Parallelization of Ensemble Kalman Filter (EnKF) for Oil Reservoirs

Md. Khairullah
Parallelization of Ensemble Kalman Filter (EnKF) for Oil Reservoirs

Master’s Thesis in
Computer Simulations for Science and Engineering (COSSE)

Delft Institute of Applied Mathematics
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
The Netherlands

Md. Khairullah
Student id: 4187326

20th June 2012
Author
Md. Khairullah

Title
Parallelization of Ensemble Kalman Filter (EnKF) for Oil Reservoirs

MSc presentation
July 05, 2012

Graduation Committee
Prof. dr. ir. A.W. Heemink (chair) Delft University of Technology
Prof. dr. ir. C. Vuik Delft University of Technology
Prof. dr. ir. H.X. Lin Delft University of Technology
dr. R.G. Hanea Delft University of Technology
dr. ir. Harald Köstler Friedrich-Alexander University
Abstract

This thesis describes the design and implementation of a parallel algorithm for data assimilation with ensemble Kalman filter (EnKF) for oil reservoir management. The implemented application works on large number of observations from time-lapse seismic, which lead to a large turnaround time for the analysis step, in addition to the time consuming simulations of the realizations. Provided that parallel resources are used for the parallel simulations of the realizations, the analysis step also deserves parallelization. Our experiments show that parallelization of the analysis step in addition to the forecast step also scales well, exploiting the same set of resources with some additional efforts.
Preface

The work lying before you is my Master of Science thesis, in which a description is given of the work done and the results achieved since September 2011 at the Delft Institute of Applied Mathematics (DIAM), Faculty of EEMCS, Delft University of Technology. Having my bachelor degree in Computer Science and Engineering, I feel lucky enough to apply my achieved programming and analytical ability to some applied and practical topic of the real world in my current studies. I was very fond of parallel computing after I learnt Socket Programming in Java during my undergrad studies. It was a nice opportunity for me to work on parallelization of data assimilation for reservoir management.

Before continuing with my thesis, I would like to thank some people who have been of great help during my Masters project. First of all, I would like to thank my supervisors Dr. Lin and Dr. Hanea for giving me their valuable time for answering all my stupid and wise questions related to the project and also for their comments, suggestions and supports at different stages of the project. I want to thank the graduation committee members for their collective and individual contributions. I would like to thank Dr. J. D. Jansen for his support at the very beginning of the project to teach me some basic things of the simsim simulator. I would also thank ir. C.W.J. Lemmens for his technical supports. I would like to thank my family members for their patience and support for my life in abroad. Special thanks go to Dr. Harald Köstler, FAU, Erlangen for his supports during my Erlangen life. Last but not the least, I express my deep gratitude to European Union for funding my studies through Erasmus Mundus Scholarship.

Md. Khairullah

Delft, The Netherlands
20th June 2012
# Contents

**Preface** .................................................. v

1 Introduction .............................................. 1

2 Parallel Data Assimilation with EnKF .................. 5
   2.1 Problem settings .................................. 5
   2.2 Previous works .................................. 6
   2.3 Proposed parallel algorithm ....................... 7
   2.4 Parallel performance ................................ 11

3 Basics of Reservoir Engineering ....................... 15
   3.1 Definitions ....................................... 15
   3.2 Recovery processes ................................ 16
   3.3 Different data sets................................ 19
      3.3.1 Production history data ...................... 19
      3.3.2 Time-lapse seismic data ..................... 20
      3.3.3 Prior knowledge ................................ 20

4 Data Assimilation with EnKF ............................ 21
   4.1 Review of the Kalman Filter ....................... 22
      4.1.1 Variance minimizing analysis scheme ............ 22
      4.1.2 Kalman filter ................................ 23
   4.2 Ensemble Kalman filter ............................ 23
      4.2.1 Representation of error statistics ............... 24
      4.2.2 Analysis scheme ................................ 24
   4.3 Practical implementation ......................... 26
      4.3.1 Ensemble representation of the covariance ....... 26
      4.3.2 Measurement perturbations ...................... 27
      4.3.3 Analysis equation ............................. 27
      4.3.4 EnKF for combined parameter and state estimation 29

5 Parallel Computing with MPI and ScaLAPACK .......... 33
   5.1 Parallel computing ................................ 33
   5.2 MPI .................................................. 35
5.3 Parallel numerical libraries for SVD .......................... 36
  5.3.1 ScaLAPACK ........................................ 37

6 Implementation .............................................. 43
  6.1 Reservoir model ......................................... 43
  6.2 Software structure ...................................... 45
  6.3 Optimization ............................................ 47
    6.3.1 Load balancing ..................................... 47
    6.3.2 Minimizing data communication ...................... 48

7 Results and Discussions .................................... 53
  7.1 Verification ............................................ 53
  7.2 Performance of the parallel algorithm ............... 54
    7.2.1 Difference in speedup: forecast vs analysis step .... 59
    7.2.2 Super-linear speedup ................................ 64
    7.2.3 Parallel scalability ................................ 64
    7.2.4 Difference in performance: 48 realizations vs 96 realizations 65
    7.2.5 Difference in speedup and performance: LAN cluster vs SARA Lisa cluster .... 65
    7.2.6 Effect of ScaLAPACK process grid .................. 65
    7.2.7 Effect of ScaLAPACK block size .................... 69
    7.2.8 Effect of non blocking communication .............. 71

8 Conclusion and Future Work .............................. 75
  8.1 Conclusion ............................................... 75
  8.2 Future work ............................................. 77
List of Figures

1.1 The forecast of global demand of energies [2] .................................................. 2
1.2 The forecast of global production of energies [2] ............................................... 2

2.1 Typical logical domain decomposition for the forecast step (top) and for the analysis step (bottom). ................................................................. 8
2.2 An example of physical domain decomposition used for the ocean circulation model in [21] ................................................................. 8
2.3 Serial implementation of the EnKF for data assimilation ........................................... 9
2.4 1st level parallelization of the simulator ............................................................... 9
2.5 2nd level parallelization of the simulator ............................................................ 10
2.6 A multi level parallelization .............................................................................. 10
2.7 Proposed parallel EnKF for data assimilation ....................................................... 11

3.1 Reservoir management depicted as a closed loop model-based controlled process [19] ................................................................. 19

4.1 Schematic of how data assimilation (DA) works and adds value to observational and model information. The data shown are various representations of ozone data of a particular day [22] ........................................ 22
4.2 The ongoing discrete Kalman filter cycle. The time update projects the current state estimate ahead in time. The measurement update adjusts the projected estimate by an actual measurement at that time. ........ 24
4.3 The procedure of data assimilation with EnKF for the ensemble member j. ................................................................. 26

5.1 Shared memory architecture ............................................................................ 34
5.2 Distributed memory architecture: MPI’s work place .............................................. 35
5.3 ScaLAPACK Software Structure ....................................................................... 38
5.4 Data distribution examples: column blocked (left) and column cyclic (right) .................. 40
5.5 Data distribution examples: column blocked cyclic (left) and 2D blocked cyclic ................................................................. 41
5.6 An example of view of data distribution in ScaLAPACK [3] ................................... 42

6.1 Location of the wells in the model reservoir field ................................................... 44
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>rms differences between the true and assimilated log permeabilities at different time steps</td>
<td>54</td>
</tr>
<tr>
<td>7.2</td>
<td>Hardware specification of the LAN cluster for initial tests and measurements</td>
<td>55</td>
</tr>
<tr>
<td>7.3</td>
<td>Performance of the parallel implementation for 48 realizations in the LAN cluster</td>
<td>59</td>
</tr>
<tr>
<td>7.4</td>
<td>Performance of the parallel implementation for 48 realizations in the SARA Lisa cluster</td>
<td>59</td>
</tr>
<tr>
<td>7.5</td>
<td>Performance of the parallel implementation for 96 realizations in the LAN cluster</td>
<td>60</td>
</tr>
<tr>
<td>7.6</td>
<td>Performance of the parallel implementation for 96 realizations in the SARA Lisa cluster</td>
<td>61</td>
</tr>
<tr>
<td>7.7</td>
<td>Execution time (in minute) of the analysis step for different process grid orientation for 48 realizations</td>
<td>69</td>
</tr>
<tr>
<td>7.8</td>
<td>Execution time (in minute) of the analysis step for different block sizes for 48 realizations</td>
<td>71</td>
</tr>
<tr>
<td>7.9</td>
<td>Execution time (in minutes) of the analysis step for blocking and non-blocking data communication</td>
<td>73</td>
</tr>
</tbody>
</table>
# List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A single step of basic EnKF</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>A single step of basic parallel EnKF</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>A single step of the optimized parallel EnKF</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td>A single step of the final parallel EnKF</td>
<td>51</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Because of rapidly increasing population of the world and the necessary industrialization, the natural resources, such as petroleum, are becoming scarce day by day. For more than 150 years after the discovery, oil continues to play an essential role for the mankind and in the global economy. At the same time, renewable energy becomes more important. Its production is less harmful to the environment and starts to become economically competitive compared to the production of energy from fossil fuels. Though the use of renewable fuels is expected to grow strongly over the coming years, still oil and natural gas will remain the main energy sources until at least 2050.

Both history and the future projections show that demand for oil is a crying need for the world. Since the energy shocks of the 1970s and 1980s [1], which proved extreme dependency of the developed world on petroleum products and also the extreme vulnerability of its shortfalls in supplies, oil remains the top source of energy, though it has fallen off its position of high adoration. The figures 1.1 and 1.2 from EIA give the forecast of demand and production of oil and other sources for the next 20 years, showing clear deficiency of the production to meet up the demand, which infers the urgency of promoting the production of petroleum reservoirs. The exploration and production industry is struggling to keep up with this demand. Oil companies find it each year more and more difficult to meet this huge demand for fossil fuels. First, a lot of large fields are already at a mature stage. Other fields, discovered recently, are often too small to be exploited efficiently. It is therefore essential to develop new technologies that allow for reduction of costs of maintaining oil fields and increase the rate of recovery of oil from the existing fields. A sound reservoir management is a key requirement to increase the production from petroleum reservoirs.
Data assimilation is an important tool for closed loop oil reservoir management. The ensemble Kalman filter (EnKF) has gained huge popularity for data assimilation for non linear systems in the recent years. However, EnKF takes large turn around time in terms of computing, particularly to run the simulations of hundreds of realizations. The situation worsens when we have large number of observations. Also the analysis step becomes time consuming for large number of observations. But fortunately enough, both the forecasting and the analysis part of EnKF have inherent parallelism.

We implement a parallel EnKF for the data assimilation for oil reservoirs and
present the experimental results. The results show that by parallelization of the whole EnKF process, significant speed up can be achieved utilizing the same set of parallel hardware, which are used for the forecast step only or the forecast step and partially for the analysis step in some previous works.

The remainder of the thesis contains 7 chapters and are organized as follows. In chapter 2, we describe the problem setting and the corresponding survey of previous works, and also the formulation of a new parallel algorithm for data assimilation with EnKF. In chapter 3, we study some necessary basics and foundations for modern reservoir engineering in order to be familiar with some common terminologies. Chapter 4 describes a general background and overview of data assimilation and ensemble Kalman filter (EnKF). Chapter 5 describes the parallel computing tools MPI and ScaLAPACK which we use for the parallel EnKF. In Chapter 6 we describe the detailed implementation of the parallel EnKF. In Chapter 7 we describe the experimental results and correctness of the implemented parallel EnKF as well as performance and efficiency of the implemented EnKF. Chapter 8 includes the concluding remarks, and also some recommendations for potential future research directions.
Chapter 2

Parallel Data Assimilation with EnKF

In this chapter we define our problem settings. Then we go through a brief discussion on previous works in this topic. Then we present a parallel algorithm for data assimilation with EnKF for oil reservoirs. Finally, we briefly discuss some metrics to measure the benefits of parallel applications, which we will use in chapter 7.

2.1 Problem settings

Data assimilation is an important component of modern reservoir management systems. The Kalman filter, a nice data assimilation algorithm, uses a series of measurements observed over time, containing noise and other inaccuracies, and produces estimates of unknown variables that tend to be more precise than those that would be based on a single measurement alone. However, Kalman filter is an optimal filter for linear systems and needs derivation of tangent linear operator to work with non linear systems, which is impossible or very compute intensive for some systems. The ensemble Kalman filter (EnKF), a suboptimal approximation of Kalman filter, is a recursive filter suitable for problems such as geophysical models with a large number of variables. EnKF consists of two major parts:

1. **forecast or simulation:** A group or ensemble of realizations/models of the system run independently, generating an ensemble of predictions or state vectors of the system. This step is computing intensive in systems with both small and large number of observations.

2. **analysis or update:** After completion of the forecast/simulation, the ensemble of the predicted state vectors are gathered and compared with available measurements in hand. Assuming Gaussian noises both in the measurements and the model forecasts, some computations are done to decide the weights of the forecast and measurements to include in the updated prediction with better approximation.
The forecast step is embarrassingly parallel as each ensemble member can run independently in separate processing element of computers. Hence most parallel implementations solely depend on this step only. The analysis or update step is critical for only systems with large number of observations and parallel implementation of this step is not simple because of the intermediate inter process communication requirement. Most of the implementations of this step depend on some sort of domain decomposition to avoid communicating large volume of data. Our goal is to develop a parallel EnKF for oil reservoirs with the whole domain.

2.2 Previous works

In [20] data assimilation experiments are performed using a parallel ensemble Kalman filter (EnKF) implemented for a two layer shallow water model where advantage is taken of the inherent parallelism in the EnKF by running each ensemble member on a different processor of a parallel computing system. The Kalman filter analysis step is parallelized by letting each processor handle the observations from a different division of the whole domain. Figure 2.1 depicts the logical domain decomposition used in [20], also a common decomposition technique for any parallel EnKF. Each shade of different colors in the figures represents the state vector of a different ensemble member. In the top picture each bounded cell represents a processor. The alphabetical letters in a cell represents division of the state vector based on some physical decomposition of the domain. The ensemble members (six in this example) are distributed one per processor. In the bottom picture the domain decomposition used for the analysis step is shown. The memory of each processor contains the same state-vector elements (of the same physical division of the domain; from one of the six in this example) from each ensemble member. In [21] a multivariate ensemble Kalman filter (MvEnKF), implemented on a massively parallel computer architecture and developed for the ocean circulation model, has been presented. Parallelism for the analysis step is achieved by regionalization of the error covariances of the ensemble. Each processing element (PE) collects elements of a matrix measurement functional from nearby PEs. In figure 2.2 the concept of physical domain decomposition is depicted. This is used for the analysis step for the ocean circulation model in [21]. The outer rectangle delimits the area, A, from which the data assimilated on one processor are collected. The innermost rectangle depicts the boundary of the processor’s private area, B. The ellipse delimits the influence region of the processor private area’s south-eastern corner cell. The shaded area contains the ellipses for all grid cells contained in B. The region C, popularly known as the ‘halo’ region, contains the regions which must be influenced and overlapped between adjacent processors and be updated after all private regions of each processor have been updated. The authors implemented domain decomposition to avoid allocating and loading large state vectors from each processor on the basis of the fact that observations from a distant location should have insignificant affect on the covariances compared to the observations from a nearby
region. However with the advent of computer memory technology, allocating large state vectors on each processing element is not be a big concern at present or in the near future.

