PARALLELISM IN ADAPTIVE MULTIGRID SOLVERS
PARALLELISM IN ADAPTIVE MULTIGRID SOLVERS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus,
Prof. Drs. P.A. Schenck,
in het openbaar te verdedigen
ten overstaan van een commissie
door het College van Dekanen
daartoe aangewezen,
op 22 februari 1990 te 16.00 uur
door

PETER HANS MICHIELSE

egeboren te Papendrecht,
wiskundig ingenieur.
Dit proefschrift is goedgekeurd door de promotor Prof. Dr. H.A. van der Vorst.

Promotiecommissie:

Prof. Dr. J. Arbocz
Prof. Dr. Ir. L. Dekker
Prof. Dr. Th. J. Dekker
Prof. Dr. Ir. P.M. Dewilde
Dr. G.H. Schmidt
Prof. Dr. H.A. van der Vorst
Prof. Dr. Ir. P. Wesseling

This work has been supported by the Koninklijke/Shell Exploratie en Productie Laboratorium (KSEPL, Shell Research b.v.).
VOORWOORD

Aan de totstandkoming van dit proefschrift heeft een aantal mensen in meerdere of mindere mate bijgedragen. Allereerst is daar mijn promotor, Prof. Dr. H.A. van der Vorst, wiens aanwijzingen en suggesties van groot belang zijn geweest voor het gereedkomen van dit werk. Zijn enthousiasme was telkens weer stimulerend, waarvoor ik hem dan ook zeer erkentelijk ben.

Het onderwerp van dit proefschrift is afkomstig van de afdeling General Research van het Shell-laboratorium (KSEPL) in Rijswijk. Dr. G.H. Schmidt, Dr. Ir. D.T. van Daalen en Drs. J.A. Meyerink van deze afdeling hebben mij wegwijks gemaakt in de numerieke reservoir simulatie en fungerden als discussiepartners tijdens de werkbesprekingen die wij hadden. Ook hen geldt mijn dank. Het project, waarvan dit proefschrift de neerslag is, is tevens gefinancierd door het KSEPL in Rijswijk.

Vervolgens wil ik Pauline Wilting bedanken voor haar assistentie bij het typen van het manuscript.

Tot slot bedank ik mijn collega's van de vakgroep Toegepaste Analyse van de Faculteit der Technische Wiskunde en Informatica voor hun interesse in mijn werk. Ook de belangstelling en morele steun van familie en vrienden heb ik altijd op prijs gesteld.
aan Lonneke
CONTENTS

1 GENERAL INTRODUCTION
   1.1 Introduction 1
   1.2 Introduction to reservoir engineering 3
   1.3 Parallel computers 4

2 PARALLEL SOLUTION OF BANDED LINEAR SYSTEMS
   2.1 Introduction: recursive doubling and cyclic reduction 12
   2.2 The partition method for tridiagonal linear systems 17
   2.3 The modified partition method 21
   2.4 Experimental results and analysis 25
   2.5 Some comments on Amdahl's law 31

3 NUMERICAL RESERVOIR SIMULATION
   3.1 Introduction 36
   3.2 Discretization and datastructures 40
   3.3 The hyperbolic update 46
   3.4 The elliptic update 48

4 DOMAIN DECOMPOSITION AND PARALLEL COMPUTING
   4.1 Introduction 54
   4.2 Parallel multigrid: an overview 57
   4.3 Parallel domain decomposition 63
   4.4 Additional datastructures and agglomeration processes 69
   4.5 Load-balancing 75
5 A PARALLEL HYPERBOLIC UPDATE
5.1 Introduction and sequential algorithm  80
5.2 Parallel hyperbolic algorithm  81
5.3 Implementation and experimental results: NCube  87
5.4 Implementation and experimental results: Alliant  101

6 A PARALLEL ELLIPTIC UPDATE
6.1 Introduction and sequential algorithm  112
6.2 Parallel elliptic algorithm  114
6.3 Implementation and experimental results: NCube  121
6.4 Implementation and experimental results: Alliant  139

7 CONCLUDING REMARKS  149

REFERENCES  153

SUMMARY  157

SAMENVATTING  159

CURRICULUM VITAE  161
1

GENERAL INTRODUCTION

§ 1.1 Introduction

For economic reasons, the subject of reservoir engineering is an important topic and is encountered in, e.g., the exploration and production of oil–reservoirs or in groundwater hydraulics. From a mathematical point of view, modelling of the fluid flow in reservoirs is a quite complicated, but also a challenging task. Numerical models have been developed in order to simulate the behaviour of reservoirs by computers. However, for accurate numerical computations large reservoir models are required, thus leading to large computing times. Reduction of these computing times can be realized through improvements in the underlying numerical methods, as well as through the use of fast vector– and parallel computers.

In this thesis the problem of numerical simulation of fluid flow in oil–reservoirs is considered. We will look for parallelism in the numerical model for oil–reservoir engineering, such that implementation of the parallel simulator leads to a significant reduction of execution times on commercially available parallel computers. It should be stressed that we are interested only in the underlying algorithms, especially in the modifications that are necessary to obtain efficient parallel algorithms, and not in specific parallel computers.

We have focussed on discovering and creating as much parallelism as possible in a realistic mathematical model of an oil–reservoir, which includes the features of adaptive grid refinement and multigrid. We have tried to get insight into fundamental restrictions for parallel implementations of the reservoir–model with adaptive grid refinement and multigrid. So far, in many parallel implementations these (complicating) factors are left out. We believe that our analysis for the complicated reservoir–model can also be used in less complex simulators, which do not apply adaptive grid refinement and multigrid.
It will be clear then that our results can be extrapolated to more processors, provided that the problem size is large enough compared with the number of processors.

Our parallel algorithms will be tested on machines that are generally regarded as representatives of two different classes of parallel computers. Attention will be paid to some well-known bottlenecks for efficient parallel execution, as there are synchronization overhead, sequential overhead, data transfer and their impact on parallel performance.

Before studying parallel aspects of reservoir engineering, some simpler problems (banded linear systems) are dealt with in chapter 2 of this thesis. In chapter 3, details about reservoir simulation will be given, whereafter parallel aspects of the numerical reservoir model are considered in chapters 4 to 6. We continue this chapter with a brief introduction to reservoir engineering and a few remarks about the computers on which our parallel algorithms have been tested.
§ 1.2 Introduction to reservoir engineering

Reservoir engineering is widely used in oil-industry to model the recovery of oil from reservoirs. In general, three phases are present in an oil-reservoir: gas, oil and water, which flow either miscible or immiscible through the reservoir. The reservoir itself can be viewed as a porous medium, formed of a rock, like sandstone. The oil is trapped in the microscopic pores of the rock and will flow through the rock only under the influence of extremely large pressure gradients. The heterogeneous nature of the rock in most reservoirs greatly complicates the mathematical modelling process.

Fluid motions in porous media are governed by fundamental laws that are based on conservation of mass, momentum and energy. Additional equations which must be specified are rate equations, usually a form of Darcy's law, and equations of state. Furthermore, in order to have a complete mathematical model, a description of the reservoir, initial conditions and boundary conditions must be specified.

To simulate the fluid flow in the reservoir, we are considering a numerical method which is based on a mixed finite element discretization of a system of partial differential equations that may be regarded as representative for the phenomena arising in reservoir simulation. A reservoir simulator can be viewed as an alternation of flash calculations and transport calculations on a computational grid. The flash calculations compute the equilibrium phase and are perfectly suited to parallel processing, since computations are restricted to individual grid blocks. During the transport calculations data exchange between neighbouring grid blocks is required, which means that the transport calculations are not a priori suited for efficient parallel computing. Therefore, for an efficient parallel implementation of the reservoir simulator, the transport calculations have to be considered carefully. Also because adaptive grid refinement and a multigrid solver are applied, the parallelization strategy is not obvious.

In chapter 3 we will discuss reservoir engineering in some more detail, including an outline of adaptive grid refinement and multigrid.
§ 1.3 Parallel computers

It has become clear for years now that the classical "Von Neumann" concept for computers can not answer the speed requests of scientific computing. For this reason, computer manufacturers have been looking for other ways to increase the computational speed of their computers. This has led to serious research on parallel computing, in a broad sense. In this section we briefly describe two classes of parallel computers which we will use for testing our parallel algorithms. We will focus on aspects that are relevant for numerical use and hence we will avoid technical details.

Roughly spoken, the types of rather widely used computers that support parallel processing can be subdivided into two classes [Flynn]. The first class, denoted by SIMD–computers (= Single Instruction Multiple Data), is characterized by the fact that one instruction performs many identical operations on multiple operands. E.g., two vectors (multiple data) are added elementwise (the same operation on each pair of elements). Many supercomputers are based upon this so–called pipeline concept. Examples of these (vector–)computers are the CRAY–1, CYBER 205, and the Japanese vectorcomputers. The parallelism exploited by these computers is known as fine–grained parallelism.

Real parallelism, in the sense that different instructions can be executed in parallel, is at present also used in order to further increase the computing speed. Such multiprocessor systems include, e.g., CRAY–XMP, Intel Vx, CONVEX C–240 and Alliant FX/8. Such machines should be classified as MIMD–machines of which the processors support pipelined operations themselves. The second class of parallel computers consists of these MIMD–computers (= Multiple Instruction Multiple Data). Hence, the parallelism exploited by MIMD–computers may be coarse–grained as well as fine–grained (per processor). Coarse–grained parallelism is defined as parallelism, in which the processors have their own, rather large subtasks of a complete job.

A further subdivision of MIMD–machines is obtained by classifying their memory management. When its processors are directly connected to one large, common memory, the computer is called a shared memory machine (i.e. the memory is shared by the processors; each processor has access to the complete memory). Examples are the CRAY X–MP, the Alliant FX–series and the Convex C2–series. On the other hand, opposed to shared memory machines, there are local memory machines in which each processor has its own local memory. This means that, if one processor needs data which are in the local memory of another processor, transfer of data between processors is required. Hence, for data transfer, physical connections between processors are necessary. For a
small number of processors (4, maybe 8), it is possible to connect all processors with each other. In case of more processors, full connection is technically more difficult to achieve.

In order to determine whether a parallel algorithm is an efficient parallel algorithm, the speed-up over the original algorithm is defined as:

**Definition 1.3.1:** The speed-up $S(p)$, with $p$ the number of processors, of a parallel algorithm for a problem $P$ is defined as

$$ S(p) = \frac{\text{execution time of original algor. for } P}{\text{execution time of parallel algor. for } P, \text{ on } p \text{ procs.}} $$

Definition 1.3.1 implies that the speed-up $S(p)$ of a parallel implementation on $p$ processors is bounded by $p$. If the upperbound $p$ is reached, then the sequential algorithm for problem $P$ has been split into $p$ independent subtasks, each of equal size. In many practical situations, however, the number of processors $p$ is an unachievable upperbound, since the parallel algorithm for $P$ is often computationally more complex. Furthermore, overhead, such as the initial distribution of data over the processors, synchronization of the processors or physical data transfer, has a negative impact on speed-up values.

Related to the speed-up is the efficiency $E(p)$, which expresses how well the processors are utilized:

**Definition 1.3.2:** The efficiency $E(p)$ is defined as

$$ E(p) = \frac{S(p)}{p} $$

In section 1.1 we argued that we are interested only in algorithmic aspects of parallel computing. At the start of the project, January 1986, shared memory machines like the CRAY X–MP were most likely to be available, but our parallelization strategy does not assume a shared or local memory specifically. The only restriction we have accepted in our approach is that the number of processors of the parallel computer is small compared with the problem size of the reservoir simulator. To test our parallel algorithms we use
machines that are generally regarded as representatives for local memory and shared memory MIMD—computers. In the remainder of this section we briefly discuss an NCube/four and an Alliant FX/4, on which our experiments will be carried out.

Local memory machines: NCube/four

We will discuss an NCube/four as an example of a local memory parallel computer. Experiments on local memory machines, reported in the remainder of this thesis, have been carried out on an NCube/four. An NCube/four consists of 16 processors, whereas the largest configuration (NCube/ten) is equipped with 1024 processors. As was noted before, local memory machines require a connection network to connect the processors. This is shown schematically in figure 1.3.1:

![Diagram](network_diagram)

**Figure 1.3.1:** General local memory architecture, with p for processor and m for local memory.

The most popular connection network is the hypercube structure, which is also applied in the NCube systems. For this reason, we regard the NCube/four as representative for a class of local memory machines. If \( d \) denotes the dimension of the hypercube, then the system consists of \( 2^d \) processors. A hypercube of dimension \( d \) is formed by two hypercubes of dimension \( d-1 \), such that the corresponding nodes of the two subcubes are connected. Furthermore, one node of a hypercube of dimension \( d \) is directly connected to \( d \) other nodes. Our hypercube has dimension 4. It is also possible to define and use hypercubes of smaller dimensions, since they are easily allocated from the full (dimension 4) system. A node of the hypercube represents a single processor, while
connections of the nodes of the hypercube represent interprocessor connections. Figure 1.3.2 shows the hypercube network for some small values of $d$:

![Hypercube diagrams](image)

$d = 0$  
$d = 1$  
$d = 2$  
$d = 3$

**Figure 1.3.2:** Some hypercubes of dimension $d$.

For more details about hypercubes, we refer to [SaS1].

For many local memory systems (or: distributed memory systems), there is a so-called host processor, which serves as an interface between the user and the actual processors of the parallel computer. In some cases, one of the processors plays also the role of the host. For the NCube systems, the host is a different computer (specifically, the NCube/four has a PC–AT as host computer).

In general, when programming for the NCube, one has to write at least two programs: one for the host and at least one for the nodes. The host program starts with loading the node program (or node programs, since the nodes do not necessarily have to execute identical programs) on the nodes, which leads to data transfer from the host to the nodes. If the node programs need input information, then the host has to be instructed to send data to the nodes, possibly different data for each node. The node program starts execution on each node, and after its completion, it sends its relevant data to the host.

Each node starts its node program as soon as it has arrived from the host. Data is received from the host and each node executes its program. For many applications, communication (e.g., exchange of data) between the nodes, apart from host–nodes communication, is required to guarantee correct execution of the program. This communication, which implies physical data transfer, is handled by a send–receive mechanism: node A sends a message to node B, which contains the required information, the number of the destination processor (node B) and some identification number. Node B receives the message, if the source processor (node A) and the identification number are known. Hence, one has to take care for matching send and receive messages, in order
to avoid the notorious deadlock situations, in which a node waits for a message that will never arrive.

In order to determine the speed-up of parallel implementations over the original implementation, it is necessary to carry out accurate timing measurements. Each node starts its clock as soon as all other nodes have received their nodeprograms (including the required data) from the host. This is achieved in the following way: the node which is assumed to be the last one to receive its data, sends a message to all other nodes upon which their clocks are started, whereafter it starts its own clock. At the end of the execution, a similar procedure is applied: the node which is believed to be the last one to finish, sends all other nodes a message instructing them to stop their clocks, and then stops its own clock too. Hereafter, each node sends its output information to the host. This procedure does not take into account the time to load the nodeprograms on the nodes and the time for input and output communication between the host and the nodes. However, for practical applications as discussed in chapters 5 and 6, initial data are already available on the nodes, whereas output results are used as initial data for subsequent parts of the program.

The time for communication between nodes, and between nodes and host, during the computational process is included in the total computing time. For most local memory machines, the time for sending a message from one processor to another can be split into two parts. The first part represents a start-up time $\gamma$ for the complete message, while the second part reflects the costs of data transfer per element ($\delta$). Hence, the time $t_m$ required for sending a message of length $m$ elements is given by

$$t_m = \gamma + m\delta. \quad (1.3.1)$$

The hypercube connection network does not connect a processor to all other processors. Therefore, a distance between processors can be defined. Processors that are directly connected to each other are called neighbouring processors, and have distance one to each other. The distance between other processors is determined by the minimum number of processors that have to be passed for a message to travel from one processor to another processor. For the NCube/four, we measured for an ideal situation (two neighbouring processors): $\gamma \approx 0.44$ ms and $\delta \approx 0.022$ ms. It should be noted that for modern systems data transfer can overlap with computational work. For such systems data transfer does not have to be a bottleneck.

For more information about NCube systems, we refer to [NCSI]. More general information about local memory machines can be found in, e.g., [Care].
Shared memory machines: Alliant FX/4

The shared memory machines consist of a number of processors that are connected to one shared memory. Each processor has full access to this memory, and for that reason, shared memory machines do not have a network to connect the processors.

In more and more cases, the separate processors of shared memory machines are vectorprocessors. An example of such a computer is the CRAY X-MP, with a maximum of 4 vectorprocessors. The maximum number of processors connected to one shared memory is small at the moment, in the order of 8 or 16, say. However, currently shared memory computers of up to 64 processors are under design (CRAY-4). Another example of a shared memory machine, equipped with vectorprocessors is the Alliant FX computer, with up to 8 processors. The Alliant FX computers are so-called minisupercomputers (performance less than 100 Mflops). In the remainder of this thesis, experiments for shared memory machines are carried out on an Alliant FX/4, with 4 vectorprocessors, although our reservoir simulation model is not particularly suited for vectorcomputing, nor have we attempted to do so.

The programming for shared memory machines seems to be easier than it is for local memory machines. The operating system of the computer exploits parallelism by splitting relatively simple do-loops, whereafter the parts are distributed over the processors. For most shared memory machines, it is also possible to distribute independent parts of the job (so-called tasks) over the processors. E.g., it is possible to execute in parallel the same subroutine for different values of the subroutine parameters. For this kind of parallelism, however, the user should explicitly indicate those tasks that can be executed in parallel.

The fact that the processors are connected to one shared memory has some consequences. First, it is possible that data have already been changed by one processor, while the original values are still needed by another processor. Second, two or more processors may read data from or write data to the same memory address simultaneously, which may lead to different output for two runs of the same parallel program for identical input. This is caused by a different execution order in both runs. Sometimes this is acceptable in problems in which the order is not that important (in order to avoid synchronization overhead). E.g., for a Gauss-Seidel iteration process, a somewhat different order in general does not have too much impact on convergence rates. Third, the fact that more processors have access to the complete memory may also lead to memory bank conflicts, which may degrade the performance. Especially for vectorprocessors, it is known that memory bank conflicts are obstacles that may reduce the computing speed significantly.
The distribution of tasks over the processors leads to some additional work (overhead) for the operating system. The same is true after the completion of the tasks by the processor: synchronization between processors is required, again leading to overhead for the operating system. Timing measurements include this overhead, since distribution and synchronization points are inevitable for parallel execution. It is much easier to do timing measurements, as compared with local memory machines. Since there is only one program, operations to start and to stop the clock occur only at the start and at the end of the program, respectively.

For the Alliant FX—series, as well as for other MIMD shared memory machines, compiler directives are used to identify tasks of the job that can be run in parallel. It is also possible to force loops to be executed in parallel. The way parallelism will be exploited in our programs for the Alliant FX/4 is done by means of concurrent calling of subroutines. The compiler directive "cncall" is used, in combination with the compiler option "--recursive". For more details, we refer to [Alma].
PARALLEL SOLUTION OF BANDED LINEAR SYSTEMS

Abstract

In this chapter we will consider the efficient solution of bidiagonal and tridiagonal linear systems of equations. Tridiagonal systems can be solved in parallel with common techniques, such as recursive doubling and cyclic reduction. An alternative and more promising method is the partition method. This method may suffer from the amount of data transfer, that is required between the processors of a local memory computer. A modified version of the partition method seems to be more efficient in this respect. This was actually confirmed by experimental results, which have led to models that describe computational and communicational work quite satisfactorily. The experimental results for the modified partition method are used in a discussion on Amdahl’s law.
§ 2.1 Introduction: recursive doubling and cyclic reduction

Tridiagonal systems of linear equations (or, in general, banded systems) lie at the heart of many programs for scientific computing. Their solution is a computing time consuming kernel of various applications. With the development and availability of parallel and vector computers, new algorithms have appeared for solving systems of linear equations efficiently on such computers. In this chapter we will study some of these algorithms. The parallel solution of bidiagonal and tridiagonal systems will be considered in some detail. Since vector and parallel solution are closely related in many cases, vector aspects are discussed also.

In this section two methods for parallel solution of bidiagonal systems will be described: recursive doubling and cyclic reduction. In section 2.2 we will consider an alternative method, known as the partition method, while in section 2.3 the partition method is modified, in order to reduce communication between the processors. Experimental results are presented and discussed in section 2.4. Section 2.5 is devoted to more general aspects concerning parallel computing, for which the results of section 2.4 serve as examples.

Consider the n–th order bidiagonal system $Ax = b$:

$$
\begin{bmatrix}
1 \\
-a_2 & 1 \\
& -a_3 & 1 \\
& & & \ddots \\
& & & & \ddots \\
& & & & & -a_n & 1
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
\vdots \\
x_n
\end{bmatrix}
=
\begin{bmatrix}
b_1 \\
b_2 \\
b_3 \\
\vdots \\
b_n
\end{bmatrix}
\tag{2.1.1}
$$

The standard way of solving (2.1.1) is represented by the recursion formula

$$x_1 = b_1, \quad x_j = b_j + a_j \cdot x_{j-1}, \quad j = 2,3,\ldots,n. \tag{2.1.2}$$

The recursion formula (2.1.2) itself is not suitable for parallel or vector computing; $x_j$ can be computed only after $x_{j-1}$ is known. In order to obtain the solution of (2.1.1) by parallel computing techniques, some already classical methods have been developed.
In this section we will discuss two of these methods: recursive doubling [Ston], generalized in [DuRo], and cyclic reduction [LaVo].

The recursive doubling method for system (2.1.1) starts with the multiplication of (2.1.1) by the (bidirectional) matrix \(-A+2I\), in which I denotes the \(n\)-th order identity matrix. This leads to the system \(A'x = b'\):

\[
\begin{bmatrix}
1 & 0 & 1 \\
-a_2 & a_3 & 0 & 1 \\
0 & -a_4 & a_5 & 0 & 1 \\
\vdots & \ddots & \ddots & \ddots & \ddots \\
0 & \cdots & 0 & -a_{n-1} & a_n & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_n \\
\end{bmatrix}
=
\begin{bmatrix}
b_1 \\
b_2 + a_2 b_1 \\
b_3 + a_3 b_2 \\
b_4 + a_4 b_3 \\
b_5 + a_5 b_4 \\
\vdots \\
b_n + a_n b_{n-1} \\
\end{bmatrix}
\tag{2.1.3}
\]

After this first step, two independent bidiagonal systems are obtained, each of order \(\frac{n}{2}\). The first system only contains the odd numbered elements of \(x\) \((x_{2j-1}, j = 1, \ldots, \frac{n}{2})\), whereas the second system only contains the even numbered elements of \(x\) \((x_{2j}, j = 1, \ldots, \frac{n}{2})\). (For convenience, we will further assume \(n\) to be an integer power of 2.) The two independent bidiagonal systems can be solved in parallel on two processors, each by the recursion formula (2.1.2). The matrix multiplication itself can be executed in parallel as well, which means that this recursive doubling process is a completely parallel process for two processors.

Of course, this parallelization process can be continued by another matrix multiplication. If (2.1.3) is multiplied by \(-A' + 2I\), then 4 processors can be used to solve 4 independent bidiagonal systems, each of order \(\frac{n}{4}\), etcetera. After each matrix multiplication (i.e. each recursive doubling step) parallelism is increased by a factor of 2. In figure 2.1.1, the algorithm for the recursive doubling process is given (\(k\) denotes the number of recursive doubling steps). The second outer loop of the algorithm (of which the body is denoted by \(\Pi\), in figure 2.1.1) is equivalent to the solution of \(p\) independent bidiagonal systems and is therefore ideally suited for distribution over \(p\) processors (with \(p = 2^k\), or, \(k = \log(p)\)). The body \((I)\) of the first outer loop describes a matrix multiplication of the underlying system, and can be distributed over \(p\) processors as well.
\[ \text{for } i = 1, k \]
\[
\begin{align*}
\text{for } j = n, 2^{i-1} + 1, -1 & \quad b_j = b_j + a_j b_{j-2^{i-1}} \\
\text{for } j = n, 2^i + 1, -1 & \quad a_j = a_j a_{j-2^i-1}
\end{align*}
\]
\[ p = 2^k \]
\[ \text{for } i = 1, p \]
\[ \begin{align*}
\text{II} & \quad \text{for } j = p+i, n, p \\
& \quad b_j = b_j + a_j b_{j-p}
\end{align*}
\]

Figure 2.1.1: Recursive doubling algorithm, \( k \) steps.

In [Mic1] it is shown that the amount of parallel multiplications and additions (i.e. the number of multiplications and additions in each of the parallel parts) is given approximately by

\[ \frac{3k+2}{p} n \quad \text{flops.} \quad (2.1.4) \]

Neglecting overhead, such as the (re-)distribution of elements over the processors, the speed-up of the recursive doubling process over the standard recursion (2.1.2), which takes approximately \( 2n \) operations, amounts to (note that \( k = \log(p) \))

\[ S(p) \approx \frac{2p}{3 \cdot \log(p) + 2} \quad (2.1.5) \]

For \( p \geq 8 \), the speed-up \( S(p) \) becomes larger than one. The efficiency of the recursive doubling process is given by

\[ E(p) \approx \frac{2}{3 \cdot \log(p) + 2}, \quad (2.1.6) \]

which tends to zero if \( p \) becomes large.

If the number of processors \( p \) equals the number of unknowns \( n \), then the number of parallel operations (2.1.4) becomes proportional to \( \log(n) = \log(p) \). The efficiency of the recursive doubling process then becomes very low, even if overhead is neglected. The
impact of overhead (e.g. communication between processors, or physical data transfer), however, may even be more serious, thus degrading further the parallel performance. In the next sections, in the analysis of the partition method we try to model such overhead.

**Remark.** In [Mic1], it is shown that vectorization of the body (I) of the first outer loop (matrix multiplication) improves the parallel efficiency considerably (of course, this is only possible if the parallel computer has vectorprocessors).

Another well-known parallelization technique is the method of cyclic reduction [LaVo]. Although cyclic reduction is frequently used in a vectorcomputing environment, it also has nice parallel properties. Focussing again at the bidiagonal system (2.1.1), then the idea behind cyclic reduction is to eliminate $x_1$ from the second equation, $x_3$ from the fourth equation, and in general $x_{2j-1}$ from the $2j$–th equation. These eliminations can be carried out independently, i.e. in parallel. If one elimination step is completed, then the even numbered unknowns are no longer coupled to the odd numbered unknowns:

$$
\begin{bmatrix}
1 & & & & \\
-a_3a_4 & 1 & & & \\
& -a_5a_6 & 1 & & \\
& & \ddots & \ddots & \\
& & & -a_{n-1}a_n & 1 \\
\end{bmatrix}
\begin{bmatrix}
x_2 \\
x_4 \\
x_6 \\
\vdots \\
x_n \\
\end{bmatrix}
=
\begin{bmatrix}
b_2 + a_2b_1 \\
b_4 + a_4b_3 \\
b_6 + a_6b_5 \\
\vdots \\
b_n + a_nb_{n-1} \\
\end{bmatrix}
$$

The linear system (2.1.7) of order $\frac{n}{2}$ can be solved, whereafter the unknowns with an odd index can be computed in parallel, from the unknowns with an even index. However, the elimination of unknowns from (2.1.7) can be repeated, thus leading to a system of order $\frac{n}{4}$, etcetera. Each elimination step can be done in parallel, but the resulting bidiagonal system (of small order) still has to be solved by one single processor. Each substitution step for computing the eliminated unknowns can be done in parallel again. The cyclic reduction algorithm is more geared to vectorcomputing, since all steps, except for the solution of the eventual bidiagonal system, can be vectorized. For more details, we refer to [VoKa].
So far, we have not considered the distribution of data over the processors. In the recursive doubling process, each processor may operate on a number of rows of the matrix for the matrix multiplications. However, for the parallel solution of the independent bidiagonal systems, re-arrangement of data among the processors is necessary, since each smaller order bidiagonal system does not consist of the same rows as used during the matrix multiplication. For the cyclic reduction process, a similar argument holds: e.g., the sequential solution of the small order bidiagonal system requires collection of data by one processor.

For local memory machines, re-distribution of data among the processors requires physical data transfer between the processors. In the next sections, we will study this aspect in more detail. The algorithm we will consider is the partition algorithm, which may be used for the parallel solution of (banded) systems of linear equations.

Remark. The recursive doubling method and the cyclic reduction method seem to have much in common. However, the main difference is that after one recursive doubling sweep two independent bidiagonal systems are obtained, each of order $\frac{n}{2}$, while after one cyclic reduction sweep the original system has been reduced to one bidiagonal system of order $\frac{n}{2}$. After one recursive doubling sweep the odd and even unknowns can be computed simultaneously, while after one cyclic reduction sweep the even numbered unknowns have to be computed before the odd numbered unknowns can be determined.
§ 2.2 The partition method for tridiagonal linear systems

As an alternative for the recursive doubling and cyclic reduction methods, the so-called partition method has been developed. A more general version of the method was originally described in [CKSa], although many references are made to Wang, who proposed the method for tridiagonal systems in [Wang]. The name "partition method" also originates from Wang. In [Meie] the partition method has been generalized for banded systems of equations. In this section we will discuss the partition method as described in [Wang]. We will focus on aspects of the method related to re-distribution of data (i.e. exchange of data between the processors).

We consider a tridiagonal system of equations of order \( n \). The matrix \( A \) is subdivided into \( p \) groups of \( m \) consecutive rows. We assume that \( p \) denotes the number of processors and we assume that \( n \) can be factored as \( n = p \times m \):

\[
\begin{bmatrix}
d_1 & f_1 \\
e_2 & d_2 & f_2 \\
\vdots & \ddots & \ddots & \ddots \\
e_m & d_m & f_m & & \\
e_{m+1} & d_{m+1} & f_{m+1} & & \\
\vdots & \ddots & \ddots & \ddots & \ddots \\
e_{2m} & d_{2m} & f_{2m} & & \\
e_{2m+1} & d_{2m+1} & f_{2m+1} & & \\
& \ddots & \ddots & \ddots & \ddots \\
e_n & d_n & & & \\
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_m \\
x_{m+1} \\
\vdots \\
x_{2m} \\
x_{2m+1} \\
\vdots \\
x_n \\
\end{bmatrix}
= 
\begin{bmatrix}
b_1 \\
b_2 \\
\vdots \\
b_m \\
b_{m+1} \\
\vdots \\
b_{2m} \\
b_{2m+1} \\
\vdots \\
b_n \\
\end{bmatrix}
\tag{2.2.1}
\]

It is further assumed that the processors have local memory and that initially the data have been spread over the local memories as follows. The local memory of processor \( i \) contains only the matrix and vector elements of the \( i \)-th group of \( m \) rows of system (2.2.1), i.e. it contains the following data:

\[
d_{(i-1)m+1} \cdots \cdots, d_{im};
\]
\[
f_{(i-1)m+1} \cdots \cdots, f_{im};
\]
\[
e_{(i-1)m+1} \cdots \cdots, e_{im};
\]
\[
b_{(i-1)m+1} \cdots \cdots, b_{im}.
\]
The partition method for (2.2.1) can be split into two steps.

Step 1: The elements $e_j$, $2 \leq j \leq n$ and $j \neq m+1, 2m+1, \ldots, (p-1)m+1$, are eliminated, in parallel, followed by eliminating the elements $f_j$, $1 \leq j \leq n-1$ and $j \neq m-1, 2m-1, \ldots, (p-1)m-1$, in parallel. Each processor eliminates the $e_j$ and $f_j$ within its own group. Within one group, the $f_j$ are eliminated in reverse order. Since the processors run in parallel, the result of the first step is fill-in:

\[
\begin{bmatrix}
    d_1 & \cdots & f_1 \\
    d_2 & \cdots & \\
    \vdots & \ddots & \ddots \\
    d_m & \cdots & f_m \\
    e_{m+1} & d_{m+1} & f_{m+1} \\
    \vdots & \ddots & \ddots \\
    e_{2m} & d_{2m} & f_{2m} \\
    e_{2m+1} & d_{2m+1} & f_{2m+1} \\
    \vdots & \ddots & \ddots \\
    e_{3m} & d_{3m} & d_{3m+1} \\
    \vdots & \ddots & \ddots \\
    d_n & \cdots & \\
\end{bmatrix}
\begin{bmatrix}
    x_1 \\
    \vdots \\
    x_m \\
    x_{m+1} \\
    x_{2m} \\
    x_{2m+1} \\
    \vdots \\
    x_{3m} \\
    x_{3m+1} \\
    \vdots \\
    x_n \\
\end{bmatrix} = \begin{bmatrix}
    b_1 \\
    \vdots \\
    b_m \\
    b_{m+1} \\
    b_{2m} \\
    b_{2m+1} \\
    \vdots \\
    b_{3m} \\
    b_{3m+1} \\
    \vdots \\
    b_n \\
\end{bmatrix} \tag{2.2.2}
\]

Note that the elements $d_j$, $e_j$, $f_j$, and $b_j$ have been changed during both elimination steps. Furthermore, during the elimination of $f_{im}$, $i=1, \ldots, p-1$, it is necessary to transport elements $b_{im+1}, d_{im+1}, e_{im+1}$ and $f_{im+1}$, $i=1, \ldots, p-1$, from processor $i+1$ to processor $i$. This means that $(p-1)^2$ elements have been transported after completion of step 1.

