises and falls periodically with the number of processors. This is related to the issue with batch sizes discussed section 4.4.1.5. We also see that the parallel efficiency drops quite steeply when the speedup flattens because the added processors are not being utilized.

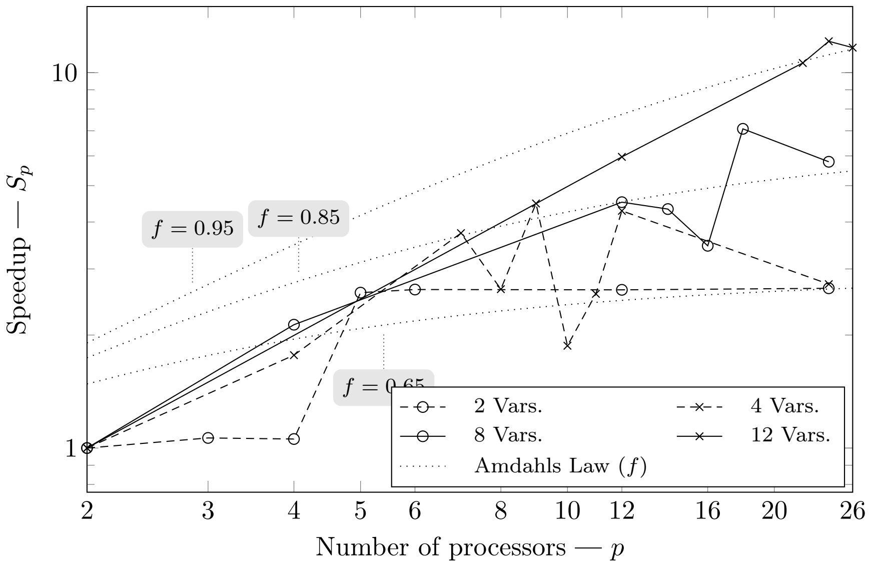


Figure 4.10: Plot of speedup for the cases with 2, 4, 8 and 12 variables. The dotted lines show the predictions from Amdahls Law (equation (2.5)) for various values of f.

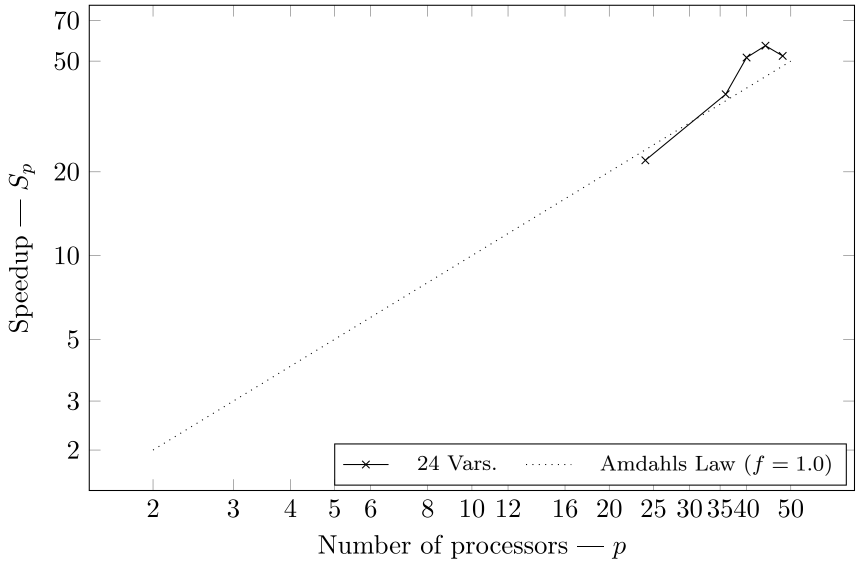


Figure 4.11: Speedup for the case with 24 variables calculated using an estimated value for the serial execution time from equation (4.1).