In [23] a parallel implementation of weighted EnKF for oil reservoir is presented which parallelize each ensemble member in a number of processors and this is described as two levels of parallelism. In [29] a multilevel parallization for oil reservoir simulation by EnKF is proposed. The first level of parallelization is usually during the forecast step, where each ensemble member runs on a separate processor. A second level of parallelization which uses a parallel reservoir simulator for each realization is implemented. It proposes an algorithm which partially parallelize the analysis step and describes it as the third level of parallelization. The main computational gain of parallelization of the analysis step comes from the fact that the matrix-vector multiplications can be parallelized efficiently. Figure 2.3 depicts the work flow of a serial implementation of the method whereas figures 2.4, 2.5 and 2.6 show the simplified work flows of the parallel implementations done so far. In these examples the EnKF method works with $N$ realizations. In figure 2.3 all realizations and the analysis step are executed on a single processor $P_1$. In figure 2.4 each realization runs on a different processors, requiring total $N$ processors, namely $P_1, P_2, \ldots, P_N$ and the analysis step runs on the single processor $P_1$. In figure 2.5 each realization runs on three processors, involving a total $3N$ processors, namely $P_1, P_2, \ldots, P_{3N}$ and the analysis step runs on the single processor $P_1$. In 2.6 each realization runs on three processors, involving a total $3N$ processors, namely $P_1, P_2, \ldots, P_{3N}$ and the analysis step is partially parallelized on $N$ processors, the so-called master processors.

2.3 Proposed parallel algorithm

In detailed discussion of EnKF in chapter 4 we will see that the EnKF update is expressed as a matrix multiplication

$$A^u = AX$$

(2.1)

where the matrix $A$ is the ensemble of the forecasted state vectors, $A^u$ is the ensemble of the updated state vectors and $X$, let us call it the updating coefficient matrix, is computed in the analysis step based on the forecast and the measurements, to minimize the variances of the state vectors among the ensemble members. So the EnKF analysis step consists of construction of the update coefficient matrix $X$, and the matrix multiplication of $A$ and $X$ in (2.1). It is obvious that in the above mentioned parallel algorithms, the first two levels of parallelization work for the construction of the matrix $A$ and the third level is used only for the matrix-matrix multiplication in (2.1).

But the construction of the matrix $X$ in (2.1) would be very compute intensive for large number of observations. We will see in later chapters that the construction of $X$ involves a series of singular value decomposition (SVD) compu-
Figure 2.1: Typical logical domain decomposition for the forecast step (top) and for the analysis step (bottom)

Figure 2.2: An example of physical domain decomposition used for the ocean circulation model in [21]
Figure 2.3: Serial implementation of the EnKF for data assimilation

Figure 2.4: 1st level parallelization of the simulator
Compute the updating coefficient matrix
Update on $P_1$
Update on $P_4$
...
...
Update on $P_{3N-5}$
Update on $P_{3N-2}$
$J=1$
$J=2$
$J=N-1$
$J=N$

$P_1$ $P_2$ $P_3$
$P_4$ $P_5$ $P_6$
...
...
$P_{3N-5}$ $P_{3N-3}$
$P_{3N-2}$ $P_{3N}$

Figure 2.5: 2nd level parallelization of the simulator

Figure 2.6: A multi level parallelization
Figure 2.7: Proposed parallel EnKF for data assimilation

tation and matrix-matrix multiplications and these two operations can be carried out efficiently in parallel computers due to availability of nice parallel algorithms and corresponding libraries. Hence also the construction of $X$ can be parallelized and significant reduction in assimilation time can be obtained. However, there are techniques to compute $X$ with less computations, based on different assumptions. Provided parallel computing resources for running the simulations in parallel, we can exploit them for the large computations for the analysis steps to get exact results, instead of depending on the assumptions. As we said, the matrix $X$ is termed as the updating coefficient matrix in the flowcharts. And by the statement ‘update’ in the flowcharts we mean the matrix-matrix multiplication in (2.1).

2.4 Parallel performance

A measurement is needed to express the performance gain achieved by parallelising a given application over a sequential implementation. Here we list some common parallel performance metrics and related terminologies [15].

- **execution time:** The time elapsed between the beginning and end of execution on a sequential computer is called the serial runtime and usually is denoted by $T_S$. The time elapsed from the start of the parallel computation to the end of execution by the last processing element (PE) is called the parallel runtime and usually is denoted by $T_P$. These are not practically used as
metrics.

• **speedup** ($S_P$): It demonstrates the relative benefit of solving a problem in parallel and is calculated by taking the ratio of the serial runtime $T_S$ of the best sequential algorithm to the time $T_P$ taken to solve the same problem with $P$ processors:

$$S = \frac{T_S}{T_P} \quad (2.2)$$

This is the mostly used metric for parallel performance. Some people, for simplicity, defines the speedup by the ratio of the single processor runtime $T_1$ of the parallel application to the $P$ processor run time $T_P$:

$$S = \frac{T_1}{T_P} \quad (2.3)$$

The maximum speedup theoretically achievable is $P$ (linear speedup), although super-linear speedup might occur in practise due to a non-optimal sequential algorithm or hardware characteristics. Cache memory is a very fast memory in the computer memory hierarchy. When the collective cache memory of some processors can hold the required data of the application which is impossible for a single or less number of processors, super-linear speedup can be observed due to the cache memory performance, in addition to the performance due to more number of processors [16]. However, the well known Amdahl’s law states an important general limitation on the speedup of parallel application with increasing number of processors [6]. If $N$ is the number of processors, $s$ is the amount of time spent (by a serial processor) on serial parts of a program and $p$ is the amount of time spent (by a serial processor) on parts of the program that can be done in parallel, then Amdahl’s law says that speedup achieved by parallel processing is given by

$$Speedup = \frac{s + p}{s + p/N} = \frac{1}{s + p/N}, \quad (2.4)$$

where we have set total time $s + p = 1$ for algebraic simplicity. When $s \neq 0$, the speedup decreases with increase in $N$, the number of processors. However, Gustafson’s law, a counter point of Amdahl’s law, states that with increased number of processors a larger problem can be solved with the same efficiency [17]. Gustafson proposes a new metric ‘scaled speedup’ in this context.

• **cost** (C): It is the sum of the times that all processors spend solving the problem in a parallel system. Cost is used for comparing the performance of parallel algorithms to their sequential counterparts, and is defined by

$$C = PT_P \quad (2.5)$$

12
• **efficiency** \((E_P)\): It is another performance metric, closely related to speedup, which is the fraction of the time for which each processor is utilized usefully.

\[
E = \frac{S}{P} = \frac{P_s}{PT_P} = \frac{T_S}{C}
\]  

This is mostly used in conjunction with speedup.

• **parallel scalability:** A common task in parallel computing is measuring the scalability of an application. This measurement indicates how efficient an application is when using increasing numbers of parallel processing elements. **Strong scaling** is defined as how the solution time varies with the number of processors for a fixed total problem size. On the other hand, **weak scaling** is defined as how the solution time varies with the number of processors for a fixed problem size per processor.

• **overhead:** The performance depends on the overhead inflicted on a parallel algorithm during execution. The overhead of the parallel algorithm is the difference between its cost and the runtime of the fastest known serial algorithm for solving the same problem.

\[
T_O = C - T_S = PT_P - T_S
\]

Some major sources that cause overhead inevitably in a parallel computing are:

1. **communication overheads:** caused by the time taken to exchange information between processors, which obviously does not occur in sequential algorithms.

2. **synchronisation overheads:** caused by idling processors, because there is no work in the work queue or a processor has to wait at a synchronisation point. The most important factor in synchronisation overhead is the load imbalance.

• **granularity:** It is the number and size of basic tasks into which a problem is decomposed. It therefore bounds the maximum level of concurrency and is of great influence to the overhead factors. A decomposition into a large number of small tasks is called fine-grained and a decomposition into a small number of large tasks is called coarse-grained. Generally, coarser-grained reduces communication overhead, but increases the synchronisation overhead.
Chapter 3
Basics of Reservoir Engineering

In this chapter we describe some very fundamental concepts of reservoir engineering, which are essential for discussion of any reservoir application, and introduce some common terminologies used in reservoir engineering, and also in this thesis. We start with some formal definitions of reservoir engineering and then briefly describe the recovery processes and conclude by describing the data sets used in data assimilation.

3.1 Definitions

Reservoir engineering grew out of the recognition that recovery of oil and gas from reservoirs could be made more predictable (and actually could be increased), if the reservoir were analyzed and well managed. Reservoir engineering started in the 1930s with some milestone research works concerning measurement of fundamental rock properties including porosity and permeability; measuring fluid properties with subsurface samples of oil and gas; and energies driving oil and gas out of the reservoirs [31].

Petroleum reservoir management is the application of state-of-the-art technologies to a known reservoir system within a given management environment. Reservoir management can be thought of as a set of operations and decisions by which a reservoir is identified, measured, produced, developed, monitored and evaluated from its discovery through depletion and final abandonment [35].

Sound reservoir management practices rely on the utilization of available human, technological and financial resources to maximize profits from a reservoir by optimizing recovery while minimizing capital investments and operating expenses. “Reservoir management involves making certain choices: either let it happen, or make it happen” [27].

Though before 1970, reservoir engineering was strongly believed to be the major technical importance in reservoir management, in the 1970s and 1980s the collaboration between geo-scientists and reservoir engineers proved to be very successful. Then the value of detailed reservoir description with geological, geophysical
and reservoir simulation techniques was emphasized [30].

By reservoir simulation, we mean the process of inferring the behaviour of a real reservoir from the performance of a model of that reservoir. The model may be physical, such as a scaled laboratory model, or mathematical [26].

3.2 Recovery processes

Sandstones and limestones are two general sources of hydrocarbon. Oil is trapped in small microscopic pores (the empty spaces among the sand grains) in the rock. A reservoir is well described by its rock and fluid properties. Rock properties are generally static and are called parameters in common practice. Several properties of the porous rock are important for hydrocarbon extraction. Two most important properties are:

1. **porosity**: the fraction of the rock that can be occupied by fluids, common symbol is $\phi$, has no unit.

2. **permeability**: the ability of the rock to transmit fluids, common symbol is $k$ and unit is Darcy ($1 \text{ Darcy} = 10^{-12} \text{m}^2$).

Fluid properties of a reservoir are generally dynamic and depend on the rock properties and the reservoir management. They are called state variables in common practice. Two mostly used state variables are:

1. **pressure**: the pressure of fluids within the pores of a reservoir, the common symbol is $P$ and the unit is pascal or Newton per square meter ($N/m^2$).

2. **saturation**: the ratio of the volume that a fluid occupies to the pore volume, has the symbols $S_w$, $S_o$ etc. meaning water saturation and hydrocarbon saturation respectively. For a two phase flow consisting of only hydrocarbon and water, we have

$$S_w + S_o = 1 \quad (3.1)$$

Being a ratio of two volumes, saturation has no unit.

If the pores are connected such that the fluids can easily flow through these linked pore paths, the rock is called permeable. If hydrocarbons are present in the pores of a reservoir rock, they must be able to move out of them. Hydrocarbons remain locked in place unless they can move from pore to pore. The operating company considers a reservoir suitable, economical and feasible to drill and produce if the rock is porous, permeable, and off course contain enough hydrocarbons. The properties of reservoir rock have a big influence on the ability of the fluids to flow through the reservoir and often determine the strategy used during hydrocarbon recovery.

The hydrocarbon recovery process can be divided into three phases in general.
1. **primary recovery:** In this phase the pressure in the reservoir is often high enough to force hydrocarbon or gas to flow through the porous medium and eventually out of the well. Extraction of hydrocarbon causes a decrease of the pressure and subsequently a decrease of the recovery rates. At some point in time, when recovery rates become too small, the economic production can only be maintained after applying special techniques. Very often the primary recovery leaves 70-80% or more of hydrocarbons still in the reservoir.

2. **secondary recovery:** It consists of injecting gas or water into the reservoir to increase the pressure and to push the remaining hydrocarbon out of the pores. Usually an injection occurs through the wells that are located at some distance from the production wells. The injected fluid should then displace some of the hydrocarbon and push it towards production wells. The secondary recovery phase increases the recovery factor, but still 50% or more hydrocarbons remain in the reservoir.

3. **tertiary recovery or enhanced hydrocarbon recovery (EOR):** In this phase the fluid properties are altered to ease its flow. Two tactics are mainly used. In thermal recovery steam is injected into the reservoir to heat the hydrocarbon to increase its fluidity. In chemical recovery several different chemicals like polymers or surfactants are used to reduce the oil viscosity, or increase the viscosity of the injected water, or lower the capillary pressure that stops oil droplets from flowing through a reservoir.

All these difficulties are result of the spatial heterogeneity of reservoir rock properties. The existence of preferable paths, through which injected fluid is moving toward the production well, are very common. Hydrocarbons located outside this path is not influenced by the injection of fluids. This causes the production of injected fluid instead of hydrocarbons at an early stage. Having the knowledge of the spatial distribution of rock properties, one would be able to design the production strategy to postpone water breakthrough in the wells and maximize the recovery. However, the spatial heterogeneity and lack of direct measurements of rock properties, which are only known in well locations, introduces a lot of uncertainties that need to be addressed, if reliable future predictions of reservoir performance are to be expected.

After the exploration phase, in which potential reservoirs are identified and exploration wells are drilled, initial geological models are created based on the knowledge gathered from seismic surveys and well data. Initial predictions for the future reservoir performance are made, and if those predictions are financially profitable the reservoir enters a field development phase. When developing a field, the main target is to maximize the economic criterion, most often in terms of oil revenues. Choices are made about the number and locations of wells, the surface facilities that need to be built and the required infrastructure.

A detailed geological model of a given reservoir is derived based on all available information and an ‘upscaled’ simpler version is used for flow simulation in
practice. This numerical reservoir model should ideally mimic all the processes occurring in the reservoir. If the numerical model would adequately describe the real reservoir, it would be possible to predict the reservoir behavior properly and plan the best strategy to maximize the recovery from a given field. Unfortunately, a numerical reservoir model is only a crude approximation to the truth, mainly for two reasons. First, not all the processes occurring in a real reservoir can be modelled in an appropriate way. Very often some simplifications are imposed on the model, to make the problem easier to tackle. Second, there is usually a large uncertainty in the parameter values of the simulation model. Many rock properties, that influence reservoir flow, are poorly known, while there are also uncertainties in fluid properties and the amount of hydrocarbons present in the reservoir. The uncertainties involve the reservoir structure, the initial fluid contacts, and the values of permeabilities, porosities, and fault transmissibilities, etc. These reservoir-related parameters are presumed to be known in numerical simulations. However, neglecting the uncertainties lead to contradictions between the results produced by numerical reservoir models and the data gathered from the real field. It’s then difficult to make decisions based only on the output of a numerical model. Therefore, the measured data together with numerical simulations should be used in reservoir management for improving the production rates and increasing the recovery from a field.