Step 2: The partition method proceeds as follows. First, consider the second group of $m$ rows (i.e. rows $m+1, \ldots, 2m$). The elimination of $e_{m+1}, \ldots, e_{2m}$ by row $m$ can be distributed over the $p$ processors. Then, the elimination of $e_j$ belonging to the third group can be distributed over the processors, etc., until all $e_j$, $j=2, \ldots, n$ are eliminated. Note that the parallel elimination process of the $e_j$ within one group of $m$ rows now requires the use of other processors too. This then leads to a large amount of data transfer between the processors. One group of $m$ rows contains $n=4m$ elements, which are, after step 1, still located in the local memory of one processor. When the $p$
processors operate simultaneously on a group of \( m \) rows, then each processor eliminates \( \frac{m}{p} \) elements of the \( e_j \) and needs \( \approx \frac{3(m+1)}{p} \) elements to do so. However, only \( \approx \frac{3(m+1)}{p} \) elements of the group are already located in the "right" memory, so that for each group of \( m \) rows \( \approx (p-1). \frac{3(m+1)}{p} \) elements have to be transported. Since there are \( p-1 \) groups,

\[
z \approx \frac{3(p-1)^2(m+1)}{p} \text{ elements} \tag{2.2.3}
\]

have to be transported, in order to eliminate all \( e_j \) in parallel.

After the elimination of \( e_j \), the elimination of \( f_j \) within one group of \( m \) rows is distributed over \( p \) processors in a similar way. Starting from the original data distribution, the data transport now involves

\[
z \approx \frac{2(p-1)^2(m+1)}{p} \text{ elements.} \tag{2.2.4}
\]

Hence, from (2.2.3) and (2.2.4) we conclude that step 2 (elimination of fill-in) involves

\[
z \approx \frac{5(p-1)^2(m+1)}{p} \text{ elements} \tag{2.2.5}
\]

that have to be transported. After these two steps, the matrix \( A \) has been reduced to diagonal form and the solution of (2.2.1) is obtained straightforwardly, which concludes step 2.

The partition method has been generalized for arbitrary banded systems in [Meie]. Again, the matrix is subdivided into \( p \) groups of \( m \) rows \( (p^*m=n) \), where \( p \) denotes the number of processors, with local memory. In [MiVo], it is shown that the data transfer between the processors for execution of the generalized partition method amounts to

\[
z \approx \frac{(p-1)^2[(2r+s+2)m + r^2 + s^2 + \frac{1}{2}r(r+3) + s]}{p} \text{ elements,} \tag{2.2.6}
\]

in which \( r \) denotes the upper bandwidth and \( s \) the lower bandwidth. Note that for tridiagonal matrices \( (r=s=1) \), (2.2.6) reduces to (2.2.5). For lower or upper bidiagonal systems, (2.2.6) does not hold, since one of the two elimination steps does not have to be performed. For bidiagonal systems, the amount of data transfer is determined by either the first or the second part of step 2 of the elimination process:
\[
\frac{2(p-1)^2(m+1)}{p} \text{ elements.} \quad (2.2.7)
\]

The analysis above shows that for banded systems with a small bandwidth, the
number of elements to be transported from one processor to another on local memory
machines is of the order of the problem size \( n \). It is well-known that too much data
transfer may seriously degrade the parallel performance, and therefore reduction of data
transfer is paramount. In the next section, we will consider an alternative for the
partition method, in which the amount of data transfer is reduced considerably. In
section 2.4, the parallel performances of the original partition method and this modified
version will be compared.

**Remark 1.** Shared memory computers do not suffer from physical data transfer
between processors. However, for correct solution of many problems, it is required that
data is shared (i.e. one processor needs information that first has to be processed by
another processor). Therefore, some synchronization between processors may be
necessary. In chapters 5 and 6 synchronization will be discussed in more detail for more
complicated situations.

**Remark 2.** In this section, we have assumed that \( n=m*p \). This assumption is, of
course, not essential and only minor changes are required if it is not possible or desirable
to choose \( m \) integer, such that \( n=m*p \) (\( n \) and \( p \) are assumed to be fixed).

**Remark 3.** Step 2 of the partition method distributes \( m \) rows over \( p \) processors. In
order to avoid idle processors, \( m \) should be greater than \( p \).

**Remark 4.** The number of parallel cycles of the partition method for solving
tridiagonal systems amounts to \( 21m = 21 \frac{n}{p} \) [Wang].
§ 2.3 The modified partition method

In this section, we will discuss an alternative version of the partition method which requires a strongly reduced amount of data transfer between the processors in local memory machines. In the previous section, we have shown that the final elimination step (step 2) caused the major part of the data transfer (O(n) elements that have to be re-distributed among the processors). The modification we discuss concerns this step and it has been suggested in [DERe].

After step 1 of the original partition method, the following system of equations is obtained (cf. (2.2.2)):

\[
\begin{bmatrix}
  d_1 & \cdots & f_1 \\
  \vdots & \ddots & \vdots \\
  d_m & \cdots & f_m \\
  e_{m+1} & \cdots & f_{m+1} \\
  \vdots & \ddots & \vdots \\
  e_{2m} & \cdots & f_{2m} \\
  e_{2m+1} & \cdots & f_{2m+1} \\
  \vdots & \ddots & \vdots \\
  e_{3m} & \cdots & f_{3m} \\
  \vdots & \ddots & \vdots \\
  e_{n} & \cdots & f_{n} \\
  d_{n} & & f_{n}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  \vdots \\
  x_m \\
  x_{m+1} \\
  \vdots \\
  x_{2m} \\
  x_{2m+1} \\
  \vdots \\
  x_{3m} \\
  \vdots \\
  x_{n}
\end{bmatrix}
= \begin{bmatrix}
  b_1 \\
  \vdots \\
  b_m \\
  b_{m+1} \\
  \vdots \\
  b_{2m} \\
  b_{2m+1} \\
  \vdots \\
  b_{3m} \\
  \vdots \\
  b_n
\end{bmatrix}
\] (2.3.1)

Instead of distributing a group of m rows over p processors, the following strategy is applied. First, element \( e_{2m} \) is eliminated by row \( m \), which requires data transfer of the elements of row \( m \), since row \( m \) and row \( 2m \) are in different groups. This elimination does not create any additional fill-in. After the elimination, element \( e_{3m} \) is eliminated by row \( 2m \). Again, no additional fill-in is created, but only data transfer of the elements of row \( 2m \) is required. This (sequential) process is continued until the elimination of \( e_n \) and requires \( 3(p-1) \) elements to be transported.

After this sequential step, parallel execution continues. The remaining elements \( e_j \) can be eliminated for all the groups of rows simultaneously. Processor 2 eliminates \( e_{m+1}, \ldots, e_{2m-1} \) by using row \( 2m \), etcetera. Each group of rows keeps its "own" processor and hence no data transfer is required. However, only \( p-1 \) processors are
active, since there are only \( p-1 \) groups that contain elements \( e_j \) below the main diagonal. Therefore, compared with the original elimination of \( e_j \), the amount of parallel cycles has been increased by a factor of \( \frac{p}{p-1} \). A parallel cycle is defined as the amount of time, used by one processor to execute one floating-point operation (note that during this cycle the other processors may also carry out one floating-point operation). Furthermore, the sequential step takes approximately \( 5(p-1) \) cycles.

A similar strategy can be used for eliminating the \( f_j \). Again \( p-1 \) processors are activated simultaneously. The data transfer now involves \( 2(p-1) \) elements and after completion of the elimination the tridiagonal system (2.2.1) has been reduced to diagonal form. We conclude that the modified step 2 involves the data transfer of

\[
5(p-1) \text{ elements.} \tag{2.3.2}
\]

The original partition method involved ((2.2.5)) the transport of

\[
\frac{5(p-1)^2(m+1)}{p} \text{ elements,} \tag{2.3.3}
\]

which means that the total amount of data transfer has been reduced by a factor of \( O\left(\frac{p}{p}\right) \), whereas the number of parallel cycles for executing the complete algorithm has not significantly been increased (provided \( p \) is small as compared with \( m \)). For the original partition method for tridiagonal systems the number of parallel cycles amounts to \( 21m = 21\frac{n}{p} \), see remark 4 in section 2.2.

To illustrate the effect of the modified partition method, we consider the bidiagonal system (2.1.1) of section 2.1. Application of the modified partition method to bidiagonal systems has been discussed in [SaS2], see also [MiVo]. The algorithm is given in figure 2.3.1. The first and the third outer loop of the algorithm (of which the bodies are denoted by I and III, respectively) can be distributed over \( p \) processors. The first outer loop eliminates the subdiagonal elements \( a_{ij} \), leading to fill-in. The second outer loop represents the sequential step of eliminating the \( a_{im} \), \( i=2, \ldots, p \). The third outer loop is the final elimination of the fill-in. After completion of the algorithm, the solution of (2.1.1) is given by \( b_{ij} \), \( j=1, \ldots, n \). The total amount of data transfer is only \( p-1 \) elements \( (b_{im} \), \( i=1, \ldots, p-1 \)) since the main diagonal is unity. The original partition method involves \( \frac{2(p-1)^2(m+1)}{p} \) elements to be transported.
for \(i = 1, p\)
\[
\begin{align*}
\text{for } j & = (i-1)m+2, im \\
& \\
I & \\
& b_j = b_j + a_jb_{j-1} \\
& a_j = a_ja_{j-1} \\
\text{for } i & = 2, p \\
\text{II} & \\
b_{im} & = b_{im} + a_{im}b_{(i-1)m} \\
\text{for } i & = 2, p \\
\text{III} & \\
& \\
& b_j = b_j + a_jb_{(i-1)m}
\end{align*}
\]

**Figure 2.3.1:** Modified partition method for bidiagonal systems.

The amount of parallel floating-point operations (cycles) for the modified partition method (bidiagonal systems) is given by

\[
5(m-1) + 2(p-1) \approx \frac{5n}{p} + 2p \text{ cycles,} \quad (2.3.4)
\]

while for the original partition method, we have

\[
3(m-1) + \frac{2(p-1)m}{p} \approx \frac{5n}{p} \text{ cycles.} \quad (2.3.5)
\]

If \(p \ll n\) then, for both versions, the speed-up is roughly \(\frac{2n}{5p}\), when data transport is not taken into account. This result shows that the speed-up is bounded by \(\frac{2n}{5p}\) in practical situations when data transfer or synchronization plays a role.

In most practical cases, the costs of data transfer are not negligible small. Assuming that the transport of one floating-point element takes \(\beta\) seconds and that one floating-point operation takes \(\alpha\) seconds, we estimate the parallel execution time \(T_m\) for the modified algorithm by:

\[
T_m = \frac{5n}{p} \alpha + 2p \alpha + (p-1)\beta, \quad (2.3.6)
\]

and \(T_o\) for the original algorithm by:

\[
T_o = \frac{5n}{p} \alpha + \frac{2(p-1)^2n}{p^2} \beta. \quad (2.3.7)
\]
If \( \frac{n}{p} > p \), then the modified algorithm takes approximately

\[
T_m \approx \frac{5n}{p} \alpha \text{ seconds,} \tag{2.3.8}
\]

indeed leading to a speed-up of approximately \( \frac{2}{5}p \). For the original algorithm \( T_o \) is

\[
T_o \approx \frac{5n}{p} \alpha + 2n\beta \text{ seconds,} \tag{2.3.9}
\]

and hence the speed-up is

\[
S_o \approx \frac{2n\alpha}{\frac{5n}{p} \alpha + 2n\beta} = \frac{\alpha}{\frac{5}{2} \frac{\alpha}{p} + \beta}. \tag{2.3.10}
\]

\( S_o \) is then bounded by

\[
S_o \leq \min \left[ \frac{2}{\frac{5}{2} p}, \frac{\alpha}{\beta} \right], \tag{2.3.11}
\]

which shows that the ratio between one floating-point cycle and one transport cycle gives an upper-bound for the speed-up \( S_o \), independently of the number of processors \( p \) [MiVo]. For many local memory machines, the ratio \( \frac{\alpha}{\beta} \) is less than 1.0 [Mic2].

In the next section, actual implementations for both versions of the partition method will be discussed. Experimental results show that data transfer indeed seriously degrades parallel performance.

**Remark.** If the modified step 2 is used instead of the original step 2, then processor \( i \) always works on group \( i \). For tridiagonal systems, the rows \((i-1)m+1, \cdots, im-1\) are modified in the first half of modified step 2. In the second half of modified step 2, the rows \(im-1, \cdots, (i-1)m+1\) are modified. In the case of (parallel) vectorprocessors, the above processes are so-called linked triads (or vector updates), which offers another advantage over the original partition method. This also holds for general banded systems and bidiagonal systems.
§ 2.4 Experimental results and analysis

In this section, experimental results for implementations of the original partition method and the modified partition method will be discussed. Both methods have been implemented for bidiagonal systems on an NCube, a hypercube type parallel computer with local memories. The maximum dimension of the hypercube we used is 4, which corresponds to a maximum of 16 processors. For more specific details on this machine, we refer to chapter 1.

First, we implemented the standard solution method (2.1.2) for one processor (node) of the NCube. For some values of the problem size n, we measured the times in milliseconds, as shown in table 2.4.1:

<table>
<thead>
<tr>
<th>n</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>5.4</td>
</tr>
<tr>
<td>512</td>
<td>10.9</td>
</tr>
<tr>
<td>1536</td>
<td>31.9</td>
</tr>
<tr>
<td>2560</td>
<td>53.1</td>
</tr>
</tbody>
</table>

*Table 2.4.1:* Time in ms for the recursion formula on an NCube/four.

Since the recursion formula involves $2(n-1)$ floating-point operations, the kflops rate for this problem, on one processor, is approximately 95. If $\alpha$ denotes the time in ms for one operation (addition or multiplication), then the sequential execution time in ms for the recursion (2.1.2) is given by:

$$T_s = 2(n-1)\alpha \quad (2.4.1)$$

From table 2.4.1, it follows that

$$\alpha \approx 1.1 \cdot 10^{-2} \text{ ms.} \quad (2.4.2)$$

Implementation of the modified partition method has led to the timing results, as shown in table 2.4.2. The number of processors is denoted by $p$, while the speed-up $S$ for each value of $n$ and $p$ is computed relative to the timing results of table 2.4.1:
<table>
<thead>
<tr>
<th>p</th>
<th>n = 256 time</th>
<th>S</th>
<th>n = 512 time</th>
<th>S</th>
<th>n = 1536 time</th>
<th>S</th>
<th>n = 2560 time</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>8.2</td>
<td>0.66</td>
<td>15.6</td>
<td>0.70</td>
<td>44.5</td>
<td>0.72</td>
<td>74.0</td>
</tr>
<tr>
<td>4</td>
<td>5.4</td>
<td>1.0</td>
<td>8.8</td>
<td>1.24</td>
<td>23.5</td>
<td>1.36</td>
<td>38.3</td>
<td>1.39</td>
</tr>
<tr>
<td>8</td>
<td>5.6</td>
<td>0.96</td>
<td>7.3</td>
<td>1.49</td>
<td>14.4</td>
<td>2.22</td>
<td>21.7</td>
<td>2.45</td>
</tr>
<tr>
<td>16</td>
<td>8.1</td>
<td>0.67</td>
<td>9.0</td>
<td>1.21</td>
<td>12.5</td>
<td>2.55</td>
<td>16.1</td>
<td>3.30</td>
</tr>
</tbody>
</table>

**Table 2.4.2:** Time in ms and speed-up S for the modified partition method.

Table 2.4.2 shows that the speed-up is well below \( \frac{2}{p} \), the upperbound that has been derived in the previous section. Also in the previous section, we found that the computational work of the modified partition method is proportional to \( \frac{5n}{p} \), and that the communicational work (in this case, physical data transfer) is proportional to \( p-1 \). Then we have for the parallel execution time of the modified partition method:

\[
T_m(p;n) = \frac{5n}{p} \alpha + (p-1)\beta, \tag{2.4.3}
\]

in which \( \alpha \) is given by (2.4.2). The value of \( \beta \) is computed using the results of table 2.4.2, and amounts to

\[
\beta \approx 0.50 \text{ ms.} \tag{2.4.4}
\]

The speed-up \( S_m(p;n) \) is given by

\[
S_m(p;n) \approx \frac{2n\alpha}{\frac{5n}{p}\alpha + (p-1)\beta}. \tag{2.4.5}
\]

Experiments with other values of \( n \) turned out to be in agreement with (2.4.3) and (2.4.5). Result (2.4.5) also shows that the speed-up \( S_m \) tends to zero for large \( p \) (\( n \) fixed), and to the upperbound \( \frac{2}{3p} \) for large \( n \) (\( p \) fixed). Furthermore,

\[
\frac{\partial S_m}{\partial p} = \frac{2n\alpha(5n\alpha - \beta^2)}{(5n\alpha + \beta(p-1))^2}. \tag{2.4.6}
\]
Hence, for fixed $n$, the speed-up $S_m$ is maximal if $p$ is choosen as

$$p_{\text{opt}} = \sqrt{\frac{5n\alpha}{\beta}}.$$  \hfill (2.4.7)

For the experiments of table 2.4.2, we find (with $\alpha$ and $\beta$ given):

<table>
<thead>
<tr>
<th>$n$</th>
<th>$p_{\text{opt}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>6</td>
</tr>
<tr>
<td>512</td>
<td>8</td>
</tr>
<tr>
<td>1536</td>
<td>14</td>
</tr>
<tr>
<td>2560</td>
<td>18</td>
</tr>
</tbody>
</table>

*Table 2.4.3*: Optimum number of processors for fixed $n$; modified partition method.

The results of table 2.4.2 are consistent with those in table 2.4.3.

The original partition method has also been implemented for the NCube. Table 2.4.4 shows the timing results:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$n = 256$ time</th>
<th>$n = 256$ S</th>
<th>$n = 512$ time</th>
<th>$n = 512$ S</th>
<th>$n = 1536$ time</th>
<th>$n = 1536$ S</th>
<th>$n = 2560$ time</th>
<th>$n = 2560$ S</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12.8</td>
<td>0.42</td>
<td>24.7</td>
<td>0.44</td>
<td>72.5</td>
<td>0.44</td>
<td>120.4</td>
<td>0.44</td>
</tr>
<tr>
<td>4</td>
<td>19.2</td>
<td>0.28</td>
<td>31.9</td>
<td>0.34</td>
<td>82.2</td>
<td>0.39</td>
<td>132.4</td>
<td>0.40</td>
</tr>
<tr>
<td>8</td>
<td>44.5</td>
<td>0.12</td>
<td>56.4</td>
<td>0.19</td>
<td>104.7</td>
<td>0.30</td>
<td>152.7</td>
<td>0.35</td>
</tr>
<tr>
<td>16</td>
<td>148.3</td>
<td>0.04</td>
<td>159.9</td>
<td>0.07</td>
<td>205.8</td>
<td>0.16</td>
<td>251.8</td>
<td>0.21</td>
</tr>
</tbody>
</table>

*Table 2.4.4*: Time in ms and speed-up $S$ for the original partition method.

Table 2.4.4 shows that speed-up results for the original partition method are far below 1.0, which means that the original partition method is not efficient at all for parallel computation on local memory machines. In section 2.3, we have shown the computational work to be proportional to $\frac{5n^p}{p}$. The number of elements to be re-distributed among the processors amounts to $\frac{2(p-1)^2n}{p^2}$ (result (2.3.7) for bidiagonal
systems), i.e. \((p-1)^2\) messages, of length \(\frac{2n}{p^2}\) each.

For local memory machines, a frequently used model for physical data transfer involves a start-up time \(\gamma\), and a transfer time per element \(\delta\) (cf. section 1.2):

\[
\gamma + k\delta,
\]

in which \(k\) denotes the number of elements to be transported. In many cases, \(\delta \ll \gamma\), which implies that for small \(k\), data transfer costs are represented largely by the start-up time \(\gamma\) (modified partition method: \(k = 1\), and hence \(\beta = \gamma + \delta\)). The parallel execution time for the original partition method is then:

\[
T_o(p;n) = \frac{5n}{p} \alpha + (p-1)^2(\gamma + \frac{2n}{p^2}\delta).
\]

(2.4.9)

The speed-up \(S_o(p;n)\) is given by

\[
S_o(p;n) = \frac{2n\alpha}{\frac{5n}{p}\alpha + (p-1)^2(\gamma + \frac{2n}{p^2}\delta)}.
\]

(2.4.10)

For fixed \(p\), we have

\[
\lim_{n \to \infty} S_o(p;n) = \frac{2\alpha}{\frac{5\alpha}{p} + \frac{(p-1)^2}{p^2} \cdot 2\delta} \leq \left(\frac{p}{p-1}\right)^2 \cdot \frac{\alpha}{\delta}.
\]

(2.4.11)

Furthermore, \(\frac{\partial S_o}{\partial n} \geq 0\), and we conclude that \(S_o(p;n)\) is bounded from above by

\[
S_o(p;n) \leq \left(\frac{p}{p-1}\right)^2 \cdot \frac{\alpha}{\delta} \cdot \frac{\alpha}{\delta},
\]

for each \(n\) and \(p\) not too small.

To determine \(\gamma\) and \(\delta\), we first consider the case of two processors \((p=2)\). The results of table 2.4.4, for \(n=256\) and \(n=512\), lead to \((p=2)\):

\[
S_o(p;n) \leq \left(\frac{p}{p-1}\right)^2 \cdot \frac{\alpha}{\delta} \cdot \frac{\alpha}{\delta}.
\]

(2.4.12)
\[ \gamma_2 \approx 0.89 \text{ ms and } \delta_2 \approx 0.038 \text{ ms.} \quad (2.4.13) \]

Application of (2.4.9) for \( n = 1536 \) and \( n = 2560 \) then yields estimated parallel execution times of 72.3 ms and 119.9 ms, respectively, which agree very well with the corresponding results in table 2.4.4 (\( p = 2 \)). For \( p = 2 \), we find that \( \frac{\alpha}{\delta_2} \approx 0.29 \), which implies that, for each \( n \), \( S_0(2; n) \leq 1.16 \).

In the case of 4 processors (\( p = 4 \)), we also compute \( \gamma \) and \( \delta \) from the results of table 2.4.4 concerning \( n = 256 \) and \( n = 512 \). This yields (\( p = 4 \))

\[ \gamma_4 \approx 0.72 \text{ ms and } \delta_4 \approx 0.032 \text{ ms.} \quad (2.4.14) \]

Again, estimated parallel execution times (by (2.4.9)) agree with the results of table 2.4.4 (\( p = 4 \)).

Comparison of (2.4.13) with (2.4.14) suggests that in the case of 4 processors (dimension of hypercube is 2), physical data transfer of a message from one processor to another is cheaper than the transfer of the same message in the case of 2 processors (dimension of hypercube is 1). This is not true, of course, but we have not mentioned one aspect of data transfer yet. In the case of 4 processors, one processor sends each of the 3 other processors a message. This means that, at some time, there are 3 messages in the system, each of them being sent to another processor. To some extent then, data transfer is carried out in parallel. Since in section 2.2, and in formula (2.4.9), data transfer was assumed to be sequential, lower values of \( \gamma \) and \( \delta \) (effective \( \gamma \) and \( \delta \)) reflect partly parallel data transfer. For this reason, again, \( \gamma \) and \( \delta \) for \( p = 8 \) and \( p = 16 \) are smaller than for \( p = 4 \):

<table>
<thead>
<tr>
<th>P</th>
<th>( \gamma_P )</th>
<th>( \delta_P )</th>
<th>( n = 1536 )</th>
<th>( n = 2560 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>estim. time</td>
<td>estim. time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>meas. time</td>
<td>meas. time</td>
</tr>
<tr>
<td>2</td>
<td>0.89</td>
<td>0.038</td>
<td>72.3</td>
<td>119.9</td>
</tr>
<tr>
<td>4</td>
<td>0.72</td>
<td>0.032</td>
<td>82.9</td>
<td>133.8</td>
</tr>
<tr>
<td>8</td>
<td>0.66</td>
<td>0.026</td>
<td>104.1</td>
<td>151.9</td>
</tr>
<tr>
<td>16</td>
<td>0.61</td>
<td>0.024</td>
<td>207.3</td>
<td>254.1</td>
</tr>
</tbody>
</table>

|                |                |               |              |              |
|                |                |               |              |              |
|                |                |               |              |              |
|                |                |               |              |              |

Table 2.4.5: Estimated and measured execution times (in ms) for the original partition method.
The rate of reduction of $\gamma_p$ and $\delta_p$ is decreasing for increasing $p$, due to the fact that the average distance between the processors of the hypercube becomes larger for higher dimensions.

From tables 2.4.2 and 2.4.4 it has become clear that the modified partition method offers a great advantage over the original partition method. In figure 2.4.1, speed-up values for the modified ($S_m$) and the original partition method ($S_o$) are depicted:

![Graph showing speed-up values for different $n$ values.](image)

**Figure 2.4.1:** Speed-up of the modified ($S_m$) and the original partition method ($S_o$).

The upper (straight) line represents the upper-bound $\frac{2}{5}p$ for the speed-up. The next 3 curves represent the modified partition method for $n=5120$, $n=2560$ and $n=1536$, respectively, while the last 2 curves reflect the original partition method for $n=2560$ and $n=1536$.

The results in this section show that, for local memory machines, data transfer between processors may seriously degrade parallel performance. Analysis of computational work and communicational work leads to models that can accurately predict parallel execution times. The type of modelling, used in this section, will also be used in chapters 5 and 6, for more complicated problems.
§ 2.5 Some comments on Amdahl’s law

In the previous section, the original partition method and the modified partition method have been modelled by analyzing the computational and communicational work. This has led to models that predict parallel execution times and speed-up factors quite well. These models are based upon detailed knowledge of the underlying algorithm and computer system, and experimental results which determine the parameters of the model. However, there are also models that do not consider the underlying algorithm or computer system. In this section, we will discuss briefly some aspects of Amdahl’s law, which describes one of such models.

Amdahl’s law [Amda] is based upon the observation that in most realistic situations, not all operations in a complex algorithm are executable with the same computational speeds. E.g., for vectorcomputers part of a job can be executed at vectorspeed, while the other part of the job is run only at scalar speed. It turns out that the lower of these speeds (scalar) may have a quite dramatic influence on the overall performance. For parallel processing, Amdahl’s law distinguishes between the part of the algorithm that is executed in parallel, and the part that is executed in one-processor mode (sequential). This can be formulated as follows.

*Amdahl’s law:* Let algorithm A be executed on $p_o$ processors, leading to a speed-up $S(p_o)$ over the fastest sequential algorithm. Define the parallel fraction $f$ of A by:

$$S(p_o) = \frac{p_o}{f + (1-f)p_o}.$$ 

Then we have for the speed-up $S(p)$:

$$S(p) = \frac{p}{f+(1-f)p}, \quad p > p_o.$$

As an example of Amdahl’s law, suppose that $S(2) = 1.94$. Then the definition of $f$ leads to: $f \approx 0.97$. The speed-ups $S(p)$ for execution of the algorithm on 4 and 8 processors then are 3.67 and 6.61, respectively, according to Amdahl’s law. For very large $p$, the speed-up $S(p)$ is bounded by 33.3.

In practice, these speed-up values for 4 and 8 processors are not always achieved. Amdahl’s law does not count for, e.g., synchronization overhead or data transfer. Furthermore, in many cases, algorithms have to be modified in order to profit fully from
the available additional processors. In most cases, this leads to an algorithm which is computationally more complicated and hence the speed-up values predicted by Amdahl's law are too optimistic. In [DDSV], it is proven that Amdahl's speed-up values are upperbounds for the real speed-up values, provided that the sequential part of the algorithm \((1-f)\) does not change. In chapter 6, we will show that this latter condition is essential (remark 2 in section 6.3).

In table 2.4.2, we have listed some speed-up results for the modified partition method. In the case of \(p=2, n=2560\), the measured speed-up is 0.72. Application of Amdahl's law then leads to a negative parallel fraction \(f\) and of course Amdahl's law is not applicable in this case. The negative value of \(f\) is caused by the fact that the computational complexity of the modified partition method is about \(5n\) flops, versus \(2n\) flops for the recursion formula. It is therefore preferable to use Amdahl's law with respect to the performance of the parallel algorithm on 2 processors, or, in general, \(p_o \geq 2\) processors:

**Alternative formulation of Amdahl's law:**

Let algorithm A be executed on \(p_o\) and \(p_1 ( \geq p_o)\) processors, leading to parallel execution times \(T_{p_o}\) and \(T_{p_1}\), respectively. Define \(f\) by

\[
S(p_1; p_o) = \frac{T_{p_1}}{T_{p_o}} = \frac{p_1}{p_o} f + (1-f)\frac{p_1}{p_o} \tag{2.5.1}
\]

Then we have for the speed-up \(S(p; p_o)\), for \(p > p_1\):

\[
S(p; p_o) = \frac{p}{p_o} f + (1-f)\frac{p}{p_o} \tag{2.5.2}
\]

For the modified partition method \((n=2560)\), this leads to the results \((p_o=2, p_1=4, f=0.964)\) as shown in table 2.5.1. This table shows that measured values of the speed-up for the modified partition method \((S_m)\) stay well behind the predicted values \((S)\), especially for large \(p\). The asterisk * denotes application of model (2.4.3), since only
16 processors were available.

<table>
<thead>
<tr>
<th>p</th>
<th>S(p;2)</th>
<th>measured S_m(p;2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>1.93</td>
<td>1.93</td>
</tr>
<tr>
<td>8</td>
<td>3.61</td>
<td>3.41</td>
</tr>
<tr>
<td>16</td>
<td>6.39</td>
<td>4.60</td>
</tr>
<tr>
<td>32</td>
<td>10.39</td>
<td>3.72*</td>
</tr>
<tr>
<td>64</td>
<td>15.12</td>
<td>2.20*</td>
</tr>
<tr>
<td>128</td>
<td>19.58</td>
<td>1.16*</td>
</tr>
</tbody>
</table>

*Table 2.5.1: Predicted ((2.5.1), (2.5.2)) and measured speed-up values with respect to 2 processors.*

In many cases there are speed-up values for different small values of p available. It seems natural then to attempt to use these speed-up values for obtaining sharper upperbounds for the speed-up with more processors. This is illustrated by the following example, which is based upon table 2.5.1. Suppose, S_m(4;2) and S_m(8;2) are known (1.93 and 3.41, respectively). Result (2.5.2) has led to $f \approx 0.964$, which can be viewed as the fraction of the parallel algorithm on 2 processors that can take full advantage of 4 processors. Now, define $g$ as the fraction of $f$ that can take full advantage of 8 processors. Then $g$ is defined as:

$$S(8;2) = \frac{T_2}{T_8} = 3.41 = \frac{\frac{8}{2}}{g \cdot f + (1-g) \cdot f \cdot \frac{8}{4} + (1-f) \cdot \frac{p}{2}}, \tag{2.5.3}$$

which gives $g \approx 0.933$. Then, for $p > 8$, we have

$$S(p;2) = \frac{\frac{p}{2}}{g \cdot f + (1-g) \cdot f \cdot \frac{8}{4} + (1-f) \cdot \frac{8}{2}}, \tag{2.5.4}$$

leading to table 2.5.2, which gives somewhat better upperbounds than table 2.5.1:
<table>
<thead>
<tr>
<th>( p )</th>
<th>( S(p;2) )</th>
<th>( S_m(p;2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>5.53</td>
<td>4.60</td>
</tr>
<tr>
<td>32</td>
<td>8.03</td>
<td>3.72</td>
</tr>
<tr>
<td>64</td>
<td>10.37</td>
<td>2.20</td>
</tr>
<tr>
<td>128</td>
<td>12.15</td>
<td>1.16</td>
</tr>
</tbody>
</table>

*Table 2.5.2: Predicted \((2.5.4)\) speed–up values.*

The procedure described above can be continued for \( p=16 \), \( p=32 \), etc., provided that the respective speed–up values are known. In the case of \( p=32 \), we encounter an interesting phenomenon, since the measured speed–up is less than for \( p=16 \). This leads to a negative fraction of the algorithm that is to be executed on 32 processors. For more processors, then, the predicted speed–up values are decreasing, which is in accordance with the measured speed–up values.