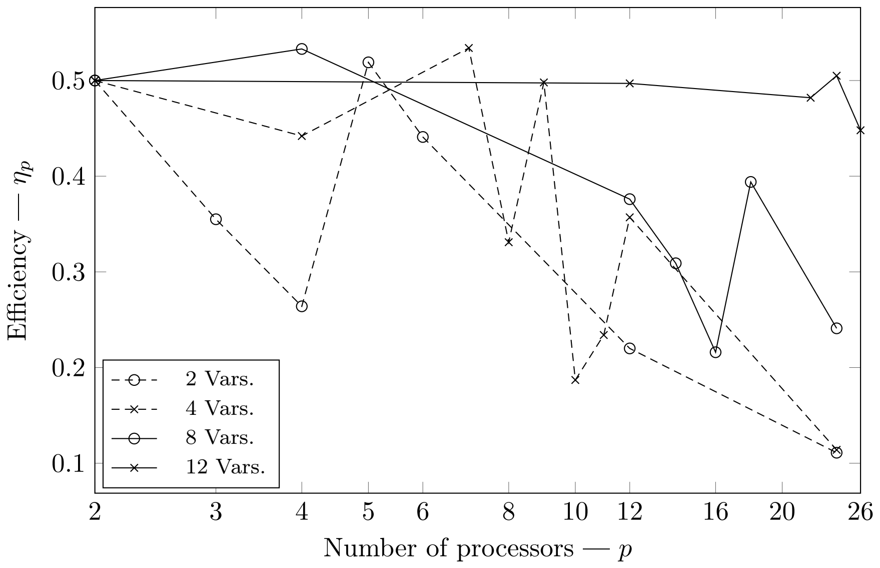


Figure 4.12: Efficiency.

Chapter 5

Conclusions & Recommendations for Further Work

This thesis has dealt with the development of a software framework that couples topics from mathematical theory, i.e., optimization algorithms, with topics from petroleum engineering, i.e., field development problems. The main goal of this framework is to facilitate research work that builds upon the integration of these two topics. A main priority has been to create software that runs efficiently on distributed hardware architectures. A second focus has been to make the code easily accessible and extensible through simplification and modularization and extensive documentation of the code, to encourage further development of the software by its users. At the moment, we feel FieldOpt can utilize large, distributed hardware architectures quite well. However, only time will show whether or not we have met our second goal. It was easy for us to implement a relatively simple optimization algorithm, but this might not turn out to be a straightforward process for other users. We might need to develop support-procedures based on user-feedback to aid this process.

Core Functionality There is still room for significant improvement of the core functionalities in FieldOpt. For example, the code requires much cleanup in general, and some of the code ported from ResOpt – particularly the model – should be completely rewritten to make it more flexible and easy to maintain. Moreover, while we have extended the documentation substantially, there remains a lot of work in this area, given FieldOpt is intended to be modified and extended by a range of developers with differing backgrounds.

Optimizer Interface At the moment, the optimizer interface we have implemented is clear and well-defined. Though the current interface should be suitable for most pattern search algorithms, we have thus far only implemented one algorithm: the simple compass search algorithm. In the future, the interface will likely need to be modified to cater to more complex types of pattern search algorithms, as well as to a wider range of different kinds of algorithms developed either in-house, or coming from ready-made solver packages.

Load Balancing The code that handles load balancing works well for problems that deal with a relatively large number of variables, with reservoir models that are not too demanding. However, the load balancing procedure is less suitable for problems with few variables and time-consuming models. In our current algorithm implementation, few variables mean that the optimizer will generate few cases per iteration, which in turn means that we can exploit fewer processors. A straightforward solution to this problem is to enable a second level of parallelization in our solution design, at the simulation level. Implementing support for a reservoir simulator that itself uses parallelization would allow a larger number of processors to be utilized even when the optimization algorithm yields few cases per iteration. Another possible solution is to implement a peer-to-peer communication paradigm (similar to the one used by the APPS algorithm [6]), instead of the current master/slave paradigm. The peer-to-peer paradigm would allow all the available processors to perform simulations, meaning no core(s) would be reserved solely for control and optimization purposes.