The new concept of a closed-loop model-based reservoir management, depicted in figure 3.1 seems to be a proper framework for reservoir monitoring and management [19]. It involves the use of (uncertain) reservoir and production models and combines them with available data from a real field, such as production history data or time-lapse seismic data, to continuously update the numerical models and reduce the uncertainties. The system, at the top of the figure represents reality: the reservoir with wells and other facilities. This system is monitored with the help of various kinds of sensors, which give knowledge about the state of the reservoir, such as bottom hole pressures (BHP), oil and water flow rates, in the form of measured outputs. However, the information coming from these measurements is not perfect. Also the inputs into the system can be uncertain and be subject to noise. When production begins and data become available, it is possible to update the numerical reservoir and sensor models those simulate the behavior of the real reservoir. The gathered data are used to reduce the uncertainties of the parameters through identification and update in the data assimilation loop. The updated reservoir models are then used in an optimization loop where a new optimized production strategy can be determined. In the figure, we see that data assimilation is a vital part of reservoir management work loop. Our goal is to enhance the performance of the data assimilation process by utilizing it’s inherent parallelism.
3.3 Different data sets

Data available from different sources can be used in the data assimilation loop to improve the characterization of the reservoir and the reliability of the flow predictions. Generally, data can be divided into two classes.

1. **sparse data**: available only at well locations

2. **dense data**: gathered everywhere in the reservoir

These data sets differ in accuracy and field coverage, but both contain valuable information about the reservoir and the parameters influencing the flow, and should be used together to obtain a better reservoir model.

3.3.1 Production history data

Usually production history data, obtained from wells in the form of bottom hole pressures and flow rates, are used in history matching algorithms to update the uncertain parameters. This type of data is typically acquired with an accuracy between 5%-20%. However, because the number of model parameters to be estimated is very large, the production history data has a limited resolving power. It provides some information on the unknown properties in the neighborhood of the wells, but not further away from them.
3.3.2 Time-lapse seismic data

Due to the developments in geophysics, especially in the field of seismic, it is possible to determine not only the position of the reservoir, but also to track the fluids movements in the reservoir itself. This additional information in the form of time-lapse seismic data can be utilized, together with production data, to narrow the solution space when minimizing the mismatch between gathered measurements and their forecasts from the numerical model [28]. Time-lapse seismic is the process of carrying out a seismic survey before the production begins and then repeating surveys over the producing reservoir. Seismic data is sensitive to static properties like lithology and pore volume; and also to dynamic (i.e. time varying) properties like fluid saturation and pore pressure. Sensors yield seismic data in the form of travel time information and amplitudes. These seismic attributes undergo an inversion process like in [32], in which they are translated into time-lapse changes in density, acoustic propagation velocity and shear modulus. Time-lapse seismic data are available everywhere in the reservoir. Although less accurate than production data, they contain information about the reservoir properties everywhere and can be used to infer parameter values away from wells.

3.3.3 Prior knowledge

History matching is an ill-posed problem i.e. many different parameter sets may result in the same measurement predictions. In petroleum engineering, due to the complexity and high dimensionality of the models, and due to the lack of direct measurements of the unknown parameters, the prior knowledge is usually described by an ensemble of models that all conform to the geological information. Most of the available data assimilation schemes take into account only the mean value and the covariance matrix derived from the ensemble, indirectly assuming a Gaussian distribution for the unknown parameters.
Chapter 4

Data Assimilation with EnKF

In this chapter we go through the foundation of the data assimilation concepts. We start with reviewing the Kalman filter. Later on we present the ensemble Kalman filter (EnKF) and finish this chapter with the practical implementation details of the EnKF.

For any dynamic system, we have two broad sources of information:

1. measurements of the system or observations

2. understanding of the temporal and spatial evolution of the system or models

Mathematics provides a foundation to address questions such as: “What combination of the observation and model information is optimal?”, and provides an estimate of the errors of the “optimum” or “best” estimate \[22\]. This is known as “data assimilation”.

Consider a dynamical model along with initial and boundary conditions and a set of measurements that can be related to the model state. Then the state estimation problem is defined as finding the estimate of the model state that best fits the model equations, the initial and boundary conditions, and the observed data in some weighted measure. The problem may become overdetermined and consequently no general solution exists, unless we relax the equations and allow some or all of the dynamical model, the conditions, and the measurements to contain errors. We often assume Gaussian distributions for the error terms. We also assume that errors in the measurements are uncorrelated with errors in the dynamical model. The problem is then formulated as a quadratic cost function whose minimum defines the best estimate of the state. Fig 4.1 shows the relation among the participants of a data assimilation process.
4.1 Review of the Kalman Filter

4.1.1 Variance minimizing analysis scheme

The Kalman Filter (KF), a variance minimizing algorithm, updates the state estimate whenever measurements are available. Let us assume that Gaussian noise is present both in the model prediction and the measured data, and that no correlation is present between these noises. Let a vector of variables stored in $\psi(x, t)$ be defined on some spatial domain $D$ with spatial coordinate $x$. When $\psi(x, t)$ is discretized on a numerical grid, representing the spatial model domain, it can be represented by the state vector $\psi_k$ at each time instant $t_k$. Let new measurements are available at the time $t_k$. Then we define the new estimate for the state vector as

$$\psi_k^a = \psi_k^f + K_k (d_k - M_k \psi_k^f), \quad (4.1)$$

where $\psi_k^a$ and $\psi_k^f$ are the analysed and forecast estimates respectively, $d_k$ is the vector of measurements, $M_k$ is the measurement operator mapping the model state $\psi_k$ to the measurements $d_k$ and $K_k$ is the weighting factor or the Kalman gain.

Then the covariance of the analysed state is given by

$$(C_{\psi\psi})_k^a = (1 - K_k M_k)^2 (C_{\psi\psi})_k^f + K_k^2 (C_{ee})_k, \quad (4.2)$$

where $(C_{\psi\psi})_k^f$ is the error covariance of the predicted model state, and $(C_{ee})_k$ is the measurement error covariance matrix. An optimal value for $K_k$ can be found by choosing $K_k$ such that the covariance of $\psi_k^a$ is minimum. The solution is given by

$$K_k = (C_{\psi\psi})_k^f M_k^T (M_k (C_{\psi\psi})_k^f M_k^T + (C_{ee})_k)^{-1} \quad (4.3)$$
and the corresponding covariance is given by

\[(C_{\psi\psi})^a_k = (I - K_k M_k)(C_{\psi\psi})^f_k \] (4.4)

### 4.1.2 Kalman filter

We assume that the true state \(\psi^t\) evolves in time according to the dynamical model

\[\psi^f_k = F\psi^a_{k-1} + q_{k-1}, \] (4.5)

where \(F\) is a linear model operator and \(q_{k-1}\) is the unknown model error over one time step from \(k - 1\) to \(k\) and the superscript \(t\) is the acronym for true. In this case a numerical model evolves according to

\[\psi^f_k = F\psi^a_{k-1}, \] (4.6)

where the superscripts \(a\) and \(f\) denote analysis and forecast respectively. That is, given the best possible estimate (traditionally called analysis) for \(\psi\) at time \(t_{k-1}\), a forecast is calculated at time \(t_k\), using the approximate equation (4.6).

The error covariance equation is then derived by subtracting (4.6) from (4.5), squaring the result and taking expectation we have

\[C^f_{\psi\psi}(t_k) = FC^a_{\psi\psi}(t_{k-1})F^T + C_{qq}(t_{k-1}), \] (4.7)

where the error covariance matrices for the predicted and analysed estimates are defined as

\[C^f_{\psi\psi} = (\psi^f - \psi^t)(\psi^f - \psi^t)^T, \] (4.8)

\[C^a_{\psi\psi} = (\psi^a - \psi^t)(\psi^a - \psi^t)^T. \] (4.9)

Here the over line is an expectation operator, which is equivalent to averaging over an ensemble of infinite size.

Figure [4.2] depicts the working principle of data assimilation by Kalman filter.

### 4.2 Ensemble Kalman filter

While the classical Kalman filter provides a complete and rigorous solution for state estimation of linear systems under Gaussian noise, the estimation problem for non-linear systems remains a problem. Results of rigorous solutions to the non-linear problem are either too narrow in applicability or are computationally expensive [14]. The ensemble Kalman filter is a Monte Carlo [34] approach for approximating Kalman filter to overcome the above mentioned limitations.
Figure 4.2: The ongoing discrete Kalman filter cycle. The time update projects the current state estimate ahead in time. The measurement update adjusts the projected estimate by an actual measurement at that time.

4.2.1 Representation of error statistics

The error covariance matrices $C_f$ and $C_a$ for the predicted and analysed estimate in the Kalman filter are defined in terms of the true state in (4.8) and (4.9). However, since the true state is never known, we redefine the ensemble covariance matrices in terms of the ensemble mean $\bar{\psi}$ instead of the true state, according to

\[
(C^{e_f}_{\psi\psi})_k = (\bar{\psi}_f - \bar{\psi})(\bar{\psi}_f - \bar{\psi})^T,
\]

(4.10)

\[
(C^{e_a}_{\psi\psi})_k = (\bar{\psi}_a - \bar{\psi})(\bar{\psi}_a - \bar{\psi})^T,
\]

(4.11)

where now the over line denotes an average over the ensemble. Thus, we can use an interpretation where the ensemble mean is the best estimate and the spreading of the ensemble around the mean is a natural definition of the error in the ensemble mean.

4.2.2 Analysis scheme

We now derive the update scheme in the KF using the ensemble covariances as defined by (4.10) and (4.11). We now drop the time index $k$ in the following equations for convenience. It is essential that the observations be treated as random
variables having a distribution with mean equal to the observed value and covariance equal to $C_{ee}$ \[8\]. We start by defining an ensemble of observations

$$d_j = d + \epsilon_j, \ j = 1 \ldots N$$ \hspace{1cm} (4.12)

where $N$ is the number of ensemble members. Next the ensemble covariance matrix of the measurement errors is defined as

$$C_{ee} = \epsilon \epsilon^T.$$ \hspace{1cm} (4.13)

This matrix converges to the actual error covariance matrix $C_{ee}$ used in the Kalman filter when we have infinity number of realizations.

In the analysis step in EnKF, updates are performed on each of the ensemble members and is given by

$$\psi^a_j = \psi^f_j + (C_{\psi|\psi}^e)^f M^T(M(C_{\psi|\psi}^e)^f M^T + C_{ee}^e)^{-1}(d_j - M \psi^f_j).$$ \hspace{1cm} (4.14)

We should note that with a finite ensemble size, the use of the ensemble covariances is an approximation of the true covariances. Furthermore, the matrices $M(C_{\psi|\psi}^e)^f M^T$ and $C_{ee}^e$ are singular when the number of measurements is larger than the number of ensemble members, and we must use pseudo inversion instead of inversion.

Equation (4.14) implies that

$$\psi^a_j = \psi^f_j (1 - K_e M) + K_e (d_j - \bar{d}),$$ \hspace{1cm} (4.16)

where $\bar{d} = d$, since we have a zero mean ensemble of measurement perturbations. Thus, we have the same relation between the analysed and predicted ensemble mean, as we have between the analysed and predicted state in the standard Kalman filter, apart from using $(C_{\psi|\psi}^e)^f$ and $C_{ee}^e$ instead of $C_{ff}$ and $C_{ee}$ respectively. We should also note that the introduction of an ensemble of observations does not affect the update of the ensemble mean.

We now show that, by updating each of the ensemble members using the perturbed observations, we can create a new ensemble with the correct error statistics. We derive the analysed error covariance estimate as a result of the above analysis scheme, although we keep the standard Kalman filter form for the analysis equations. First, we obtain using (4.14) and (4.15)

$$(C_{\psi|\psi}^e)^a = (1 - K_e M)(C_{\psi|\psi}^e)^f + K_e (\bar{d} - \psi^f).$$ \hspace{1cm} (4.17)

Using equation (4.11), we derive the error covariance update as

$$(C_{\psi|\psi}^e)^a = (I - K_e M) (C_{\psi|\psi}^e)^f.$$

(4.18)
For this derivation, we had to assume that the model uncertainties and the measurement uncertainties are independent. However, using a finite ensemble size and neglecting the cross term introduces sampling errors.

When the Kalman gain matrix is singular, the inverse in (4.17) can be replaced with the pseudo inverse, and we can write the Kalman gain as

$$K_e = (C_{e\psi}^e)^f M^T (M (C_{e\psi}^e)^f M^T + C_{ee}^e)^+.$$  \hspace{1cm} (4.19)

When the matrix in the inversion is of full rank, (4.19) becomes identical to (4.17).

Figure 4.3 depicts the working principle for a single ensemble member in EnKF.

4.3 Practical implementation

4.3.1 Ensemble representation of the covariance

We define the matrix $A$ whose columns are the ensemble members $\psi_i \in \mathbb{R}^n$ by

$$A = (\psi_1, \psi_2, \ldots, \psi_N) \in \mathbb{R}^{n \times N},$$ \hspace{1cm} (4.20)

where $N$ is the number of realizations and $n$ is the size of the model state vector. The ensemble mean is stored in each column of $\overline{A}$, which is defined as

$$\overline{A} = A1_N,$$ \hspace{1cm} (4.21)
where $1_N \in \mathbb{R}^{N \times N}$ is the matrix whose entries are all equal $1/N$. The ensemble perturbation matrix is defined as

$$A' = A - \overline{A} = A(I - 1_N).$$  \hfill (4.22)

The ensemble covariance matrix $C_{\psi\psi}^e \in \mathbb{R}^{n \times n}$ can be defined as

$$C_{\psi\psi}^e = \frac{1}{N-1} A'(A')^T.$$  \hfill (4.23)

### 4.3.2 Measurement perturbations

Given a vector of measurements $d \in \mathbb{R}^m$, where $m$ is the number of measurements, we define the $N$ vectors of perturbed observations as

$$d_j = d + \epsilon_j, \; j = 1 \ldots N,$$  \hfill (4.24)

which are stored in the columns of the matrix

$$D = (d_1, d_2, \ldots, d_N) \in \mathbb{R}^{m \times N},$$  \hfill (4.25)

while the zero mean ensemble of perturbations are stored in the matrix

$$E = (\epsilon_1, \epsilon_2, \ldots, \epsilon_N) \in \mathbb{R}^{m \times N},$$  \hfill (4.26)

from which we construct the ensemble representation of the measurement error covariance matrix

$$C_{\epsilon\epsilon}^e = \frac{1}{N-1} EE^T.$$  \hfill (4.27)

### 4.3.3 Analysis equation

The analysis equation (4.14), expressed in terms of the ensemble matrices, is

$$A^a = A + C_{\psi\psi}^e M^T (MC_{\psi\psi}^e M^T + C_{\epsilon\epsilon}^e)^{-1} (D - MA),$$  \hfill (4.28)

where the measurement matrix is $M \in \mathbb{N}^{m \times n}$.