In this way Amdahl's law can be modified to give somewhat sharper upperbounds for speed–up values. However, in section 2.4 we have shown that it is preferable to predict parallel execution times by analyzing the underlying algorithm. Therefore, this technique will also be applied in chapters 5 and 6, in order to study the performance of more complicated algorithms on both local memory and shared memory machines.
NUMERICAL RESERVOIR SIMULATION

Abstract

In this chapter a general introduction to reservoir simulation is given. The basic equations for reservoir simulation are briefly discussed, leading to a system of equations which serves as a simple model for a reservoir simulator. Several features of the model are discussed, including the splitting into a hyperbolic part and an elliptic part. Discretization is performed by means of mixed finite elements, leading to a system of coupled equations for the elliptic part, which will be solved by a multigrid method with a (block) Gauss-Seidel smoother. The solution of the hyperbolic part is fairly easy, though complicated by a grid (un)refinement technique to resolve the flow of fluids in the reservoir in an adequate way. The physical model problem consists of the injection—production mechanism, leading to fluid flow from injection to production wells. This chapter is largely based upon the description of the reservoir simulator as given in [ScJa].
§ 3.1 Introduction

Oil is found together with gas and water within porous layers of earth enclosed by an impermeable rock formation. Gravity forces lead to a separation of gas, oil and water: water is found at the bottom, gas at the top and oil in between. Such a physical system is called an oil-reservoir, and is depicted in figure 3.1.1 [Kett]:

![Figure 3.1.1: Cross-section of an oil-reservoir.](image)

During the first few years of the production time of an oil-reservoir, it is in many cases not necessary to drill injection wells: the pressure in the reservoir is such that oil can be produced fairly easy. After some time, the pressure has dropped and, in order to produce the resulting oil, it may be advantageous to increase the pressure artificially. In this case, injection wells are drilled and water, steam or chemicals are injected, so that the water-oil and oil-gas interfaces will move. The objective of reservoir engineering is to determine an optimum production strategy, i.e. by means of selecting numbers and locations of wells, types and rates of injection, etcetera.

To predict the performance of a reservoir under various exploitation schemes, reservoir simulation is applied to the underlying reservoir. The physical properties of the reservoir, such as the shape of the reservoir and the porous medium, are modelled. Furthermore, the reservoir is subject to gravity, capillary forces and viscous forces, while injection and production wells represent sources. The resulting flow of fluids can be modelled mathematically and simulated on a computer.

In reservoir simulation, one distinguishes between phases and components. The phases are gas, oil and water; the components are chemical compounds which may exist in any or all of the phases. If we consider only single-fluid flow (one component, one phase), then the continuity equation (mass conservation) is given by:
\[ \frac{\partial a}{\partial t} = q - V(\rho u), \]  

(3.1.1)

in which \( a \) denotes the fluid mass per unit volume, \( q \) denotes the fluid mass injection rate per unit volume, \( \rho \) is the density and \( u \) is the velocity vector. The porosity \( \phi \) is defined as the volume fraction of the fluid in the unit volume. Then

\[ a = \phi \rho. \]  

(3.1.2)

Let us now assume to have 2 immiscible fluid phases. Then \( s_m, m = 1,2, \) is defined as the volume fraction, occupied by phase \( m \). This leads to:

\[ \rho = \rho_1 s_1 + \rho_2 s_2, \quad s_1 + s_2 = 1, \]  

(3.1.3)

with \( \rho_m, m = 1,2, \) the density for phase \( m \).

Immiscible flow implies that there is no mass transfer between phases and hence (3.1.1) holds for each phase \( m \):

\[ \frac{\partial (\phi \rho_m s_m)}{\partial t} = q_m - V(\rho_m u_m), \]  

(3.1.4)

in which \( q_m \) denotes the fluid mass injection rate per unit volume for phase \( m \) and \( u_m \) is the velocity of phase \( m \).

The empirical relationship between flow rate and pressure gradient is known as Darcy's law. Neglecting gravity terms and capillary pressures, Darcy's law for phase \( m \) is expressed as:

\[ u_m = -\frac{1}{\mu_m} K_m \cdot \nabla p, \]  

(3.1.5)

where \( K_m \) is the absolute permeability tensor for phase \( m \), \( \mu_m \) denotes the dynamical fluid viscosity for phase \( m \) and \( p \) is the fluid pressure. Defining

\[ k_m \equiv \frac{1}{\mu_m} K_m, \quad m = 1,2, \]  

(3.1.6)

we have

\[ u_m = -k_m \cdot \nabla p. \]  

(3.1.7)
Introducing
\[ u = u_1 + u_2, \quad k = k_1 + k_2, \quad q = q_1 + q_2, \]  
then summation of (3.1.7) over the phases yields
\[ u = -k \cdot \nabla p. \]  
Furthermore, substitution of \( \nabla p = -k^{-1}u \) into (3.1.5) gives
\[ u_m = -k_m k^{-1}u. \]  
The relationship between \( p \) and \( \rho \) is expressed by an equation of state. The definition of fluid compressibility, \( c \), may serve as such an equation of state:
\[ c = \left. \frac{1}{\rho} \frac{\partial \rho}{\partial p} \right|_T, \]  
which is valid at a fixed temperature \( T \). For two-phase flow, the pressure and densities for each phase are related by equations of state for each phase \( m \). For small compressibilities, the phase densities \( \rho_m \) are assumed to be linear in the pressure \( p \). It turns out (e.g., [Ewin]) that after summation over the phases and neglecting some terms, (3.1.4) can be modelled as
\[ c \cdot \partial p = q - \nabla u, \]  
in which the compressibility \( c \) is small in many cases.

The system that can be viewed as representative for isothermal immiscible two-phase flow is given by (3.1.4), (3.1.10), (3.1.12) and (3.1.9):
\[ \frac{\partial s_m}{\partial t} = q_m - \nabla u_m, \quad m = 1,2, \]  
\[ u_m = k_m \cdot k^{-1}u, \quad m = 1,2, \]  
\[ c \cdot p + \nabla u = q, \]  
\[ \nabla p + k^{-1}u = \nu. \]
In (3.1.13c) \( \frac{\partial p}{\partial t} \) has been substituted by \( p \), while \( y \) denotes a right-hand side in (3.1.13d). Initial saturations are given:

\[
s_m = s_m^0, \quad t = 0, \quad m = 1,2, \tag{3.1.14}
\]

and at the boundary of the computational domain \( \Omega \), "no normal-flow conditions" hold (\( \nu \) denotes the outward normal on \( \partial \Omega \)):

\[
u_m \cdot \nu = 0, \quad x \in \partial \Omega, \quad m = 1,2. \tag{3.1.15}
\]

As a model problem, a (two-dimensional) square region \( \Omega \) is considered, with injection in one corner and production in the opposite corner, as depicted in figure 3.2.1. Injection of water leads to a front of water, which moves in the direction of the production well, which produces oil. After some time, also water will be produced (if production is continued).

![Model problem for reservoir simulation](image)

Figure 3.1.2: Model problem for reservoir simulation.

In this section, only a very brief introduction to reservoir simulation has been given. For an extensive treatment of reservoir simulation, we refer to, e.g., [Ewin] and [Peac]. In the remainder of this chapter, we will study a discretization of (3.1.13) – (3.1.15) by mixed finite elements (section 3.2). In section 3.2 we will also describe a splitting of (3.1.13) into a hyperbolic and an elliptic part. Local grid refinement is applied to follow the moving front of water. In sections 3.3 and 3.4 we consider the hyperbolic and the elliptic part, respectively. The elliptic part involves a multigrid solver. In general, our presentation in the sequel of this chapter has been based upon [ScJa].
§ 3.2 Discretization and datastructures

In the previous section, we observed that isothermal two-phase fluid flow through porous media can be modelled by the following system of equations:

\[
\frac{\partial s_m}{\partial t} = q_m - v u_m \quad m = 1, 2, \quad (3.2.1a)
\]

\[
u_m = k_m \cdot k^{-1} u \quad m = 1, 2, \quad (3.2.1b)
\]

\[
c \cdot p + v u = q \quad (3.2.1c)
\]

\[
v p + k^{-1} u = v, \quad (3.2.1d)
\]

together with initial values for the saturations \( s_m \), and boundary values for \( u \).

System (3.2.1) can be split into an elliptic part and a hyperbolic part, similar to the theory of partial differential equations. In this respect, the part of the response of the reservoir which is immediately recognized in the reservoir, is defined to be elliptic. If, e.g., the pressure is increased in one well, then only a few days later (which is immediately, compared with a production period of years), pressure in other wells has changed. The elliptic part of (3.2.1) is given by (3.2.1c) and (3.2.1d). According to the character of the elliptic part, discretization of (3.2.1c), (3.2.1d) should lead to a system of coupled equations, for which advanced solution methods may be (or better: must be) applied. The hyperbolic part of (3.2.1) is represented by (3.2.1a) and (3.2.1b). The part of the response of the reservoir, which is recognized after a few months (i.e. which moves through the reservoir with a limited speed) is called hyperbolic. If, e.g., water is injected in one well, then saturations in other wells do not change within a few months. For this reason, discretization of the hyperbolic part does not lead to a system of coupled equations, thus allowing relatively simple solution methods to solve the hyperbolic part of the reservoir simulation equations. The solution method of (3.2.1) described here is known as the IMPES--scheme (implicit in pressure, explicit in saturations).

Figure 3.1.2 shows a model problem for reservoir simulation: a front of water moves from the injection well towards the production well. It is obvious that the mathematically most interesting phenomena take place just before, at, and just behind the front. Other regions of the reservoir (containing just water or just oil, in the most simple case) are numerically of less interest. Therefore, an efficient numerical model that takes into account these observations should almost necessarily lead to a numerical
method that uses local grid refinement: everywhere in the reservoir, the discretization is so fine (or coarse) that the local physical phenomena are described accurately. This means that the computational grid is fine at the front, while in other regions the grid may be coarser. The local grid refinement is complicated by the fact that we have to deal with a moving front, thus leading to an adaptive procedure: when the front is approaching, the grid should be refined; when the front has passed, the grid may be coarsened.

In this section we already observed that the work for the elliptic part consists mainly of the solving of a (large) set of linear equations, which can be done more efficiently on regular grids than on locally refined grids. Therefore, the local grid refinement as mentioned above and as much regularity as possible should be balanced somehow. This can be done by a method outlined in [Bra1], which effects locally refined grids by dynamically switching on and off the individual grid points in a regular grid structure. This makes it possible to solve the elliptic part by means of a multigrid method. In section 3.4 we will consider the multigrid method in more detail.

In order to keep the datastructure on locally refined grids manageable, a low—order approximation of the fields is selected. Since the elliptic part is formulated as a set of first order differential equations, lowest—order mixed finite elements of Raviart and Thomas [RaTo] may be used to approximate the elliptic part. The basic element is a grid block, with pressures and saturations approximated in the center of the block, and flows through interfaces, as in figure 3.2.1:

![Figure 3.2.1: Basic element.](image)

Integration of (3.2.1c) over grid block i then yields:

\[
q_i p_i + \sum_j (\vec{v}_j \cdot \vec{u}_j) = q_i, 
\]

(3.2.2)
in which \( j \) runs over the interfaces of block \( i \); \( h_j \) is the mesh width of interface \( j \), while \( c_i \) and \( q_i \) are integrals of \( c \) and \( q \) over block \( i \), respectively. Discretization of (3.2.1d) leads to (it reflects the positions of the interface neighbours of block \( i \)):

\[
h_j \sum_i (s_i p_i) + \sum_l w_{jl} u_l = v_j \quad (3.2.3)
\]

Index \( i \) runs over the neighbouring blocks of interface \( j \). The discrete version of \( k^{-1} \mathbf{u} \) (\( W \mathbf{u} \) with \( W \equiv k^{-1} \)) is obtained through introduction of the indicator function \( \theta_j \): the direction of \( \theta_j \) is perpendicular to interface \( j \), while its size is one at interface \( j \) and zero at other interfaces. Furthermore, \( \theta_j \) is linear:

![Diagram](image)

**Figure 3.2.2**: Indicator function \( \theta_j \) for interface \( j \).

Then the approximation of \( \mathbf{u} \) is \( \sum_j u_j \theta_j \), and hence

\[
W \cdot \mathbf{u} \to \sum_l w_{jl} u_l \quad \text{in which} \quad w_{jl} = \intl_{\Omega} (W \theta_j) \cdot \theta_l \, d\Omega. \quad (3.2.4)
\]

and furthermore:

\[
\mathbf{v} \to v_j = \intl_{\Omega} \mathbf{v} \cdot \theta_j \, d\Omega. \quad (3.2.5)
\]

In (3.2.4), index \( l \) runs over all interfaces. Only a few (7 at most) terms \( w_{jl} \) do not vanish. The sparsity pattern is shown in figure 3.2.3:
Figure 3.2.3: Sparsity pattern for (3.2.4). If matrix $W$ is diagonal (isotropic problem), then only 3 terms are non-zero.

The hyperbolic part is discretized in the same way. Integration of (3.2.1a) over block $i$ leads to:

\[
a_i \frac{s^k_{m,i} - s^{k-1}_{m,i}}{\Delta t} + \sum_j (s^{h,j})u^{n-1}_{m,j} = q^k_{m,i},
\]

(3.2.6)

in which:

- $n$ : time level,
- $\Delta t$ : time step,
- $a_i$ : area of block $i$,
- $u^k_{m,j}$ : flow of phase $m$ through interface $j$,
- $q^k_{m,i}$ : integral of $q_m$ over block $i$.

In the summation, $j$ runs over all interface neighbours of block $i$. Equation (3.2.1b) is discretized up-stream, as:

\[
u^k_{m,j} = \frac{k_{m,iup}}{\Sigma_{m',iup}^m} u^{n-1}_{j},
\]

(3.2.7)

with $k_{m,iup}$ the permeability for phase $m$, evaluated in the up-stream block neighbour $iup$ of interface $j$. In the summation, $m'$ runs over the phases.

The complete solution process of the reservoir simulation equations (3.2.1) consists of an alternating sequence of hyperbolic and elliptic updates, on a locally refined grid. In
the hyperbolic update, grid blocks are refined or unrefined, according to some criterion. The elliptic update deals with a fixed grid on which the system of coupled equations (3.2.2), (3.2.3) is solved at each time step. This time step is determined in the hyperbolic part and represents the minimum time that a fluid particle stays in a grid block. For more details about the discretization we refer to [ScJa].

As was mentioned before a few times, computations are carried out on a locally refined grid, which is adapted in the hyperbolic part of the computational process. Together with a suitable datastructure to describe the locally refined grid, we consider refinement of grid blocks itself. The basic refinement refines one grid block into 4 identical smaller ones, as in figure 3.2.4:

![Basic refinement](image)

**Figure 3.2.4:** Basic refinement.

The basic refinement automatically leads to levels of grid blocks. The starting grid always consists of 4 grid blocks ($2 \times 2$-grid). We define blocks of this coarsest grid to have level one. When a grid block is refined (figure 3.2.4), the level of the 4 new grid blocks (sons) is one level higher. The same holds for interfaces. Furthermore, we define fine and coarse blocks:

**Definition 3.2.1:** A grid block in a (uniform or locally refined) grid is called *fine* if it is not refined. All refined blocks are called *coarse*.

Definition 3.2.1 implies that grid blocks of different levels may build the finest grid, which is formed by fine grid blocks. Both the hyperbolic and elliptic update seek for a solution defined on the finest grid.

The datastructure that describes locally refined grids is the following [ScJa]:

---
* for each block : 4 interface neighbours,
* for each interface : 2 block neighbours (which are not necessarily of the same level),
* for each block : number of its first son (only for coarse blocks),
* for each interface : number of its first son (only for coarse interfaces),
* for each block : level number and coordinates of the center,
* for each interface : orientation (horizontal or vertical).

The complete datastructure is used in the elliptic part, while only part of it is used in the hyperbolic part (orientation and interface sons are not used there).

The hyperbolic part of the reservoir simulator deals with local grid (un)refinement. Since blocks and interfaces may vanish, but can also be created, it is quite difficult to keep the datastructure regular. The character of the datastructure, in combination with the (hyperbolic and elliptic) update, makes the complete algorithm as it is not very well-suited to implementation on vectorcomputers. With the purpose to increase the performance of the algorithm, we will study the algorithm for its parallel properties in chapters 4 to 6. In the next sections we discuss the hyperbolic and elliptic part of the simulator in more detail. We conclude this section with an example of a locally refined grid, which is representative for the type of grids that occur during the computational process, as in figure 3.2.5:

![Example of a locally refined grid](image)

*Figure 3.2.5: Example of a locally refined grid.*
§ 3.3 The hyperbolic update

In the previous section, we observed that numerical solution of the reservoir simulation equations consists of an alternating sequence of hyperbolic and elliptic updates. Discretization of the hyperbolic part yields (cf. (3.2.6) and (3.2.7)):

\[
a_i \frac{s_{m,i}^n - s_{m,i}^{n-1}}{\Delta t} + \sum_j (s_{h,j}^n) u_{m,j}^{n-1} = q_{m,i}^n.
\]

(3.3.1)

\[
u_{m,j}^n = \frac{k_{m,i^{up}}}{\sum_{m'} k_{m',i^{up}}} u_{j}^{n-1}.
\]

(3.3.2)

For each phase m the new saturations \(s_{m}^n\) are solved from (3.3.1) which is called the saturation update. The phase flow update (3.3.2) determines the phase flows \(u_{m,j}^n\). The hyperbolic update itself is an alternating sequence of saturation updates and phase flow updates, including some refinement/unrefinement technique, in order to mark the moving waterfront by local grid refinement. This leads to the following hyperbolic algorithm:

do each time step
saturation update
if (necessary) then
refine/unrefine current grid
determine new time step
endif
phase flow update

**Figure 3.3.1:** Algorithm for the hyperbolic update.

In general, it is not necessary to adapt the computational grid for each time step. Therefore, the if–statement in figure 3.3.1 controls the refinement/unrefinement procedure. Adapting the grid only at each third time step is a quite common strategy. Since the time step itself strongly depends on the size of the fine grid blocks (i.e. minimum size: finest–level grid blocks), it is necessary to adapt the time step after a grid update.
As was mentioned in the previous section, local grid refinement should be applied when the front of water approaches, while the grid should be unrefined (coarsened) when the front has passed. If the waterfront is approaching, then we have typically the following situation, as in figure 3.3.2:

![Diagram](image)

**Figure 3.3.2:** Approaching (from the left) waterfront; local refinement.

If, e.g., the waterfront has just reached blocks i2 and i3, then part of the oil in blocks i2 and i3 has flowed to block i1, and has been substituted by water. A straightforward way to check this is to compare saturations of neighbouring grid blocks. For each fine grid block (i.e. not refined) saturations are compared with saturations of neighbouring, up-stream, fine grid blocks. For block i1, this means that saturations of blocks i2 and i3 have to be considered (and probably of some other neighbours). If the discrepancy exceeds some upper limit, then block i1 will be refined. A similar procedure is applied for unrefinement. If saturation differences are found to be below some lower limit, then the underlying grid block will be unrefined. This concept is called the indicator concept: for each fine block, saturation gradients are determined (indicators), whereafter blocks are refined or unrefined. Blocks with indicators in between upper and lower limits will not be refined or unrefined.

Figure 3.3.1 shows that the hyperbolic update is complicated by the application of local grid refinement. In the next section we will consider the elliptic update, which is carried out on the fixed grid dictated by the hyperbolic update.
§ 3.4 The elliptic update

In sections 3.1 and 3.2, we have observed that the modelproblem (3.1.13) — (3.1.15) for isothermal, immiscible two–phase flow through porous media can be split into a hyperbolic and an elliptic part. In the previous section, we discussed the solution of the (discretized) hyperbolic part. In this section, we will consider the solution of the elliptic part, which has been diocretized by mixed finite elements.

The elliptic part of the reservoir simulation equations is given by:

\[ c \cdot p + V u = q, \quad (3.4.1) \]
\[ \nabla p + W \cdot u = v, \quad (3.4.2) \]

with "no normal–flow" boundary conditions, i.e. no flow outside the reservoir:

\[ u \cdot n = 0. \quad (3.4.3) \]

The pressure p and the total flow u have to be determined in the elliptic part. In [ScJa], it is shown that solving \((p, u)\) from (3.4.1) — (3.4.3) is equivalent to solving

\[ \min \left[ \int_{\Omega} \left[ \frac{1}{2} c p^2 + \frac{1}{2} W \cdot u - v \right] d\Omega \right], \quad (3.4.4) \]

under the constraint (3.4.1).

The mixed finite element discretization of (3.4.1), (3.4.2) leads to (cf. section 3.2):

\[ c_i p_i + \sum_j (h_j) u_j = q_i, \quad (3.4.5) \]
\[ h_j \sum_i (p_i) + \sum_k w_{jk} u_k = v_j, \quad (3.4.6) \]

for each finest–level block \(i\) and interface \(j\), respectively. To solve (3.4.5), (3.4.6), an iterative solution process (a Gauss–Seidel type method) is studied in [ScJa]. If we consider a uniform refined grid, then one iteration involves all combinations of 4 grid blocks, having one vertex in common. (Note that these grid blocks have the same level, since the grid is uniform.) Such a combination of 4 grid blocks is called a quadruple.
The computational work related to a quadruple consists of the solution of a linear system of 8 equations. Each quadruple is related to 4 blocks and 4 internal interfaces:

![Diagram of a quadruple](image)

**Figure 3.4.1:** Quadruple, which contains 4 blocks (x) and 4 internal interfaces (n).

There are 4 equations of type (3.4.5) and 4 equations of type (3.4.6) that are associated with these blocks and interfaces, respectively. Flows through boundary interfaces (o) are considered as known. Solution of the 8x8 linear system (a quadruple update) leads to new values of pressures in blocks (x) and to new values of flows through internal interfaces (n). One iteration then is a complete series of quadruple updates.

In [ScJa], the convergence of this process is proven, provided that the series of quadruple updates touches the complete computational grid (i.e. each interface, except boundary interfaces, is at least once an internal interface of a quadruple, during each iteration). However, convergence can be extremely slow, especially in the case of many grid blocks. An interesting property of the iterative process is that after the first iteration, the equations of type (3.4.5) have been solved. The iterative process is then continued until the equations of type (3.4.6) are satisfied up to a given accuracy.

In the case of compressible flows (c > 0), the linear system corresponding to a quadruple is non-singular and can be solved by standard Gaussian elimination. However, for incompressible flows (c = 0) the system is singular. Hence, some alternative solution technique is required to solve it. For more details about incompressible flows we refer to [ScJa]. For the remainder of this section we will implicitly assume compressible flows.

The convergence of the previously described Gauss-Seidel process may also be extremely slow in situations with practically relevant values of the permeability. In [ScJa], multigrid is used to speed-up the convergence of the iterative process. For a general introduction to multigrid methods, we refer to, e.g., [StTr], [Bra1] and [HKWZ].
As was done in section 3.2, we start with a coarsest grid consisting of 4 grid blocks (2×2, level 1). If the basic refinement is applied to each block of the coarsest grid maxl−1 times, then a series of uniform grids \( l = 1, \ldots, \text{maxl} \) is formed. The solution \((p,u)\) is desired at the finest grid (which is formed by all grid blocks of level maxl). At each grid (level), the iteration process (i.e. smoothing process) consists of the series of quadruple updates as described before. A quadruple at level \( l \) is a combination of 4 grid blocks of level \( l \), having one vertex in common. At coarser levels \( l = 1, \ldots, \text{maxl}−1 \), the series of quadruple updates leads to a correction of the finest–level solution.

In multigrid methods the relation between grids of different level is defined by prolongation and restriction operators. The prolongation operator maps quantities from one level to the next finer level. Distinction is made between block and interface quantities. The prolongation of block quantities (e.g., pressure) is denoted by \( P_{\phi} \), which assigns the block quantities to all 4 sons of the underlying block. Prolongation of interface quantities (e.g., flows) is given by \( P_{u} \). For so-called interface–born interfaces, the two son–interfaces receive the same value as the father–interface. Block–born interfaces receive the average of the quantities, belonging to the two parallel, one level coarser, interfaces. Note that interface–born interfaces have a father–interface, in contrast with block–born interfaces. The adjoint mappings of \( P_{\phi} \) and \( P_{u} \) are denoted by \( P_{\phi}^{*} \) and \( P_{u}^{*} \), respectively.

Restriction operators are mappings from one level to the next coarser level. Again, there is a distinction between blocks and interfaces. The block quantities belonging to the 4 sons of one coarse block are summed and divided by 4 (\( R_{\phi} \)), while the interface quantities corresponding to the two sons of a coarse interface are summed and divided by 2 (\( R_{u} \)). The adjoint mappings of \( R_{\phi} \) and \( R_{u} \) are given by \( R_{\phi}^{*} \) and \( R_{u}^{*} \), respectively.

The multigrid process uses coarser grids to improve the solution on the finest grid. Before stating the equations on coarser grids we describe the discrete equations (3.4.5), (3.4.6) in matrix vector representation (the superscript \( f \) denotes the finest level):

\[
C_{p}^{f} f + D_{u}^{f} f = q^{f},
\]

\[
-D_{p}^{f} f + W_{u}^{f} f = y^{f},
\]

in which \( C^{f} \) is a diagonal matrix, representing the compressibility, \( D^{f} \) and \( -D_{p}^{f} \) represent the divergence and gradient operator, respectively, and \( W^{f} \) is the matrix, corresponding
to the inverse permeability $k^{-1}$ ($\equiv W$). Furthermore, $\mathbf{q}^f$ and $\mathbf{v}^f$ are vectors representing the right-hand sides of (3.4.5) and (3.4.6).

In [ScJa] the multigrid method is implemented as a FAS-scheme, which means that full approximations of $\mathbf{p}^f$ and $\mathbf{u}^f$ are computed on coarser grids. The difference between the new and the old approximations on the coarse grid is prolonged to the finest level and is added to the old approximation on the finest level. The equations on coarser grids are similar to the equations on the finest grid:

$$
C^c \mathbf{p}^c + D^c \mathbf{u}^c = \mathbf{q}^c \quad (= P^*_\phi (\mathbf{q}^f - C^f \mathbf{p}^f) + C^c R_{\phi, \mathbf{p}^f}),
$$

$$
-D^c \mathbf{p}^c + W^c \mathbf{u}^c = \mathbf{v}^c \quad (= P^*_u (\mathbf{v}^f - W^f \mathbf{u}^f) + W^c R_{\mathbf{u}^f}).
$$

in which the superscript $c$ refers to the coarser grids. The matrices $C^c$, $D^c$ and $W^c$ are determined as:

$$
C^c = P^*_\phi C^f P_{\phi}, \quad D^c = P^*_\phi D^f P_u, \quad W^c = P^*_u W^f P_u.
$$

The vectors $\mathbf{q}^c$ and $\mathbf{v}^c$ represent the right-hand sides $\mathbf{q}^c$ and $\mathbf{v}^c$ (for more details we refer to [ScJa]). The matrices $C^c$, $D^c$ and $W^c$ can be computed once and for all before the start of the multigrid process, since they do not change during the process. Their sparsity pattern is the same as the sparsity pattern of the corresponding finest-level matrices (cf. section 3.2). The vectors $\mathbf{q}^c$ and $\mathbf{v}^c$ depend on the finest-level solution, and therefore change after each multigrid iteration.

So far, we have assumed to use uniform, not locally refined grids. In the case of locally refined grids, the multigrid algorithm as given in figure 3.4.2 does not change. Following the method outlined in [Bral], smoothing at level $l$ is equivalent to a series of quadruple updates, consisting exclusively of blocks of level $l$. In this way, it is not necessary to consider other, more complicated regions than quadruples. The multigrid algorithm, using this smoother, supports local refinement in a natural way.
The final multigrid algorithm for solving the elliptic part of the reservoir simulation equations then may be represented as figure 3.4.2:

\begin{verbatim}
compute \( C^f, D^f, q^f, W^f \) and \( v^f \) at finest level \( \text{maxl} \)
for \( l = \text{maxl}-1,1,-1 \)
\hspace{1cm} compute \( C^l, D^l, W^l \)
initial guess \( p^f, u^f \) at finest level \( \text{maxl} \)
do until convergence
\hspace{1cm} for \( l = \text{maxl}-1,1,-1 \)
\hspace{2.8cm} compute restrictions \( p^l, u^l \)
\hspace{1cm} compute \( q^l, v^l \)
\hspace{1cm} solve exact at coarsest level 1
\hspace{1cm} prolongate \( p^1, u^1 \) to level 2
\hspace{1cm} for \( l = 2, \text{maxl} \)
\hspace{1.9cm} smooth at level \( l \)
\hspace{1cm} if \( (l \neq \text{maxl}) \) then
\hspace{1cm} \hspace{1cm} prolongate \( p^l, u^l \) to level \( l+1 \)
\hspace{1cm} endif
\hspace{1cm} compute block and interface residuals at finest level
\end{verbatim}

\textbf{Figure 3.4.2:} Multigrid algorithm (sawtooth scheme) for the elliptic part of the reservoir simulator.

The smoother at each level \( l, l \geq 2 \), is just the series of quadruple updates. Since the coarsest grid consists of 4 grid blocks \((2*2)\), the exact solver is identical to the quadruple update, i.e. on the coarsest grid, the smoother reduces to a direct solver.

In the next chapter, we will study the reservoir simulator for its parallel properties and certain parallelization strategies will be chosen. Chapters 5 and 6 will cover in detail the parallel implementation of the hyperbolic and elliptic update, respectively.
4

DOMAIN DECOMPOSITION AND PARALLEL COMPUTING

Abstract

In this chapter we consider domain decomposition techniques for the parallelization of grid-oriented problems. The application we have in mind (reservoir simulation with local grid refinement) uses multigrid and adaptive grid refinement, in order to simulate the physical process in an efficient way. Both multigrid and adaptive grid refinement complicate the parallel method and have their impact on datastructures. Also load-balancing of the work-load over the processors is studied in this chapter.

Parts of this chapter and chapter 5 have been accepted for publication in Parallel Computing, to appear in 1990.
§ 4.1 Introduction

In the previous chapter we have introduced our model problem, which is described by the following set of equations for \( N \) phases instead of 2:

\[
\begin{align*}
\frac{\partial s_m}{\partial t} &= q_m - v u_m & m = 1, \cdots, N \\
u_m &= k_m k^{-1} u & m = 1, \cdots, N \\
c \cdot p + v u &= q & (4.1.1c) \\
v p + k^{-1} u &= v & (4.1.1d)
\end{align*}
\]

We consider a domain \( \Omega \subset \mathbb{R}^2 \), in which the number of phases is denoted by \( N \), phase flows by \( u_m (m = 1, \cdots, N) \), the pressure by \( p \) and saturations by \( s_m (m = 1, \cdots, N) \). In the most simple case, \( N \) equals 2 and the phases are oil and water. The permeabilities \( k_m (m = 1, \cdots, N) \), which are given functions of the saturations and space coordinates, and the source terms \( q_m (m = 1, \cdots, N) \) are assumed to be known. In the mass conservation equation (4.1.1c), \( c \) denotes the compressibility (non-negative), while in Darcy's equation (4.1.1d) the right-hand side \( v \) represents external forces, e.g. gravity. Initial saturations are given:

\[
s_m = s_m^0 \hspace{1cm} t = 0, \ m = 1, \cdots, N, \hspace{1cm} (4.1.2)
\]

and at the boundary of \( \Omega \), "no normal-flow" conditions hold (\( \nu \) denotes the outward normal):

\[
u_m \cdot \nu = 0 \hspace{1cm} x \in \partial \Omega, \ m = 1, \cdots, N. \hspace{1cm} (4.1.3)
\]

Furthermore,

\[
u = \sum_{m=1}^{N} u_m, \ k = \sum_{m=1}^{N} k_m, \ q = \sum_{m=1}^{N} q_m. \hspace{1cm} (4.1.4)
\]

Equations (4.1.4) reflect total flow, total permeability and total source term, respectively.
The system of equations (4.1.1) may be split into a hyperbolic part (4.1.1ab) and an elliptic part (4.1.1cd). Coupling between the two parts is established by the total flow $u$ and the total permeability $k$.

A mixed finite element method is used to discretize (4.1.1), as described in section 3.2. Local grid refinement and dynamical grid adaptation are applied in order to reduce computer execution time and memory space, without affecting accuracy and convergence of the computational process.

Since local grid refinement and dynamical grid adaptation are used, it is not obvious how to vectorize the solution process of (4.1.1). Moreover, only vectorization of large parts of a computer code may lead to a significant reduction of computer execution time (Amdahl's law, [Amda]; for a generalization, see [DDSV]). We have been looking for other ways to speed-up the complete computational process. One such other way is provided by parallel computing. In some parallel computers, processors are vector-processors themselves. However, though lack of obvious vectorization has led us to parallel computing, and though we are not looking for speed-up from the vector facilities of the processors, we expect that our parallel approaches may be combined with vectorization techniques for problems large enough to have some significant local uniformity.