Asynchrony Another way to increase the performance in FieldOpt is to implement an asynchronous optimization algorithm. Currently, our asynchronous broker handles the load balancing issue arising from non-homogeneous simulation times quite well, but towards the end of evaluating a batch, an increasing number of processors will be idle. If the optimization algorithm itself is capable of providing new cases mid-iteration when given the results of the cases evaluated so far, we could potentially exploit all processors close to 100

Non-Homogenous Simulation Times While our brokers' handling of non-homogeneous simulation times works quite well in our test case, it is not good enough if a five-fold increase in simulation time, similar to the one seen in Chapter 4, occurs when running a model that normally takes several hours to simulate. Such situations with highly non-homogeneous simulation times must be handled systematically. A first approach is to develop more specific parameter settings at reservoir level to diminish the likelihood of troublesome configurations. Along these lines, we could though it would be significant undertaking, perform an in-depth study of the relationship between the range of feasible configurations and the numerical stability of the particular model. If possible, we could develop constraints to limit our search to ranges that are likely to yield stable solutions. Another option on the reactive side is for the broker level to include some form of early detection for this type situations, i.e., a monitor function. This monitoring would enable the broker to terminate simulations that are taking too long or behaving irregularly.

Driver Files Finally, the main way of configuring FieldOpt is, as with most reservoir simulators, through a driver file. FieldOpt uses ResOpt's driver files with only small additions, and they are are intended to be written manually using a text editor. These driver files "feel" outdated. For example, knowing what to put in them requires extensive reading of manuals, and finding errors is often time-consuming. For further development, we propose creating a graphical user interface that would allow the user to create and edit driver files for FieldOpt. These files should be in a machine-readable format (e.g. XML or JSON), since this would significantly reduce the complexity of the code needed to read and write the files. The reduced complexity would in turn make it easier to add new configuration options.

Glossary

Boost Free, cross-platform, peer-reviewed C++ library.. 26

C++ A programming language. 26

Efficiency A measure of the average of the average utlilization of the available processors when executing a program.. 21

MPI Message Passing Interface (MPI) is a standardized, portable interface for message passing. It is primarily used when programming parallel computers with distributed memory.. 16, 26

MRST MATLAB Reservoir Simulation Toolbox is a MATLAB toolbox developed by SINTEF Applied Mathematics.. 32

OpenMP Open Multi-Processing, an API supporting shared-memory multiprocessing.. 16

POSIX Portable Operating System Interface. Provides an interface to the undelying operating system, including a standard multi-threading library.. 16

Qt Cross-platform application and UI framework for C++.. 26

Speedup A measure of the reduction in execution time for a program when the number of available processors are increased.. 21

Bibliography

[1] J. Nocedal and S. J. Wright, *Numerical optimization.* New York: Springer, 2006.

[2] M. C. Bellout, D. Echeverría Ciaurri, L. Durlofsky, B. Foss, and J. Kleppe, "Joint optimization of oil well placement and controls," *Computational Geosciences*, vol. 16, no. 4, pp. 1061–1079, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10596-012-9303-5

[3] R. Hooke and T. A. Jeeves, ""direct search" solution of numerical and statistical problems." *J. ACM*, vol. 8, no. 2, pp. 212–229, 1961. [Online]. Available: http://dblp.uni-trier.de/db/journals/jacm/jacm8.html#HookeJ61

[4] V. Torczon, "On the convergence of pattern search algorithms," *SIAM Journal on Optimization*, vol. 7, no. 1, pp. 1–25, feb 1997. [Online]. Available: http://dx.doi.org/10.1137/S1052623493250780

[5] T. G. Kolda, R. M. Lewis, and V. Torczon, "Optimization by direct search: New perspectives on some classical and modern methods," *SIAM Review*, vol. 45, no. 3, pp. 385–482, 2003. [Online]. Available: http://dx.doi.org/10.1137/S003614450242889

[6] P. D. Hough, T. G. Kolda, and V. J. Torczon, "Asynchronous parallel pattern search for nonlinear optimization," *SIAM Journal on Scientific Computing*, vol. 23, no. 1, pp. 134–156, 2000. [Online]. Available: http://dx.doi.org/10.1137/S1064827599365823

[7] D. W. Peaceman, *Fundamentals of numerical reservoir simulation.* Amsterdam; New York: Elsevier Scientific Pub. Co. : Distributors for the U.S. and Canada, Elsevier North-Holland, 1977, ch. 1, pp. 2–5.