We define the ensemble of innovation vectors as

$$D' = D - MA,$$  \hfill (4.29)

where $(D' \in \mathbb{R}^{m \times N})$. Using $D'$, along with the definitions of the ensemble error covariance matrices in (4.23) and (4.27), the analysis can be expressed as

$$A^a = A + A' A'^T M^T (MA' A'^T M^T + EE^T)^{-1} D',$$  \hfill (4.30)

where all references to the error covariance matrices are eliminated.
We now define the matrix \( S \in \mathbb{R}^{m \times N} \), holding the measurements of the ensemble perturbations, by
\[
S = MA' \tag{4.31}
\]
and the matrix \( C \in \mathbb{R}^{m \times m} \),
\[
C = SS^T + (N - 1)C_{ee}. \tag{4.32}
\]
Here we can use the full-rank, exact measurement error covariance matrix \( C_{ee} \) or the low-rank representation \( C_{ee}' \) defined in (4.27).

The analysis equation (4.30) can then be written as
\[
A^a = A + A'S^T C^{-1}D' = A + (I - 1_N)S^T C^{-1}D'
= A(I + (I - 1_N)S^T C^{-1}D') = AX, \tag{4.33}
\]
where we used the identity in (4.22). The matrix \( X \in \mathbb{R}^{N \times N} \) is defined as
\[
X = I + (I - 1_N)S^T C^{-1}D'. \tag{4.34}
\]
Based on the assumption \( 1_N S^T \equiv 0 \), (4.33) can also be written as
\[
A^a = A(I + S^T C^{-1}D') = AX. \tag{4.35}
\]

Then (4.34) becomes
\[
X = I + S^T C^{-1}D'. \tag{4.36}
\]

When number of measurements is larger than the number of realizations, the matrix \( C \) is singular and instead of using inverse we should use pseudo-inverse of \( C \) in the above expressions. One popular way to compute pseudo-inverse of a matrix \( C \) is by SVD using the following formulation
\[
C^+ = VS^+ U^T, \tag{4.37}
\]
where \( U \) and \( V \) are the left and the right singular vectors of \( C \) respectively and \( S \) is the diagonal matrix containing the singular values of \( C \) in descending order and \( S^+ \) is defined as
\[
S^+ = \begin{cases} 
\frac{1}{S_{ij}}, & \text{if } i = j \text{ and } S_{ij} \neq 0 \\
0, & \text{else} 
\end{cases} \tag{4.38}
\]
The other way to write (4.32) is
\[
C = SS^T + EE^T, \tag{4.39}
\]
\[
\Rightarrow C = (S + E)(S + E)^T, \tag{4.40}
\]
assuming ensemble perturbations and measurement perturbations are uncorrelated, that is
\[
SE^T = MA'E^T = 0. \tag{4.41}
\]
If we compute the SVD

\[ S + E = U \Sigma V^T, \quad (4.42) \]

then (4.30) becomes

\[ A^a = A + A' S^T U \Lambda^{-1} U^T D', \quad (4.43) \]

where

\[ \Lambda = \Sigma \Sigma^T. \quad (4.44) \]

This is another approach for calculating the analysis avoiding inversion of a large matrix when \( m \gg N \), based on the above mentioned assumption.

There are two major sources of sampling errors in EnKF: the use of a finite ensemble of stochastic model realizations and the introduction of stochastic measurement perturbations [12], [13].

### 4.3.4 EnKF for combined parameter and state estimation

When using EnKF to estimate poorly known model parameters, we start by representing the prior probability density functions (pdf) of the parameters by an ensemble of realizations, which is augmented to the state ensemble matrix \( A \) at the update steps. The poorly known parameters are then updated using the variance minimizing analysis scheme, where the covariances between the predicted data and the parameters are used to update the parameters.

Algorithm 1 based on (4.33) and (4.34), summarizes the above discussion on working procedure of a single step of an ensemble Kalman filter. For quick reference, we recapitulate the dimensions of the matrices used in the algorithm: \( A_1, A_2, \ldots, A_N \in \mathbb{R}^n; A, A^a, A' \in \mathbb{R}^{n \times N}; 1_N, X_2, X_3, X_4 \in \mathbb{R}^{N \times N}; D, D', E, S, X_1 \in \mathbb{R}^{m \times N}; d, d_j, \epsilon_j \in \mathbb{R}^m; M \in \mathbb{R}^{m \times n}; C, C^+, U, \Sigma, V \in \mathbb{R}^{m \times m}. \)
Algorithm 1 A single step of basic EnKF

1: procedure EnKF
2: Run simulators to generate state vectors \( A_1, A_2, \ldots, A_N \)
3: Construct \( A = [A_1 \ A_2 \ \ldots \ A_N] \)
4: \( I_N = \frac{1}{N} \text{ones}(N,N) \)
5: \( A' = A(I-1_N) \quad O(nN^2) \)
6: \( S = MA' \quad O(mnN) \)
7: Load measurement \( d \)
8: Generate Gaussian noise \( \epsilon_j \), for \( j=1,2,\ldots,N \)
9: Compute \( d_j = d + \epsilon_j \), for \( j=1,2,\ldots,N \)
10: Construct \( D = [d_1 \ d_2 \ \ldots \ d_N] \)
11: \( D' = D - MA \quad O(mnN) \)
12: Construct \( E = [\epsilon_1 \ \epsilon_2 \ \ldots \ \epsilon_N] \)
13: \( C = SS^T + EE^T \quad O(m^2N) \)
14: \( [U, \Sigma, V] = \text{svd}(C) \quad O(m^3) \)
15: \( C^+ = V\Sigma^+U^T \quad O(m^3) \)
16: \( X_1 = C^+D' \quad O(m^2N) \)
17: \( X_2 = S^TX_1 \quad O(mN^2) \)
18: \( X_3 = (I-1_N)X_2 \quad O(N^3) \)
19: \( X_4 = I + X_3 \quad O(N) \)
20: \( A^a = AX_4 \quad O(nN^2) \)
21: return \( X^a \)
22: end procedure

Algorithm 2 is the very primitive translation of algorithm 1 to the parallel version. We call this algorithm ‘very primitive’ because it simply states the order of execution of the statements in parallel, without any consideration for optimizations. Here we see several chunks of communications are interleaved between computing whereas some chunks communicate large volume of data and also many serial works are done by the root, indicating this is certainly an inefficient parallel algorithm. Many levels of optimizations, which makes this algorithm very efficient, are described in detail in chapter 6.
Algorithm 2 A single step of basic parallel EnKF

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td><strong>procedure</strong> EnKF</td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>Run simulator to generate state vector $A_i$</td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>(root) Collect states $A_1, A_2, \ldots, A_N$</td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>(root) Construct $A = [X_1 \ X_2 \ \ldots \ X_N]$</td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td>Communicate $A$</td>
<td>$O(nN)$</td>
</tr>
<tr>
<td>6:</td>
<td>(root) $1_N = \frac{1}{N}$ ones(N,N)</td>
<td></td>
</tr>
<tr>
<td>7:</td>
<td>Communicate $1_N$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>8:</td>
<td>$A' = A(I-1_N)$</td>
<td>$O(nN^2)$</td>
</tr>
<tr>
<td>9:</td>
<td>Communicate $A'$</td>
<td>$O(nN)$</td>
</tr>
<tr>
<td>10:</td>
<td>$S = MA'$</td>
<td>$O(mnN)$</td>
</tr>
<tr>
<td>11:</td>
<td>Communicate $S$</td>
<td>$O(mN)$</td>
</tr>
<tr>
<td>12:</td>
<td>Load measurement $d$</td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>(root) Generate Gaussian noise $\epsilon_j$, for $j=1, 2, \ldots, N$</td>
<td></td>
</tr>
<tr>
<td>14:</td>
<td>(root) Construct $E = [\epsilon_1 \ \epsilon_2 \ \ldots \ \epsilon_N]$</td>
<td></td>
</tr>
<tr>
<td>15:</td>
<td>Communicate $E$</td>
<td>$O(mN)$</td>
</tr>
<tr>
<td>16:</td>
<td>(root) Compute $d_j = d + \epsilon_j$, for $j=1, 2, \ldots, N$</td>
<td></td>
</tr>
<tr>
<td>17:</td>
<td>(root) Construct $D = [d_1 \ d_2 \ \ldots \ d_N]$</td>
<td></td>
</tr>
<tr>
<td>18:</td>
<td>$D' = D - MA$</td>
<td>$O(mnN)$</td>
</tr>
<tr>
<td>19:</td>
<td>Communicate $D'$</td>
<td>$O(mN)$</td>
</tr>
<tr>
<td>20:</td>
<td>$C = SS^T + EE^T$</td>
<td>$O(m^2N)$</td>
</tr>
<tr>
<td>21:</td>
<td>Communicate $C$</td>
<td>$O(m^3)$</td>
</tr>
<tr>
<td>22:</td>
<td>$[U, \Sigma, V] = \text{svd}(C)$</td>
<td>$O(m^3)$</td>
</tr>
<tr>
<td>23:</td>
<td>Communicate $U, V$, and $\Sigma$</td>
<td>$O(m^3)$</td>
</tr>
<tr>
<td>24:</td>
<td>$C^+ = V\Sigma^+U^T$</td>
<td>$O(m^3)$</td>
</tr>
<tr>
<td>25:</td>
<td>Communicate $C^+$</td>
<td>$O(m^3)$</td>
</tr>
<tr>
<td>26:</td>
<td>$X_1 = C^+D'$</td>
<td>$O(m^2N)$</td>
</tr>
<tr>
<td>27:</td>
<td>Communicate $X_1$</td>
<td>$O(mN)$</td>
</tr>
<tr>
<td>28:</td>
<td>$X_2 = S^TX_1$</td>
<td>$O(mN^2)$</td>
</tr>
<tr>
<td>29:</td>
<td>Communicate $X_2$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>30:</td>
<td>$X_3 = (I-1_N) \ X_2$</td>
<td>$O(N^3)$</td>
</tr>
<tr>
<td>31:</td>
<td>Communicate $X_3$</td>
<td>$O(N^3)$</td>
</tr>
<tr>
<td>32:</td>
<td>$X_4 = I + X_3$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>33:</td>
<td>Communicate $X_4$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>34:</td>
<td>$A^a = AX_4$</td>
<td>$O(nN^2)$</td>
</tr>
<tr>
<td>35:</td>
<td>(root) Distribute $A^a$ for further use by the simulators</td>
<td>$O(nN)$</td>
</tr>
<tr>
<td>36:</td>
<td><strong>end procedure</strong></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5

Parallel Computing with MPI and ScaLAPACK

In this chapter we briefly introduce parallel computing and some related common technologies. First we go through the basic parallel architecture concepts. Then we describe the message passing interface (MPI), a key software technology for parallel computing in distributed memory architecture. Then we list some parallel numerical computing libraries with a special and detailed description of ScaLAPACK.

5.1 Parallel computing

Frequency scaling limitation due to the physical constraints, and also power consumption and heat generation by computers have become a key concern in recent years. As a result parallel computing has become the dominant paradigm in computer architecture. In parallel computing, many computations are carried out simultaneously. It is based on the principle that large problems can often be divided into smaller ones, which are then solved concurrently.

Some broad forms of parallelism in computing are:

1. **Bit level**: Doubling the computer word size can halve the number of instructions required to carry out operations working on operands of two times the word size. It was a common trend of speed up until 1986. At present a similar approach is taken in vector processing, where a single processor register is large enough to hold multiple scalars and a single instruction operates on multiple scalars (hence the term vector), significantly speeding up a single processor operations.

2. **Instruction level**: To carry out a single operation, an instruction goes through several stages such as instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM) and register write back (WB). These are ac-
accomplished by different hardware in the processor and sequence of instructions can be pipelined to enhance the overall process in a single processor.

3. **Data level:** This type of parallelism are inherent in program loops, focusing on distributing the data across different processing elements to be processed in parallel. Data dependency is a typical challenge for data level parallelism. This is a very conventional parallelism from a view of multiple processors.

4. **Task level:** For some applications, a parallel program can do different calculations on either the same or different sets of data. This is another conventional parallelism for multiple processors.

On the other hand followings are the major categories of parallel computing from hardware point of view:

1. **Shared memory multiprocessing or Symmetric multiprocessing (SMP):** Here multiple identical processors share the same memory (at least the same logical address space) and connect via a bus. It has limited number of processors to ensure sufficient bandwidth and hence has limited scaling.

2. **Distributed memory multiprocessing:** Here the processing elements are connected via a communication network. Performance depends much on the network type and also the amount as well as the number of data communications required. There is no limitation on the number of processors and hence it is highly scalable. It is the mostly used architecture for parallel computing.

Figure 5.1 and figure 5.2 depicts the basic differences among these two hardware architectures.
5.2 MPI

Message Passing Interface (MPI), a standardized and portable message passing system, defines the syntax and semantics of a core of library routines for writing portable message-passing programs in Fortran or C programming language. The very initial implementation of the MPI 1.x standard was MPICH. LAM/MPI was another early open implementation. LAM/MPI and a number of other MPI efforts recently merged to form Open MPI, whereas the commercial organizations like IBM, HP, Intel, SGI, and Microsoft have their own implementations of MPI.

Although MPI is a complex and multifaceted system having more than 500 subroutines, typically 10 to 30 are useful and we can solve a wide range of problems using just six of its functions: \texttt{MPI\_Init}, \texttt{MPI\_Finalize}, \texttt{MPI\_Comm\_size}, \texttt{MPI\_Comm\_rank}, \texttt{MPI\_Send}, \texttt{MPI\_Recv} \cite{18}.

Tasks run throughout the program execution can be divided into four categories:

1. \textbf{Start up phase}: launches tasks and establishes communication context among all tasks. The set of the routines which we need for our application is:
   - \texttt{MPI\_Init}: initiates an MPI computation.
   - \texttt{MPI\_Comm\_size}: determines number of processes.
   - \texttt{MPI\_Comm\_rank}: determines the process identifier.

2. \textbf{Point-to-point data transfer}: usually takes place between pairs of processes and usually in a coordinated fashion. The transfer may be blocking or non-blocking. Explicit synchronization is needed for non-blocking transfers. Here we list the routines which we require for our application:
• **MPI Send**: sends a message to a particular destination. This routine is blocking, that means it returns after the transfer is completed.