In chapter 2, we have observed that many algorithms are not well-suited for parallel computers. It is therefore necessary to reconsider our reservoir simulator, for its parallel possibilities, and eventually develop a parallel algorithm which takes advantage of the parallel architecture. In order to select a practical parallelization method it is, of course, first necessary to have some insight in commercially available parallel computers, and secondly to study the main aspects of the reservoir simulator (local grid refinement, dynamical grid adaptation).

As was mentioned in chapter 1, several types of parallel computers have entered the market during the last few years. For our application, we consider parallel computer systems with a small number of processors, relative to the problem size. Such systems, with shared or local memory, are at the moment more common than systems with a (very) large number of processors. In chapters 5 and 6, the relation between number of processors and problem size will be investigated in more detail.

According to chapter 3, the hyperbolic part (4.1.1ab) of the reservoir simulation equations (4.1.1) may be characterized as local. In the hyperbolic part, saturations and phase flows are updated. Both updates only need local quantities: the saturation update in a grid block is determined by phase flows through neighbouring interfaces, while phase flows are determined by up-stream saturations. The elliptic part (4.1.1cd), however, has
global properties. Discretization of (4.1.1cd) leads to a system of equations for the pressure and the total flow, which should be solved simultaneously. This global character is reflected in the solution method for the elliptic part, which consists of a multigrid process.

Roughly spoken, a standard multigrid process has four essential ingredients: smoothing, restriction, prolongation and coarse-grid correction. The smoothing process is typically a local process, which means that computations concerning a grid block (or point) only involve neighbouring grid blocks (or points). E.g., Jacobi, Gauss–Seidel and line Gauss–Seidel, all act more or less locally. The smoothing is performed on a sequence of coarse grids, which is called the coarse-grid correction. A direct method then solves the coarsest-grid equations. It is the fine-to-coarse-grid sequence which represents the global characteristics of the physical problem. The coarsest grid consists of only a few blocks, i.e. computation of the solution in one coarsest-grid block requires information of all other coarsest-grid blocks, which represents global information. The prolongation and restriction transfer quantities from one grid to another.

The choice of an efficient parallelization method is often negatively affected by global effects, since local computations can often be executed more easily in parallel. Since multigrid has a global character, it is necessary to study the parallel properties of multigrid processes. In section 2 of this chapter we will consider attempts proposed in literature to parallelize multigrid. In section 3 our final choice for the parallel method is described, which will be based upon domain decomposition methods. Sections 4 and 5 describe additional datastructures and load-balancing aspects of the domain decomposition.
§ 4.2 Parallel multigrid: an overview

It is well-known that multigrid methods are among the fastest methods for solving partial differential equations. For certain problems, multigrid methods have been shown to be of optimal complexity order, i.e., the computational work to reach a solution within the discretization error is proportional to the number of unknowns, see, e.g., [StTr]. After parallel computers entered the market, many investigators have tried to combine the advantages of both multigrid and parallel computing. This has led to two distinct approaches. The first of these approaches is to study the parallel properties of standard multigrid algorithms, while the other approach is to develop essentially new multigrid–like algorithms, that are more suitable for parallel processing.

Brandt was among the first to recognize the opportunities of combining multigrid and parallel computing as early as in 1981. In [Bra2] he suggests various ways of programming multigrid methods for parallel computers, although preferring standard multigrid algorithms for their rather general convergence proofs. In such standard multigrid algorithms, computations on one level at a time are performed by the processors. A few years later, several authors considered new multigrid–like algorithms, in order to efficiently use (highly) parallel architectures. In these algorithms, computations on more than one level at a time are performed in parallel. In this section we will give an overview of both approaches, which inspired the choice for our problem.

A straightforward way to parallelize grid problems is based on domain decomposition methods. The computational domain is split into a number of subdomains, equal to the number of processors of the parallel computer. We first assume this number to be small compared with the grid size. If the underlying grid is uniform, then the domain decomposition can be done in such a way that the work–load is equally balanced over the processors. In, e.g., [BeBo] and [MCQu], partitioning strategies for non–uniform problems, balancing the work–load over the processors, are considered.

In many cases, the smoothing part of multigrid methods is a local process, as described in the previous section. Operations on grid points have a local character, which means that only neighbouring grid points are involved in computations. Therefore, computational work concerning internal points of a subdomain is restricted to the subdomain itself. However, points near an internal boundary may need information from points of other subdomains. To handle this situation, an overlap of subdomains is proposed in, e.g., [BHMQ] and [Thol], as is shown in figure 4.2.1:
The dotted lines mark the overlap regions. The overlap between the upper two subgrids has been drawn. In this way, boundary points of one subdomain are internal points of a neighbouring subdomain. At least after each smoothing sweep, the solution in overlap regions should be updated for the next smoothing sweep, thus leading to communication between the subdomains. In local memory machines, communication implies physical data transfer between processors. Since data transfer costs generally depend upon the distance between the processors, neighbouring subdomains should be mapped onto neighbouring processors.

In multigrid methods, a sequence of grids is used. So far, we have only considered the finest-level grid. On coarser grids usually the same finest-level smoothing process is applied. Since the number of subdomains is small, neighbouring subdomains at the finest level are also neighbouring subdomains at a number of coarser levels. Therefore, smoothing on coarser grids makes efficiently use of the finest-level subdomain mapping. However, on very coarse grids, some subdomains may run out of points, which implies that the corresponding processors are idle during the smoothing process on these grids, possibly degrading parallel performance. However, in [BHQ] it is shown that these idle processors do not seriously degrade parallel efficiency, if the number of processors is small compared with the problem size.

The treatment of coarse grids, in view of parallelism, is extensively studied in [BHQ]. Firstly, C-level is defined as the coarsest level where essentially full concurrency is maintained, i.e. subdomains do not yet run out of points. Several techniques are suggested to handle the multigrid levels which are below C-level. A very simple approach is not to go below C-level. In this case, V-cycles (see [StTr]) become U-cycles, which degrade convergence: low-frequency error components corresponding to
levels below C–level are not damped anymore, which leads to lower convergence rates. Of course, these lower convergence rates are avoided if C–level equations are solved exactly. However, if C–level is too fine this may be quite expensive. Another approach suggests another solver for the C–level equations. If C–level occurs on a rather fine level, this solver may be quite expensive too.

An alternative way to handle grids below C–level is based upon the concept of sleeping nodes. If a subdomain runs out of points, the corresponding processor is put asleep, while other processors continue computations on coarser grids, until the coarsest grid, which is handled by only one or a very few processors. When execution visits finer levels again, sleeping nodes are awakened by receiving a message, whereafter they continue their computational task. This method was actually implemented in [BHMQ] on an Intel iPSC hypercube of dimension 5 (32 nodes), which led to a speed–up of 13.3, on a finest grid of 64*64 grid points. In [Thol] similar results are reported on the CalTech hypercube (U–cycle).

We can actually show that idle processors on coarse levels do not have much impact on overall parallel efficiency, provided that the problem size is sufficiently large compared with the number of processors. To this end, suppose that we have a finest grid of \( 2^l \times 2^l = 4^l \), a coarsest grid of \( 2^2 \times 2^2 \), \( l \) multigrid levels and a sawtooth cycle ("upward" V–cycle, see [StTr]). The work for one processor is proportional to

\[
4^1 + 4^{l-1} + 4^{l-2} + \ldots + 4^1 = \frac{4^{l+1} - 4}{3}.
\]  \hspace{1cm} (4.2.1)

If 8 processors are available and if we assume that all levels \( l \geq 3 \) can take full advantage of the processors, while levels 2 and 1 can only use 4 and 2 processors, respectively, then the parallel computational work is proportional to

\[
\frac{4^1}{8} + \frac{4^{l-1}}{8} + \ldots + \frac{4^3}{8} + \frac{4^2}{4} + \frac{4^1}{2} = \frac{4^{l+1} - 4^3}{3 \cdot 8} + 6.
\]  \hspace{1cm} (4.2.2)

Disregarding any communicational costs, theoretical speed–ups and efficiencies become

\[
S(8) \approx 7.4; \quad E(8) \approx 0.92, \quad \text{for } l = 4,
\]  \hspace{1cm} (4.2.3)

and

\[
S(8) \approx 7.8; \quad E(8) \approx 0.98, \quad \text{for } l = 5.
\]  \hspace{1cm} (4.2.4)
These results indeed show that idle processors on coarse levels do not seriously degrade parallel performances, provided that problem sizes are large enough.

Apart from the treatment of coarse grids, communication between subdomains must be considered carefully. As is shown in figure 4.2.1, overlap regions are used during the smoothing process in order to handle internal boundaries. If, e.g., red–black relaxation is used then it is necessary to perform a communication sweep after each half–relaxation (i.e. only the red points or only the black points), thus updating grid points in overlapping regions. Since communication is performed only between neighbouring subdomains, we consider the subdomain configuration created by the domain decomposition. Roughly spoken, there are two possibilities: squares and strips, leading to 4 neighbouring subdomains and 2 neighbouring subdomains, respectively. In local memory machines each processor has to send 4 messages to neighbouring processors, in the case of squares, while in the case of strips, only two messages have to be sent.

Which subdomain configuration is the optimal one depends on the specific local memory machine. If messages can be sent simultaneously, then the squares seem to be in favour, since for squares message lengths are smaller than for strips. In other cases experiments have to indicate the optimal subdomain configuration. For shared memory machines data transfer is not required, but synchronization between processors is still necessary.

In the above lines, we have argued that standard multigrid methods can be implemented rather efficiently on parallel computers, if the problem size is large compared with the number of processors. Under these circumstances, idle processors on coarse levels do not seem to have much impact on speed–up factors. Although we have excluded highly parallel machines like the Connection Machine (64k one–bit processors at most) from our study, we briefly consider the influence of coarse grids to the parallel performance of multigrid implementations on such machines. It is clear that for these machines the condition concerning number of processors and problem size does not hold, since each processor can handle only a very small amount of data. Actually, in [McBr], an implementation of a standard multigrid method on the Connection Machine assigns only one grid point to each processor, while each coarser–grid point is always assigned to the same processor as its corresponding fine–grid point. Parallel computing times on fine grids are equal to parallel computing times on coarse grids now, which means that coarse grids have become as important as fine grids for the parallel performance.
This phenomenon becomes more clear if we consider again our previous example of \( l \) multigrid levels, with a coarsest grid of 2×2 and a sawtooth cycle. If 256 processors are available, then the parallel computational work is proportional to:

\[
\frac{4^l}{256} + \cdots + \frac{4^4}{256} + 1 + 1 + 1 = \frac{1}{256} \cdot \frac{4^{l+1} - 4^4}{3} + 3. \tag{4.2.6}
\]

Note that at the coarser levels (levels 1 to 3, in this case) more and more processors are idle. Again disregarding communicational costs, theoretical speed-ups and efficiencies for \( l = 4 \) and \( l = 5 \) become:

\[
S(256) = 85; \quad E(256) \approx 0.33, \quad \text{for } l = 4, \tag{4.2.7}
\]

and

\[
S(256) \approx 171; \quad E(256) \approx 0.67, \quad \text{for } l = 5. \tag{4.2.8}
\]

Results (4.2.7) and (4.2.8) show that idle processors at coarser levels can seriously degrade parallel efficiency, if the number of processors is proportional to the problem size. For this reason, it is clear that V–cycles are more favoured than W–cycles.

The above example shows that parallel implementations of standard multigrid algorithms on massively parallel computers are unable to use effectively all processors on coarse grids. In [FrMB] the idle processors of the Connection Machine are used to solve many coarse problems simultaneously and to combine the results in an optimal way by some prolongation operator, in order to increase parallel efficiency. The basic idea, in \( d \) dimensions, is that for each fine grid, there are \( 2^d \) natural coarse grids. If the coarse–grid corrections are combined in an appropriate way, this non–standard multigrid method may lead to faster convergence rates than for standard multigrid methods. A similar technique, which also uses idle processors to accelerate convergence, is proposed in [ChTu]. The coarse–grid residual is split into a number of components, e.g., a component dominated by low frequencies and a component dominated by high frequencies. The subproblems are then handled by idle processors.

Originally developed for massively parallel computers, this method of so–called residual filtering can also be applied to systems with fewer processors. In [ChTu], it is shown that this multigrid–like method can be effectively mapped onto a hypercube. In [DSMM] an aggregation/disaggregation technique is used to create a number of independent subproblems, which can be solved in parallel. Again, an appropriate
combination of results may lead to faster convergence and parallel efficiency. In [Muld] a semi-coarsening technique is applied to create more than one coarse-grid problem. This technique is similar to the one described in [FrMB].

Finally, we mention some other techniques for parallel multigrid, which do not fit into the ones described above. For example, in [GaRo], a reformulation of a standard multigrid method leads to a variation, which is called concurrent iteration. The concurrent iteration algorithm performs operations at all levels in parallel, as well as operations per level in parallel. This method uses a large number of processors (order of problem size), and requires very large amounts of data transfer between processors to maintain stability. In [HaMC] another interesting technique is proposed. A composite grid is formed by a global coarse grid and a local fine grid, which represents a subregion of the global coarse grid. Both grids are uniform, and are used to find a solution on the composite grid in parallel. Boundary values for the fine grid are provided by the coarse grid. This method is called asynchronous fast adaptive composite grid method (AFAC), and handles locally refined grids in parallel.

An alternative method to parallelize multigrid methods with local grid refinement is proposed in [Mier]. First, a domain decomposition technique is used to split all uniform grids. Then, another domain decomposition is applied to locally refined regions of the computational domain. In this way, each local grid is split into a number of subregions. Theoretical studies concerning computational and communicational complexity show that for a priori known local grids, a similar speed-up can be achieved as for problems without local refinement.

In this section, we have observed that many grid problems are parallelized by means of some domain decomposition method. Since we are interested in parallel computers with a few (relative to the problem size) processors, it seems natural to apply such a domain decomposition technique to the reservoir simulation problem, which is complicated by the adaptive local grid refinement, as described in section 3.3. In the next sections we will consider local grid refinement and its impact on the parallel domain decomposition method.
§ 4.3 Parallel domain decomposition

In the previous section we observed that many parallel algorithms for solving grid problems (e.g. multigrid) are based on domain decomposition techniques. Especially for parallel computers with a few quite powerful processors, domain decomposition seems to be a natural and straightforward way to parallelize the original sequential algorithm. For such machines (either shared or local memory), synchronization and/or communication may seriously degrade the parallel performance, as was shown in chapter 2. It is therefore essential to reduce the number of synchronization or communication points, in order to reach a significant speed-up over the one-processor algorithm. This reduction of synchronization and communication points can be achieved by splitting the computational domain $\Omega$ into only a small number of subdomains, which together build the complete domain. Again, we assume that the number of subdomains is small compared with the problem size, but has the same order of magnitude as the number of processors.

The domain decomposition techniques, as described in section 4.2, have been designed for grid problems of a regular (uniform) structure. For such types of grids it is fairly easy to split the domain once and for all into a number of subdomains of equal size. If the number of subdomains is equal to the number of processors, then each subdomain can be assigned to one processor. Each processor handles its own subdomain, which means that the computational process for a subdomain is completely performed by the corresponding processor. The domain decomposition introduces internal (or: artificial) boundaries between the subdomains. In order to obtain a correct solution, it is, of course, necessary to treat these internal boundaries in a somewhat different way than the internal of the subdomains. This leads to exchange of data between the subdomains, across the internal boundaries, which involves synchronization of the processors. In the case of a local memory machine, exchange of data between subdomains also involves physical data transfer between the processors. In figure 4.3.1, we show a uniform grid and a possible domain decomposition into 4 subdomains. Since the subdomains are of equal size, each processor has the same work-load. In this case there are no serious synchronization points, hence no significant loss of parallel efficiency, caused by idle processors, is encountered. However, for local memory machines, physical data transfer between the processors is still required, which probably degrades the parallel efficiency.
Figure 4.3.1: Uniform grid and possible domain decomposition.

So far, we have only considered uniform grids, for which balancing of the work–load over the processors was a fairly easy task. However, for the reservoir simulator, as described in chapter 3, we have to deal with two additional problems: multigrid and local grid refinement. A very simple example of a locally refined grid is shown in figure 4.3.2:

Figure 4.3.2: Example of a locally refined grid.

It is not a priori clear how to split the computational domain of figure 4.3.2 into a number of subdomains, such that the work–load is equally balanced over the processors. In the case of 4 subdomains, e.g., a possible domain decomposition may be as follows:
Figure 4.3.3: Possible subdomain configuration of figure 4.3.2.

Figure 4.3.3 shows a domain decomposition which balances the work-load over 4 processors. However, problems arise if a multigrid solver is applied to obtain a correct solution on the complete grid.

The coarse-grid correction, one of the ingredients of multigrid processes, corrects the approximation of the finest-level solution, using coarser grids. The one-level coarser grid, associated with the grid of figure 4.3.2, is the following:

Figure 4.3.4: One-level coarser grid, associated with the grid of figure 4.3.2.

From figure 4.3.3 we observe that the coarse-grid correction does not fit into the subdomains of this specific domain decomposition. The internal boundaries between the
subdomains intersect individual coarser blocks, an effect that becomes only worse if another coarser grid is used. Hence, a coarse-grid correction, using the subdomain configuration of figure 4.3.3, will actually lead to exchange of many data between the subdomains, therefore potentially degrading the parallel efficiency. To avoid this problem, one might strive for internal boundaries, which do not intersect coarse blocks:

![Alternative subdomain configuration](image)

**Figure 4.3.5**: Alternative subdomain configuration of figure 4.3.2.

Note that no overlap regions between the subdomains are used. The argument described above may be summarized as the "multigrid condition":

**Definition 4.3.1**: The multigrid condition is defined as the practical restriction on the domain decomposition not to intersect grid blocks of some specified (coarse) level.

This multigrid condition, although favourable for handling coarser grids in parallel, strongly reduces the number of possible domain decompositions. In figure 4.3.5, the work-load is still perfectly balanced over 4 processors, but in many cases this property will not hold. Moreover, if dynamical grid adaptation is used, then the refinement structure of each subdomain grid will change as a consequence of refining and unrefining grid blocks, thus unbalancing the work-load over the processors. This load-balancing problem may be solved by dynamically adapting the configuration of subdomains, in order to have the work-load balanced over the processors, during the complete computational process. The multigrid condition, however, should be satisfied, which
implies that only coarse blocks with their refinement structure, may be re-assigned from one subdomain to another.

The considerations above concerning multigrid and parallel efficiency in combination lead to a parallel domain decomposition method, which can be outlined as follows.

The computational domain $\Omega$ is split into a (small) number of subdomains. The amount of computational work in each subdomain should ideally be equal. This condition usually conflicts with the multigrid condition, which forces internal boundaries not to intersect coarse grid blocks. It is clear that some compromise has to be made. Furthermore, the amount of work along internal boundaries has to be small, in order to avoid too much synchronization and communication. This can be achieved by keeping the subdomains connected, which puts another restriction on the dynamical load—balancing process. An additional restriction for this load—balancing is forced by the multigrid condition, which permits only coarse blocks to be re-assigned from one subdomain to another.

Application of this parallel domain decomposition method leads to some problems that have to be solved. In the first place, the internal boundaries between the subdomains have to be described in an appropriate way. The internal boundary datastructure must be able to handle local grid refinement and dynamical grid adaptation in a flexible way, guaranteeing no significant loss of parallel efficiency. Second, the load—balancing process also has its impact on internal boundaries, since it re-configures the subdomain configuration. Moreover, the load—balancing process itself should be efficient, because it is included in the computational process. For local memory machines, the load—balancing process may be expensive, since possibly large amounts of data are transferred from one processor to another. We will discuss these problems in sections 4.4 and 4.5, respectively.

Another problem that needs our attention, is the treatment of very coarse grids in the multigrid process. In section 3.4, the coarsest grid consists of only 4 blocks. Evidently, for this coarsest grid there is just one way to create 4 subdomains: one coarse block per subdomain. Therefore, we usually apply domain decomposition to a one level finer, uniform, grid, consisting of 16 blocks. Special techniques, as already mentioned in section 4.2, will be used to handle the coarsest grid. In the next section, we will describe these so—called agglomeration and de—agglomeration techniques in detail.

An example of a fine grid, split into 4 subdomains, is shown in figure 4.3.6. In order to make the splitting more visible, the subdomains have been separated.
In figure 4.3.6, the numbers denote the number of finest-level blocks. The ratio between the maximum number of finest-level blocks and the average number, which is a measure for the degree of load-balancing (1.0 is optimum), is 1.50.

In this section, the number of subdomains was not explicitly fixed to the number of processors. In the case of regular, uniform grids, however, the number of subdomains should be equal to an integer multiple of the number of processors, for obvious reasons. This also holds in the case of locally refined grids, at least for local memory machines. Since increase of the number of subdomains leads to more internal boundaries between the subdomains, and hence to more physical data transfer between the processors, it is then recommended to have the number of subdomains exactly equal to the number of processors. However, for shared memory machines, it may sometimes be profitable to create more subdomains than there are processors. In this way, the operating system of the computer (e.g., Alliant) may balance the work-load by directing the subdomains to the processors, which may lead to a speed-up over the case in which there are as many subdomains as processors. In a computer environment with more users, creation of more subdomains leads to more subjobs, which may positively affect the throughput of the machine [Care]. This actually happens on, e.g., CRAY-XMP and CONVEX C-240 machines.
§ 4.4 Additional datastructures and agglomeration processes

In the previous section, we have seen that the domain decomposition method introduces internal boundaries between the subdomains. In order to obtain a correct solution, some computational work across the internal boundaries has to be done. The multigrid process on very coarse grids requires a somewhat different treatment than the multigrid process on finer grids. As pointed out in section 4.2, agglomeration and de-agglomeration techniques may be necessary to handle very coarse grids efficiently. In this section, we will consider the consequences of both internal boundaries and multigrid on coarse grids, in relation with our datastructures.

The domain decomposition introduces internal boundaries between the subdomains. These internal boundaries may be described by an additional datastructure, the choice of which is complicated by essentially the following situation. Suppose, the internal boundary is just between i1 and i2 in figure 4.4.1:

```

<table>
<thead>
<tr>
<th>i1</th>
<th>i2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

left subdomain       right subdomain
```

*Figure 4.4.1: No refinement yet.*

In the hyperbolic part of the reservoir simulator, adaptive local grid refinement is applied (cf. section 3.3). During the computational process, it therefore may happen that block i1, in the left subdomain, will be refined, while block i2, in the right subdomain, will not (yet) be refined (figure 4.4.2):

```

  |     | i2 |
  |     |    |
  |     |    |

left subdomain       right subdomain
```

*Figure 4.4.2: Only refinement in the left subdomain.*
In this situation, the interface coinciding with the internal boundary in the left subdomain is refined, while its geometrically equivalent interface in the right subdomain is not refined. (In this respect, geometrically equivalent means the same interface, but in another subdomain. Note that this interface exists in the left subdomain, and independently in the right subdomain datastructure.) Since the internal boundary in the right subdomain does not change, we only want to adapt the internal boundary datastructure in the left subdomain. This implies that the internal boundary datastructure has to satisfy some locality condition.

In section 4.3, we discussed the domain decomposition in connection with the multigrid process. As a consequence of the multigrid condition (definition 4.3.1), the internal boundaries do not intersect grid blocks of some specified coarse level. Therefore, internal boundaries coincide with interface neighbours of these grid blocks. Since load–balancing is also applied to these grid blocks, this coarse level will be referred to as the load–balancing level, denoted by minl (to be compared with C–level in [BHMQ]). Interfaces, which correspond to the load–balancing level, are called basic interfaces. As we already noticed in section 4.3, the load–balancing level and the coarsest level are usually not identical (minl ≥ 1).

Since internal boundaries now coincide with basic interfaces, we consider one such basic interface, in one subdomain. The refinement structure of some basic interface \( j_1 \) may be described by a tree structure:

```
interface                     level
                                  \( j_1 \)                                                \( \text{minl} \)
                                  \( j_2 \)                                                   \( . \)
                                  \( j_3 \)                                                   \( . \)
                                  \( j_4 \) \( j_5 \) \( j_6 \) \( j_7 \)
                                  \( 0 \) \( 0 \) \( j_{10} \) \( j_{11} \) \( j_{12} \) \( j_{13} \) \( 0 \) \( 0 \) \( \text{maxl} \)
```

*Figure 4.4.3:* Tree structure of a basic interface.

Note that when, e.g., interface \( j_4 \) is refined, we only have to adapt the two zeroes in the lower–left corner of the tree, which is a local change. If basic interface \( j_1 \) is part of an internal boundary, then an analogous tree can be built in the subdomain of the geometrical equivalent of \( j_1 \). Both trees can be matched, and it is immediately clear which interfaces are equivalent. The number of interfaces, which coincide with a basic
interface (e.g., \( j_1 \)) is bounded by

\[
\max_1 \sum_{k=\min_1}^{\max_1} 2^{k-\min_1} = 2^{\max_1-\min_1+1} - 1.
\]

(4.4.1)

So far, the internal boundary datastructure is local (i.e. restricted to a subdomain). Refinement or unrefinement of interfaces in one subdomain does not have any impact on the datastructure of other subdomains. In general, each subdomain has more than one basic interface which coincides with an internal boundary. For each such basic interface, an interface tree is built, which leads to a segmented description of the internal boundary, with segment lengths given by (4.4.1). In order to distinguish between basic interfaces and to match the correct interface trees, we additionally need some non-local properties. Therefore, for each basic interface, coinciding with an internal boundary, we assignate:

* the number of the neighbouring subdomain, and
* a pointer to the geometrical equivalent basic interface in the neighbouring subdomain.

These two quantities are the only non-local properties for each basic interface. Furthermore, note that each interface should have two block neighbours. Since internal boundary interfaces have only one block neighbour in their own subdomain, its index in the segmented internal boundary datastructure is assigned to it as the second block neighbour, which is again of local type.

In section 4.2, we observed that complete parallel execution of standard multigrid processes on coarse grids is difficult to achieve without overhead, since coarse grids consist of only a few grid blocks (or points). Some ways to handle these coarser grids are studied in, e.g., [BHMQ] and [Thol] (cf. section 4.2). One of these ways introduces the concept of sleeping processors ([BHMQ]), using fewer and fewer processors to handle coarser grids, until the coarsest grid is reached, which is handled by only one processor. The sleeping processors become idle during this so-called agglomeration process. Closely related to this concept is the stepwise agglomeration process, which agglomerates two subdomains to one subdomain, until the coarsest grid is reached. (Note the subtle difference: in the stepwise agglomeration, subdomains do not run out of points.) The stepwise de-agglomeration is just the agglomeration process in reverse order. During the de-agglomeration process, processors are awakened again.
As is shown in figure 4.4.4, the stepwise agglomeration process assumes a regular subdomain configuration. Since two subdomains are agglomerated to one subdomain (step 1), whereafter restriction to a coarser grid is performed (step 2), this coarser grid should fit into the structure of the subdomain configuration. The final agglomeration step is given by step 3.

However, this regular subdomain configuration has dramatic consequences for the load-balancing process. The load-balancing process re-assigns only grid blocks of load-balancing level from one subdomain to another. This leads, e.g., to the following subdomain configuration (cf. section 4.3):

Figure 4.4.5: Load-balanced subdomain configuration.

If the stepwise agglomeration is applied to the subdomain configuration of figure 4.4.5, then there arises, after step 1:
Figure 4.4.6: Subdomain configuration of figure 4.4.5, after step 1 of the stepwise agglomeration.

The one-level coarser grid of figure 4.4.6 does not fit into the structure of the subdomain configuration. Hence, stepwise agglomeration and load-balancing are two conflicting options.

One way to avoid this conflict is to apply another, more flexible, way of agglomeration, which may be characterized as direct agglomeration: grid blocks of load-balancing level are agglomerated directly to a grid on one processor, which handles this grid and coarser grids. In [BHMQ] this concept is known as a shared multigrid solver. The subdomain configuration of figure 4.4.5, e.g., is directly agglomerated in the following way:

Figure 4.4.7: Direct agglomeration/de-agglomeration.
Apart from the datastructure for coarse grids that are encountered during the stepwise agglomeration, the stepwise agglomeration itself does not require an additional datastructure. This is due to the fact that the subdomain configuration is regular and does not change during the computational process (no load-balancing). For the direct agglomeration process, which allows load-balancing and hence irregular subdomain configurations, we need a simple mapping, which describes step 1 of figure 4.4.7. The processor that handles the coarser grids, contains the datastructure for these grids. In this way, each grid block of load-balancing level, being part of one of the subdomains is mapped onto its equivalent grid block in the agglomerated grid. Note that we also need the inverse mapping. If grid blocks are re-assigned from one subdomain to another, we only have to adapt this mapping. This additional datastructure, which may be regarded as some shadow-grid, introduces only a little overhead over the usual datastructures.

During either type of agglomeration there are idle processors. In some theoretical oriented papers these idle processors are identified to form a serious problem in maintaining parallel efficiency. In [BHMQ] and in section 4.2, however, it is shown that idle processors on coarse grids do not seriously degrade parallel efficiency, if the problem size is large as compared with the number of processors. Our experimental results in chapter 6 are in agreement with the results in [BHMQ]. We considered both stepwise and direct agglomeration. We will show that these techniques, when applied carefully, have only a small impact on parallel efficiency.
§ 4.5 Load–balancing

In chapter 3 we have observed that it suffices to apply local grid (un)refinement only in the hyperbolic part of the reservoir simulator, since the hyperbolic part has a local character. Therefore, the load–balancing process as proposed in section 4.3 should be used only in this hyperbolic part, after a grid update. In this section we will discuss the load–balancing process in more detail.

As was outlined before, the work–load for each processor changes as a consequence of grid (un)refinement. The load–balancing process tries to balance the work–load equally over the processors, during the complete simulation. To satisfy the multigrid condition, only blocks of load–balancing level, including their refinement structure, may be re–assigned from one subdomain to another. This re–assignment, which is a sequential process, requires communication between the subdomains (local memory machines: data transfer). For this reason, load–balancing should be applied only if the work–load distribution has become too unbalanced.

Multigrid methods are of optimal complexity order for certain partial differential equations. Therefore, it seems natural to assume the work–load to be proportional to the number of finest–level grid blocks. A perfectly balanced work–load then corresponds to an equal number of finest–level blocks in each subdomain. However, the multigrid condition (definition 4.3.1) inhibits, in general, a perfect load–balancing. The load–balancing process tries to minimize the ratio between maximum number of fine blocks and average number of fine blocks, over the subdomains. Note that this ratio is bounded from below by 1.0. Furthermore, some tolerance ratio should be given. As soon as the ratio maximum/average is below the tolerance, the load–balancing process stops and usual computations continue. Typical values of this tolerance are in the range 1.1 – 1.3.

Apart from the multigrid condition, the load–balancing process is restricted by another condition. Since communication degrades parallel efficiency, it is essential to minimize the amount of communicational work, which implies that the length of internal boundaries should be small. This can be achieved by keeping the subdomains connected, which means that blocks of load–balancing level (minl) can only be moved from one subdomain to a neighbouring subdomain.

As was remarked before, the load–balancing process requires communication between the subdomains. In order to keep the amount of data that has to be communicated limited, the refinement structure of a locally refined block of load–balancing level should be represented in a compact form. As an example of this compact form, we consider the following example of a locally refined block of load–
balancing level (note that it is only necessary to know whether a grid block is refined or not):

![Diagram of load balancing level with refinement](image)

**Figure 4.5.1:** Block of load—balancing level with refinement.

This block of load—balancing level is split into the following sequence:

![Diagram of refinement sequence](image)

**Figure 4.5.2:** Refinement sequence of figure 4.5.1.

In figure 4.5.2, we numbered the 4 sons of a refined block as is shown. This sequence can be defined uniquely by storing the numbers of refined blocks:

\[
\begin{align*}
n\text{seq}(1) &= 1, \\
n\text{seq}(2) &= 1.2, \\
n\text{seq}(3) &= 1.3, \\
n\text{seq}(4) &= 1.4, \\
n\text{seq}(5) &= 1.2.3, \\
n\text{seq}(6) &= 1.3.2, \\
n\text{seq}(7) &= 1.4.1, \\
n\text{seq}(8) &= 1.4.4.
\end{align*}
\]
Apart from this sequence, other information (e.g., coordinates) concerning blocks and interfaces has to be transferred to the new subdomain. However, using the sequence described above, it is not necessary to transfer block and interface numbers.

The concepts of this section lead to the following load-balancing algorithm:

```
if (load-balancing necessary) then
    L1
    determine ratio maximum/average
    if (ratio > tolerance) then
        find block of load-balancing level, such that
        * ratio maximum/average decreases most
        * subdomains remain connected
        if (block is found) then
            [make refinement sequence
             communicate between subdomains
             connect block to new subdomain
             disconnect block from old subdomain
             adapt internal boundaries]
            go to L1
        else
            go to L2
        endif
    else
        go to L2
    endif
endif
L2 continue (load-balancing terminates)
```

**Figure 4.5.3**: Load-balancing algorithm.

Note that the part of the algorithm in brackets is performed at the level of subdomains; the other part is done at some central stage.