[8] SINTEF, "Sintef mrst project web page," 2015, accessed: 29.4.2015. [Online]. Available: http://www.sintef.no/Projectweb/MRST/

[9] F. Begali, *Algorithms and Parallel Computing*, 1st ed., ser. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Inc, 2011.

[10] NTNU HPC Group, "Kongull hardware," HPC Wiki, 2015, accessed: 08.05.2015. [Online]. Available: https://www.hpc.ntnu.no/display/hpc/Kongull+Hardware

[11] D. Eager, J. Zahorjan, and E. Lazowska, "Speedup versus efficiency in parallel systems." *Computers, IEEE Transactions on*, vol. 38, no. 3, pp. 408–423, mar 1989. [Online]. Available: http://dx.doi.org/10.1109/12.21127

[12] N. Matloff, *Programming on Parallel Machines*, 2015. [Online]. Available: http://heather.cs.ucdavis.edu/parprocbook

[13] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice.* Upper Saddle River, NJ: Addison-Wesley, 2013.

[14] The Qt Company, "Qt framework web page," 2015, accessed: 11.03.2015. [Online]. Available: http://www.sintef.no/Projectweb/MRST/

[15] B. Schäling, *The boost C++ libraries.* XML Press, 2014.

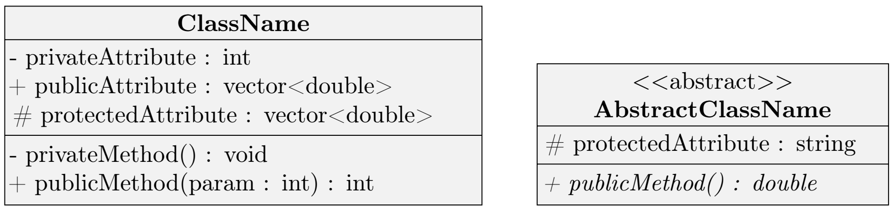
[16] Indiana University, "Open mpi project home page," 2015, accessed: 20.05.2015. [Online]. Available: http://www.open-mpi.org/

Appendix A

Software Development

A.1 Class Diagrams

This section is a short summary of how the various symbols in class diagrams are used in this report.


ClassName
- privateAttribute : int
+ publicAttribute : vector<double>
# protectedAttribute : vector<double>
- privateMethod() : void
+ publicMethod(param : int) : int
<<abstract>>
AbstractClassName
# protectedAttribute : string
+ *publicMethod() : double*

(a) Basic class properties. **(b)** An abstract class.

Figure A.1: Examples of a basic class and a basic abstract class.

A basic *class* is shown as in Figure A.1a.

- The *name* of the class is in bold on the first line.
- Methods always have parentheses after the name.
- *Private* attributes and methods are are prefixed with a '-'. Private attributes and methods are only available to the class itself.
- *Public* attributes and methods are prefixed with a '+'. Public attributes and methods are available for all parts of the program.
- *Protected* attributes and methods are prefixed with a '#'. Protected attributes and methods are only visible to the class itself and its subclasses.

- The type of an attribute and the return-type of a method is indicated after the last ':'.
- The type of a parameter is indicated after a ':' inside the parentheses.
- When a data structure type such as 'vector' or 'list' is followed by angle brackets, the word inside the brackets indicates the type of the elements which make up the data structure.

An *abstract class* is shown in Figure A.1b. Abstract classes cannot be instantiated on their own, only classes implementing their virtual methods may be instantiated. Virtual methods are shown in cursive. They *must* be implemented by classes that *implement* the abstract class.