• **MPI Recv**: receives a message from a particular source. The routine, a blocking one, returns after the transfer is completed.

• **MPI Isend**: sends a message to a particular destination. The routine returns immediately after the call and does not guarantee the completion of the transfer. We need special mechanisms to be sure of the completion of the data transfer.

• **MPI Irecv**: receives a message from a particular source. The non blocking routine returns immediately after the call and does not guarantee the completion of the data transfer. We need special mechanisms to be sure of the completion.

• **MPI Wait**: The sender and the receiver wait for the completion of the non blocking data transfer routines MPI Isend and MPI Irecv respectively.

3. **Collective communication**: happens between all tasks or a subgroup of tasks and only in blocking fashion. Broadcasts, reductions, scatter/gather operations are some examples. When appropriate, they are preferred for efficient library calls. We need the following five routines for our application.

  • **MPI Gather**: All the processes send a fixed amount of data to the root and the root arranges all the received data sequentially.

  • **MPI Gatherv**: Different process sends different amount of data to the root and the root arranges all the received data sequentially.

  • **MPI Scatter**: The root process evenly distributes some data to all the processes.

  • **MPI Scatterv**: The root process unevenly distributes some data to all the processes.

  • **MPI Barrier**: Blocks until all processes in the communicator have reached this routine.

4. **Clean up phase**: does the reverse tasks done in the start up phase. This is done with the simplest function MPI Finalize without any argument.

### 5.3 Parallel numerical libraries for SVD

For parallelization of EnKF analysis step we mainly depend on matrix-matrix multiplications and SVD computations. Matrix multiplication is less complex than the SVD computation with respect to computation time. There are several parallel numerical libraries to handle this two problems. There are differences among the algorithms implemented in these libraries. Some depend on iterative methods,
some use preconditioning, some use block algorithms. For eigenvalue algorithms, some compute a partial set of eigenvalues/vectors. Some are useful for special types of matrices (ex. sparse), whereas the matrices in our application are dense. Most of the libraries in the following list have special syntaxes (ex. data types, array and matrix structures) and handle data communication by themselves. We choose ScaLAPACK because of its closeness to the original C and MPI syntaxes and to have some grips on the data communication so that we can use non blocking communications where as other libraries mostly use blocking communications internally. Moreover, ScaLAPACK uses the popular and easy to use LAPACK style interfaces.

1. **SLEPC**: The *Scalable Library for Eigenvalue Problem Computations* is a specialized software library for the solution of large scale sparse eigenvalue problems on parallel computers. It is an extension of PETSc and can be used for either standard or generalized eigen problems. It can also be used for computing a partial SVD of a large, sparse, rectangular matrix, and to solve quadratic eigenvalue problems.

2. **BLOPEX**: The *Block Locally Optimal Preconditioned Eigenvalue Solvers* is a suite of routines for the scalable (parallel) solution of eigenvalue problems.

3. **Lis**: The *Library of Iterative Solvers for linear systems* is a scalable parallel library for solving systems of linear equations and standard eigenvalue problems with real sparse matrices using iterative solvers.

4. **ELPA**: The *Eigenvalue solvers for Petaflop Applications* is a Fortran-based high-performance computational library for the (massively) parallel solution of symmetric or Hermitian, standard or generalized eigenvalue problems.

5. **The NAG C Library**: is a collection of numerical analysis routines developed by the *Numerical Algorithms Group* which can be called from user applications running on a wide variety of hardware platforms.

6. **ParPACK**: is a portable *parallel* version of the ARPACK library. ARPACK is a collection of Fortran77 subroutines designed to solve large scale eigenvalue problems. The package is designed to compute a few eigenvalues and corresponding eigenvectors of a general n by n matrix A.

### 5.3.1 ScaLAPACK

ScaLAPACK (Scalable Linear Algebra PACKage) is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD (Multiple Instruction Multiple Data) computers and networks of workstations supporting PVM and/or MPI. It is provided by a consortium of universities as well as an industry, consisting of University of Tennessee; University of California, Berkeley; University of Colorado Denver; and NAG Ltd. ScaLAPACK solves dense and
also banded linear systems but not sparse systems, least squares problems, eigen-value problems, and singular value problems. They can also handle many associated computations such as matrix factorizations or estimation of condition numbers. The fundamental building blocks of the ScaLAPACK library are distributed-memory versions of the Level 1, Level 2, and Level 3 BLAS, called the Parallel BLAS (PBLAS), and a set of Basic Linear Algebra Communication Subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations. Figure 5.3 depicts the structure of ScaLAPACK. ScaLAPACK includes the following key ideas [4]:

1. block cyclic data distribution (for dense matrices) and block data distribution (for banded matrices): the distribution information is used as parameters during runtime;
2. block-partitioned algorithms: in order to minimize the frequency of data movement;
3. well-designed low-level modular components: to simplify the task of parallelizing the high level routines by making their source code the same as in the sequential case.

The goals of ScaLAPACK are the same as those of LAPACK:

1. efficiency (to run as fast as possible),
2. scalability (as the problem size and number of processors grow),
3. reliability (including error bounds),

4. portability (across all important parallel machines),

5. flexibility (so users can construct new routines from well-designed parts),

6. ease of use (by making the interface to LAPACK and ScaLAPACK look as similar as possible).

In ScaLAPACK, generally there are three steps required to divide the computations among the parallel processors [7]. They are:

1. Initialize the library (using standard routines).

2. Set up a processor grid.

3. Map the data structure used (array, vector, etc.) onto the processor grid.
   - Create an array descriptor vector for each global array.
   - Map global array elements in a 2-D blocked-cyclic manner onto each processor grid.

After these steps are completed, each processor will be responsible for the calculations on some subset of the data. This general approach is referred to by many names such as data decomposition, domain decomposition, data distribution, and so on.

**Process grid**

ScaLAPACK uses a processor grid during parallel work distribution. One of the member libraries under the ScaLAPACK umbrella is the BLACS (Basic Linear Algebra Communication Subroutines) library. In addition to array-operand communication calls, the BLACS library contains important routines for creating and examining the processor grid, which is expected and used by all ScaLAPACK routines. The processor grid is a two-dimensional array whose size and shape is controlled by the program. A processor is identified by its row and column number in the processor grid rather than its traditional MPI rank [9].

The processor grid diagram for eight processors used in some parallel program arranged in a $2 \times 4$ grid with row major ordering will be

```
0 1 2 3
0 0 1 2 3
1 4 5 6 7
```

The MPI rank for a processor is contained within each element of the grid. For example, the processor with MPI rank 6 has grid coordinates (1, 2).
Data distribution

An important performance criteria for ScaLAPACK is the data distribution phenomena. We said earlier that ScaLAPACK data distribution method is block cyclic for dense matrices. To clarify this we show the differences among the column block, the column cyclic, the column block cyclic and the block cyclic distributions for a 16x16 array among 4 processors in figures 5.4 and 5.5. In simple column block distribution each processor gets 4 consecutive columns and 16 rows of the original array. In column cyclic distribution each processor gets 4 columns and 16 rows but now the columns are not consecutive. In column block cycling distribution each processor gets 4 columns and 16 rows but now every two columns are consecutive in the original matrix. In block cyclic distribution, which is simultaneously column block and row block distribution, each processor gets 4 rows and 4 columns where every two rows and every two columns are consecutive in the original matrix. The rules for block cyclic data distribution are:

1. Divide the global array into blocks with \( mb \) rows and \( nb \) columns.

2. Place the first row of array blocks across the first row of the processor grid in order. If you run out of processor grid columns, cycle back to the first column of the first row.

3. Repeat the previous step with the second row of array blocks and use the second row of the processor grid. Cycling back if you need to.

4. Continue for the remaining rows of array blocks.

5. If you run out of processor grid rows, cycle back to the first processor row and keep repeating the steps above.
The diagram in figure 5.6 illustrates a two-dimensional, block-cyclic distribution of a 9×9 global array with 2×2 array blocks over a 2×3 processor grid. (The colors represent the IDs of the six different processors.)

Useful routines

We need the following ScaLAPACK routines for our application:

1. Cblacs_pinfo: It reports the rank for the calling process, and the total number of processes available before the BLACS environment set up.

2. Cblacs_get: It reports information about the BLACS state. It can be used to return the system default context needed to construct a first BLACS grid and its associated context. The default context corresponds to the MPI default context of MPI_COMM_WORLD.

3. Cblacs_gridinit: The BLACS grid created is identified by a context handle that is subsequently used to identify that particular process grid among the many that may be generated. This routine takes the available processes and assign or map them into a BLACS process grid. In other words, it establishes how the BLACS coordinate system will map into the native machine’s process numbering system.

4. Cblacs_gridinfo: It reports information about a BLACS context. Returns information on the current grid, particularly the coordinate of the process calling it.

5. Cblacs_barrier: This routine holds up execution of all processes within the indicated scope until they have all called the routine.
6. **Cblacs_exit**: This routine should be called when a process has finished all use of the BLACS. It frees all BLACS contexts and releases all memory the BLACS have allocated.

7. **numroc**: This function computes the local number of rows or columns of a block-cyclically distributed matrix contained in a process row or process column, respectively for the calling process.

8. **descinit**: It initializes the descriptor vector with 8 input arguments: number of row and columns in the operand matrix, row and column wise block sizes, row and column indices of the process containing the first row and the first column of the matrix, the BLACS context handle, the leading dimension of the local array.

9. **indxl2g**: It computes the global row or column index of a local element of a block-cyclically distributed matrix.

10. **pdgesvd**: This subroutine computes the singular values and, optionally, the left and/or right singular vectors of a real or complex general matrix.

11. **pdgemm**: It computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product.
Chapter 6

Implementation

In this chapter we describe the details of the development of our application. First we describe the oil reservoir model, which we use in our application. Then we describe the various components of the application’s software structure. We conclude by describing some optimization techniques adopted for better performance of the parallel application.

We stated in chapter 2 that only a few researches have been carried out for parallelization of EnKF for oil reservoirs. Moreover, they work with only sparse measurement data like BHP, and oil and water flow rates from the wells and parallelize the embarrassingly parallel forecast step only and as a result have a limited speedup with large number of processors and consequently show limited scaling. On the other hand, dense data like saturation, calculated from time-lapse seismic from every grid block, though more noisy than the direct measurement data from the wells, give better estimation of the reservoir states, applying data assimilation.

We incorporate time-lapse seismic data in our parallel implementation of EnKF for oil reservoirs for the first time. Inclusion of time-lapse seismic data makes the EnKF process more time consuming as the analysis step becomes larger than the forecast step in terms of execution time. In addition to this, the analysis part is more complex than the forecast step as it requires intermediate inter process communications. It is challenging to maintain or improve the speedup of the previous research works with the added complex analysis step which requires communication of large matrices. However, by intelligent use and runtime configurations of the ScaLAPACK routines we minimize the requirement of data communications effectively and achieve very good speedup values. Even with large number of processors, our implementation shows better scaling.

6.1 Reservoir model

We work with a rectangular reservoir field with length 1980 m, width 1020 m and height 5 m. We assume all the fluid and rock properties of the reservoir to be constant along the height and hence we consider it as a two dimensional reservoir. For
Figure 6.1: Location of the wells in the model reservoir field

computational purpose the field is logically divided into 5049 blocks in a $51 \times 99$ grid of square blocks of size 20 m in each side. We have two injection wells located at the coordinates (2, 2) and (50, 17) and two production wells located at the coordinates (17, 62) and (41, 98) as shown in figure 6.1. The black dots in the left side of the figure are the injection wells, whereas the red dots in the right side are the production wells.

The typical state vector for oil reservoir consists of the following variables

1. pressure ($p$)
2. saturation ($s$)
3. bottom hole pressures in the wells ($BHP$)
4. oil flow rate in the wells ($q_o$)
5. water flow rate in the wells ($q_w$)

We consider observations of saturations (in form of time-lapse seismic) at every block and bottom hole pressures, and oil and water flow rates in the wells (in form of measurements). We consider the following standard deviations of the measurement noise (Gaussian) for observations: 15% for saturation, 5% for BHP and oil flow rate, and 10% for water flow rate. We want to estimate the log permeability ($\log k$) also. Then our state vector has the following form.

$$ A = \begin{bmatrix} p \\ s \\ BHP \\ q_o \\ q_w \\ \log k \end{bmatrix} $$
We assume a constant porosity field in figure 6.2 for all ensemble members and we do not consider it in our analysis process.

Our measurement network excludes pressure values and permeability values. Then for computation of S and D in analysis steps described in chapter 4, we use the following portion (of length 5061) of the state vector:

\[ A = \begin{bmatrix} s \\ BHP \\ q_o \\ q_w \end{bmatrix} \]

And as estimating the log permeability values (\( \log k \)) is our primary goal of the assimilation and updating other state values are of minor importance (and we neglect them) and hence in the update statement in 4.33 we use only the following portion (of length 5049) of the state vector.

\[ A = [ \log k ] \]

### 6.2 Software structure

For the forecast or simulation step we used simsim (simple simulator), written in MATLAB, developed by Prof. Jan Dirk Jansen, TU Delft. For parallel data assimilation we used MPI in C language. To interface with simsim we developed a driver named simsimdriver which contains all data assimilation computations as well as function calls for running the simulator for the forecast step. To make the interface simple we developed two wrapper functions in MATLAB:

1. **initSimsim**: It initializes and prepares all the required resources such as model, control and parameter informations in each processor. It also provides assumed permeabilities of the ensemble.
2. **runSimsim**: It works with the updated assumption of permeabilities for a defined time period and returns the seismic data for saturation and measurement data of bottom hole pressure, and oil and water flow rates measured in the wells.

These wrapper functions and the original `simsim` functions are compiled to a library, usable in C programming, by the MATLAB compiler `mcc` tool with appropriate settings. The driver program in C and the compiled library needs to be compiled to an executable with the MATLAB compiler `mbuild` utility.

Figure 6.3 depicts the relation and interaction among different components of the parallel application in the simplest 2 processors environment for 4 realizations, where each processor processes 2 realizations. State vectors and the permeability data for each realization are represented with different colors. In the figure we see the data/work flow sequence of the application. The vertical dotted line defines the division between the processors P0 and P1. The horizontal dotted line indicates the division of the execution platform between MATLAB and C in each processor. The MATLAB part in the application is handled by MATLAB Compiler Runtime (MCR). The original simulator `simsim` and the wrapper functions `initSimsim` and `runSimsim` are executed by MCR. We also observe that the simulation part in each processor is independent of the other, whereas the analysis part needs to be carried out in a coordinated fashion. Also notable, the data (state vectors A0 or A1) returned by one processor after the simulation step need to be distributed among all
the processors for the analysis step and the data (updated permeabilities \( K_0 \) or \( K_1 \)) fed to the simulators in one processor need to be collected from all the processors. The reason is explained in the next section. We also notice that the steps 4 to 7 in the figure represents a single cycle for a single assimilation step.