The connection of a block of load-balancing level to its new subdomain mainly consists of adapting the grid datastructure. The refinement sequence is used to build the refinement structure again, in such a way that new block and interface numbers fit in the already existing datastructure of the new subdomain. The disconnection of the underlying block of load-balancing level is also done by adapting the grid datastructure
of its previous subdomain. Renumbering of blocks and interfaces is necessary, since blocks and interfaces have disappeared from the datastructure. Note that connection and disconnection can be done simultaneously on two processors.

The load-balancing process causes internal boundaries between the subdomains to change. Since the load-balancing may be done such that also neighbour relations between subdomains change, some global operations are required to adapt the internal boundaries. The shadow-grid, which was introduced in the previous section, is updated too.

We conclude this section with an example of a load-balanced grid after a number of re-assignment steps. The original domain decomposition, at the start of the simulation, has split the domain into 4 strip-shaped subdomains.

![Diagram](image)

**Figure 4.5.4:** Finest-level grid, split into 4 subdomains, with load-balancing.

In figure 4.5.4, the ratio between the maximum number of finest-level blocks and the average number of finest-level blocks equals 1.15 (cf. 1.50 in figure 4.3.6).

In the next two chapters we will discuss in more detail the parallel implementations for the hyperbolic part and the elliptic part of the reservoir simulator.
A PARALLEL HYPERBOLIC UPDATE

Abstract

In this chapter we consider a parallel algorithm for the hyperbolic part of the reservoir simulation equations, based upon the concepts discussed in chapter 4. After recalling briefly the sequential hyperbolic update, we study a parallel update, with special attention to communicational work. Parallel implementations on both local memory and shared memory machines, which are represented by an NCube/four and an Alliant FX/4, respectively, are considered. Experimental results, together with analysis and expectations for more processors, are presented. These experimental results have been obtained by a code, including the implementation of the dynamical load-balancing process on both types of computers.
§ 5.1 Introduction and sequential algorithm

In this chapter, the parallel domain decomposition, described in chapter 4, will be applied to the hyperbolic part of the reservoir simulation equations. In this first section we repeat briefly the sequential algorithm for the hyperbolic update, as it was discussed in more detail in section 3.3.

The hyperbolic part of the reservoir simulator is discretized by mixed finite elements. The basic element is a grid block, with the saturations discretized in the center of the block, and flows through interfaces of the block. The hyperbolic update consists of an alternating sequence of saturation updates and phase flow updates, including a refinement/unrefinement technique, in order to follow the moving waterfront by locally finer grids. In algorithm, we have:

```
do each time step
    saturation update
    if (necessary) then
        refine/unrefine current grid
        determine new time step
    endif
    phase flow update
```

*Figure 5.1.1: Sequential hyperbolic algorithm.*

In section 5.2, a parallel version of the hyperbolic algorithm, based on domain decomposition, is studied, including computational and communicational aspects and load-balancing. Section 5.3 discusses implementation aspects and experimental results, including modelling of computational and communicational costs on a local memory machine (NCube). Section 5.4 deals with implementation, experiments and performance modelling on a shared memory machine (Alliant).
§ 5.2 Parallel hyperbolic algorithm

In chapter 4 we have proposed a domain decomposition technique to parallelize the problem of numerical reservoir simulation, with local grid refinement and multigrid. The domain decomposition splits the computational domain into a few subdomains, which are handled by the processors. For a correct solution, communication between the subdomains, across internal boundaries, is needed, leading to data transfer between processors in case of local memory machines. In this section the elements of the sequential hyperbolic algorithm of figure 5.1.1 are studied for their parallel properties, thus leading to a parallel hyperbolic algorithm.

The first part of the sequential hyperbolic algorithm consists of the saturation update, which was given by (3.3.1):

\[
a_i \frac{s^{n+1}_{m,i} - s^n_{m,i}}{\Delta t} + \sum_j (\Delta h_j) u^n_{m,j} = q^n_{m,i}.
\]

(5.2.1)

Only one grid block at a time is involved in (5.2.1). The index \( j \) in the summation runs over all interface neighbours of \( i \). Since internal boundaries coincide with coarse interfaces (of load–balancing level), the saturation update (5.2.1) does not have to cross internal boundaries, provided that phase flows through finest–level interfaces at the internal boundaries are correct. Hence, for each block \( i \), the saturation update is restricted to just one subdomain and complete parallel execution is guaranteed.

So far, to obtain a correct hyperbolic update, no communication between subdomains is required, as long as phase flows through interfaces are correct. The phase flow update itself is given by (3.3.2):

\[
u^n_{m,j} = \frac{k_{m,i,up}}{\sum_{m'} k_{m',i,up}} u^{n-1}_j.
\]

(5.2.2)

For each interface \( j \), permeabilities are evaluated in the up–stream block neighbour of \( j \). (Note that up–stream is with respect to the direction of the total flow \( u \) through interface \( j \).) If interface \( j \) is in the internal of a subdomain, then its two block neighbours are in the same subdomain, thus leading to normal execution of (5.2.2). However, problems arise when interface \( j \) is part of an internal boundary, while its up–stream block neighbour is in a neighbouring subdomain:
Figure 5.2.1: Up-stream block neighbour of interface $j$ in neighbouring subdomain.

First, we assume that we have not applied local grid refinement yet. Then the geometrically equivalent interface $j'$ of $j$ exists in the up-stream subdomain, and is not refined:

![Diagram](image)

Figure 5.2.2: No local grid refinement.

Note that $u_j'$ equals $u_j$. During the phase flow update in the up-stream subdomain, phase flows through interface $j'$ have been updated by permeabilities, evaluated in block iup. To update the phase flows through interface $j$ (which should equal phase flows through $j'$), only these permeabilities are needed, together with an appropriate pointer to interface $j$. This pointer to interface $j$ is generated by the internal boundary datastructure, as described in section 4.4. The second block neighbour of interface $j'$ is a pointer to the index of $j'$ in the segmented internal boundary datastructure of the up-stream subdomain (step 1). This latter datastructure then contains the index of interface $j$ in the internal boundary datastructure of the down-stream subdomain (step 2), which is used to find $j$ (step 3):
Figure 5.2.3: Matching of equivalent internal boundary interfaces, no local refinement.

The case of local refinement does not lead to an essential different situation. In the up-stream subdomain, permeabilities corresponding to finest-level interfaces are collected, together with the pointer to the corresponding interface in the down-stream subdomain. Only the computation of phase flows in the down-stream subdomain is complicated by the local grid refinement, which is characterized by one of the following situations.

Figure 5.2.4: Situation 1: equivalents of $k'$ and $l'$ do not exist.

In situation 1, the geometrically equivalent interfaces of $k'$ and $l'$ do not exist. In this case, phase flows through interface $j$ are updated as

$$u_{m,j} = \frac{1}{2}(u_{m,k'} + u_{m,l'})$$

$$m = 1, ..., N.$$  \hspace{1cm} (5.2.3)
The tree structure of a segment of the internal boundary datastructure is used to find interface \( j \):

![Diagram of segmented internal boundary datastructure]

**Figure 5.2.5:** Matching in case of local refinement, situation 1.

Compared with figure 5.2.3, we have one additional step: the tree structure of a segment is used to find the equivalent interface of the father–interface of \( k' \) and \( l' \).

In situation 2, interface \( j \) of the down-stream subdomain is refined, while its equivalent \( j' \) in the up-stream subdomain is not refined:

![Diagram of situation 2: equivalents of \( k \) and \( l \) do not exist]

**Figure 5.2.6:** Situation 2: equivalents of \( k \) and \( l \) do not exist.

In this situation,

\[
u_{m,k} = u_{m,l} = u_{m,j'} \quad m = 1, \ldots, N, \tag{5.2.4}
\]

while matching is done again by using tree structures:
By the above figures, we have shown in which way communication concerning the phase flow update is handled. Note, that it is not known a priori which subdomains are up-stream subdomains (it is even possible that subdomains have up-stream blocks, as well as down-stream blocks, along just one internal boundary). This has consequences for physical data transfer (local memory machines), as we will see in the next section.

As was pointed out in section 3.3, local grid (un)refinement is performed by using the concept of indicators. For each finest-level grid block, saturations in up-stream block neighbours are considered. For grid blocks near an internal boundary, communication with the neighbouring subdomain may be necessary to compute correct indicators. Saturations are collected, together again with an appropriate pointer, for each up-stream grid block near an internal boundary. The pointer is identical to the one used in the phase flow update.

Local grid (un)refinement has another consequence for the parallel execution of the hyperbolic update. As was mentioned in section 3.3, the length of the time step is determined from information of all finest-level blocks. In a parallel environment, this implies that global communication is necessary to determine the correct time step. First, a minimum time step for each subdomain is computed, whereafter communication is needed to determine the global minimum. After collection of the local minimum time steps for the subdomains, the global minimum is broadcasted to the other subdomains.

Using the concepts of this section, together with the load-balancing process as described in section 4.5, we obtain the following parallel hyperbolic update algorithm:
do each time step/each subdomain
  saturation update
  if (necessary) then
    compute indicators
    communicate with neighbouring subdomains to correct indicators
    refine/unrefine current grid
    compute new local time step
    communicate globally to determine global minimum time step
    if (necessary) then
      load-balancing
    endif
  endif
  phase flow update
  communicate with neighbouring subdomains to complete phase flow update

Figure 5.2.8: Parallel hyperbolic algorithm.

In the remainder of this chapter, we consider implementations of the parallel hyperbolic algorithm on both local memory and shared memory machines, together with analysis and experimental results.
§ 5.3 Implementation and experimental results: NCube

In this section we will consider an implementation of the parallel algorithm for the hyperbolic part of the reservoir simulation equations, as discussed in the previous section. The parallel algorithm has been implemented on an NCube/four, a local memory machine based on the hypercube architecture with a maximum dimension of 4. As was described in chapter 1, a send–receive mechanism takes care of physical data transfer between the processors. Furthermore, a host computer loads nodeprograms and data on the nodes.

The parallel hyperbolic algorithm consists of 4 major parts: saturation update, complete phase flow update, (un)refinement and determination of a new minimum time step. No communication between subdomains is involved in the saturation update. Since each node handles its own subdomain, with only an additional datastructure to describe internal boundaries, the sequential saturation update can be copied to the parallel algorithm for execution on the NCube–nodes. (Note that the number of subdomains equals the number of nodes.)

In section 5.2, we have observed that during the phase flow update communication between neighbouring subdomains is necessary to obtain correct phase flows. In up–stream subdomains, for each finest–level internal boundary interface, up–stream permeabilities, together with an appropriate pointer, have to be sent to the down–stream neighbouring subdomain. Since data transfer between processors may be quite expensive, the permeabilities and pointers are collected in buffers, which are sent to the correct processors. The direction of the communication between two subdomains equals the direction of the total flow. Since the direction of the total flow may vary during the simulation, the direction of the communication may also change. In order to implement this phenomenon in a robust way (i.e. to exclude a deadlock situation), it is necessary to have two–way messages between two neighbouring subdomains. In many cases, one of the messages is empty. Furthermore, note that for each finest–level internal boundary interface the number of elements to be sent equals the number of phases plus one.

The correct computation of indicators also requires communication between neighbouring subdomains. Again, the direction of the total flow determines the direction of the communication. The same remarks as for the phase flow update hold, which means that message–sending is in two directions.

As was pointed out in chapter 1, the distance between two processors in the hypercube affects the data transfer time. Therefore, it is desirable to distribute the
subdomains over the processors in such a way that neighbouring subdomains are mapped onto neighbouring processors. In [SaS1] it is shown how to map grids on hypercubes using Gray–codes. In [ChSa] multigrid algorithms on a hypercube are considered, in such a way that coarser grids can be handled efficiently (with respect to data transfer times). In our case, we have only a few subdomains (processors), and it is straightforward how to map the subdomains on the processors, such that neighbouring subdomains are handled by neighbouring processors. Since subdomains do not run out of grid blocks at coarser levels (cf. section 4.3), we do not have to consider coarser grids separately.

After each (un)refinement step, a new minimum time step has to be computed (cf. section 3.3). In each subdomain, a local minimum time step is computed, after which the global minimum over all subdomains has to be determined. In [SNNi], some schemes to determine global quantities (e.g., convergence checking or time step computation) are considered. In our case, again, we have only a few subdomains and global quantities are computed in a straightforward way: collect the local quantities on one processor (host), compute the global quantity on the host and broadcast this quantity to all processors.

Experiments on the NCube/four have shown that simultaneously sending messages between neighbouring processors may cause a deadlock situation: processors are waiting for messages that will never arrive. The hypercube system also suffers from a situation in which too many messages are sent at the same time. To avoid such situations, messages are scheduled such that neighbouring processors do not write simultaneously to each other. This is achieved by numbering the subdomains, and allowing corresponding processors only to send messages after they received messages from processors corresponding to lower–numbered subdomains. For two subdomains, this leads to the following communication schedule. First, subdomain 1 fills the buffer corresponding to its internal boundary, and sends it to subdomain 2 (A). Subdomain 2 receives the message and processes its internal boundary (B), whereafter subdomain 2 fills a buffer and sends it to subdomain 1 (C). If we assume subdomain 1 to be the up–stream subdomain, then this latter message is empty, but has to be received by subdomain 1, before usual computations (D) in subdomain 1 continue. This communication schedule can be defined in a formal way:

**Definition 5.3.1:** The communication schedule between a number of subdomains is defined as the sequence of processes, which update internal boundary quantities, including data transfer.
For two subdomains, we have:

\[
\begin{array}{cccc}
2 & \cdots & \cdots & | & 1 & \cdots & \cdots & | & \cdots & \cdots & | & \cdots & \cdots & | & \cdots & D
\end{array}
\]

\[
\begin{array}{cccc}
D & | & A & | & D
\end{array}
\]

\[
\begin{array}{cccc}
\longrightarrow & \text{time}
\end{array}
\]

**Figure 5.3.1**: Communication schedule for 2 subdomains.

From figure 5.3.1, we observe that there are 3 steps (A, B and C), during which internal boundaries are processed (filling buffers (twice) and adapting internal boundary flows). In the case of more subdomains, the number of communication steps will increase. During each step, the internal boundary is processed. We will assume each step to be of equal costs, denoted by \( \beta \). The number of steps is given by \( k \). The communication schedule is used for phase flow updates and for correction of indicators. The phase flow update is performed at each time step, whereas (un)refinement of grid blocks is not. We assume that (un)refinement takes place each 3 time steps, thus computation of indicators is necessary only each 3 time steps.

Apart from this communicational work, which also reflects physical data transfer, there is computational work in the internal of the subdomains. The saturation update, the computation of indicators and the phase flow update are local processes in the sense that only neighbouring blocks and interfaces are involved in computations. Therefore, we estimate the computational work to be proportional to the number of finest–level grid blocks. The computational work per finest–level grid block per time step is denoted by \( \alpha \). Furthermore, the number of subdomains (processors) is denoted by \( p \), the number of finest–level grid blocks by \( N \) and the number of time steps by \( n \). Then the execution time of the parallel algorithm, assuming perfect load–balancing, is given by:

\[
T_p = n \left[ \frac{N}{p} \alpha + \left(1 + \frac{1}{3}\right) k \beta \right].
\] (5.3.1)

Note that in the sequential case \( (p=1) \), the execution time of the sequential algorithm \( (k=0) \) is also given by (5.3.1).

In order to verify the model given by (5.3.1), we have run a test problem which did not suffer from local grid (un)refinement. As usual, the domain \( \Omega \) is square, with a finest grid of 32*32 grid blocks. We consider two–phase flow, in which water flows in at the
lower–left corner, while oil flows out at the upper–right corner. Permeabilities are choosen fairly simple, i.e. the permeability \( k_m \) for phase \( m \) equals the saturation \( s_m \) of phase \( m \). Note that no local grid (un)refinement means that no indicators have to be computed, and hence no communication between subdomains is necessary. However, we computed indicators correctly, in order to test the model with full internal boundary work. The total simulation time is 1.5 time units, which corresponds to approximately 150 time steps. The subdomains are strip–shaped (figure 4.3.6). Table 5.3.1 shows some timing results for this problem (testproblem 1):

<table>
<thead>
<tr>
<th>( p )</th>
<th>( \text{time in seconds} )</th>
<th>( S )</th>
<th>( E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>586.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>302.1</td>
<td>1.94</td>
<td>0.97</td>
</tr>
<tr>
<td>4</td>
<td>174.0</td>
<td>3.37</td>
<td>0.84</td>
</tr>
<tr>
<td>8</td>
<td>115.4</td>
<td>5.10</td>
<td>0.64</td>
</tr>
</tbody>
</table>

*Table 5.3.1:* Time in seconds on NCube/four for testproblem 1.

\( S \) denotes the speed–up, relative to the sequential algorithm on one processor, and \( E \) the efficiency.

Substituting \( p=1, N=1024, n=150 \) and \( k=0 \) in (5.3.1), we compute \( \alpha \) from

\[
T_1 = n \cdot \frac{N}{p} \cdot \alpha = 586, \quad (5.3.2)
\]

leading to

\[ \alpha = 3.8 \cdot 10^{-3} \text{ s.} \quad (5.3.3) \]

Then, the value of \( \beta \) is computed from \( T_2 \) (\( k=3 \), as pointed out in figure 5.3.1):

\[ \beta = 1.7 \cdot 10^{-2} \text{ s.} \quad (5.3.4) \]

In the case of 4 subdomains, the domain can be split into square–shaped subdomains or strip–shaped subdomains. If strips are used and the subdomains are numbered as in figure 4.3.6 (bottom–up), then the following communication schedule is applied:
subdomain

4 --|--
   ---+------------------
   2 4 comp.

3 --|--
     +++
     1 3 comp.

2 --|--
     +++
     1 3 comp.

1 --|--
     +
     2 comp.

|---| time

Figure 5.3.2: Communication schedule for 4 strip-shaped subdomains. The numbers denote the destination subdomains of the messages.

From figure 5.3.2, we observe that the first communication schedule, when parallel computations in the processors are synchronized, consists of 9 steps. The next communication schedule then takes effectively only 5 steps, since communication partly overlaps with computations. However, the determination of the minimum time step requires global communication, and hence parallel computations are more or less synchronized. This means that the situation in figure 5.3.2 only occurs within one time step, and thus only when indicators for local grid (un)refinement are computed. Formula (5.3.1) then becomes

\[ T_p = n \left[ \frac{N}{\nu} \alpha + (k + \frac{1}{3}k_e)\beta \right], \]  

(5.3.5)
in which \( k_e \) denotes the effective number of communication steps, necessary for correct computation of indicators.

In the case of 8 subdomains, we have the subdomain configuration, as depicted in figure 5.3.3:
Figure 5.3.3: Subdomain configuration for 8 subdomains.

The communication schedule then becomes:

```
<table>
<thead>
<tr>
<th>subd.</th>
<th>comp.</th>
<th>comp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 --&gt;</td>
<td>6 7</td>
<td>6 7</td>
</tr>
<tr>
<td></td>
<td>------+--</td>
<td>------++--</td>
</tr>
<tr>
<td>7 --&gt;</td>
<td>5 8</td>
<td>5 8</td>
</tr>
<tr>
<td></td>
<td>------++-</td>
<td>------++--</td>
</tr>
<tr>
<td>6 --&gt;</td>
<td>4 5 8</td>
<td>4 5 8</td>
</tr>
<tr>
<td></td>
<td>------+++</td>
<td>------+++--</td>
</tr>
<tr>
<td>5 --&gt;</td>
<td>3 6 7</td>
<td>3 6 7</td>
</tr>
<tr>
<td></td>
<td>------+++</td>
<td>------+++--</td>
</tr>
<tr>
<td>4 --&gt;</td>
<td>2 3 6</td>
<td>2 3 6</td>
</tr>
<tr>
<td></td>
<td>------+++</td>
<td>------+++--</td>
</tr>
<tr>
<td>3 --&gt;</td>
<td>1 4 5</td>
<td>1 4 5</td>
</tr>
<tr>
<td></td>
<td>------+++</td>
<td>------+++--</td>
</tr>
<tr>
<td>2 --&gt;</td>
<td>1 4</td>
<td>1 4</td>
</tr>
<tr>
<td></td>
<td>------++-</td>
<td>------++--</td>
</tr>
<tr>
<td>1 --&gt;</td>
<td>2 3</td>
<td>2 3</td>
</tr>
<tr>
<td></td>
<td>------++-</td>
<td>------++--</td>
</tr>
<tr>
<td></td>
<td>------++-</td>
<td>------++--</td>
</tr>
<tr>
<td></td>
<td>------++-</td>
<td>------++--</td>
</tr>
<tr>
<td></td>
<td>------++-</td>
<td>------++--</td>
</tr>
<tr>
<td></td>
<td>------++-</td>
<td>------++--</td>
</tr>
</tbody>
</table>
```

Figure 5.3.4: Communication schedule for 8 subdomains.
Figure 5.3.4 shows that $k=17$ and $k_e=8$. If we estimate the execution times for 4 and 8 processors (subdomains), using the aforesaided model, then the following table arises:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$T_p$ in seconds</th>
<th>measured time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>173.1</td>
<td>174.0</td>
</tr>
<tr>
<td>8</td>
<td>123.1</td>
<td>115.4</td>
</tr>
</tbody>
</table>

*Table 5.3.2: Estimated times for testproblem 1 with uniform grid.*

For 4 subdomains, the model (5.3.5) estimates the measured time in an excellent way. For 8 subdomains, however, there is a discrepancy between estimated and measured times. This may be explained by the following observation. Figures 5.3.2 and 5.3.4 show that asynchronous execution of computational work may lead to lower (effective) communication costs: communication partly overlaps with computational work, and hence processors are less idle. The determination of the minimum time step requires global communication. The new minimum time step is broadcasted to the processors, which then continue their computational task. For 8 subdomains, this broadcasting is somewhat more expensive, and hence processors are less synchronized than for 4 subdomains. If this lack of synchronization decreases the value of $k$ from 17 to, e.g., 14, then the estimated execution time becomes 115.5 seconds, which should be in very good agreement with the measured 115.4 seconds. The phenomenon described here may be compared with the reduction in effective data transfer, as encountered in section 2.4 for the original partition method, in the case of more processors.

If we have another look at figure 5.3.2, then we observe that the communication schedule is completely sequential. This is caused by the fact that subdomains communicate with neighbouring subdomains in an increasing order, i.e. a subdomain first writes to its lowest-numbered neighbour, etc. In the case of 4 strips, in particular for our uniform testproblem, it may be profitable to communicate in decreasing order with neighbouring subdomains, since a lower-numbered subdomain neighbour is always an up-stream neighbour. This strategy leads to the communication schedule, as shown in figure 5.3.5. This figure shows that $k=7$ and $k_e=5$, to be compared with $k=9$ and $k_e=5$ in the original communication schedule (figure 5.3.2). Application of (5.3.5) to this new situation leads to an estimate of 168.0 seconds, to be compared with an experimental execution time of 170.0 seconds.
subdomain

\[
\begin{array}{cccc}
4 & -\mathbf{--}| & 3\text{ comp.} & \mathbf{++++}|-\mathbf{------------}| & 3\text{ comp.} & \mathbf{++++}|-\mathbf{----}\mathbf{-} \\
3 & -\mathbf{--}| & 4\text{ comp.} & \mathbf{++++}|-\mathbf{------------}| & 4\text{ comp.} & \mathbf{++++}|-\mathbf{----}\mathbf{-} \\
2 & -\mathbf{--}| & 3\text{ comp.} & \mathbf{++++}|-\mathbf{------------}| & 3\text{ comp.} & \mathbf{++++}|-\mathbf{----}\mathbf{-} \\
1 & -\mathbf{--|--|--}| & 2\text{ comp.} & \mathbf{++++}|-\mathbf{------------}| & 2\text{ comp.} & \mathbf{++++}|-\mathbf{----}\mathbf{-} \\
\end{array}
\]

\[\text{time}\]

**Figure 5.3.5:** Alternative communication schedule for 4 strip-shaped subdomains.

In the case of 8 subdomains, we get the following figure:

\[
\begin{array}{cccc}
8 & --& | & 7\text{ comp.} & 8\text{ comp.} & \mathbf{76}\text{ comp.} & \mathbf{76}\text{ comp.} & \mathbf{76}\text{ comp.} & \mathbf{76}\text{ comp.} \\
7 & --& | & 8\text{ comp.} & 8\text{ comp.} & \mathbf{85}\text{ comp.} & \mathbf{85}\text{ comp.} & \mathbf{85}\text{ comp.} & \mathbf{85}\text{ comp.} \\
6 & --& | & 8\text{ comp.} & 8\text{ comp.} & \mathbf{854}\text{ comp.} & \mathbf{854}\text{ comp.} & \mathbf{854}\text{ comp.} & \mathbf{854}\text{ comp.} \\
5 & --& | & 7\text{ comp.} & 7\text{ comp.} & \mathbf{763}\text{ comp.} & \mathbf{763}\text{ comp.} & \mathbf{763}\text{ comp.} & \mathbf{763}\text{ comp.} \\
4 & --& | & 6\text{ comp.} & 6\text{ comp.} & \mathbf{632}\text{ comp.} & \mathbf{632}\text{ comp.} & \mathbf{632}\text{ comp.} & \mathbf{632}\text{ comp.} \\
3 & --& | & 5\text{ comp.} & 5\text{ comp.} & \mathbf{541}\text{ comp.} & \mathbf{541}\text{ comp.} & \mathbf{541}\text{ comp.} & \mathbf{541}\text{ comp.} \\
2 & --& | & 4\text{ comp.} & 4\text{ comp.} & \mathbf{41}\text{ comp.} & \mathbf{41}\text{ comp.} & \mathbf{41}\text{ comp.} & \mathbf{41}\text{ comp.} \\
1 & --& | & 3\text{ comp.} & 3\text{ comp.} & \mathbf{32}\text{ comp.} & \mathbf{32}\text{ comp.} & \mathbf{32}\text{ comp.} & \mathbf{32}\text{ comp.} \\
\end{array}
\]

\[\text{time}\]

**Figure 5.3.6:** Alternative communication schedule for 8 subdomains.
Now we have $k=12$ and $k_e=7$ (to be compared with $k=17$ and $k_e=8$), and substituting these values in (5.3.5) leads to an estimated time of 109.5 seconds. Experimentally, we observed 106.9 seconds. The small discrepancy may be explained again by asynchronous computational tasks, which overlap with communicational work. For 16 subdomains (with a 4*4 subdomain configuration: squares), we get $k=28$ and $k_e=9$ for the original communication schedule (increasing order) and $k=19$ and $k_e=9$ for the alternative schedule (decreasing order), leading to estimated times of 115.5 and 92.6 seconds, respectively. For the original schedule, no additional speed-up as compared with 8 processors is reached. In the alternative case, however, the parallel performance is improved, regarding the speed-up of 6.33, relative to the sequential algorithm.

In figure 5.3.2, the communication schedule for 4 strip-shaped subdomains was shown. Instead of 4 strip-shaped subdomains, it is also possible to have square-shaped subdomains, in the following way:

![Diagram](image)

*Figure 5.3.7: Square-shaped subdomains.*

The original communication schedule for 4 square-shaped subdomains is depicted in figure 5.3.8. We observe that $k=8$ and $k_e=6$, and substitution in (5.3.5) yields 171.5 seconds, while 168.4 seconds was experimentally found. The alternative communication schedule amounts to $k=7$ and $k_e=5$, and hence to 168.1 seconds. We measured 165.4 seconds. These results show that for the uniform testproblem, square-shaped subdomains are somewhat better than strip-shaped subdomains.
The speed-up $S$, for $p$ processors, is defined as the ratio between $T_1$ and $T_p$:

$$S(p;N) = \frac{T_1}{T_p} = \frac{nN\alpha}{n[N\alpha + (k+\frac{1}{3}k_e)\beta]} = \frac{1}{\frac{1}{p} + \frac{1}{N}(k+\frac{1}{3}k_e)\cdot\frac{\beta}{\alpha}} \quad (5.3.6)$$

From (5.3.3) and (5.3.4) we find that $\beta = 4.5\alpha$. Furthermore, $k+\frac{1}{3}k_e$ is an increasing function of $p$ only, which means that the speed-up $S$ goes to $p$, if the problem size $N$ becomes very large ($p$ fixed). Actually, if $p=16$, then we can compute the problem size we need for $S = \frac{p}{2}$:

$p=16, k+\frac{1}{3}k_e = 31 \implies N=2232$ (original comm. schedule)

$p=16, k+\frac{1}{3}k_e = 22 \implies N=1584$ (alternative comm. schedule)

However, if the number of processors becomes very large, for fixed problem size $N$, then the speed-up $S$ goes to zero. The parameter $\beta$ partly reflects physical data transfer, while $\alpha$ completely reflects computational work. It is clear from (5.3.6) then, that $S$ is reduced if the ratio $\frac{\beta}{\alpha}$ is increased: the computational speed becomes relatively faster than the speed of data transfer. If data transfer is very fast and can be ignored relative to computational costs, then work across internal boundaries is completely computational, and is proportional to the length of the boundary: $n\sqrt{N}$, in the case of strips, thus leading to the same asymptotic results. Again, the speed-up $S$ tends to zero for increasing $p$, since the number of internal boundaries increases as a function of $p$. 

---

**Figure 5.3.8**: Communication schedule for 4 square–shaped subdomains.
We conclude the discussion concerning testproblem 1 with a table, which summarizes some experimental and theoretical results.

<table>
<thead>
<tr>
<th>p</th>
<th>subdomain configuration</th>
<th>comm. schedule</th>
<th>estimated time in seconds</th>
<th>measured time in seconds</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>586.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>strips/ squares</td>
<td>increas./ decreas.</td>
<td>—</td>
<td>302.1</td>
<td>1.94</td>
</tr>
<tr>
<td>4</td>
<td>strips</td>
<td>increas.</td>
<td>173.1</td>
<td>174.0</td>
<td>3.37</td>
</tr>
<tr>
<td>4</td>
<td>strips</td>
<td>decreas.</td>
<td>168.0</td>
<td>170.0</td>
<td>3.45</td>
</tr>
<tr>
<td>4</td>
<td>squares</td>
<td>increas.</td>
<td>171.5</td>
<td>168.4</td>
<td>3.48</td>
</tr>
<tr>
<td>4</td>
<td>squares</td>
<td>decreas.</td>
<td>168.1</td>
<td>165.4</td>
<td>3.54</td>
</tr>
<tr>
<td>8</td>
<td>squares</td>
<td>increas.</td>
<td>123.1</td>
<td>115.4</td>
<td>5.08</td>
</tr>
<tr>
<td>8</td>
<td>squares</td>
<td>decreas.</td>
<td>109.5</td>
<td>106.9</td>
<td>5.48</td>
</tr>
<tr>
<td>16</td>
<td>squares</td>
<td>increas.</td>
<td>115.5</td>
<td>—</td>
<td>5.07</td>
</tr>
<tr>
<td>16</td>
<td>squares</td>
<td>decreas.</td>
<td>92.6</td>
<td>—</td>
<td>6.33</td>
</tr>
</tbody>
</table>

*Table 5.3.3: Summary of theoretical and experimental results.*

The second testproblem we have run was influenced by adaptive local grid (un)refinement, during a simulation time of 2.0 time units. Since the work-load for each processor changes each 3 time steps, the computational work is not exactly known. In order to compute idle times of processors, we therefore measured the average time over the processors, required for computational work, apart from the total execution time. The results are given in table 5.3.4:

<table>
<thead>
<tr>
<th>p</th>
<th>execution time in seconds</th>
<th>average comp. time in seconds</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>534.0</td>
<td>534.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>310.6</td>
<td>210</td>
<td>1.72</td>
<td>0.86</td>
</tr>
<tr>
<td>4</td>
<td>197.6</td>
<td>92</td>
<td>2.70</td>
<td>0.68</td>
</tr>
<tr>
<td>8</td>
<td>125.1</td>
<td>45</td>
<td>4.27</td>
<td>0.53</td>
</tr>
</tbody>
</table>

*Table 5.3.4: Times in seconds on an NCube/four for testproblem 2.*
From table 5.3.4, we observe two interesting phenomena. The first is that average idle times (approximately equal to execution time minus average computational time) do not grow absolutely if the number of processors is increased. This can be explained by the fact that in the case of more processors, for each processor the work–load is smaller, and hence idle times are shorter.