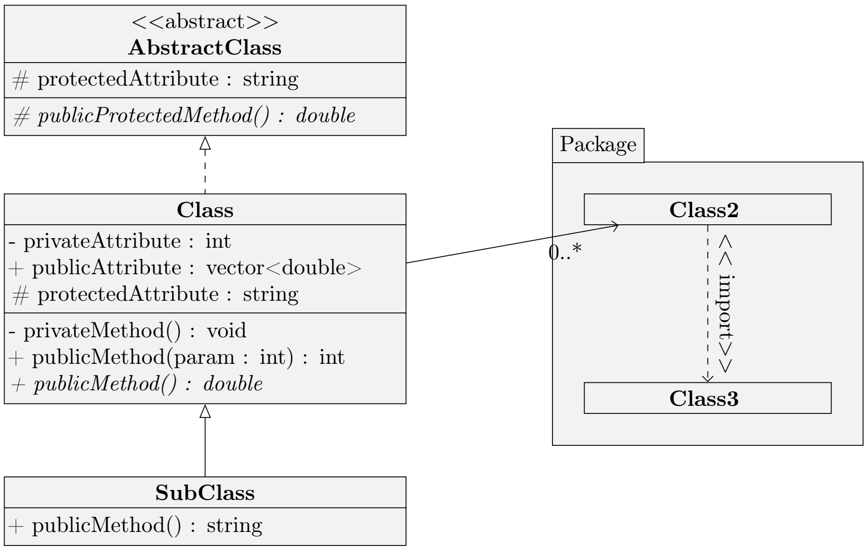

<<abstract>>
AbstractClass
# protectedAttribute : string
# publicProtectedMethod() : double
Class
- privateAttribute : int
+ publicAttribute : vector<double>
# protectedAttribute : string
- privateMethod() : void
+ publicMethod(param : int) : int
+ publicMethod() : double
SubClass
+ publicMethod() : string
Package
Class2
0..*
<<import>>
Class3

Figure A.2: Inheritance.

Figure A.2 shows how *inheritance* is indicated in class diagrams.

- The arrow with the dotted line indicates that the Class implements the AbstractClass. The in this report we say that a class implementing an abstract class implicitly implements all it's virtual methods.
- The arrow with the solid line indicates that the SubClass inherits from the Class. All of Class' public and protected attributes and methods are available to SubClass, and so is the protected (and public) attributes and methods.
- The open arrow (→) indicates that `Class` is associated with zero or more instances of `Class2`.

- The dotted line with the open arrow with `«import»` indicates that `Class2` imports `Class3`.
- Packages indicate “parts” of the system. I.e. `Class2` and `Class3` are part of a subsystem which is, to some extent, decoupled from other parts of the system.

A.2 Sequence Diagrams

This section contains a short description of how sequence diagrams in this should be read. A simple example is shown in Figure A.3.