### 6.3 Optimization

We already mentioned that we have two parts to parallelize:

1. **Forecast or simulation:** This part is embarrassingly parallel. Each processor will run a number of ensemble members to generate an ensemble of forecasts without any requirement of communication with other processors or other members in the ensemble. Only the updated permeabilities need to be collected from and predicted state values need to be sent to the root processor for gathering for the analysis step.

2. **Analysis or update:** We saw the analysis step involves SVD computation as well as a series of matrix multiplications. These operations also nicely scale in a parallel computing environment, though not as nicely as the forecast step because of the need for inter processor communication. Also there are some preparatory serial tasks for the analysis step.

We adopt the two most common ways to optimize parallel algorithms, load balancing and communication minimization, and discuss below.

#### 6.3.1 Load balancing

We statically distribute equal number of realizations to each processor to run the simulation. As we need no processor-to-processor communication during the forecast step, dynamic load balancing is very complex in this step. We ensure proper load balancing in the analysis step by choosing small block sizes of the operand matrices for ScaLAPACK operations. We choose block size 64 in case of the long dimensions (for example the rows in the matrices \( A, S, D, X_1, k \); and both rows and columns in the matrices \( C, C^+, U, V \)) and \( \frac{N}{nproc} \) in case of short dimensions (for example columns of the matrices \( A, S, X_1, k \); and both rows and columns of the matrices \( I - 1_N, X_2, X_3, X_4 \)), where \( nproc \) is the number of processes we utilize for the parallel computing. In fact the smallest possible block size 1 ensures 100% load balancing, but then the number of communications for ScaLAPACK operations increases, leading to bad performance. ScaLAPACK suggests a \( 64 \times 64 \) two dimensional blocks for a balanced load as well as tolerable communication overhead for optimum performance.
6.3.2 Minimizing data communication

To get the maximum performance we must reduce data communication. The embarrassingly parallel forecast step of EnKF is executed in a column block distribution (figure 5.4) with respect to the predicted state ensemble $A$. If we want to execute also the analysis step in this orientation, we will lose performance significantly for the ScaLAPACK matrix operation routines \textit{pdgesvd} and \textit{pdgemm}, as ScaLAPACK suggests a square or at least a close-to-square rectangular processor grid arrangement for the best performance, because this arrangement ensures the least internal communication for ScaLAPACK operations. So we have to sacrifice some communication overheads to gain significant performance from the ScaLAPACK routines. That is, we will go column block orientation for the forecast step and two dimensional block cyclic distribution for the analysis step. To do this we have to gather the forecast/simulation results (state vectors) to convert to 2D block cyclic distribution from column block distribution and vice versa after the analysis step (to feed the updated parameter values).

Other than the internal communications of the ScaLAPACK operators, we put significant efforts to minimize data communication by two ways.

Construction and computations of the local matrices locally:

In the basic parallel EnKF step, listed in algorithm 2, there are a lot of data communication statements. The reason behind this is our over dependence on the master-slave relationship among the available processors. We assumed that each operand and resultant matrix in the analysis step must be either constructed by the root processor or assembled together by the root processor after parallel operation and transferred to the other processors for further parallel operation on it. But the local parts of the matrices in each processor can be constructed locally, mitigating the need for data communication. For example, in algorithm 2 the local parts of the matrix $(I - 1_N)$ can be constructed locally. It needs computing which local matrix element corresponds to the diagonal elements of the global matrix. After construction of $(I - 1_N)$ and successful transfer of the matrix $A$ and $E$, the local parts of the matrices $A'$, $S$, $D$, $D'$ can be computed locally, significantly reducing communication requirement in the steps 7, 9, 11 and 19 in the reference algorithm. An important point to note here is that this is valid only if later matrix operations involving these matrices can be carried out with the current ScaLAPACK data distribution. This requirement can be ensured by fixing the block sizes, as we discussed earlier. In the same way we get rid of the communicating large matrices $C$, $U$, $V$, $C^+$; and also $X_1$, $X_2$, $X_3$, and $X_4$ in steps 21, 23, 25, 27, 29, 31 and 33 respectively. All these improvements lead to the algorithm 3 with very little data communication requirement. An important thing to note at this point is that in the practical implementation of this algorithm we do not use the measurement operator $M$ in steps 8 and 15, as we consider observations of saturations from every grid block of the reservoir. We included the measurement
operator $M$ only for notational purpose and to comply with previous discussion and algorithms. Also note that in the practical implementation, the dimension and contents of the matrix $A$ may change in step 24 due to the facts described in section 6.1. For quick reference, we recapitulate the dimension of the matrices used in the algorithm: $A_1, A_2, \ldots, A_N \in \mathbb{R}^n$; $A, A^a, A' \in \mathbb{R}^{n \times N}; 1_N, X_2, X_3, X_4 \in \mathbb{R}^{N \times N}; D, D', E, S, X_1 \in \mathbb{R}^{m \times N}; M \in \mathbb{R}^{m \times n}; d, d_j, \epsilon_j \in \mathbb{R}^m; C, C^+, U, \Sigma, V \in \mathbb{R}^{m \times m}$.

**Algorithm 3** A single step of the optimized parallel EnKF

1: **procedure** EnKF
2: Run simulator to generate state vector $A_i$
3: (root) Collect states $A_1, A_2, \ldots, A_N$
4: (root) Construct $A = [A_1 \ A_2 \ldots \ A_N]$
5: Communicate $A$ $O(nN)$
6: $1_N = \frac{1}{N} \text{ones}(N,N)$
7: $A' = A(1-1_N)$ $O(nN^2)$
8: $S = MA'$ $O(mnN)$
9: Load measurement $d$
10: (root) Generate Gaussian noise $\epsilon_j$, for $j=1, 2, \ldots, N$
11: (root) Construct $E = [\epsilon_1 \ \epsilon_2 \ldots \ \epsilon_N]$
12: Communicate $E$ $O(mN)$
13: Compute $d_j = d + \epsilon_j$, for $j=1, 2, \ldots, N$
14: Construct $D = [d_1 \ d_2 \ldots \ d_N]$
15: $D' = D - MA$ $O(mnN)$
16: $C = SS^T + EE^T$ $O(m^2N)$
17: $[U, \Sigma, V] = \text{svd}(C)$ $O(m^3)$
18: $C^+ = V \Sigma^+ U^T$ $O(m^3)$
19: $X_1 = C^+ D'$ $O(m^2N)$
20: $X_2 = S^T X_1$ $O(mN^2)$
21: $X_3 = (1-1_N) \ X_2$ $O(N^3)$
22: $X_4 = I + X_3$ $O(N)$
23: $A^a = AX_4$ $O(nN^2)$
24: (root) Distribute $A^a$ for further use by the simulators $O(nN)$
25: **end procedure**

Choosing non blocking communication:

Then we have only four data communication statements in algorithm in step 3 to collect the forecast results from the column block oriented processor grids, in step 5 to transfer the matrix $A$, in step 12 to transfer the matrix $E$ and in step 24 to distribute the updated permeability values to the column block oriented processor grid for the forecast step. Communications for the matrices $A$ and $E$ can be done
in non blocking fashion by overlapping communication and computations. To do that we have to rearrange the steps in the above algorithm without affecting the computational results. The finally optimized version is in algorithm 4. Other than the rearrangements, it divides the single statement $C = SS^T + EE^T$ into two different statements $E_E = EE^T$ and $C = SS^T + E_E$ for better rearrangement.

Unfortunately, the communications in steps 3 and 24 in algorithm 3 can not be done in non blocking fashion due to lack of availability of significant work which can be overlapped. Also the root processor does some unavoidable works in serial fashion. But fortunately these tasks are very small in terms of computation time.

To achieve non blocking communication, we have to do the data transfers manually, using the routines `MPI_Isend` and `MPI_Irecv`, instead of simple and easy to use collective transfers by the routines `MPI_Gather` and `MPI_Scatter`. Also there are nice and simple CBLACS and ScaLAPACK data transfer routines `Cdgesd2d`, `Cdgerv2d`, `Cdgebs2d`, `Cdgebr2d`; and `pdgmr2d` respectively. We can not use these handy tools because all these routines are blocking communications methods, instead we utilized the mostly used triad routines for non blocking communication: `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`. 
Algorithm 4 A single step of the final parallel EnKF

1: procedure $\text{EnKF}$
2: 
3: Load measurement $d$
4: (root) Generate Gaussian noise $\epsilon_j$, for $j=1, 2, \ldots, N$
5: (root) Construct $E = [\epsilon_1 \ \epsilon_2 \ \ldots \ \epsilon_N]$
6: Communicate $E$ (asynchronously) $O(mN)$
7: Run simulator to generate state vector $A_i$
8: (root) Collect states $A_1, A_2, \ldots, A_N$
9: (root) Construct $A = [A_1 \ A_2 \ \ldots \ A_N]$
10: Communicate $A$ (asynchronously) $O(nN)$
11: $I_N = \frac{1}{N} \text{ones}(N,N)$
12: $E_iE = EE^T$ $O(m^2N)$
13: Compute $d_j = d + \epsilon_j$, for $j=1, 2, \ldots, N$
14: Construct $D = [d_1 \ d_2 \ \ldots \ d_N]$ $O(nN^2)$
15: $A' = A(I-1_N)$ $O(nN^2)$
16: $S = MA'$ $O(mnN)$
17: $C = SS^T + E_iE$ $O(m^2N)$
18: $[U, \Sigma, V] = \text{svd}(C)$ $O(m^3)$
19: $C^+ = V\Sigma^+U^T$ $O(m^3)$
20: $D' = D - MA$ $O(mnN)$
21: $X_1 = C^+D'$ $O(m^2N)$
22: $X_2 = S^TX_1$ $O(mN^2)$
23: $X_3 = (I-1_N)X_2$ $O(N^3)$
24: $X_4 = I + X_3$ $O(N)$
25: $A^a = AX_4$ $O(nN^2)$
26: (root) Distribute $A^a$ for further use by the simulators $O(nN)$
27: end procedure
Chapter 7

Results and Discussions

In this chapter we present and discuss some experimental results. For any parallel algorithm, the first requirement is the correctness compared to the serial counterpart. First we present data and graphs representing the correctness of our parallel implementation of EnKF. Then we present some data and graphs representing the speedup and efficiency, and other related issues of the implemented algorithm on different number of processors. We worked on two different clusters, first one is in a local area network (LAN) with heterogeneous hardware and the other one is the SARA Lisa high performance compute cluster. In both clusters, we used a single core of each processor to have homogeneous data communication among all the processing elements.

7.1 Verification

The main goal of data assimilation is to minimize the difference between the predictions and the true values for the reference variables or parameters. So, we should show that with time (or assimilation steps) the difference is reducing in general. And also, the assimilation performance of EnKF should be better with a large number of realizations (say, a hundred) than that with a few realizations (say, a dozen). Root mean square (rms) difference between the assimilated parameters with the true values is a good tool to perceive the result.

In our experiments, the models simulate the reservoir operations for ten years, whereas data assimilations take place in the first three years at six months interval, resulting in six assimilation steps. Table 7.1 shows the rms difference of the assimilated log permeability and true log permeability when executed on four processors with 16, 48 and 96 realizations. A decaying curve indicates the correctness of the implemented parallel algorithm. A significant difference between the curves can be observed for 16 and 96 realizations. A similar difference between the curves for 16 and 48 realizations is observed. As said earlier, with large number of realizations, the EnKF performs better. Additionally we observe that with large number of realizations the performance of the EnKF is smoother. However the difference
Table 7.1: rms differences between the true and assimilated log permeabilities at different time steps

<table>
<thead>
<tr>
<th>Assimilation time (year)</th>
<th>rms (16 rea)</th>
<th>rms (48 rea)</th>
<th>rms (96 rea)</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial (no assimilation)</td>
<td>0.577885</td>
<td>0.53903</td>
<td>0.553806</td>
</tr>
<tr>
<td>0.5</td>
<td>0.583252</td>
<td>0.514202</td>
<td>0.507765</td>
</tr>
<tr>
<td>1</td>
<td>0.498312</td>
<td>0.430884</td>
<td>0.453633</td>
</tr>
<tr>
<td>1.5</td>
<td>0.524888</td>
<td>0.441685</td>
<td>0.409763</td>
</tr>
<tr>
<td>2</td>
<td>0.511347</td>
<td>0.412005</td>
<td>0.379721</td>
</tr>
<tr>
<td>2.5</td>
<td>0.431661</td>
<td>0.369525</td>
<td>0.382082</td>
</tr>
<tr>
<td>3</td>
<td>0.440173</td>
<td>0.350035</td>
<td>0.337289</td>
</tr>
</tbody>
</table>

between 48 and 96 realizations is not so obvious other than the smoother curve for 96 realizations. In fact, in presence of large number of observations 48 realizations seem to be sufficient to perform a good assimilation whereas doubling the number of realizations does not increase the EnKF performance significantly. The graph in figure 7.1 visualizes the results in table 7.1.

Figures 7.2, 7.3 and 7.4 present the log permeability graphs for visual comparison.

7.2 Performance of the parallel algorithm

Initial tests and measurements were carried out in a LAN cluster with heterogeneous processing speed and memory size configurations. Table 7.2 lists the specifications for the 16 computers used. For the single processor run we used the 2.66 GHz Intel(R) Core(TM)2 Quad CPU Q8400 machine with 6MB cache and 4 GB main memory. This machine has roughly the average configuration of the available hardware. However, the machine with the best available configuration 3.33 GHz Intel(R) Core(TM) i5 660 with 4 MB cache and 8 GB main memory was used only for the run with 16 processors, so that the combinations of processors for different experiments with varying processor numbers have a well balance of the hardware configurations. The interconnection network is switch based with data rate 100 Mbps.

Then measurements were taken in the SARA Lisa cluster having homogeneous hardware and faster interconnection network. In the Lisa cluster we used 8 core processors with 2.26 GHz speed, 8 MB cache and 24 GB main memory, and 4x DDR Infiniband network with bandwidth 1600 MB/sec and Latency < 6 µsec.