The second phenomenon is that the average time for computational tasks decreases with a factor of more than 2, if the number of subdomains is increased with a factor of 2. This is due to the fact that local grid (un)refinement requires some updating of the datastructure which describes the grid. If, e.g., blocks and interfaces vanish from the grid because unrefinement is applied, then renumbering of blocks and interfaces is needed to keep the datastructure efficient. Suppose, we have a large array of \( N \) elements and we have to remove \( m \) elements. The remaining elements should be renumbered. A simple (but not very efficient) way to achieve this is to start renumbering at the end of the array, for each element to be removed. The number of operations necessary to renumber completely then amounts to \( O\left(\frac{mN}{2}\right)\):

\[
\begin{array}{c}
\text{N,}\text{m} \\
\text{x} \quad \text{x} \quad \text{x} \quad \text{x} \quad \text{x} \\
\end{array}
\]

\( O\left(\frac{mN}{2}\right) \)

**Figure 5.3.9:** Renumbering on one processor.

If two processors (subdomains) are used and if we assume approximate equal work–loads, then the number of parallel operations reduces to \( O\left(\frac{mN}{4}\right) = \frac{1}{4} O\left(\frac{mN}{2}\right)\):

\[
\begin{array}{c}
\frac{N}{2},\frac{m}{2} \\
\text{x} \quad \text{x} \quad \text{x} \\
\text{x} \quad \text{x} \quad \text{x} \\
\end{array}
\]

\( O\left(\frac{mN}{4}\right) \)

**Figure 5.3.10:** Renumbering on 2 processors.

Hence, the number of parallel operations is reduced by a factor of 4, if the number of subdomains is increased by a factor of 2. Since the grid datastructure is quite complicated (local refinement, coupling of blocks and interfaces in several ways), this simple renumbering strategy is implemented, thus leading to a more than expected (at
first sight) reduction in computational work.

Table 5.3.4 shows that some processors are idle for a significant amount of time. This idleness is mainly caused by the fact that the work-load is not equally balanced over the subdomains (processors). The load-balancing procedure, as described in section 4.5, tries to balance the work-load more equally, by moving blocks of load-balancing level from one subdomain to another, preserving the connectedness of the subdomains. The ratio between the maximum number of fine blocks and the average number of fine blocks should be bounded from above by some tolerance.

Since application of the load-balancing procedure involves additional computational and communicational work, the choice of the tolerance depends on the underlying problem and computer. In the case of large problems it may be profitable to choose the tolerance in the range 1.1–1.3, while in other cases (e.g., local memory machines with slow data transfer) the tolerance may be somewhat larger. In table 5.3.5, we show some results:

<table>
<thead>
<tr>
<th>p</th>
<th>execution time without load-balancing, in secs.</th>
<th>tolerance</th>
<th>execution time with load-balancing, in secs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>310.6</td>
<td>1.3</td>
<td>281.6</td>
</tr>
<tr>
<td>2</td>
<td>310.6</td>
<td>1.4</td>
<td>296.4</td>
</tr>
<tr>
<td>4</td>
<td>197.6</td>
<td>1.7</td>
<td>199.1</td>
</tr>
<tr>
<td>4</td>
<td>197.6</td>
<td>2.0</td>
<td>192.9</td>
</tr>
</tbody>
</table>

*Table 5.3.5: Times for testproblem 2 with load-balancing.*

The application of the load-balancing procedure does not seem to be very succesful, according to the small gain in execution times. However, if larger problems (more grid blocks and larger simulation time) are considered, the load-balancing process may be of more use. Especially for larger simulation times, when grids do not vary too rapidly, the load-balancing procedure is applied a relatively few times, thus leading to less computational and communicational overhead. The more equally balanced work-load then positively affects the parallel performance, more than is the case in table 5.3.5 for smaller problems. Furthermore, it is noticed that if the tolerance is choosen too small, the load-balancing procedure takes too much time to reach the tolerance (or does not reach it at all) and does not lead to any reduction in execution time. In figure 5.3.11, a grid without application of the load-balancing procedure, and the same grid, but with application of load-balancing during the complete simulation, are shown.
Figure 5.3.11: Grid without load-balancing (left) and with load-balancing (right).

We conclude this section by remarking that the model which described communicational work did not explicitly take data transfer costs into account. In many cases, data transfer costs are modelled as $\gamma + m\delta$, in which $\gamma$ represents a start-up time and $\delta$ the transfer costs per element. In section 1.3 we mentioned that $\gamma \approx 0.44$ ms and that $\delta \approx 0.022$ ms, for the NCube/four. For almost each problem, there are additional costs (e.g., filling of buffers to be sent), thus leading to some amount of communicational work. For our problem, the value of $\beta$ (0.017 seconds) models communicational work, which includes computational work concerning internal boundaries, in a satisfactory way.

Conclusions:

* communicational work becomes more and more important if the number of processors increases;
* computational and communicational work can be modelled such that parallel execution times can be predicted accurately;
* dynamical load-balancing pays off if the underlying problem is not too small (large simulation time).
§ 5.4 Implementation and experimental results: Alliant

This section discusses an implementation of the parallel hyperbolic algorithm on a shared memory machine: the Alliant FX/4, a parallel computer with 4 processors (which are equipped with vector units, although relatively slow), connected to one shared memory. The coarse-grained parallelism, which our parallel algorithm is based upon, is reflected by parallelism on the level of subroutines, which is forced by the use of compiler directives (cncall). For more details, we refer to, e.g., [Alma], [Care] and chapter 1.

The computational domain is split into a number of subdomains (which will not explicitly be fixed to the number of processors, as was the case in the NCube implementation). The datastructure is split into parts, the number of them equal to the number of subdomains. For each subdomain, we only need a pointer to its part of the datastructure.

In the NCube implementation, communication between neighbouring subdomains was achieved by exchange of buffers between corresponding processors. In the Alliant implementation, the procedure is somewhat different. Each subdomain fills part of a buffer with data to be used by up-stream neighbouring subdomains. When the buffer is filled completely (synchronization), each subdomain reads the complete buffer and extracts the information it needs. This simultaneously reading is not problematic, since two or more subdomains never need the same information. This strategy means that internal boundaries are processed completely in parallel, in contrast with the treatment of internal boundaries in the NCube implementation.

The actual implementation of the parallel hyperbolic algorithm on the Alliant implicitly contains distribution points and synchronization points. The concurrent calling of subroutines (i.e. concurrent processing of subdomains) at distribution points causes the operating system to do some additional work, which may be regarded to be proportional to the number of subdomains to be distributed. At the end of concurrent calls, synchronization of processors is necessary, before execution of the program continues. If distribution points are denoted by $D$ and synchronization points by $S$, then the parallel hyperbolic algorithm on the Alliant is given in figure 5.4.1. Note that it is not needed to state distribution points and synchronization points explicitly in the program. Distribution points and synchronization points are enforced by the cncall--directive. If local (un)refinement is applied each 3 time steps, then the number of distribution and synchronization points is given by $3 + \frac{2}{3}$, for each time step.
do each time step
D
saturation update for each subdomain
S
if (necessary) then
D
compute indicators for each subdomain
S
D
correct indicators for each subdomain refinement/unrefinement in each subdomain
computation of local minimum time step in each subdomain
S
endif
D
phase flow update for each subdomain
S
D
update internal boundary phase flows for each subdomain
S
determine global minimum time step

Figure 5.4.1: Actual parallel hyperbolic algorithm on Alliant FX/4.

In order to get an estimate for the parallel execution time as a function of the number of processors p and the number of subdomains nsec (nsec≥p), we first consider the grid of load—balancing level (cf. section 4.3; recall that blocks of load—balancing level build the subdomains). So far, for each testproblem, we used a grid of load—balancing level, built of 16 blocks. If we consider a uniform testproblem (no local refinement), then the work—load can be balanced equally over nsec subdomains, if the number of subdomains equals a power of 2 (e.g., if nsec=8, then each subdomain consists of 2 blocks of load—balancing level, which are refined uniformly). First we will assume that the subdomains have equal work—loads. Furthermore, we define α to be the computational work in seconds per finest—level grid block per time step, β the time for one synchronization point, γ the distribution overhead per subdomain, and n the number of time steps. The parallel execution time T(nsec,p) then amounts to: 
\[ T(n\sec,p) = n \left[ \frac{n\sec}{p} \cdot \frac{N}{n\sec} \alpha + \left(3+\frac{2}{3}\right)(k\beta + n\sec \cdot \gamma) \right]. \]  

(5.4.1)

Synchronization is only involved if two or more processors finish their tasks at about the same moment. Otherwise, execution of the program continues right after the running processor (just one!) finished its task, and hence no synchronization is necessary. This explains the occurrence of \( k \) in (5.4.1):

\[
\begin{align*}
  k &= 0, \quad \text{if } \text{mod}(n\sec,p) = 1, \\
  k &= 1, \quad \text{otherwise}.
\end{align*}
\]

(5.4.2)

In the NCube implementation in section 5.3, the computational work \( \alpha \) was computed with respect to the sequential algorithm. In chapter 1, we already mentioned that compiler options significantly affect execution times on the Alliant FX/4. This is reflected in the following way. Subroutines that have to be called concurrently, should be compiled under some special option (–recursive), which leads to slower code. In order to have a correct model (5.4.1), (5.4.2) (which is not valid for just one subdomain on one processor), the value of \( \alpha \) has to be determined from the sequential algorithm, compiled under the same options as the parallel algorithm. Note, of course, that speed-up factors are computed relative to the fastest implementation of the sequential algorithm. For this reason, we first run the sequential algorithm under the special option, necessary for concurrent calls of subroutines. As a testproblem, we use testproblem 1 of the previous section, i.e. 1024 finest–level grid blocks, no local refinement, but correct computation of indicators, leading to an execution time of 100.6 seconds. Then

\[ 150 \cdot 1024 \alpha = 100.6 \implies \alpha = 6.5 \cdot 10^{-4} \text{ s}. \]

(5.4.3)

In (5.4.3), 150 denotes the number of time steps.

In order to determine \( \gamma \), we have chosen a situation in which \( k \) vanishes, i.e. mod(nsec,p) = 1. Such a situation occurs if nsec=4 (power of 2 for load–balancing over the subdomains) and p=3, which takes 54.7 seconds:

\[ 54.7 = 150 \cdot \left[ 2 \cdot \frac{1024}{4} \alpha + \left(3+\frac{2}{3}\right) \cdot 4 \gamma \right] \implies \gamma = 2.2 \cdot 10^{-3} \text{ s}. \]

(5.4.4)

The value of \( \beta \) can be computed if \( k=1 \), e.g., if nsec=p=2. Experimentally, we found 53.1 seconds:
53.1 = 150 \cdot \left[ \frac{1024}{2} \alpha + (3+\frac{2}{3})(\beta+2\gamma) \right] \implies \beta = 1.4 \cdot 10^{-3} \text{ s.} \quad (5.4.5)

In table 5.4.1, we list some estimated and measured times. Note that the work-load is still equally balanced over the subdomains. For the sequential algorithm, 87.4 seconds is the fastest we can get:

<table>
<thead>
<tr>
<th>p</th>
<th>nsec</th>
<th>estimated time in seconds</th>
<th>measured time in seconds</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>—</td>
<td>87.4</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>—</td>
<td>53.1</td>
<td>1.65</td>
<td>0.83</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>55.5</td>
<td>55.8</td>
<td>1.57</td>
<td>0.79</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>60.4</td>
<td>62.4</td>
<td>1.40</td>
<td>0.70</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>—</td>
<td>54.7</td>
<td>1.60</td>
<td>0.53</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>47.8</td>
<td>47.6</td>
<td>1.84</td>
<td>0.61</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>30.5</td>
<td>30.5</td>
<td>2.87</td>
<td>0.72</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>35.4</td>
<td>32.9</td>
<td>2.58</td>
<td>0.65</td>
</tr>
</tbody>
</table>

*Table 5.4.1: Estimated and measured times for testproblem 1.*

If the number of subdomains (nsec) equals the number of processors (p), then (5.4.1) becomes

\[ T_p = n \left[ \frac{N}{p} \alpha + (3+\frac{2}{3})(k\beta + p\gamma) \right]. \quad (5.4.6) \]

The value of \( \alpha \) \((6.5 \cdot 10^{-4} \text{ s})\) was given by (5.4.3), but does not reflect the fastest sequential algorithm. For the fastest sequential algorithm, we have

\[ 150 \cdot 1024 \overline{\alpha} = 87.4 \implies \overline{\alpha} = 5.7 \cdot 10^{-4} \text{ s.} \quad (5.4.7) \]

In formula, the speed-up \( S \) then is

\[ S(p;N) = \frac{T_1}{T_p} = \frac{nN\overline{\alpha}}{n \left[ \frac{N}{p} \overline{\alpha} + (3+\frac{2}{3})(k\beta + p\gamma) \right]}, \quad (5.4.8) \]

or
\[ S(p;N) = \frac{1}{\frac{1}{p} \alpha \cdot N + \frac{1}{N \alpha} \cdot (3 + \frac{2}{3}) (k \beta + p \gamma)} \] 

(5.4.9)

with \( \frac{\alpha}{\alpha} = 1.11 \). This means that an upperbound for the speed-up is given by \( \frac{p}{1.11} = 0.9 \cdot p \), no matter how large the problem size \( N \) is!

It is also possible to have subdomain configurations that do not balance the work-load equally over the processors. In general, such configurations are not the most efficient, but they can serve as a testcase for model (5.4.1), (5.4.2). Note that formula (5.4.1) should be changed such that the term representing the computational work equals the maximum work-load of the processors. In the case of 7 subdomains, e.g., the subdomains consist of 3,3,2,2,2,2 and 2 blocks of load-balancing level, respectively. In the case of 3 processors then, the work-load is distributed over the processors as follows:

\[
\begin{array}{cccc}
3 & 2 & 2 & 2 \\
2 & 3 & 2 & \\
1 & 3 & 2 & \\
\end{array}
\]

\[ \text{Figure 5.4.2: Computational work-load for 3 processors, with 7 subdomains.} \]

Figure 5.4.2 shows that for nsec=7, p=3, the maximum work-load of the processors is found in processor 3, which handles 6 blocks of load-balancing level. Formula (5.4.1) then becomes:

\[ T(nsec,p) = n \left[ \frac{n_{\max}}{1.6} N \alpha + (3 + \frac{2}{3}) (k \beta + nsec \cdot \gamma) \right], \quad (5.4.10) \]

in which \( n_{\max} \) denotes the maximum number of blocks of load-balancing level to be handled by one processor. Also the definition of \( k \) has been changed:

\[
k = 0, \quad \text{if } n_{\max} \text{ is reached in exactly one processor,}
\]

\[
k = 1, \quad \text{otherwise.} \quad (5.4.11)
\]
Table 5.4.2 shows some results, obtained both experimentally and theoretically using (5.4.10), (5.4.11):

<table>
<thead>
<tr>
<th>p</th>
<th>nsec</th>
<th>estimated time in seconds</th>
<th>measured time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>62.2*</td>
<td>63.7</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>57.9</td>
<td>58.4</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>64.6*</td>
<td>66.5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>41.1*</td>
<td>42.4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>44.3*</td>
<td>43.8</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>44.7*</td>
<td>44.0</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>45.9*</td>
<td>46.6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>43.5*</td>
<td>43.2</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>39.2</td>
<td>38.1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>39.7*</td>
<td>38.4</td>
</tr>
</tbody>
</table>

**Table 5.4.2:** Estimated and measured times for testproblem 1, using some non-optimal subdomain configurations.

In table 5.4.2, the asterisk * indicates that no synchronization was necessary (k=0). Tables 5.4.1 and 5.4.2 show that parallel execution times can be predicted quite accurately. Apparently, the problem fits in the cache of the Alliant, and hence cache effects do not affect the parallel performance, for testproblem 1. If we apply the models (5.4.1), (5.4.2) and (5.4.6), (5.4.7) to more processors, then the following estimated times are computed. Note, however, that increase of the number of processors generally also changes other characteristics of the computer (e.g., size of the cache), which may affect values of α, β and γ.

<table>
<thead>
<tr>
<th>p</th>
<th>nsec</th>
<th>estimated time in seconds</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>31.0</td>
<td>2.82</td>
<td>0.56</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>22.9</td>
<td>3.82</td>
<td>0.48</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>26.4</td>
<td>3.31</td>
<td>0.21</td>
</tr>
</tbody>
</table>

**Table 5.4.3:** Some estimated times for testproblem 1, using more processors.
Table 5.4.3 shows that for 8 processors the efficiency of the parallel implementation has reduced to less than 50%. If we compare the speed-up results of the Alliant FX/4 with these obtained on the NCube/four, then we observe that these latter speed-up values are better. This suggests that overhead on the Alliant FX/4 (distribution and synchronization) is relatively more important than overhead on the NCube (communication schedule). This is shown by comparison of the computational and communicational work for the NCube and Alliant. The ratio between computational work on the NCube and on the Alliant is given by the ratio of the parameters \( \alpha \):

\[
\frac{\alpha_{\text{NCube}}}{\alpha_{\text{Alliant}}} = \frac{3.8 \cdot 10^{-3}}{6.5 \cdot 10^{-4}} = 5.85. \quad (5.4.12)
\]

In the case of 4 strip-shaped subdomains, the communicational work per time step for the NCube amounts to \((k=9, k_e=5, \beta=1.7 \cdot 10^{-2})\):

\[
(k + \frac{1}{3}k_e)\beta = 0.181, \quad (5.4.13)
\]

while for the Alliant, communicational work amounts to \((p=4, k=1, \beta=1.4 \cdot 10^{-3}, \\
\gamma=2.2 \cdot 10^{-3})\):

\[
(3+\frac{2}{3})(k\beta + p\gamma) = 0.037. \quad (5.4.14)
\]

The ratio between (5.4.13) and (5.4.14) then equals 4.89, which shows that communicational work in the Alliant implementation is relatively more expensive than in the NCube implementation. In the case of 2 processors (subdomains), this conclusion is even more clear:

\[
\begin{align*}
\text{NCube (k=3, k_e=3)}: & \quad (k + \frac{1}{3}k_e)\beta = 0.068, \\
\text{Alliant (k=1, p=2)}: & \quad (3+\frac{2}{3})(k\beta + p\gamma) = 0.021, \\
\text{ratio (NCube/Alliant)} & = 3.24.
\end{align*}
\quad (5.4.15)
\]

Furthermore, in the case of 4 subdomains (processors) on the NCube, execution times for square-shaped and strip-shaped subdomains differ. On the Alliant, strips or squares does not make much difference: 30.5 seconds versus 30.7 seconds.
If testproblem 1 is considered with 4096 grid blocks of the finest-level (i.e. one level finer than the original testproblem 1), then we expect ((5.4.1) or (5.4.6)):

<table>
<thead>
<tr>
<th>p</th>
<th>nsec</th>
<th>estimated time in seconds</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>350.2</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>105.4</td>
<td>3.32</td>
<td>0.83</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>60.4</td>
<td>5.79</td>
<td>0.72</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>45.1</td>
<td>7.76</td>
<td>0.49</td>
</tr>
</tbody>
</table>

*Table 5.4.4: Estimated times for testproblem 1 (4096 grid blocks).*

We have also executed our second testproblem, which was the one with adaptive local grid refinement, during a simulation time of 2.0 time units (see the previous section). We obtained the following results:

<table>
<thead>
<tr>
<th>p</th>
<th>nsec</th>
<th>measured time in seconds</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>69.9</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>43.9</td>
<td>1.59</td>
<td>0.80</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>32.1</td>
<td>2.18</td>
<td>0.73</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>27.7</td>
<td>2.52</td>
<td>0.63</td>
</tr>
</tbody>
</table>

*Table 5.4.5: Timing results for testproblem 2.*

As was the case in the NCube implementation, speed-up factors are positively affected by updating the grid datastructure, as a consequence of unrefinement of grid blocks. In table 5.4.4, the number of subdomains equals the number of processors. For two reasons, it may be interesting to have more subdomains than processors. We have already notified the first reason: updating the grid datastructure may be more efficient in the case of more subdomains, as was pointed out in detail in section 5.3. The second reason is that more subdomains may lead to a more favourably balanced work-load of the processors. (It should be noted, however, that more subdomains also cause a larger distribution overhead, according to (5.4.1) or (5.4.6)). In table 5.4.6, we show some results for more subdomains than processors, for testproblem 2:
Table 5.4.6: Some more timing results for testproblem 2.

In table 5.4.1 we have observed that perfect load-balancing for 2 processors led to a maximum speed-up of 1.65. Observing table 5.4.6 then, we conclude that the first reason (updating the grid datastructure) mostly affects the speed-up in the case of more subdomains. In the previous section, we already considered the renumbering strategy (figure 5.3.10, the number of subdomains has been taken equal to the number of processors):

\[
N, m \\
\times \times \times \times \times \times \times \times \times \\
O\left(\frac{m^N}{2p}\right)
\]

Figure 5.4.3: Renumbering in one subdomain.

Since the grid datastructure is quite complicated, we assume that the best renumbering algorithm amounts to \(O(N)\) operations. If we assume the work-load to be approximately balanced over \(p\) processors, then the renumbering strategies take \(O\left[\frac{m \cdot N}{2p}\right] = O\left[\frac{m}{2p^2} \cdot N\right]\) and \(O\left[\frac{N}{p}\right]\) parallel operations, respectively. This means that if \(m=2p\), both renumbering procedures require the same number of parallel operations. If \(p\) further increases, then the sequentially more expensive procedure becomes superior over the \(O(N)\)-procedure! If \(m=8\) or \(m=16\) (which are quite common values), then for more than 4 or 8 processors, respectively, it pays off to implement the "expensive" renumbering
procedure, instead of the $O(N)$-procedure (assuming that each processor has to remove $mP$ elements from its datastructure).

We conclude this section with the dynamical load-balancing process, as described in section 4.5. It is observed that, for testproblem 2, we do not reach any additional speed-up by application of the load-balancing process. This is caused by the fact that the computational speed of the Alliant-processors is too large, relative to the additional (non-parallel) work for the load-balancing. For larger problems (simulation time or problem size larger) and in the next chapter, however, the load-balancing process may be of more value. Table 5.4.7 lists a few results, obtained with dynamical load-balancing ($S$ and $E$ are computed with load-balancing):

<table>
<thead>
<tr>
<th>p</th>
<th>nsec</th>
<th>time without load-balancing</th>
<th>time with load-balancing</th>
<th>tolerance</th>
<th>$S$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>43.9</td>
<td>47.9</td>
<td>1.1</td>
<td>1.46</td>
<td>0.73</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>43.9</td>
<td>40.1</td>
<td>1.3</td>
<td>1.74</td>
<td>0.87</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>27.7</td>
<td>34.1</td>
<td>1.1</td>
<td>2.05</td>
<td>0.51</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>27.7</td>
<td>28.2</td>
<td>1.3</td>
<td>2.48</td>
<td>0.62</td>
</tr>
</tbody>
</table>

Table 5.4.7: Timing results with load-balancing.

Conclusions:

* communicational work (distribution and synchronization) for the Alliant implementation takes relatively more time than communicational work for the NCube implementation;
* load-balancing on shared memory machines can be applied automatically, if the number of subdomains is chosen larger than the number of processors;
* dynamical load-balancing does not seem to pay off for the parallel hyperbolic implementation for shared memory machines;
* the model to predict parallel execution times on the NCube/four seems to be more accurate than the model that predicts parallel execution times on the Alliant FX/4.
6

A PARALLEL ELLIPTIC UPDATE

Abstract

In this chapter a parallel algorithm, including multigrid, for the elliptic part of the reservoir simulation equations is considered. Again based upon the domain decomposition discussed in chapter 4, special attention is paid to internal boundaries between the subdomains. Treatment of (very) coarse grids is done by means of agglomeration processes. Experimental results, focussing on CPU-times and convergence properties, together with analysis and expectations for more processors and larger problems, are presented. For shared memory machines, also aspects of the dynamical load-balancing process are studied.
§ 6.1 Introduction and sequential algorithm

In the previous section, we discussed parallel implementations of the hyperbolic part of the reservoir simulation equations on both local memory and shared memory machines. These implementations are based upon the domain decomposition method, as described in chapter 4. This chapter deals with the parallel implementation of the elliptic part of the reservoir simulator, using a multigrid solver, as studied in chapter 3.

The elliptic part of the reservoir simulation problem is given by

\[ c \cdot p + v_u = q, \quad (6.1.1a) \]
\[ v_p + k^{-1} u = v, \quad (6.1.1b) \]

in which the compressibility \( c \) is non-negative, \( q \) denotes a source term depending on the hyperbolic part of the simulator, \( k \) denotes the total permeability and \( v \) represents external forces, e.g., gravity or viscous forces. Equation (6.1.1a) represents mass conservation, while equation (6.1.1b) is known as Darcy's law. The system (6.1.1) has to be solved for the unknowns \( p \) (pressure) and \( u \) (total flow). Boundary conditions are such that there is no normal-flow:

\[ u \cdot \nu = 0. \quad (6.1.2) \]

As was pointed out in section 3.2, mixed finite elements are used to discretize the system (6.1.1):

\[ c_i p_i + \Sigma_j (s_i h_j) u_j = q_i, \quad (6.1.3a) \]
\[ h_j \cdot \Sigma_i (s_i p_i) + \Sigma_j w_{jj'} u_{jj'} = v_j. \quad (6.1.3b) \]

In (6.1.3), index \( i \) represents grid blocks while index \( j \) represents interfaces. The discretization of the matrix vector multiplication \( k^{-1} u \) (define \( W = k^{-1} \)) is given by the second term of (6.1.3b). For more details about the discretization, we refer to section 3.2 and the references therein.

In section 3.4, we have discussed a multigrid solver for the system (6.1.3). Definitions of the smoothing process, coarse-grid matrices, prolongations and restrictions were given in section 3.4. Since we are interested in the parallel properties of the elliptic part of the simulator, our testproblems are quite simple, with constant
(isotropic) permeabilities. In this sense, isotropic means that the permeability matrix $k$ is diagonal, with identical elements. The computation of the elements $w_{jj}$, for the finest grid and for all coarser grids is not too complex, then. However, only minor changes, not having any consequences for parallel features or communicational work, will be necessary to satisfy the complete specifications of section 3.4. The only simplification consists of isotropic, constant permeabilities. For more details about the multigrid process, we refer to section 3.4 and [ScJa].

To conclude this section, the sequential algorithm for the elliptic part of the simulator, including the multigrid solver, can be outlined as follows (cf. section 3.4 for the matrix vector notation and the computation of residuals: superscript $f$ denotes the finest level):

```
compute equations (6.1.3) at finest level maxl
for $l = \text{maxl}-1,1,-1$
    compute coefficients of (6.1.3) at level $l$
initial guess $p^f, u^f$ at finest level maxl
do until convergence
    compute residuals $q^f - C^f \cdot p^f$ and $\gamma^f - W^f \cdot u^f$ at finest level maxl
    for $l = \text{maxl}-1,1,-1$
        compute restrictions $p^l, u^l$
        compute restrictions of residuals
    solve exact at coarsest level 1
    prolongate $p^1, u^1$ to level 2
    for $l = 2, \text{maxl}$
        smooth at level $l$
        if ($l \neq \text{maxl}$) then
            prolongate $p^l, u^l$ to level $l+1$
        endif
    compute block and interface residuals at finest level maxl
```

**Figure 6.1.1:** Sequential algorithm (sawtooth multigrid) for the elliptic part of the reservoir simulator.

In sections 6.3 and 6.4, parallel implementations of the multigrid algorithm on both local memory and shared memory machines are considered, including analysis and experimental results.
§ 6.2 Parallel elliptic algorithm

As was done in section 5.2 for the sequential hyperbolic algorithm, we will study the building parts of the sequential elliptic algorithm for their parallel properties in this section. In section 5.2, we observed that operations which are restricted to just one grid block (e.g., saturation update or determination of a new local time step) can be run in parallel, without any communication needed between the subdomains. In this section, we will observe, again, that communication is necessary for interface computations, involving neighbouring blocks, but not for computational work restricted to just one grid block.

The discrete compressibility $c_i$ and source term $q_i$ for a finest-level block $i$ are computed as integrals over grid block $i$, and hence do not require any communication. The same is true for the discretization of the gradient in (6.1.1b), which contains mesh widths. Furthermore, the computation of block residuals and its restrictions to coarser levels are also restricted to grid blocks, and do not cross internal boundaries. Finally, the restrictions and prolongations of pressures (block quantities) and flows (interface quantities) can be computed independently in each subdomain.

In the above lines, coarser levels are considered until level minl, the load-balancing level (cf. section 4.3). For levels below the load-balancing level, some agglomeration procedure is applied to handle multigrid operations at these levels. A mapping between subdomain grids and agglomerated grids is essential to compute coarser-level quantities in a correct way (cf. section 4.4). In sections 6.3 and 6.4, agglomeration procedures are considered in more detail.

So far, no communication between subdomains is required for correct parallel execution of the elliptic algorithm. Before we study the parts of the elliptic update that do require communication, we first consider the computation of the elements $w_{jj'}$, for all levels, and the right-hand side $\mathbf{v}^{f}$.

As was pointed out in section 3.4, inverse permeabilities at the finest level are given by:

$$w_{jj'} = \int_{\Omega} (W \cdot \Theta_j) \cdot \Theta_{j'} \, d\Omega,$$

(6.2.1)

in which $\Theta_j$ and $\Theta_{j'}$ are indicator functions, belonging to interfaces $j$ and $j'$, respectively. If the matrix $W$ is diagonal (isotropy), then for each interface $j$ there are only 3 non-zero quantities $w_{jj'}$ (note that $\Theta_j$ and $\Theta_{j'}$ are parallel with either the $x$- or $y$-axis):
Figure 6.2.1: Only $w_{j,j1}$, $w_{j}j$ and $w_{j,k1}$ are non-zero, in the isotropic case.

If interface $j$ is part of an internal boundary, then we have the following situation:

![Diagram of subdomains](image)

Figure 6.2.2: Computation of $w_{jj}$ at internal boundary.

Interfaces $j$ and $k$ are equivalent. In the left subdomain, $w_{j,j1}$ is computed in the usual way, as is $w_{k,k1}$ in the right subdomain. If computation of $w_{jj}$ is restricted to only the left subdomain, and computation of $w_{kk}$ to only the right subdomain, then incorrect (i.e. incomplete) values of $w_{jj}$ and $w_{kk}$ are obtained. However, the sum of $w_{jj}$ and $w_{kk}$ is the correct value for both $w_{jj}$ and $w_{kk}$. In the actual implementations, we do not explicitly compute the correct $w_{jj}$ or $w_{kk}$. During computation of internal boundary residuals and internal boundary smoothing, we keep in mind that $w_{jj}$ (or $w_{kk}$) is incomplete. If an anisotropic problem is considered, then the cross-terms $w_{j,j2}$, $w_{j,j3}$, $w_{k,k2}$ and $w_{k,k3}$ in figure 6.2.1 do not vanish. In figure 6.2.2, computation of these cross-terms does not lead to any additional communication across the internal boundary, which implies that the communicational work in the anisotropic case is equivalent to the communicational work in the isotropic case. The right-hand side $\mathbf{v}^f$ is computed in a similar way as the elements $w_{jj}$. Local refinement along internal boundaries, although more complicated, is handled in the same way as uniform refinement.
In section 3.4, a description of the smoothing process was given. The smoother consists of a sequence of so-called quadruple updates. A quadruple is a combination of 4 grid blocks of the same level, having one vertex in common. For convergence of the multigrid process, it is necessary that each quadruple of the sequence is handled at least once [ScJa]. This means that also quadruples have to be handled, which are intersected by internal boundaries. In this case, at least 2 neighbouring subdomains are involved in such quadruple updates:

![Figure 6.2.3](image1)

**Figure 6.2.3:** Quadruple update across an internal boundary; interfaces j1 and k1, and j2 and k2 are equivalent, respectively.

In figure 6.2.3, the quadruple update for blocks i1, i2, i3 and i4 should be done in either the left subdomain or the right subdomain. If we choose the left subdomain, then information concerning grid blocks i3 and i4, which belong to the right subdomain, should be available in the left subdomain. This can be achieved by communication between the 2 subdomains (which leads to physical data transfer in case of local memory machines):

![Figure 6.2.4](image2)

**Figure 6.2.4:** Information that has to be available in the left subdomain.
Again, as in section 5.2, only one pointer is necessary to match equivalent internal boundary interfaces. Using this pointer, it becomes easy to identify correct quadruples, i.e. if an equivalent interface does not exist (local refinement), then no quadruple update is possible. For each internal boundary interface, boundary flows of the quadruple \((u)\), the block residual or block right-hand side of the neighbouring block \((b)\) and the interface residuals of inner interfaces \((r)\) have to be communicated from the right subdomain to the left subdomain. After solution of the quadruple problem, again communication is necessary to store the new pressures and flows in the right subdomain.

From figure 6.2.4 we observe that the interface residuals of interfaces \(k_1\) and \(k_2\) are not used in the left subdomain (which is assumed to be the basis for quadruple updates). This implies that the interface residuals of \(j_1\) and \(j_2\) have to be correct, which means that communication between the subdomains must have been applied earlier. In the next figures, we will indicate how to correct interface residuals or right-hand sides.