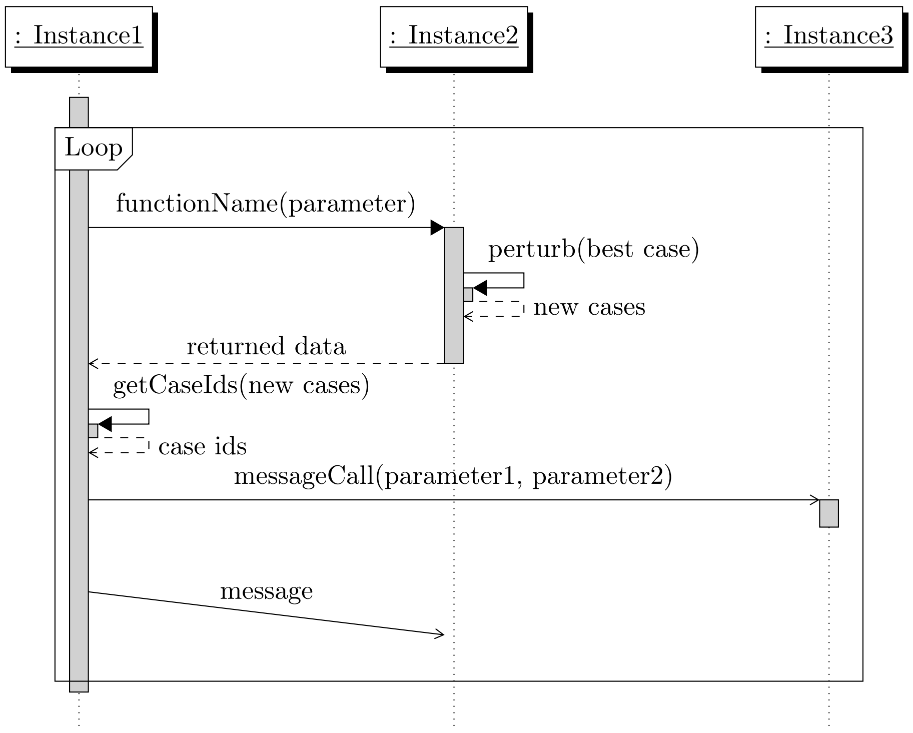


Figure A.3: A sequence diagram.

- The events describes in a sequence diagram increases from top to bottom.
- Object instances are indicated with their names on the top row, and dashed lines down from them.
- Solid blocks superimposed on the dashed lines down from objects indicate when they are active.

- Solid lines with solid arrows indicate synchronous messages (function/method calls).
- Dashed lines with open arrows indicate return messages.
- Solid, tilted lines with open arrows indicate asynchronous, non-instant messages (MPI messages).
- Loops are indicated with a frame and a label. The events inside a loop occur repeatedly until some termination condition is met.

[illegible]
Lagt inn i [illegible]

NTNU
Det skapende universitet

MASTERKONTRAKT
- uttak av masteroppgave

1. Studentens personalia

Etternavn, fornavn **Baumann, Einar Johan Moen**	Fødselsdato **16. jul 1990**
E-post **einarjba@stud.ntnu.no**	Telefon **92851325**

2. Studieopplysninger

Fakultet **Fakultet for ingeniørvitenskap og teknologi**	
Institutt **Institutt for bygg, anlegg og transport**	
Studieprogram **Master i ingeniørvitenskap og IKT**	Studieretning **Integrerte operasjoner i petroleumsindustrien**

3. Masteroppgave

Oppstartsdato **14. jan 2015**	Innleveringsfrist **10. jun 2015**
Oppgavens (foreløpige) tittel ***Extended Development of Software Framework for Petroleum Field Optimization*** **Support system for the integration of oil production problems with optimization methodology**	
Oppgavetekst/Problembeskrivelse This project focuses primarily on software development and is part of the new IPT/ITK Petroleum Cybernetics initiative at NTNU. This initiative aims at solving petroleum engineering problems through extensive use of optimization methodology from the field of control theory. A crucial component for the realization of this initiative is the development of a (software) support platform that can facilitate the articulation of oil production problems for optimization purposes. Importantly, this support framework will also include optimization algorithms that can find improved operation and design configurations to increase oil recovery in the set production problems. Tentatively, the future use of this framework in Msc and PhD coursework will further serve to integrate the areas of petroleum engineering (in particular reservoir management) and control systemsengineering at NTNU IPT and ITK departments. The software development planned in this master thesis consists of expanding the ResOpt application (discontinued Spring 2014) originally implemented by NTNU/IO-Center Postdoc Aleksander Juell. ResOpt is an optimization framework for integrated production networks in the petroleum ...	
Hovedveileder ved institutt **Professor Jon Kleppe**	Medveileder(e) ved institutt **Mathias Bellout**
Merknader **1 uke ekstra p.g.a påske.**	

4. Underskrift

Student: Jeg erklærer herved at jeg har satt meg inn i gjeldende bestemmelser for mastergradsstudiet og at jeg oppfyller kravene for adgang til å påbegynne oppgaven, herunder eventuelle praksiskrav.

Partene er gjort kjent med avtalens vilkår, samt kapitlene i studiehåndboken om generelle regler og aktuell studieplan for masterstudiet.

Trondheim, 14.01.2014
Sted og dato

[illegible]
Student

[illegible]
Hovedveileder

Originalen lagres i NTNUs elektroniske arkiv. Kopi av avtalen sendes til instituttet og studenten.