In following discussion the forecast time, analysis time and the total time are related as

\[
\text{total time} = \text{initialization time} + \text{forecast time} + \text{analysis time}. \quad (7.1)
\]

So we should be aware that total time is not the sum of the forecast time and the
Figure 7.1: rms differences of log permeability over time

Table 7.2: Hardware specification of the LAN cluster for initial tests and measurements

<table>
<thead>
<tr>
<th>Processor Name</th>
<th>Speed (GHz)</th>
<th>Cache</th>
<th>RAM</th>
<th># Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel(R) Core(TM)2 Duo E8400</td>
<td>3.00</td>
<td>6 MB</td>
<td>4 GB</td>
<td>3</td>
</tr>
<tr>
<td>Intel(R) Core(TM)2 Duo E4500</td>
<td>2.20</td>
<td>2 MB</td>
<td>8 GB</td>
<td>7</td>
</tr>
<tr>
<td>Intel(R) Core(TM)2 Duo E8500</td>
<td>3.16</td>
<td>6 MB</td>
<td>4 GB</td>
<td>1</td>
</tr>
<tr>
<td>Intel(R) Core(TM)2 Quad Q8400</td>
<td>2.66</td>
<td>6 MB</td>
<td>4 GB</td>
<td>1</td>
</tr>
<tr>
<td>Intel(R) Core(TM)2 Quad Q8400</td>
<td>2.66</td>
<td>6 MB</td>
<td>8 GB</td>
<td>2</td>
</tr>
<tr>
<td>Intel(R) Core(TM)2 Duo E8400</td>
<td>3.00</td>
<td>6 MB</td>
<td>8 GB</td>
<td>1</td>
</tr>
<tr>
<td>Intel(R) Core(TM) i5 660</td>
<td>3.33</td>
<td>4 MB</td>
<td>8 GB</td>
<td>1</td>
</tr>
</tbody>
</table>

55
Figure 7.2: log permeability fields with 16 realizations: initial (top), after 6 assimilation steps in 3 years (middle), the true log permeability (bottom)
Figure 7.3: log permeability fields with 48 realizations: initial (top), after 6 assimilation steps in 3 years (middle), the true log permeability (bottom)
Figure 7.4: log permeability fields with 96 realizations: initial (top), after 6 assimilation steps in 3 years (middle), the true log permeability (bottom)
Table 7.3: Performance of the parallel implementation for 48 realizations in the LAN cluster

<table>
<thead>
<tr>
<th># processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>forecast time (h)</td>
<td>3.04</td>
<td>1.54</td>
<td>0.75</td>
<td>0.37</td>
<td>0.25</td>
<td>0.19</td>
</tr>
<tr>
<td>analysis time (h)</td>
<td>4.62</td>
<td>2.49</td>
<td>1.31</td>
<td>0.85</td>
<td>0.65</td>
<td>0.58</td>
</tr>
<tr>
<td>total time (h)</td>
<td>7.66</td>
<td>4.25</td>
<td>2.16</td>
<td>1.29</td>
<td>0.94</td>
<td>0.81</td>
</tr>
<tr>
<td>speedup (forecast)</td>
<td>1.97</td>
<td>4.06</td>
<td>8.11</td>
<td>12.02</td>
<td>16.23</td>
<td></td>
</tr>
<tr>
<td>speedup (analysis)</td>
<td>1.85</td>
<td>3.53</td>
<td>5.46</td>
<td>7.14</td>
<td>7.95</td>
<td></td>
</tr>
<tr>
<td>speedup (total)</td>
<td>1.80</td>
<td>3.54</td>
<td>5.96</td>
<td>8.12</td>
<td>9.51</td>
<td></td>
</tr>
<tr>
<td>efficiency (forecast) %</td>
<td>100</td>
<td>98.67</td>
<td>101.48</td>
<td>101.36</td>
<td>100.13</td>
<td>101.46</td>
</tr>
<tr>
<td>efficiency (analysis) %</td>
<td>100</td>
<td>92.64</td>
<td>88.13</td>
<td>68.25</td>
<td>59.48</td>
<td>49.69</td>
</tr>
<tr>
<td>efficiency (total) %</td>
<td>100</td>
<td>90.16</td>
<td>88.59</td>
<td>74.51</td>
<td>67.69</td>
<td>59.41</td>
</tr>
</tbody>
</table>

Table 7.4: Performance of the parallel implementation for 48 realizations in the SARA Lisa cluster

<table>
<thead>
<tr>
<th># processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>forecast time (h)</td>
<td>2.99</td>
<td>1.55</td>
<td>0.75</td>
<td>0.39</td>
<td>0.20</td>
<td>0.13</td>
<td>0.06</td>
</tr>
<tr>
<td>analysis time (h)</td>
<td>3.84</td>
<td>2.08</td>
<td>0.94</td>
<td>0.44</td>
<td>0.21</td>
<td>0.18</td>
<td>0.09</td>
</tr>
<tr>
<td>total time (h)</td>
<td>6.85</td>
<td>3.65</td>
<td>1.71</td>
<td>0.84</td>
<td>0.42</td>
<td>0.33</td>
<td>0.20</td>
</tr>
<tr>
<td>$S_P$ (forecast)</td>
<td>1.93</td>
<td>3.97</td>
<td>7.73</td>
<td>15.34</td>
<td>23.01</td>
<td>47.86</td>
<td></td>
</tr>
<tr>
<td>$S_P$ (analysis)</td>
<td>1.84</td>
<td>4.10</td>
<td>8.76</td>
<td>17.97</td>
<td>21.86</td>
<td>42.17</td>
<td></td>
</tr>
<tr>
<td>$S_P$ (total)</td>
<td>1.88</td>
<td>4.00</td>
<td>8.19</td>
<td>16.22</td>
<td>20.94</td>
<td>34.43</td>
<td></td>
</tr>
<tr>
<td>$E_P$ (forecast) %</td>
<td>100</td>
<td>96.44</td>
<td>99.14</td>
<td>96.66</td>
<td>95.85</td>
<td>95.88</td>
<td>99.70</td>
</tr>
<tr>
<td>$E_P$ (analysis) %</td>
<td>100</td>
<td>92.25</td>
<td>102.47</td>
<td>109.47</td>
<td>112.31</td>
<td>91.08</td>
<td>87.85</td>
</tr>
<tr>
<td>$E_P$ (total) %</td>
<td>100</td>
<td>93.92</td>
<td>99.91</td>
<td>102.37</td>
<td>101.36</td>
<td>87.26</td>
<td>71.72</td>
</tr>
</tbody>
</table>

analysis time.

Tables 7.3 and 7.5 show the time measurements, corresponding speedup and parallel efficiency with 48 and 96 realization in the LAN cluster. The graph in figures 7.5 and 7.7 show the gained speedup with initial measurements on the LAN cluster with 48 and 96 realizations. Tables 7.4 and 7.6 list the time measurements, corresponding speedup and parallel efficiency with 48 and 96 realization in the SARA Lisa cluster. Figures 7.6 and 7.8 depict the speedup in the SARA Lisa cluster.

7.2.1 Difference in speedup: forecast vs analysis step

We said several times that the forecast step is embarrassingly parallel and need no intermediate inter process communication and hence the interconnection network has absolutely no role in the speedup of this step and the speedup should follow
Figure 7.5: Speedup of the parallel implementation for 48 realizations in the LAN cluster for different parts

Table 7.5: Performance of the parallel implementation for 96 realizations in the LAN cluster

<table>
<thead>
<tr>
<th># processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>forecast time (h)</td>
<td>6.22</td>
<td>3.03</td>
<td>1.52</td>
<td>0.98</td>
<td>0.65</td>
<td>0.49</td>
</tr>
<tr>
<td>analysis time (h)</td>
<td>4.13</td>
<td>2.66</td>
<td>1.41</td>
<td>0.78</td>
<td>0.61</td>
<td>0.55</td>
</tr>
<tr>
<td>total time (h)</td>
<td>10.37</td>
<td>6.11</td>
<td>3.15</td>
<td>1.77</td>
<td>1.29</td>
<td>1.06</td>
</tr>
<tr>
<td>speedup (forecast)</td>
<td>1</td>
<td>2.05</td>
<td>4.09</td>
<td>6.37</td>
<td>9.52</td>
<td>12.68</td>
</tr>
<tr>
<td>speedup (analysis)</td>
<td>1</td>
<td>1.55</td>
<td>2.92</td>
<td>5.30</td>
<td>6.73</td>
<td>7.46</td>
</tr>
<tr>
<td>speedup (total)</td>
<td>1</td>
<td>1.70</td>
<td>3.29</td>
<td>5.84</td>
<td>8.04</td>
<td>9.76</td>
</tr>
<tr>
<td>efficiency (forecast) %</td>
<td>100</td>
<td>102.74</td>
<td>102.31</td>
<td>79.68</td>
<td>79.35</td>
<td>79.25</td>
</tr>
<tr>
<td>efficiency (analysis) %</td>
<td>100</td>
<td>77.71</td>
<td>73.02</td>
<td>66.23</td>
<td>56.05</td>
<td>46.65</td>
</tr>
<tr>
<td>efficiency (total) %</td>
<td>100</td>
<td>84.90</td>
<td>82.37</td>
<td>73.05</td>
<td>67.01</td>
<td>60.98</td>
</tr>
</tbody>
</table>
Figure 7.6: Speedup of the parallel implementation for 48 realization on SARA Lisa cluster for different parts

Table 7.6: Performance of the parallel implementation for 96 realizations in the SARA Lisa cluster

<table>
<thead>
<tr>
<th># processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>forecast time (h)</td>
<td>6.10</td>
<td>3.03</td>
<td>1.54</td>
<td>0.80</td>
<td>0.37</td>
<td>0.26</td>
<td>0.12</td>
</tr>
<tr>
<td>analysis time (h)</td>
<td>3.72</td>
<td>1.91</td>
<td>1.00</td>
<td>0.43</td>
<td>0.21</td>
<td>0.16</td>
<td>0.09</td>
</tr>
<tr>
<td>total time (h)</td>
<td>9.84</td>
<td>4.95</td>
<td>2.58</td>
<td>1.24</td>
<td>0.62</td>
<td>0.46</td>
<td>0.24</td>
</tr>
<tr>
<td>$S_P$ (forecast)</td>
<td>1</td>
<td>2.02</td>
<td>3.97</td>
<td>7.63</td>
<td>16.43</td>
<td>23.43</td>
<td>49.93</td>
</tr>
<tr>
<td>$S_P$ (analysis)</td>
<td>1</td>
<td>1.95</td>
<td>3.74</td>
<td>8.76</td>
<td>17.44</td>
<td>22.79</td>
<td>41.09</td>
</tr>
<tr>
<td>$S_P$ (total)</td>
<td>1</td>
<td>1.99</td>
<td>3.81</td>
<td>7.91</td>
<td>15.80</td>
<td>21.46</td>
<td>41.67</td>
</tr>
<tr>
<td>$E_P$ (forecast) %</td>
<td>100</td>
<td>100.83</td>
<td>99.26</td>
<td>95.33</td>
<td>102.70</td>
<td>97.62</td>
<td>104.02</td>
</tr>
<tr>
<td>$E_P$ (analysis) %</td>
<td>100</td>
<td>97.57</td>
<td>93.50</td>
<td>109.50</td>
<td>109.00</td>
<td>94.94</td>
<td>85.61</td>
</tr>
<tr>
<td>$E_P$ (total) %</td>
<td>100</td>
<td>99.44</td>
<td>95.19</td>
<td>98.81</td>
<td>98.75</td>
<td>89.40</td>
<td>86.82</td>
</tr>
</tbody>
</table>
Figure 7.7: Speedup of the parallel implementation for 96 realizations in the LAN cluster for different parts
Figure 7.8: Speedup of the parallel implementation for 96 realization on SARA Lisa cluster for different parts
the ideal case. On the other hand in the computation of SVD and matrix-matrix multiplication, naturally we need a huge number of intermediate inter process communication and this implies that the speedup in the analysis step will be less than that of the forecast step. This is observed in all the experimental results and corresponding figures.

7.2.2 Super-linear speedup

In both the clusters super-linear speedup is observed for the forecast step. In the SARA Lisa cluster we observe this also for the analysis step. We described in chapter 2 that this occurs when number of processors becomes so large that their combined cache memory size is enough to hold the required data which is impossible for a single or small number of processors. In the Lisa cluster every processor has a cache memory of 8 MB, whereas in the LAN cluster the maximum available cache memory size is 6 MB and the minimum is 2 MB. The shortfall in the cache memory size and the mentioned slow interconnection network do not allow speedup to be super-linear for the analysis step in the LAN cluster.

However, it is important to note here, the forecast time for the same realization may vary a little bit on different runs. In our implementation of the analysis step, we need to generate random Gaussian noise. The system time works as the seed for the random number generators. Thus, noise generated by different runs of the random number or Gaussian noise generator are not same. This may result in slightly varying assimilated parameters on different runs of the same realization. So other than the larger combined cache memory effect, also this may contribute for super-linear speedup for the forecast step on all clusters.

7.2.3 Parallel scalability

For both 48 and 96 realizations on both the clusters, the efficiencies of the forecast steps almost always follow the linear speed up. This implies that the forecast step is strongly scalable regardless of the interconnection network of the cluster. In the forecast step with 96 realization the problem size is double of that with 48 realizations. This means the forecast step is also weakly scalable.

However, for both 48 and 96 realizations on both the cluster the efficiencies of the analysis steps gradually decrease with increased number of processors. This implies that the analysis step is not strongly scalable. For proper scaling of SVD computation and matrix-matrix multiplication, ScaLAPACK suggests at least a $1000 \times 1000$ matrix per processor after data distribution. By this formulation, 25 is the roughly calculated upper limit of number of processors for the model we are working with to get good efficiency for the analysis step. We observe both for 48 and 96 realizations, on the SARA Lisa cluster the efficiency is well above 90% for up to 24 processors. We said earlier that increasing the number of realizations does not increase the problem size for the analysis step. To increase the problem size
for the analysis step we need increased number of observations. So weak scaling
is not relevant in this context.

### 7.2.4 Difference in performance: 48 realizations vs 96 realizations

With more number of realizations the problem size for the forecast step increases proportionally. These are observed in the figures 7.9 and 7.10. In both cases, the forecast time for 96 realizations is almost double of that for 48 realizations. On the other hand, with more number of realizations, the problem size for the analysis part remains almost the same, as the dimension of matrix in the SVD computation and the large matrix multiplication immediately after this remain same (m×m or 5061×5061 for our working model). We observe in the figures that the analysis time remains almost the same for both 48 and 96 realizations.

However, some small matrices’ size becomes double or quadruple with double number of realizations. Though the computation time for operations involving these matrices are not very significant, we observe that in most of the cases, the analysis time decreases with increased number of realizations. In the last chapter we described that we used two types of block lengths for ScaLAPACK operations. We used a block length of 64 for long dimensions (with length 5061 or 5049) and a block length of \( \frac{N}{nproc} \) for short dimensions (with length 48 or 96), where \( N \) is the number of realizations and \( nproc \) is the number of processors used. With increased number of realizations, we can use a bit more larger block length, closer to the optimal block length 64, for the short dimensions which lead to some performance gain, as seen in the figures.