![Figure 6.2.5: Interfaces j and k are equivalent, of the finest level.](image)

The right-hand side \(v_f^f\) of interface \(j\) is given by:

\[
\text{rhs}(j) = v(j) + v(k),
\]

while the residual \(\text{res} = v_f^f - W_f \cdot u_f^f\) is given by:

\[
\text{res}(j) = \text{rhs}(j) - (w_{jj} + w_{kk})u_j - w_{j,jl} \cdot u_{jl} - w_{k,k1} \cdot u_{k1}.
\]

Formulas (6.2.2) and (6.2.3) indicate that quantities depending on \(k\) and \(k_1\) (which are interfaces of the right subdomain) have to be available in the left subdomain. Again, a pointer is needed to match interfaces \(j\) and \(k\). A similar property holds for interfaces of coarser levels, for which \(P_u^*(v_f^f - W_f \cdot u_f^f)\) have to be computed:
Now we have

\[ \text{res}(j) = \text{res}(j1) + \text{res}(j2) + \frac{1}{2} \left( \text{res}(j3) + \text{res}(j4) + \text{res}(k3) + \text{res}(k4) \right). \]  

(6.2.4)

Note that the interface residuals of j1 and j2 should have been corrected before execution of (6.2.4). A consequence of this (sequential) mechanism (which, of course, also holds in the internal of the subdomains) is that interface residuals of each level have to be computed before smoothing at the coarsest level starts. It is not possible to correct interface residuals just before the internal boundary quadruple updates, without changing the smoothing sequence (which is a special case of a V-cycle).

The procedure to compute correct interface residuals or right-hand sides is then the following. First, compute in parallel right-hand sides and residuals for interfaces that do not coincide with an internal boundary. This does not require any communication. Then, compute right-hand sides and residuals for internal boundary interfaces. This step requires communication between neighboring subdomains. If we assume the lowest-numbered subdomain to be associated with the internal boundary quadruple updates, then only one message from the neighboring subdomain is required to allow the lowest-numbered subdomain to correct residuals for levels maxl to minl. First, the finest-level right-hand sides are corrected, then the one-level coarser residuals, etc. Note that no return-message is necessary.

So far, only uniform refined grids were considered, regarding the figures 6.2.5 and 6.2.6. In case of locally refined grids, formulas (6.2.2) – (6.2.4) become more complicated. As an example we consider figure 6.2.7:
Figure 6.2.7: Local refinement and computation of internal boundary residuals.

Interface $j$ does not have an equivalent interface. To correct the right-hand side of interface $j$, the flow and the right-hand side of interface $k$ are required. Interface $k$ is equivalent to the father-interface of $j$, which can be determined easily, using the tree structured data structure of the internal boundary interfaces. If the message, coming from the right subdomain, is also tree structured, then all information for interface $j$ is contained in this message. It is then possible to correct the right-hand side (or residual, for coarser blocks) in a proper way, using both tree structures, without any additional communication between the subdomains.

Before concluding this section with the parallel elliptic algorithm, we remark that convergence of the multigrid process is determined by the norm of block and interface residuals, after each iteration. These computations can be done completely in parallel. After each quadruple update, the block and interface residuals within this specific quadruple are exactly zero. Since quadruple updates across internal boundaries are the last quadruple updates of the sequence, the residuals of interfaces, coinciding with an internal boundary, become zero after execution of all internal boundary quadruple updates and do not have to be computed. For other (internal) interfaces, computation of residuals can be done without communication between the subdomains.

Finally, the concepts discussed here lead to the following parallel elliptic update:
for each subdomain
  compute equations (6.1.3) at finest level maxl
  for $l = maxl-1,minl,-1$
    compute coefficients of (6.1.3) at level $l$,
    without crossing internal boundaries
  initial guess $p^f, u^f$ at finest level maxl
do until convergence
  for each subdomain
    compute residuals at finest level,
    without crossing internal boundaries
    for $l = maxl-1,minl,-1$
      compute restrictions $p^l, u^l$
      compute restrictions of residuals,
      without crossing internal boundaries
  communication between subdomains to correct internal boundary residuals
  agglomeration: compute restrictions up to the coarsest level 1
  apply smoothing from level 1 to level minl
for each subdomain
  prolongate $p_{minl}, u_{minl}$ to level $minl+1$
for $l = minl+1,maxl$
  for each subdomain
    smooth at level $l$ without crossing internal boundaries
  communicate between subdomains:
    set internal boundary quadruples at level $l$
    solve quadruple problems
    return new $p^l, u^l$ to neighbouring subdomains
if ($l$ .ne. maxl) then
  for each subdomain
    prolongate $p^l, u^l$ to level $l+1$
endif
for each subdomain
  compute maximum norms of residuals
  determine global residual

Figure 6.2.8: Parallel elliptic update.
§ 6.3 Implementation and experimental results: NCube

In this section we will consider an implementation of the parallel algorithm for the elliptic part of the reservoir simulation equations on an NCube/four. In the previous section, we observed that communication between subdomains is only necessary in 3 parts of the algorithm (apart from agglomeration): setting of correct internal boundary interface equations, smoothing across internal boundaries and determination of global residuals. As was done in the parallel algorithm for the hyperbolic part (section 5.3), it is assumed that the number of subdomains equals the number of processors. Furthermore, in section 5.3 we use buffers to handle physical data transfer between the processors: since data transfer is quite expensive, in comparison with floating point operations, buffers are filled with information concerning the complete internal boundary between two neighbouring subdomains and are sent as one message to the processor which handles the neighbouring subdomain. This concept will also be used in the elliptic part.

As a result of section 6.2, we take the lowest-numbered subdomain for the quadruple updates across an internal boundary. This choice implies that internal boundary interfaces in the lowest-numbered subdomain should contain correct right-hand sides or residuals. Therefore, communication between neighbouring subdomains is in the direction of the lowest-numbered subdomain, while no return-message is necessary. For levels maxl to minl, we need only one message between two neighbouring subdomains to handle all levels, provided that full advantage is taken from the internal boundary datastructure (tree structure in receiving subdomain and tree structured buffer to be sent).

In section 5.3, we argued that simultaneous data transfer between two neighbouring processors could cause a deadlock situation. Therefore, we had to take special care to implement communication schedules that avoid such situations. However, when correcting internal boundary residuals, simultaneous data transfer between two neighbouring processors never occurs and communication schedules become straightforward: each subdomain first writes messages to its lower-numbered neighbours, and then receives messages from its other neighbours. Note that for strip-shaped subdomains, each subdomain has at most one lower- and one upper-numbered neighbour.

The parallel elliptic update described in figure 6.2.8 shows that smoothing across internal boundaries is performed at each level \( l \), \( l = \text{min} + 1, \text{max}, 1 \). For two subdomains, at level \( l \), this internal boundary smoothing is performed as follows. First, subdomain 2 fills a buffer with internal boundary information concerning level \( l \), and sends it to subdomain 1. Subdomain 1 receives this buffer, performs the quadruple updates and fills
another buffer with results, sends it back to subdomain 2, and continues. At the next level, the same procedure is repeated, etcetera. The quadruple updates in subdomain 1 are performed as depicted in figure 6.3.1:

```
  i11  i12  i13
```

**Figure 6.3.1:** Internal boundary quadruple updates.

First, the quadruple, formed by the blocks i11, i12, i21 and i22 is handled. This leads, among others, to new flows through interfaces marked with x and o. Then the next quadruple, consisting of i12, i13, i22 and i23 is considered. For this quadruple update, the flows through interfaces marked with x are boundary values. This implies that the flow through the interface between blocks i21 and i22 should be adapted, in order to have a correct quadruple sequence. This has to be done in the buffer that comes from subdomain 2. Then, the next quadruple is considered, etcetera, until the internal boundary has been processed completely.

Before subdomain 1 can actually perform the quadruple update, it has to read the buffer filled in subdomain 2 in a proper way. In subdomain 2 the buffer is filled in the way, shown in figure 6.3.2:

```
  c
    d,f  h  e,g
    a,b
```

**Figure 6.3.2:** Filling of buffer in subdomain 2.
In figure 6.3.2, for each internal boundary interface, we have:

- \(a\) : the number of the neighbouring subdomain,
- \(b\) : pointer to the equivalent interface in \(a\),
- \(c,d,e\) : interface flows,
- \(f,g\) : interface residuals,
- \(h\) : block residual.

For other orientations (e.g., subdomain 1 left), the sequence \(a,b,\ldots,h\) is uniquely determined by the orientation. The buffer containing results, which is sent from subdomain 2 to subdomain 1, is filled in a similar way.

As was stated before, multigrid methods are of optimal complexity order for certain partial differential equations. The multigrid algorithm, as described in section 6.1, contains two large computational parts: the computation of restrictions and residuals for coarser blocks and interfaces, and the smoothing process, based on quadruple updates. If \(N_c\) denotes the number of coarse blocks over all levels, \(N_q\) denotes the number of quadruple updates over all levels, \(\alpha\) is the time in seconds per coarse block per iteration and \(\beta\) is the time in seconds per quadruple update per iteration, then

\[
T_s = n \cdot \left[ N_c \alpha + N_q \beta \right]
\]

(6.3.1)

denotes the execution time of the sequential algorithm, over \(n\) iterations.

The first testproblem we have chosen deals over a uniform grid on \(\Omega = (0,1) \times (0,1)\) (note that during the elliptic update the grid is fixed). Furthermore,

\[
c \equiv 1, \quad q = \left[ \frac{2}{\pi} + \pi \right] \cdot \left[ \cos(\pi x) + \cos(\pi y) \right], \quad v \equiv 0.
\]

(6.3.2)

The problem is isotropic

\[
W = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}
\]

(6.3.3)

and has the exact solution

\[
p = \frac{2}{\pi} \left[ \cos(\pi x) + \cos(\pi y) \right], \quad u = \left[ \sin(\pi x), \sin(\pi y) \right]^T.
\]

(6.3.4)
For timing experiments on the NCube/four, we used \( n = 5 \), after which the residuals have been reduced to \( O\left(10^{-7}\right) \), in single precision arithmetic (\( \approx 7 \) decimal places). An initial guess is given by \( p = 0, u = 0 \). As always, the coarsest multigrid level (level 1) consists of a \( 2^4 \)-grid. The number of quadruple updates for a grid of \( nx \) by \( ny \) blocks is given by \( (nx-1)*(ny-1) \). If the number of levels equals \( \text{maxl} \), then the number of quadruples and the number of coarse blocks, over all levels \( i \), equal:

\[
N_q = \sum_{l=1}^{\text{maxl}} (2^l - 1) \cdot (2^l - 1)
\]

(6.3.5)

and

\[
N_c = \sum_{l=1}^{\text{maxl}-1} 2^l \cdot 2^l = \sum_{l=1}^{\text{maxl}-1} 4^l
\]

(6.3.6)

respectively.

On one processor, we observed the following execution times:

<table>
<thead>
<tr>
<th>( \text{maxl} )</th>
<th>finest grid</th>
<th>( N_c )</th>
<th>( N_q )</th>
<th>( T_s ) in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4*4</td>
<td>4</td>
<td>10</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>8*8</td>
<td>20</td>
<td>59</td>
<td>4.04</td>
</tr>
<tr>
<td>4</td>
<td>16*16</td>
<td>84</td>
<td>284</td>
<td>18.73</td>
</tr>
</tbody>
</table>

*Table 6.3.1: Timing results for testproblem 1 on one processor.*

From the results for \( \text{maxl} = 3 \) and \( \text{maxl} = 4 \), we obtain for \( \alpha \) and \( \beta \):

\[
\alpha = 1.15 \cdot 10^{-2} \text{ s} \quad \text{and} \quad \beta = 9.8 \cdot 10^{-3} \text{ s.}
\]

(6.3.7)

Due to insufficient memory size, it was not possible to run the case \( \text{maxl} = 5 \) on one processor of the NCube/four. Therefore, we estimate

\[
T_s(\text{maxl} = 5) = 5 \cdot (340\alpha + 1245\beta) = 80.56 \text{ s.}
\]

(6.3.8)

In the case of two subdomains (processors), an explicit agglomeration procedure is
not required. The coarsest level (2*2-grid) can be handled in a similar way as quadruple updates across internal boundaries are treated: subdomain 1 performs all quadruple updates across internal boundaries. These latter quadruple updates form the only sequential part of the parallel algorithm. Quadruples in the internal of the subdomains and coarse block operations are dealt with in parallel. The number of quadruples across the internal boundary \(N_{qb}\) equals:

\[
N_{qb} = \sum_{i=1}^{\max 1} 2^i - 1.
\]

(6.3.9)

For the number of parallel internal quadruples \(N_{qi}\), we have:

\[
N_{qi} = \frac{1}{2} \left[ N_q - N_{qb} \right].
\]

(6.3.10)

while the number of parallel coarse blocks \(N_{ci}\) equals

\[
N_{ci} = \frac{1}{2} \cdot N_c.
\]

(6.3.11)

The parallel computational work (i.e. the computational work for subdomain 1) then is

\[
T_c = n \cdot \left[ N_{ci} \alpha + (N_{qi} + N_{qb})\beta \right].
\]

(6.3.12)

Apart from this computational work, involved in the multigrid process, there is some communication overhead, which comes from the additional work to correct internal boundary interface residuals, the filling of buffers and physical data transfer. In table 6.3.2, parallel execution times, together with estimated ((6.3.12)) and measured computational times, are given \((n=5)\). Due to insufficient memory size, again, the case \(\max l=5\) could not be run on two processors. Speed-up values are relative to the execution times of table 6.3.1 and result (6.3.8). Table 6.3.2 shows that the estimated computational times agree with the measured times. If model (6.3.12) is correct, then overhead (communication: 0.20 for \(\max l=2\), 0.27 for \(\max l=3\) and 0.39 for \(\max l=4\)) seems to depend on the number of multigrid levels (which may be expected in the case of two processors, since at each level there is communication between the two subdomains).
Table 6.3.2: Timing results for testproblem 1 on two processors.

A consequence of parallel execution of the multigrid process is that the sequence of quadruple updates changes. Subdomains 1 and 2 simultaneously handle their internal quadruple updates, whereafter the internal boundary is treated. This alternative quadruple sequence may lead to convergence rates that differ from the ones obtained for the sequential algorithm. For our testproblem, convergence behaviour in the parallel case can be compared with the sequential case, i.e. $O(10^{-7})$ after 5 iterations, single precision. In section 6.4, for a shared memory machine, we will discuss convergence aspects in more detail.

So far, we have given experimental results for two subdomains (processors), in which case no special treatment of the coarsest level was necessary. In section 4.3, it was pointed out that it is in general not possible to handle in parallel the coarsest grid ($2*2$), and probably also the next grid ($4*4$), on more than two processors. This led (in 4.3) to the introduction of the so-called load-balancing level (minl), which, in many cases, is one level finer than the coarsest level. In section 4.4, we discussed two ways, known as agglomeration processes, for handling coarse grids. In the sequel we will study the actual implementation of both agglomeration processes, together with some experimental results.

Stepwise agglomeration

In the case of 4 subdomains, we set the load-balancing level parameter (minl) to 2 (in general, minl = $\log(n_{sec}) = \log(p)$), and we consider strip-shaped subdomains, each consisting of 4 blocks of level 2. From figure 6.3.3, we observe that at level 2, there are no internal quadruples. This means that, at level 2, all quadruple updates are performed across internal boundaries. However, these quadruples overlap, which reduces the smoothing process (block Gauss-Seidel-like) to some block Jacobi-like smoother, if
internal boundaries are handled in parallel. Furthermore, it is not obvious how to handle the coarsest grid.

![Diagram showing strip-shaped subdomains, level 2.](image)

**Figure 6.3.3:** Strip-shaped subdomains, level 2.

The stepwise agglomeration process handles these problems by agglomerating subdomains 3 and 4, and subdomains 2 and 1:

![Diagram showing agglomerated subdomains, level 2.](image)

**Figure 6.3.4:** Agglomerated subdomains, level 2.

Restrictions to the coarsest level are performed in parallel, whereafter again agglomeration is applied (figure 6.3.5)). Smoothing at the coarsest level is done by the processor, corresponding to subdomain 2, whereafter de-agglomeration to subdomain 3
and prolongation to level 2 is applied. Subdomains 2 and 3 handle internal quadruples of level 2, while internal boundary quadruples of level 2 are considered in subdomain 2. Again de-agglomeration to subdomains 1 and 4 is carried out, prolongation (completely in parallel now) to level 3, and smoothing at level 3 starts, completely in parallel.

![Diagram](image)

**Figure 6.3.5:** Agglomerated subdomains, level 1.

Figures 6.3.4 and 6.3.5 show that, at certain multigrid levels, subdomains (processors) go to sleep and that internal boundaries between subdomains seem to vanish. In order to explain these concepts, subdomains (processors) 3 and 4 may serve as an example. Subdomain 3 waits for a message of subdomain 4, which contains information about residuals, flows and pressures. After having sent this message, subdomain 4 goes to sleep. Subdomain 3 receives the message, and adjusts its datastructure (i.e. ignores the internal boundary with subdomain 4 and switches on an additional part of its datastructure, which describes blocks that are equivalent to blocks of level 2 of subdomain 4). Subdomain 3 is expanded now, and continues with restrictions to level 1. After having sent a similar message to subdomain 2, subdomain (processor) 3 goes to sleep. It is awakened by receiving a message from subdomain 2, with updated pressures and flows of level 1. Subdomain 3 prolongates its quantities to level 2, performs its internal quadruple updates, sends an internal boundary message to subdomain 2, and waits for the results. After having received the results from subdomain 2, it awakens subdomain 4, adapts its datastructure (i.e. establishes the internal boundary with subdomain 4 and switches off the blocks of level 2 that were switched on) and continues with prolongation to level 3. Subdomain 4 receives the message from subdomain 3, wakes up and continues with prolongation to level 3. It is obvious that the
stepwise agglomeration process for 8 subdomains (processors) starts at level 3. Note also that the load—balancing level is one level finer (minl = 3).

From level minl+1 up to level maxl, there are p−1 internal boundaries between p strip—shaped subdomains. The number of internal boundaries for level minl is $\frac{P}{2} - 1$, and reduces to no internal boundaries for the coarsest level. For an internal boundary at level $l$, the number of quadruples is given by:

$$N_{qb}(l) = 2^l - 1. \quad (6.3.13)$$

The total number of parallel internal quadruples, $N_{qi}$, over all levels is:

$$N_{qi} = \sum_{l=1}^{\text{minl}} (2^l - 1) + \sum_{l=\text{minl}+1}^{\text{maxl}} \left(\frac{2^l}{2} - 1\right)(2^l - 1), \quad (6.3.14)$$

which is identical to

$$N_{qi} = \sum_{l=1}^{\text{minl}} (2^l - 1) + \sum_{l=\text{minl}+1}^{\text{maxl}} (2^l - \text{minl} - 1)(2^l - 1) \approx \frac{1}{3} \cdot 2^{2\cdot \text{maxl} + 2 - \text{minl}}, \quad (6.3.15)$$

if maxl−minl is large. The number of parallel coarse blocks, $N_{ci}$, is given by

$$N_{ci} = \sum_{l=1}^{\text{minl}-1} 2^l + \frac{1}{p} \sum_{l=\text{minl}}^{\text{maxl}-1} 2^l \cdot 2^l \approx \frac{1}{4} N_{qi}, \quad (6.3.16)$$

if maxl−minl is large. In (6.3.15) and (6.3.16), the first summation reflects the stepwise agglomeration process.

From figure 6.3.5 we have observed that there is one subdomain (processor) that performs the smoothing process (exact solution) on the coarsest grid. For this subdomain, the number of internal boundary quadruples is given by:

$$N_{qb} = \sum_{l=2}^{\text{maxl}} N_{qb}(l) \approx 2^\text{maxl} + 1 \approx \sqrt{3pN_{qi}}, \quad (6.3.17)$$

Then, the time spent for computational work in this subdomain amounts to
\[ T_c = n \cdot \left[ N_{ci} \alpha + (N_{qi} + N_{qb})\beta \right], \]

with \( N_{ci}, \ N_{qi}, \) and \( N_{qb} \) given by (6.3.16), (6.3.15) and (6.3.17), respectively. Note the equivalence between (6.3.18) and (6.3.12).

The stepwise agglomeration process is chosen such that, at coarser levels, quadruple updates across internal boundaries can be performed in parallel: quadruples simply do not overlap. For finer levels (> minl), this was already the case. Therefore, at each level, internal boundaries can be handled completely in parallel. Hence, the parallel computational work for the complete process is given by (6.3.18). With (6.3.18), we can compute an upperbound for the speed-up, relative to the sequential case (cf. table 6.3.1):

<table>
<thead>
<tr>
<th>p</th>
<th>minl</th>
<th>maxl</th>
<th>( N_{ci} )</th>
<th>( N_{qi} )</th>
<th>( N_{qb} )</th>
<th>( T_c ) in secs.</th>
<th>( S_{\text{max}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>22</td>
<td>56</td>
<td>25</td>
<td>5.23</td>
<td>3.58</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>86</td>
<td>273</td>
<td>56</td>
<td>21.07</td>
<td>3.82</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>4</td>
<td>14</td>
<td>26</td>
<td>25</td>
<td>3.30</td>
<td>5.68</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>5</td>
<td>46</td>
<td>119</td>
<td>56</td>
<td>11.22</td>
<td>7.18</td>
</tr>
</tbody>
</table>

*Table 6.3.3: Maximum speed-up values \( S_{\text{max}} \), stepwise agglomeration.*

In table 6.3.4, some experimental results are shown, with application of the stepwise agglomeration process. The discrepancy between the actual parallel execution time \( T_p \) and the parallel computational time \( T_c \) is given as \( T_{ov} \):

<table>
<thead>
<tr>
<th>p</th>
<th>minl</th>
<th>maxl</th>
<th>( T_c ) in secs.</th>
<th>( T_p ) in secs.</th>
<th>( T_{ov} ) in secs.</th>
<th>( S )</th>
<th>( E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>5.23</td>
<td>6.02</td>
<td>0.79</td>
<td>3.11</td>
<td>0.78</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>21.07</td>
<td>22.64</td>
<td>1.57</td>
<td>3.56</td>
<td>0.89</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>4</td>
<td>3.30</td>
<td>4.52</td>
<td>1.22</td>
<td>4.14</td>
<td>0.52</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>5</td>
<td>11.22</td>
<td>13.24</td>
<td>2.02</td>
<td>6.08</td>
<td>0.76</td>
</tr>
</tbody>
</table>

*Table 6.3.4: Experimental speed-up values \( S \), stepwise agglomeration.*
\( T_{ov} \) represents data transfer and additional work concerning internal boundaries and agglomeration. For both cases \((p = 4 \text{ and } p = 8)\), \( T_{ov} \) is increased by approximately 0.80 seconds, if \( maxl \) increases from 4 to 5. This means that in both cases the overhead concerning multigrid level 5 approximately equals 0.80 seconds, since no other overhead is added. This indicates that the treatment of internal boundaries for \( p = 8 \) takes full advantage of the 4 extra processors over the case \( p = 4 \).

We conclude the discussion of the stepwise agglomeration process by estimating some speed-up values for more processors. The overhead \( T_{ov} \) is estimated, using results of table 6.3.4. It is assumed that the memory size of each NCube-node is large enough to store its subdomain; in practice, this may be a problem at present.

<table>
<thead>
<tr>
<th>( p )</th>
<th>( \text{minl} )</th>
<th>( \text{maxl} )</th>
<th>( N_{ci} )</th>
<th>( N_{qi} )</th>
<th>( N_{qb} )</th>
<th>( T_c ) in secs.</th>
<th>( T_{ov} ) in secs.</th>
<th>( T_p ) in secs.</th>
<th>( S )</th>
<th>( E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>6</td>
<td>174</td>
<td>560</td>
<td>119</td>
<td>43.28</td>
<td>2.02+1.60</td>
<td>46.90</td>
<td>7.12</td>
<td>0.89</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>5</td>
<td>30</td>
<td>57</td>
<td>56</td>
<td>7.26</td>
<td>1.22+0.80</td>
<td>9.28</td>
<td>8.68</td>
<td>0.54</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>6</td>
<td>94</td>
<td>246</td>
<td>119</td>
<td>23.29</td>
<td>2.02+1.60</td>
<td>26.91</td>
<td>12.41</td>
<td>0.78</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>6</td>
<td>62</td>
<td>120</td>
<td>119</td>
<td>15.28</td>
<td>2.02+1.60</td>
<td>18.90</td>
<td>17.67</td>
<td>0.55</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>7</td>
<td>190</td>
<td>501</td>
<td>246</td>
<td>47.53</td>
<td>3.62+3.20</td>
<td>54.35</td>
<td>25.02</td>
<td>0.78</td>
</tr>
</tbody>
</table>

*Table 6.3.5: Estimated speed-up values for more processors, stepwise agglomeration.*

The sequential execution times for \( \text{maxl} = 6 \) and \( \text{maxl} = 7 \) are estimated as

\[
T_s(\text{maxl}=6) = 5 \cdot (1364\alpha + 5214\beta) = 333.9\text{ s},
\]

and

\[
T_s(\text{maxl}=7) = 5 \cdot (5460\alpha + 21343\beta) = 1360\text{ s}.
\] (6.3.19)

In table 6.3.5 we try to estimate the overhead time \( T_{ov} \). The time required for the quadruple updates across internal boundaries \((n \cdot N_{qb} \cdot \beta)\) is always larger than \( T_{ov} \), at least for the NCube. \( T_{ov} \) consists of computational work and physical data transfer, and our estimates for \( T_{ov} \) suggest that physical data transfer costs are less important than computational overhead (i.e. increase in \( T_{ov} \) mainly caused by the length of the internal boundary, and not by the number of processors). For other hypercubes [Mic2], the ratios between computational work and data transfer are of the same order as for an NCube, and therefore we expect approximately the same speed-up values as observed in the NCube case. Furthermore, note that for each smallest problem on \( p \) processors (i.e. \( \text{maxl} = \text{minl}+1 = 2^\log (p) + 1 \), efficiency rates are over 50%.
**Remark 1.** As in the case of two subdomains, convergence rates for 4 and 8 subdomains are approximately the same as in the sequential case.

**Remark 2.** In section 2.5, Amdahl’s law and the modification of Van der Vorst were considered. It is proven in [DDSV] that Amdahl’s law gives upperbounds for speed-up values, provided that the sequential part of the algorithm does not change if more processors are in use. If Amdahl’s law is applied to the results of maxl = 4 (i.e. 18.73, 10.35, 6.02 and 4.52 seconds for p = 1, p = 2, p = 4 and p = 8 respectively), then we find that the parallel fraction \( f \approx 0.895 \) for \( p_0 = 2 \). The predicted upperbound for the speed-up for 4 processors then amounts to 3.04, while we measured 3.11 (table 6.3.4). This is caused by the violation of the condition concerning the sequential part. For two processors, there is one internal boundary between the two subdomains, which is handled sequentially. In the case of 4 processors, there are 3 internal boundaries which are handled in parallel, and hence, there is no longer a specific sequential part.

**Direct agglomeration**

The direct agglomeration process, proposed in section 4.3 for irregular subdomain configurations, agglomerates subdomain grids of load-balancing level directly onto one grid, which is contained in one processor (e.g., host). A mapping \( A \) is needed to map subdomain grids onto the host grid:

![Diagram](Figure6.3.6.png)

*Figure 6.3.6: Direct agglomeration for 4 subdomains.*
As usual, restrictions are computed until the load-balancing level \( \text{minl} \) is reached. Then, internal boundary interface residuals from level \( \text{maxl} \) to level \( \text{minl} \) are computed. The direct agglomeration process then starts to communicate with the host: information from each subdomain, concerning level \( \text{minl} \), is sent to the host, applying mapping \( A \), to store the information in a correct way. Since lower-numbered subdomains contain correct internal boundary interface residuals, the communication process is sequential: in figure 6.3.6, the host first handles information from subdomain 4, whereafter information from subdomain 3 is processed. The correct interface residuals at the internal boundary between the two subdomains override some (wrong) information that was received from subdomain 4. Next, information from subdomain 2 is received, etcetera.

After having received information from all subdomains in decreasing order, the host finishes the restriction process until the coarsest level. The upward sweep of the multigrid process starts, until smoothing at level \( \text{minl} \) is performed. The subdomains (processors) are still sleeping. The host applies the inverse mapping of \( A \) to awaken the subdomains by sending them a message containing updated pressures and flows of level \( \text{minl} \). Each subdomain then continues with prolongation to level \( \text{minl} + 1 \), etcetera.

If direct agglomeration is applied, then we have

\[
N_{\text{qb}} = \sum_{l=\text{minl}+1}^{\text{maxl}} (2^l - 1) \approx 2^{\text{maxl}+1} - 2^{\text{minl}+1},
\]

(6.3.20a)

\[
N_{\text{qi}} = \sum_{l=1}^{\text{minl}} (2^l - 1)(2^l - 1) + \sum_{l=\text{minl}+1}^{\text{maxl}} \left( \frac{2^l}{p} - 1 \right)(2^l - 1) \approx \frac{1}{3} \cdot 2 \cdot 2^{\text{maxl}+2-\text{minl}}
\]

(6.3.20b)

\[
N_{\text{ci}} = \sum_{l=1}^{\text{minl}-1} 2^l \cdot 2^l + \frac{1}{p} \sum_{l=\text{minl}}^{\text{maxl}-1} 2^l \cdot 2^l \approx \frac{1}{3} \cdot 4^{\text{minl}} + \frac{1}{2} \cdot 2^{\text{maxl}-\text{minl}}
\]

(6.3.20c)

which are substituted in the parallel computational time \( T_c \), given by (6.3.18). For \( \text{maxl} - \text{minl} \) large, \( N_{\text{qb}}, N_{\text{qi}} \) and \( N_{\text{ci}} \) are dominated by higher levels, which means that direct agglomeration effects become less important. Hence, \( N_{\text{qb}}, N_{\text{qi}} \) and \( N_{\text{ci}} \) can be compared with the corresponding values of the stepwise agglomeration process, if \( \text{maxl} - \text{minl} \) is large. Table 6.3.6 shows some theoretical and experimental results:
<table>
<thead>
<tr>
<th>p</th>
<th>minl</th>
<th>maxl</th>
<th>Nci</th>
<th>Nqi</th>
<th>Nqb</th>
<th>Tc in secs</th>
<th>Tp in secs</th>
<th>Tov in secs</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>24</td>
<td>62</td>
<td>22</td>
<td>5.50</td>
<td>7.25</td>
<td>1.75</td>
<td>2.58</td>
<td>0.65</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>88</td>
<td>279</td>
<td>53</td>
<td>21.33</td>
<td>23.12</td>
<td>1.79</td>
<td>3.48</td>
<td>0.87</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>4</td>
<td>28</td>
<td>74</td>
<td>15</td>
<td>5.97</td>
<td>15.25</td>
<td>9.28</td>
<td>1.23</td>
<td>0.15</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>5</td>
<td>60</td>
<td>167</td>
<td>46</td>
<td>13.89</td>
<td>22.81</td>
<td>8.92</td>
<td>3.53</td>
<td>0.44</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>6</td>
<td>188</td>
<td>608</td>
<td>109</td>
<td>45.94</td>
<td>54.94</td>
<td>9.00</td>
<td>6.08</td>
<td>0.76</td>
</tr>
</tbody>
</table>

Table 6.3.6: Theoretical and experimental results for testproblem 1, direct agglomeration.

Again, Tov represents data transfer and additional work concerning internal boundaries. It is clear that for p=8, the data transfer and overhead during the direct agglomeration process dominate Tov. In the case of p=8, maxl=5, Tov is of the same order as in the case p=8, maxl=4. This effect is probably caused by a similar phenomenon as was encountered in section 5.3: lack of synchronization, caused by data transfer between host and 8 nodes may lead to overlap in computational and communicational work. The communicational work across internal boundaries for multigrid level 5 may be handled during computational work, then.

Table 6.3.6 shows that speed-up results for p=4 are equivalent to speed-up results for p=4, using stepwise agglomeration. However, in the case of p=8, speed-up results are clearly worse. For larger problems (i.e. maxl–minl larger), the sequential part (direct agglomeration) becomes less important, and speed-up values become better. Application of model (6.3.18), (6.3.20) to, e.g., p=8, maxl=6, minl=3 leads to a computational time Tc of 45.94 seconds. If the overhead Tov is estimated as 9.0 seconds, we obtain a parallel execution time Tp of 54.94 seconds, to be compared with 46.90 in the case of stepwise agglomeration. The treatment of levels minl+1 up to maxl is equivalent for both cases, which implies that speed-up values in the case of direct agglomeration should approach speed-up values in the case of stepwise agglomeration, for large maxl.