### 7.2.5 Difference in speedup and performance: LAN cluster vs SARA Lisa cluster

The forecast step needs no intermediate inter process communication. So the speedup of this step should not be affected by the interconnection network of the used cluster. We see in the figures, for both the clusters the speedup of the forecast step follows almost the ideal case. However, as SARA Lisa uses faster interconnection network, the effect of the intermediate inter process communications of the analysis step should be less severe to it, compared to the LAN cluster and we follow from the figures that for the SARA Lisa cluster, the analysis step speedup is slightly less than the forecast step speedup. We see in the figure 7.11 the forecast time is almost equal in both clusters, but as the SARA Lisa cluster uses faster interconnection network, it consumes very less time for analysis step compared to the LAN cluster.

### 7.2.6 Effect of ScaLAPACK process grid

Process grid orientation is an important performance issue for ScaLAPACK operations as the number and amount of internal intermediate inter process communic-
Figure 7.9: Execution time of the forecast and analysis step for 48 and 96 realizations on the LAN cluster
Figure 7.10: Execution time of the forecast and analysis step for 48 and 96 realizations on the SARA Lisa cluster
Figure 7.11: Comparison of execution time of the forecast and analysis step for 48 realizations on different clusters
Table 7.7: Execution time (in minute) of the analysis step for different process grid orientation for 48 realizations

<table>
<thead>
<tr>
<th>process grid</th>
<th>LAN cluster</th>
<th>SARA Lisa cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x16</td>
<td>77.43</td>
<td>15.97</td>
</tr>
<tr>
<td>2x8</td>
<td>44.48</td>
<td>13.78</td>
</tr>
<tr>
<td>4x4</td>
<td>35.66</td>
<td>12.83</td>
</tr>
</tbody>
</table>

The performance issue of the process grid orientation should be more vital for slow interconnection networks. Table 7.7 lists the various analysis times for possible 3 process grid arrangements of 16 processors on both the clusters used. Here we compare only the analysis time as only this will be affected by the process grid arrangements. Figure 7.12 depicts the data in the table. We observe that the $1 \times 16$ grid, which is the worst arrangement compared to a square grid arrangement, shows the worst performance whereas the $4 \times 4$ grid, the best arrangement, shows the best performance. As we said, process grid arrangement affects the intermediate internal inter process communications of the ScaLAPACK operations, hence the performance of the LAN cluster with slow interconnection network is affected severely.

### 7.2.7 Effect of ScaLAPACK block size

ScaLAPACK logically divides the data in the original matrix in two dimensional blocks and distributes the matrix in units of these blocks among the available processors in the process grid. A block with length and width 1, that is a block with just 1 element ensures the highest load balancing of the ScaLAPACK operations but increases the need for intermediate internal inter process communication for ScaLAPACK operations. So there is always a trade-off between load balancing and amount of data communication for the optimal block length and width which ensures the best performance. ScaLAPACK suggests 64 to be the optimal choice. However, at present with larger cache memory size, the optimal block length may vary for different applications. Table 7.8 lists the analysis time for different block lengths on both the clusters. We applied this only to the long dimensions (with length 5061 or 5049 for the used model) of the matrices described earlier, as short dimensions (with length $N/n_{proc}$) of matrices are very short to apply the optimal choice of the block lengths. Here also the block lengths affect only the analysis step and we compare only with respect to the analysis time. Figure 7.13 depicts the data in the table. We observe that varying lengths of the block has a less affect on the performance compared to the variation of process grid arrangements. Also here the LAN cluster with slow interconnection network is more vulnerable to the
Figure 7.12: Execution time of the analysis step for different process grid orientation for 48 realizations
Table 7.8: Execution time (in minute) of the analysis step for different block sizes for 48 realizations

<table>
<thead>
<tr>
<th>Block Length</th>
<th>SARA Lisa Cluster</th>
<th>LAN Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14.71</td>
<td>37.99</td>
</tr>
<tr>
<td>16</td>
<td>12.95</td>
<td>35.83</td>
</tr>
<tr>
<td>32</td>
<td>12.84</td>
<td>36.12</td>
</tr>
<tr>
<td>64</td>
<td>12.83</td>
<td>35.66</td>
</tr>
<tr>
<td>128</td>
<td>13.14</td>
<td>37.40</td>
</tr>
<tr>
<td>256</td>
<td>14.81</td>
<td>37.36</td>
</tr>
<tr>
<td>512</td>
<td>13.23</td>
<td>40.15</td>
</tr>
</tbody>
</table>

variation in the block lengths. In the LAN cluster block length of 64 gives the best performance and 512 gives the worst one. In the SARA Lisa cluster block length of 1 and 256 seem to be the worst choices whereas 64 gives the best performance, though 32 seems to be very close to the best choice.

7.2.8 Effect of non blocking communication

Data communication is always considered a bottleneck for parallel computing. One remedy is communication and computation overlapping through non blocking communications. We implement this approach in our application for better performance. However, it is important to note here that the four well model, which we use for our application, works with 5061 observations. The large matrices of dimension $5061 \times 5061$ need no communication as their construction and further use are done locally on each processor. However, in each assimilation step, we need communication of 3 matrices of dimension $5061 \times N$, where $N$ is the number of realizations. This implies that for the used model, data communication is not a very serious issue and we get very small difference between the performance of the blocking and non blocking communication. In order to implement the blocking counterpart of our application, we replace the `MPI_Send` with `MPI_Send`, `MPI_Recv` with `MPI_Recv` and omit the `MPI_Wait` statements in our application to keep all settings, sequence of computation, correctness and above all the simplicity. Table 7.9 lists the execution time of the analysis steps in the LAN cluster for 48 realizations and in the SARA Lisa cluster for 96 realizations. We use 96 realizations for the Lisa cluster as the interconnection network for this is very fast and we need some good figures to feel the difference. Figure 7.14 shows the data in the above table.

In the SARA Lisa cluster we observe the above mentioned claims. But in the LAN cluster we get some very unexpected results. Instead of reducing the analysis time, we see, non-blocking communication has increased the time. This may be an effect of the particular MPI implementation on the LAN cluster and also of the heterogeneous hardware components which may lead to unexpected delay for the
Figure 7.13: Execution time of the analysis step for different block sizes for 48 realizations
Table 7.9: Execution time (in minutes) of the analysis step for blocking and non-blocking data communication

<table>
<thead>
<tr>
<th></th>
<th>LAN cluster (48 realizations)</th>
<th>SARA Lisa cluster (96 realizations)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># p</td>
<td>blocking</td>
</tr>
<tr>
<td>2</td>
<td>149.46</td>
<td>192.82</td>
</tr>
<tr>
<td>4</td>
<td>78.56</td>
<td>101.90</td>
</tr>
<tr>
<td>8</td>
<td>50.71</td>
<td>62.05</td>
</tr>
<tr>
<td>12</td>
<td>38.80</td>
<td>46.19</td>
</tr>
<tr>
<td>16</td>
<td>34.83</td>
<td>35.66</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

synchronization. Anyway, for the LAN cluster we used execution time data with the blocking communications, as it performs better, in tables 7.5 and 7.7 and the corresponding figures.
Figure 7.14: Execution time of the analysis step for blocking and non-blocking data communication on the LAN cluster (top) and SARA Lisa cluster (bottom)
Chapter 8

Conclusion and Future Work

8.1 Conclusion

We mentioned in chapter [1] that oil companies are struggling to meet the increasing demand for energy. It is important, therefore, to develop new technologies and strategies that will optimize recovery from existing fields. Whatever the technology or strategy is, it requires better knowledge and characterization of the reservoir. Improvement of the reservoir description is achieved through adjustment of uncertain parameters such as permeability, using history matching or data assimilation. The ensemble Kalman filter (EnKF), a useful tool for data assimilation, is a very time consuming process. Reducing the data assimilation time in the overall reservoir management cycle is an important issue. The inherent parallelism in EnKF strongly demands the parallelization of EnKF.

In this thesis we described the parallelization of the whole EnKF process for oil reservoirs in presence of sparse measurement data and dense time-lapse seismic data. In other parallelizations of EnKF for a variety of applications, the forecast step is parallelized totally and the analysis step is parallelized depending on some form of localisation. Past parallel implementations of EnKF for oil reservoirs parallelize only the forecast step and at best the analysis step is parallelized partially. The parallel performance of our implementation of the EnKF is satisfactory compared to the other implementations of parallel EnKF for oil reservoirs. In [29] with 100 realizations, parallel efficiencies of 50.8% and 47.2% on two different clusters with 50 processors are reported. On the SARA Lisa cluster with 48 processors for 96 realizations we achieved 86.8% efficiency. In [23] with 200 realizations on 16 processors a parallel efficiency of 76.8% is reported, whereas for our implementation, the parallel efficiency with 16 processors on SARA Lisa cluster for 96 realization is 98.8%. We said earlier that both [29] and [23] work with small measurement data from the wells and the analysis step is not that much vital and these mostly parallelize the forecast step only.

Below we summarize other important issues relevant to our implementation of parallel EnKF for oil reservoirs.
1. Parallel EnKF, like the traditional serial EnKF, can do data assimilation by variance minimization and gives better predictions utilizing past predictions and the past and present measurements. Also a large number of realizations do better assimilation than a very small number of realizations, however, in presence of large number of observations when there are sufficiently large number of realizations, increasing the number of realizations does not improve data assimilation performance significantly.

2. The forecast step shows linear speedup in any type of cluster and strongly scales whereas the analysis step shows sub-linear speedup and has an upper limit of number of processors (25 for our model with 5061 observations), beyond which the scaling degrades.

3. Super-linear speedup may be observed for the forecast step in any type of cluster whereas it may be observed for the analysis step in very fast interconnection clusters for small number of processors (from 4 or 8 up to 16).

4. Increasing the number of realizations proportionally increases the problem size for the forecast step and the execution time is proportionally increased, however, the problem size for the analysis step remains almost the same and the execution time does not change significantly.

5. The forecast step is embarrassingly parallel and need no intermediate inter process communication, and hence the execution time of this step is almost indifferent of the hardware and interconnection network of the cluster used. But the analysis step requires huge number of intermediate inter process communication and the execution time significantly depends on the interconnection network of the used cluster. A cluster with fast interconnection can complete the analysis step faster.

6. ScaLAPACK grid orientation has a significant effect on the analysis step. As the grid orientation determines the number and amount of data communication, it mostly affects the performance of a cluster with slow interconnection network.

7. ScaLAPACK block lengths have a minor effect on the analysis time. Varying block lengths affect the number and amount of data communication. Hence also here the performance of a cluster with slow interconnection network is mostly affected.

8. Non blocking communication with small volume of data insignificantly improves the performance. But this should be of great benefit with models giving very large number of observations and larger number of realizations.
8.2 Future work

Implementation on GPU clusters

Further performance gain can be achieved by exploiting GPGPU or GPU computing on each node. GPGPU stands for General-Purpose computation on Graphics Processing Units. It is the use of a GPU as a co-processor to accelerate CPUs for general-purpose scientific and engineering computing. The GPU, with the massively parallel processing power, accelerates applications running on the CPU by offloading some of the compute-intensive and time consuming portions of the code where the rest of the application still runs on the CPU [5].

To get the maximum potential performance from the GPU hardware, we have to consider four key issues, namely: Memory optimizations, Execution configuration optimization, Instruction optimizations and Control flow [25]. To get good performance from GPUs the problem size needs to be large enough, so that the performance benefits outweigh the inherent overheads for GPU computing. In our EnKF analysis step, only the SVD computation and the matrix multiplication immediately after this are suitable for execution on GPUs due to the sizes of the operand matrices. To keep all other settings of our implementation we need to manually implement the GPU versions of the routines pdgesvd and pdgemm for SVD computation and matrix-matrix multiplication respectively, maintaining all the syntaxes and semantics of ScALAPACK and need to call the GPU versions instead of the original ones.

However, as we are using simsim for forecast as a third party software, total implementation of the forecast step for GPU would be very complex. Rather we should look for some functions which consume the most of the computation time and can be easily converted to CUDA kernels and simply can be called from the original MATLAB code using MATLAB parallel computing toolbox (PCT). By MATLAB profiling, it is observed that the functions simsim_trans and simsim_rel consume about 90% and 7% of the overall computation time of simsim respectively. The first function computes the oil and water transmissibility matrices based on some graph algorithms and the second one computes relative permeabilities using Corey functions and does some truncation operations. Fortunately, both of them nicely fit for GPU implementation because of their tasks. However, it is also observed that for a 10 years simulation and runtime of around 290 seconds on a 2.4 GHz Core 2 Duo and 2 GB RAM machine, these functions are called 7042 and 7227 times respectively. These data represents that currently these functions roughly run for 37064 and 2809 micro seconds respectively for a single call. With a rough estimation of 20 micro seconds for a CUDA kernel launch time, above results show potential performance gain by GPU implementation of these two functions. Again there are issues regarding data upload (GPU memory write) before the kernel execution and download (GPU memory read) after the kernel execution. These will certainly add additional overheads. Moreover GPUs are compatible with double precision floats on compute capability 1.3 or higher GPUs.
only and double precision operations are very slow due to the presence of only a single double precision unit per multiprocessor. Double precision operations can be executed with single precisions with loss of accuracies. These indicate that the performance and accuracy would certainly depend on the practical implementations of the kernels and can be an independent research issue.

However [24], [10], [33] and [11] describe some independent implementations of reservoir simulation on GPUs and may be used in place of simsim to work on GPU clusters.

Dynamic load balancing

Another research direction may be to attempt load balancing dynamically for the forecast step. Initially equal number of realizations can be assigned to each processor, and the completion time for each realizations can be tracked on different processors. In subsequent steps instead of equalization based on just the number, realizations may be equalized for completion time of the last step, as is done in most of the load balancing issues. However to do this a lot of complex data for the model, control and parameter information need to be communicated back and forth between MATLAB compiler runtime (MCR) and the simsimdriver as discussed in chapter 6 and shown in figure 6.3. Also major modifications to the wrapper functions initSimsim and runSimsim is required for that. Moreover some additional MATLAB functions may be needed.

Verifying weak scaling

We could not verify weak scaling of the analysis step for our application as we worked with a fixed model with constant number of observations. Verification of weak scaling for the analysis step can be done by using a model which allows varying number of observations.

Other EnKF schemes

EnKFs are divided into 2 broad categories: stochastic and deterministic, both having inherent advantages and disadvantages. The typical EnKF, which we parallelized, is a stochastic one. Some closely related schemes for ensemble based filters are: double EnKF (DEnKF), singular evolutive interpolated Kalman filter (SEIK), ensemble adjustment Kalman filter (EAKF), serial ensemble square root filter (ESRF), maximum likelihood ensemble filter (MLEF), deterministic EnKF (DEnKF), ensemble randomised maximum likelihood filter (EnRML), ensemble transform Kalman filter (ETKF), left-multiplied ensemble square root filter (LESRF), ensemble optimal interpolation (EnOI), etc. Depending on the applicability to oil reservoirs they can be parallelized, with little modifications to our developed application, and compared for different parallel performance metrics.
Bibliography