Local refinement

So far, only uniform grids and strip-shaped subdomains were considered. A strip-shaped subdomain configuration has the advantage that internal boundary quadruples never involve more than two subdomains. In the case of uniform refinement
and a square-shaped subdomain configuration (e.g., 4 subdomains, as was used in section 5.3), the subdomains have one point in common:

![Diagram of subdomains](image)

*Figure 6.3.7: Square-shaped subdomain configuration.*

Figure 6.3.7 shows that internal boundary smoothing can be done completely in parallel (although the "lower-numbered subdomain" strategy is not the most efficient, then). However, one quadruple (around P) has not been considered yet. To handle this quadruple, additional communication between subdomains is necessary, apart from programming effort. Omitting this individual quadruple update may cause slower convergence rates, as will be shown in section 6.4.

The problem of "corner points", like P, also occurs in irregular subdomain configurations, as encountered when dynamical load-balancing is applied (cf. sections 5.3 and 5.4) in the case of local grid refinement. For the NCube/four, we did not consider other subdomain configurations, for the aforementioned reasons. However, we have considered local grid refinement.

The hyperbolic part of the reservoir simulator adapts the grid by refining and unrefining grid blocks. The elliptic part updates the pressure and the total flow on a fixed grid, arising from the hyperbolic part. As a second testproblem for the elliptic part, we have run testproblem 1 on two locally refined grids (maxl=5 and maxl=6), which may be representative for grids encountered during the complete reservoir simulation. Figure 6.3.8 shows the grid for testproblem 2. The grid for maxl=5 is just the grid obtained from the one in figure 6.3.8 after discarding the finest grid blocks of level 6.
Figure 6.3.8: Locally refined grid for testproblem 2, maxl = 6.

The application of local grid refinement leads to more complicated computations for restrictions and interface residuals. This can be reflected in another value of $\alpha$, the amount of work per (refined) coarse grid block, per iteration. Of course, the work necessary to perform one quadruple update ($\beta$) does not change. On one processor, we measured an execution time of 43.08 seconds for the grid of figure 6.3.8 (maxl=6, n=5, interface residual approximately $10^{-6}$, single precision). Some simple counting leads to 195 coarse blocks and 560 quadruples, over all levels. Then ((6.3.1)):

$$T_s = 5 \cdot (195\overline{\alpha} + 560\beta) = 43.08 \quad \Rightarrow \quad \overline{\alpha} = 1.6 \cdot 10^{-2} \text{ s.} \quad (6.3.21)$$

In order to estimate the parallel computational work, the number of coarse blocks, the number of internal quadruples and the number of internal boundary quadruples, for each level and for each subdomain, should be counted. The number of parallel coarse blocks that determines the computational work then equals (stepwise agglomeration):

$$N_{ci} = \sum_{l=1}^{\min l-1} 2^l + \max_k \left[ \sum_{l=\min l}^{\max l-1} N'_{ci}(l,k) \right], \quad (6.3.22)$$
in which \( k \) runs over all subdomains and \( N_{qi}(i,k) \) denotes the number of coarse blocks of level \( l \) in subdomain \( k \). Note that there is no need for synchronization in coarse-block computations; otherwise, the maximum over the subdomains would be included in the summation. The number of parallel internal quadruples equals (stepwise agglomeration)

\[
N_{qi} = \sum_{l=1}^{\text{min}l} (2^l - 1) + \sum_{l=\text{min}l+1}^{\text{max}l} \left[ \max_k N_{qi}(i,k) \right],
\]

(6.3.23)

with \( N_{qb}(i,k) \) the number of internal quadruples of level \( l \) in subdomain \( k \). The number of parallel internal quadruples is given by

\[
N_{qb} = \sum_{l=2}^{\text{min}l} (2^l - 1) + \sum_{l=\text{min}l+1}^{\text{max}l} \left[ \max_k N_{qb}(i,k) \right],
\]

(6.3.24)

with \( N_{qb}(i,k) \) analogous to \( N_{qi}(i,k) \). Again, the parallel computational time is given by

\[
T_c = n \cdot \left[ N_{ci} \overline{\alpha} + (N_{qi} + N_{qb})\beta \right],
\]

(6.3.25)

with \( \overline{\alpha} = 1.6 \cdot 10^{-2} \) s and \( \beta = 9.8 \cdot 10^{-3} \) s.

In Table 6.3.7, some experimental and theoretical results are shown (convergence rates can be compared with the sequential case):

<table>
<thead>
<tr>
<th>( p )</th>
<th>maxl</th>
<th>( N_{ci} )</th>
<th>( N_{qi} )</th>
<th>( N_{qb} )</th>
<th>( T_c )</th>
<th>( T_p )</th>
<th>( T_{ov} )</th>
<th>( S )</th>
<th>( E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>55</td>
<td>145</td>
<td>16</td>
<td>12.29</td>
<td>13.01</td>
<td>0.72</td>
<td>1.66</td>
<td>0.83</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>39</td>
<td>101</td>
<td>16</td>
<td>8.85</td>
<td>9.61</td>
<td>0.76</td>
<td>2.25</td>
<td>0.56</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>23</td>
<td>51</td>
<td>28</td>
<td>5.71</td>
<td>6.84</td>
<td>1.13</td>
<td>3.16</td>
<td>0.40</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>101</td>
<td>280</td>
<td>17</td>
<td>22.63</td>
<td>24.71</td>
<td>2.08</td>
<td>1.74</td>
<td>0.87</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>85</td>
<td>246</td>
<td>17</td>
<td>19.20</td>
<td>21.01</td>
<td>1.81</td>
<td>2.05</td>
<td>0.51</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>55</td>
<td>146</td>
<td>37</td>
<td>13.37</td>
<td>15.02</td>
<td>1.65</td>
<td>2.87</td>
<td>0.36</td>
</tr>
</tbody>
</table>

*Table 6.3.7: Timing results (in secs.) for test problem 2, stepwise agglomeration.*
The speed-up results for maxl=5 are relative to 21.64 seconds, the execution time of the sequential algorithm for maxl=5. Table 6.3.7 shows that in the case of maxl=6, overhead costs decrease if the number of processors is increased, probably caused by asynchronous computational tasks. Note, furthermore, that the overhead is of the same order as the overhead for testproblem 1 (cf. table 6.3.4), with uniform grids. The reduction of the speed-up is then fully due to unbalanced work-loads over the processors. The dynamical load-balancing procedure, as carried out in the hyperbolic part, balances the work-load more equally over the subdomains (processors). If the "corner problem" is solved efficiently, then the speed-up values may increase considerably. In the next section, for a shared memory machine, we actually considered load-balanced subdomain configurations, using the multigrid process. Since the load-balancing itself is applied (and counted for) in the hyperbolic part, the elliptic part does not suffer from additional work for the load-balancing. For a local memory machine (NCube), we expect load-balancing also to increase speed-up values, although data transfer for corners and correct implementation may be a non-trivial problem.

Conclusions:

* parallel performance for uniform problems of modest size is rather well;
* internal boundary quadruples can be handled in parallel;
* application of stepwise agglomeration leads to better performances than application of direct agglomeration, if the number of processors is greater than 4;
* dynamical load-balancing introduces irregular subdomain configurations, with internal corners, which causes complicated data transfer to smooth around these corners, on local memory machines;
* computational and communicational work can be modelled for both uniform and locally refined grids, in order to predict parallel execution times in an accurate way.
§ 6.4 Implementation and experimental results: Alliant

In this section, we consider an implementation for the parallel elliptic algorithm on a shared memory machine: the Alliant FX/4. In the previous section, we discussed the use of buffers to handle communication between neighbouring subdomains: buffers were filled in a prescribed way to correct internal boundary interface equations, and to smooth across internal boundaries, which led to one-way communication and two-way communication, respectively.

In the Alliant implementation, buffers are used only to correct internal boundary interface equations. A similar procedure as in section 5.4 is used: each subdomain fills, in parallel, part of a buffer with data concerning multigrid levels minl+1 to maxl. When the buffer is completely filled (synchronization), each subdomain reads the complete buffer and takes the information it needs. As was the case in the hyperbolic part (section 5.4), no simultaneous reading of the same information by two or more subdomains occurs, since each lowest-numbered subdomain corrects the residuals of its internal boundary interfaces.

In the NCube implementation, buffers were filled with complete internal boundaries, in order to reduce data transfer costs between neighbouring processors. In the Alliant implementation, we do not fill buffers explicitly for internal boundary smoothing. The main reason to omit the buffer-principle is caused by the dynamical load-balancing procedure, which may lead to irregular subdomain configurations. For such situations, it is for convergence reasons more profitable not to use buffers for internal boundary smoothing. Furthermore, since each processor has access to the complete memory (and hence to the datastructure of all subdomains), it is not necessary to fill buffers and to send them to all other processors. Later in this section, we will discuss load-balancing in more detail.

In section 5.4, it was explicited that pointers were used to separate distinct subdomains. For shared memory machines, this implies that for an interface on an internal boundary, the internal boundary datastructure allows us to determine directly the equivalent interface in another subdomain, by using such pointers. To perform a quadruple update across an internal boundary, we can therefore immediately compute the block and interface numbers involved in this quadruple update. Resulting flows and pressures are stored directly in the correct parts of the datastructure, after each quadruple update. Figure 6.4.1 shows a quadruple update across an internal boundary, for shared memory machines:
In the NCube implementation, both types of agglomeration (stepwise and direct) were considered. If stepwise agglomeration is used, subdomains must have a prescribed regular shape, to allow unrefinement of blocks at very coarse levels. As an alternative, direct agglomeration was implemented, to handle irregular subdomain configurations, arising from load–balanced grids. In this section we will only consider the direct agglomeration process, since we are interested in the consequences of load–balancing for the multigrid method. In section 6.3 it was observed that up to 4 processors, speed–up values obtained with direct agglomeration do not differ much from speed–up values obtained with stepwise agglomeration, if problem sizes (maxl) are not too small.

As was the case in the previous section, the load–balancing level minl is proportional to $\frac{\alpha}{\log(nsec)}$, with nsec the number of subdomains. Note that the number of subdomains is not explicitly fixed to the number of processors. In our experiments, the number of subdomains is an integer multiple of the number of processors, although this is not a strict condition. The direct agglomeration process is performed by one processor; collection of data, concerning blocks of load–balancing level, is done as in the NCube implementation. Of course, no physical data transfer is necessary. After completion of the agglomeration, results of level minl are broadcasted over the subdomains, whereafter computations continue, in parallel.

In the previous section we discussed a model to predict parallel execution times of the multigrid process, based on the costs per coarse block per iteration and per quadruple update per iteration. This model can also be applied to the Alliant implementation. In the sequential case, we have (cf. (6.3.1)):

$$T_s = n \cdot \left[ N_c \alpha + N_q \beta \right],$$

with $N_c$ and $N_q$ the total number of coarse blocks and quadruples, respectively. To determine $\alpha$ and $\beta$, we have run testproblem 1 of section 6.3 (uniform refinement):

![Figure 6.4.1: Quadruple update across an internal boundary; shared memory.](image)
Table 6.4.1: Timing results for testproblem 1 on one processor.

<table>
<thead>
<tr>
<th>$\text{maxl}$</th>
<th>finest grid</th>
<th>$N_c$</th>
<th>$N_q$</th>
<th>$T_s$ in secs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>16*16</td>
<td>84</td>
<td>284</td>
<td>2.89</td>
</tr>
<tr>
<td>5</td>
<td>32*32</td>
<td>340</td>
<td>1245</td>
<td>12.46</td>
</tr>
</tbody>
</table>

From table 6.4.1, we compute ($n=5$)

$$\alpha = 4.9 \cdot 10^{-4} \text{ s} \quad \text{and} \quad \beta = 1.89 \cdot 10^{-3} \text{ s.} \quad (6.4.2)$$

If (6.4.2) is compared with equivalent values of $\alpha$ and $\beta$ in the NCube implementation, then we observe that $\alpha$ is reduced much more than $\beta$ (a factor of 23 and 5, respectively). This phenomenon may be explained partly by the vector structure of the Alliant—processors, which is exploited during coarse—block computations.

In section 5.4, the concept of distribution points and synchronization points was introduced. We observed that time overhead due to distribution points dominates synchronization costs, and therefore we will ignore synchronization points. Again, we use the number of parallel coarse blocks ($N_{ci}$), the number of internal quadruples ($N_{qi}$) and the number of internal boundary quadruples ($N_{qb}$). Then the estimated parallel execution time is given by

$$T_c = n \cdot \left[ N_{ci} \alpha + (N_{qi} + N_{qb}) \beta + nsec \cdot k \gamma \right], \quad (6.4.3)$$

in which nsec denotes the number of subdomains ($\geq p$). The value of $k$, the number of distribution points per iteration, is determined by the parallel algorithm, given in section 6.2. This leads to

$$k = 3 + 2 \cdot (\text{maxl} - \text{minl}). \quad (6.4.4)$$

In table 6.4.2 some estimated and experimental times of the multigrid process are given. The value of $\gamma$ is determined from the case nsec = $p = 2$, maxl = 4, and amounts to:

$$\gamma = 1.2 \cdot 10^{-3} \text{ s.} \quad (6.4.5)$$
We have also run some unusual combinations of nsec and p, to illustrate the Alliant performance. The experimentally observed parallel execution times are given by $T_p$:

<table>
<thead>
<tr>
<th>p</th>
<th>nsec</th>
<th>minl</th>
<th>maxl</th>
<th>$N_{ci}$</th>
<th>$N_{qi}$</th>
<th>$N_{qb}$</th>
<th>k</th>
<th>$T_c$ in secs.</th>
<th>$T_p$ in secs.</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>42</td>
<td>130</td>
<td>25</td>
<td>9</td>
<td>1.67</td>
<td>1.67</td>
<td>1.73</td>
<td>0.87</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>170</td>
<td>595</td>
<td>56</td>
<td>11</td>
<td>6.70</td>
<td>6.93</td>
<td>1.80</td>
<td>0.90</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>172</td>
<td>548</td>
<td>106</td>
<td>9</td>
<td>6.82</td>
<td>6.63</td>
<td>1.88</td>
<td>0.94*</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>180</td>
<td>491</td>
<td>184</td>
<td>7</td>
<td>7.15</td>
<td>6.66</td>
<td>1.87</td>
<td>0.94*</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>172</td>
<td>548</td>
<td>53</td>
<td>9</td>
<td>6.32</td>
<td>6.37</td>
<td>1.96</td>
<td>0.65</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>140</td>
<td>383</td>
<td>138</td>
<td>7</td>
<td>5.60</td>
<td>5.28</td>
<td>2.36</td>
<td>0.79*</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>24</td>
<td>62</td>
<td>22</td>
<td>7</td>
<td>1.02</td>
<td>0.98</td>
<td>2.95</td>
<td>0.74</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>88</td>
<td>279</td>
<td>53</td>
<td>9</td>
<td>3.57</td>
<td>3.70</td>
<td>3.37</td>
<td>0.84</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>36</td>
<td>89</td>
<td>30</td>
<td>5</td>
<td>1.45</td>
<td>1.33</td>
<td>2.17</td>
<td>0.54*</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>100</td>
<td>275</td>
<td>92</td>
<td>7</td>
<td>4.05</td>
<td>3.87</td>
<td>3.22</td>
<td>0.80*</td>
</tr>
</tbody>
</table>

**Table 6.4.2:** Theoretical and experimental results for testproblem 1, direct agglomeration. (For *, see (6.4.6))

After 5 iterations, the maximum norm of the interface residual equals $O(10^{-9})$, double precision, just as in the sequential case. Table 6.4.2 shows some interesting results. First, speed-up values and efficiencies are worthwhile and can be compared with speed-up values obtained in the NCube implementation. Secondly, in some cases there is a discrepancy between theoretical and experimental results. A comparison between $T_c$ and $T_p$ shows that smaller parallel execution times are obtained than expected, when the ratio between the number of processors $p$ and the number of subdomains nsec is less than or equal to $\frac{1}{2}$ (denoted by *):

$$\frac{p}{nsec} \leq \frac{1}{2}$$  \hspace{1cm} (6.4.6)

In nearly all other cases (except $p = nsec = 4, maxl = 4$), parallel execution times are greater than theoretical execution times. This phenomenon may be explained in the following way. If nsec subdomains are used, then the datastructure, describing the grid, is split into nsec parts. If $p$ processors are applied, then the coarse-grained parallelism (cncall-directives) first distributes $p$ subdomains over the $p$ processors, which involves a
part \( \frac{P}{nsec} \) of the datastructure. We expect the data to be loaded in the cache (256 kbytes), but apparently the complete datastructure does not fit into the cache. Therefore, the smaller \( \frac{P}{nsec} \), the more profit can be taken from the fact that information concerning the first \( p \) subdomains stays in the cache, leading to faster (effective) computations. The case \( p=2 \), \( \text{maxl}=5 \) serves as an example: if \( nsec=2 \), then the complete datastructure should be loaded in the cache, since parallel computations involve the complete grid: \( \frac{P}{nsec} = 1 \). If \( nsec=4 \), then subdomains 1 and 2 (\( p=2 \)) are handled first (in parallel). This involves only half of the datastructure: \( \frac{P}{nsec} = \frac{1}{2} \). It seems that one half of the datastructure fits in the cache, while the complete datastructure does not, regarding parallel execution times of 6.63 seconds and 6.93 seconds, respectively. Execution times were estimated as 6.82 seconds and 6.70 seconds, respectively.

The second testproblem that we have run is the same problem as testproblem 2 of the previous section (6.3): multigrid on a locally refined grid, which is representative for the type of grids obtained in the hyperbolic part of the reservoir simulator. Again, the subdomains are strip-shaped, while direct agglomeration takes care of coarser levels. The grid itself is given in figure 6.3.8. Table 6.4.3 shows some experimental results for testproblem 2.

<table>
<thead>
<tr>
<th>( p )</th>
<th>( nsec )</th>
<th>( \text{minl} )</th>
<th>( \text{maxl} )</th>
<th>( T_p ) in secs.</th>
<th>( S )</th>
<th>( E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>2.31</td>
<td>1.62</td>
<td>0.81</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>2.36</td>
<td>1.59</td>
<td>0.80</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>4.41</td>
<td>1.75</td>
<td>0.88</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>5.23</td>
<td>1.48</td>
<td>0.74</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1.71</td>
<td>2.19</td>
<td>0.73</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>3.69</td>
<td>2.09</td>
<td>0.70</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1.67</td>
<td>2.25</td>
<td>0.56</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>1.76</td>
<td>2.13</td>
<td>0.53</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>3.61</td>
<td>2.14</td>
<td>0.54</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>3.45</td>
<td>2.24</td>
<td>0.56</td>
</tr>
</tbody>
</table>

Table 6.4.3: Experimental results for testproblem 2, direct agglomeration.

In table 6.4.3 speed-up values are based upon sequential execution times of 3.75 and 7.72 seconds, for \( \text{maxl}=5 \) and \( \text{maxl}=6 \) respectively. From table 6.4.3 we observe that for
p=4, nsec=8, the speed-up is larger than for p=4, nsec=4, in the case of maxl=6. This is caused by a better load-balancing and the cache effect, as described above.

In the case of 4 processors, the parallel efficiency degrades to slightly more than 50%. For testproblem 1 (uniform refinement, perfect load-balancing), efficiencies are approximately 80% for 4 processors. Therefore, application of the load-balancing procedure, as implemented in the hyperbolic part of the reservoir simulator, may increase the efficiency of the parallel elliptic algorithm. Since the overhead of the load-balancing procedure is already counted for in the parallel hyperbolic algorithm, we apply the load-balancing to the subdomain configuration, before starting the multigrid process (including timing).

The load-balancing procedure is applied to the initial configuration of 4 strip-shaped subdomains. This leads to the following irregular subdomain configuration, with maxl = 6:

![Subdomain configuration for 4 subdomains, after load-balancing.](image)

So far, internal boundary smoothing only handled horizontal internal boundaries, i.e. internal boundaries without any corners. The load-balancing procedure introduces irregular shaped subdomains, and hence corners. To smooth around such corners, which may involve more than two subdomains, the standard internal boundary smoothing process should be extended. First of all, we ignore these extensions and do not smooth around corners. This leads to the following experimental results:
Table 6.4.4: Experimental results for testproblem 2 with load-balancing (lb).

<table>
<thead>
<tr>
<th>p</th>
<th>nsec</th>
<th>minl</th>
<th>maxl</th>
<th>$T_p$ in secs.</th>
<th>S with lb</th>
<th>E with lb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>without lb</td>
<td>with lb</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1.67</td>
<td>1.37</td>
<td>2.74</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>3.61</td>
<td>2.44</td>
<td>3.16</td>
</tr>
</tbody>
</table>

After 5 iterations, the maximum norm of the interface residuals equals $O(10^{-7})$, in both cases (maxl=5 and maxl=6, with load-balancing). If load-balancing is not applied, and if the initial strip-shaped subdomains are processed, then the interface residuals equal $O(10^{-8})$, as is the case in the sequential algorithm. These convergence results suggest that skipping the smoothing around corners does not seriously degrade multigrid convergence. However, if more iterations (n) are performed, we get the following convergence behaviour (maxl = 6):

Figure 6.4.3: Convergence behaviour for testproblem 2 (nsec=4, maxl=6):
- n: no load-balancing,
- Δ: load-balancing, without corner smoothing.
Figure 6.4.3 shows that after a few iterations, convergence slows down if no corner smoothing is applied. This phenomenon can be explained by the idea behind multigrid methods: rapidly varying error components are smoothed well, but it takes many smoothing steps to reduce the error. Therefore, a sequence of coarser grids is induced to reduce the error in a cheaper way: coarse-grid correction. In our case, in the first few iterations, error reduction is mainly caused by smoothing on the coarse grids. Since a direct agglomeration process is applied to handle the two coarsest grids (and hence has nothing to do with corners!), convergence behaviour is as usual. After the first few iterations, the smoother on the finer levels causes further convergence. Since corner-smoothing is ignored at finer levels, we expect convergence to slow down, which is depicted in figure 6.4.3. For this reason, although speed-up values are pretty good, we will not accept internal boundary smoothing without handling the corners in a proper way. Results for maxl=5 are similar.

In order to handle quadruple updates around corners, the "lowest-numbered subdomain principle" is used, as before. Apart from the usual horizontal or vertical boundaries, we have essentially one of the following types of corners:

a) 2 subdomains

b) 3 subdomains

c) 4 subdomains

![Figure 6.4.4: Corner quadruples.](image)

In sections 4.4 and 4.5, the internal boundary datastructure and the load-balancing process were discussed. Only blocks of load-balancing level minl could be moved from one subdomain to a neighbouring one. Furthermore, internal boundary interfaces of load-balancing level are roots of the segmented tree structure that describes the internal boundaries. Therefore, corners only occur at the start or at the end of such segments. Using this observation, it is possible to determine corners. In actually doing so, the fact that each processor can access the complete datastructure (shared memory) makes programming work and data transport less complicated than it should be in the case of a local memory machine.
The actual implementation of the 3 types of corners has not led to any increase in parallel execution time. But, as could be expected, convergence results are much better. If corner smoothing is applied, convergence behaviour is similar to convergence behaviour in the sequential or strip-shaped parallel cases. We conclude that it is possible, for locally refined grids, to reach speed-up values comparable with speed-up values corresponding to perfectly load-balanced (uniform) grids, with the same convergence rates as in the sequential implementation of the multigrid process for locally refined grids.

Another example for which corner smoothing is necessary to achieve good convergence rates is given by square-shaped subdomains. In the case of 4 square-shaped subdomains, uniform refinement, maxl=5, we compare convergence rates of the multigrid process with and without application of corner smoothing:

![Graph showing convergence behaviour](image)

---

**Figure 6.4.5:** Convergence behaviour for testproblem 1 (nsec=4, square-shaped, maxl=5).  
- : with corner smoothing,  
\( \Delta \): without corner smoothing.

Figure 6.4.5 shows that corner smoothing is essential to obtain good convergence rates. Timing results for square-shaped subdomains do not differ from timing results for strip-shaped subdomains.
Remark 1. In the beginning of this section we stated that we do not use buffers during internal boundary smoothing. If corners are present, the quadruple updates around the corners may use boundary values which are "old" (i.e. new values have been put in the buffer, but have not been used yet). These "old" values may lead to slower convergence, after the first few iterations. Therefore, if corners are present, it is better not to use buffers.

Remark 2. Two runs for one problem with a grid with internal corners may lead to somewhat different results. This is caused by the fact that the sequence of internal boundary quadruples, including corners, may be somewhat different for each (parallel) run, due to synchronization effects.

Remark 3. In the implementation of the hyperbolic part on the Alliant, we noticed the effect of compiler options on execution times. In section 5.4 we had to compute different values of $\alpha$, to represent the computational work for one processor and for more processors. In this elliptic part, it was not necessary to distinguish between one processor and more processors. A speculative explanation is that the size of the elliptic problem (larger than the hyperbolic problem), combined with cache effects, has influence on execution times.

Remark 4. The quantity $\gamma$, representing distribution points, is determined from the case nsec=p=2, maxl=4. It is also possible to compute $\gamma$ from other situations, assuming that $\alpha$ and $\beta$ are fixed. E.g., in the case nsec=8, p=2, maxl=5, we find that $\gamma = -5.7 \cdot 10^{-4}$ s, which partly reflects the cache effect: the cache does not have to be re-filled for each subdomain, leading to some gain in speed. Furthermore, inaccuracies in timing experiments (and hence also in the values of $\alpha$ and $\beta$) may cause the observed discrepancies.

Conclusions:

* high parallel performance for uniform problems;
* for locally refined grids, load–balancing combined with direct agglomeration, leads to high parallel performances;
* smoothing around internal corners is required, to maintain good convergence rates;
* modelling of computational and communicational work is influenced by cache effects.
CONCLUDING REMARKS

In this thesis, we have investigated the parallel properties of an experimental reservoir simulator. Our ultimate goal, besides general insight in parallel computing for non-trivial, industrial problems, was to achieve a parallel implementation of this reservoir simulator on general parallel computer architectures. Since the reservoir simulator is based upon concepts (multigrid and adaptive grid refinement) that are rather widely used, we believe that our results and techniques can be generalized to a class of scientific computing problems.

The main objective of parallel computing is reduction of wall-clock time. If we consider our speed-up values for parallel machines with 4 to 8 processors, then we believe that the results are good. In these cases, the sizes of our testproblems were large enough to have computational work dominating over communicational work. For more processors, the problem size should be larger to achieve the same efficiency as observed for the testproblems on less processors. In practice, larger problems than our testproblems are under hand, making the use of more processors attractive. These observations show that communicational tasks (physical data transfer or distribution and synchronization) may degrade parallel performance, if the computational tasks are too small compared with the number of processors. The results of chapter 2 concerning the parallel solution of banded systems of equations, agree with this conclusion.

In the parallel elliptic update (chapter 6), the parallel implementation of the multigrid method followed two distinct ways to treat the coarse grids: stepwise and direct agglomeration have been implemented. Stepwise agglomeration is more efficient than direct agglomeration, if the underlying problem is uniform. For such problems, the subdomain configuration is regular, and subdomains can easily be agglomerated. If adaptive grid refinement is applied, load-balancing of the work-load over the processors may be necessary, thus leading to irregular subdomain configurations. To continue the
multigrid process on coarser grids, then, direct agglomeration to one grid has to be applied. For more than 4 subdomains (processors), direct agglomeration (which is sequential) degrades the parallel performance. Combination of stepwise and direct agglomeration may lead to an agglomeration procedure that combines the advantages of both individual agglomeration procedures.

Apart from computational work, it is possible to model communicational work, both for the hyperbolic and the elliptic part. For local memory machines, communication schedules reflect the computational overhead and data transfer between processors. Parallel data transfer and overlap of computational and communicational work, if possible, have a positive influence on speed-up values. For shared memory machines, distribution points are relatively expensive. On the other hand, domain decomposition subdivides the underlying problem into a number of subproblems, which may compensate for the distribution points if the parallel computer is equipped with a fast cache (cf. section 6.4).

Domain decomposition is a natural way to implement grid problems on parallel computers that support coarse-grained parallelism. The internal boundaries between the subdomains require a special treatment, reflected in their own data structure. Re-arrangement of the underlying algorithm with respect to these internal boundaries is necessary in many cases. Furthermore, the programming efforts, especially for local memory machines, are quite large. A correct implementation of the node program, with communication between the processors and the host program, including loading of the node programs and the distribution of data over the processors, requires a lot of additional work. Therefore, there seems to be a tendency to use local memory machines more and more as dedicated machines, on which special applications are implemented. The amount of additional work, apart from the parallel method, to reach a correct and efficient parallel implementation is rather large for local memory machines, and only for special applications it may pay off.

For shared memory machines, coarse-grained parallelism on subroutine level is supported by compiler directives. Additional programming work is not too large for the implementation of the parallel method. The compiler is instructed to distribute the execution of the program over the processors. The implementation of parallel codes is best reflected by the fact that the "shared memory program" does not need to contain machine-dependent subroutines, and therefore can be run on each non-parallel computer, leading to a correct solution. This is not yet possible for the "local memory program", which contains machine-dependent functions for communication. Also for this reason, we do not expect, for the near future, that local memory machines can serve as general purpose parallel computers, but rather as computers on which certain
applications can be run efficiently. On the other hand, we expect shared memory machines to be more suitable to handle more general parallel problems.
REFERENCES


[SNNi] Joel H. Saltz, Vijay K. Naik and David M. Nicol, *Reduction of the effects of
the communication delays in scientific algorithms on message passing


[StTr] K. Stuben and U. Trottenberg, *Multigrid methods: fundamental algorithms,
model problem analysis and applications*, in: W. Hackbusch, U. Trottenberg,
edits., Multigrid methods, Proc. Koln–Porz, Lecture Notes in Mathematics 961,
Springer Verlag, Berlin, 1981.


algorithms in Fortran on Cray–1 and CYBER 205 supercomputers*, report

SUMMARY

In the oil-industry, it is quite common to drill injection wells, apart from production wells, during the exploration and production of oil-reservoirs. Water, steam or chemicals are injected, leading to fluid flow in the reservoir, in order to produce the resulting oil of the reservoir. It is also quite common by now to simulate the exploration and production of a reservoir using computer models. These reservoir simulators are based upon partial differential equations, describing the fluid flow through porous media. The injection-production mechanism is reflected by a simplified modelproblem, with one injection well and one production well. Fluid flows from the injection well towards the production well.

In this thesis, we have considered a reservoir simulator, based upon a mixed finite element discretization of the underlying partial differential equations as a starting-point. Adaptive grid refinement is applied, in order to mark the moving front of water and oil (in the most simple case) in an efficient way. The solution method is known as the IMPES-scheme, which consists of an implicit and an explicit part. The implicit part (computation of pressure and total flow) is the most computer time consuming part of the simulator. Multigrid is applied to solve efficiently the linear systems of equations that arise in the implicit part. Complete vectorization of the algorithm is not well possible because of the applied adaptive grid refinement technique. However, parallel computing may be an alternative way to gain computational speed.

We report on our research with respect to the parallel features of the aforedescribed reservoir simulator, with the ultimate goal to obtain an efficient version of the simulator for rather general parallel computers. Before considering the reservoir simulator itself, we have studied less complex problems (i.e. linear banded systems) for their parallel properties. It has turned out that methods which subdivide the matrix into parts (partition methods) are the most efficient ones, although communication between the processors of the parallel computer may have serious consequences for the speed-up of the parallel algorithm. It is possible to model this communicational work, apart from computational work, which means that parallel execution times and speed-up values can
be predicted.

After a brief introduction to the reservoir simulator, we have considered the parallel method to be applied. Restricting ourselves to parallel computers with a rather modest number of processors, we have chosen a domain decomposition method, that subdivides the computational domain into a number of subdomains, which are assigned to the processors. This leads to internal boundaries between the subdomains. The datastructure for these internal boundaries is complicated due to the adaptive grid refinement, which also causes the work-load of the processors to change during the simulation time. In order to balance the work-load more or less equally over the processors, the subdomain configuration is adapted, if necessary, which restricts the domain decomposition. Also the treatment of coarse grids during the multigrid process has consequences for the domain decomposition.

The parallel algorithm for both the explicit and the implicit part has been implemented on local memory and shared memory machines (NCube/four and Alliant FX/4). Similar to the parallel solution of (banded) systems of equations, computational and communicational work can be modelled, which again makes it possible to predict parallel execution times and speed-up factors for larger numbers of processors and larger problem sizes. Speed-up factors, relative to the fastest sequential algorithm, are considerably. Even for our, rather small-sized, testproblems, efficiencies can be as high as 75%.
SAMENVATTING

In de olie-industrie is het gebruikelijk om gedurende de productie van een olie-reservoir naast productie-putten ook injectie-putten te slaan. Water, stoom of chemicaliën worden het reservoir in geperst, waardoor er stroming in het reservoir ontstaat en men de olie gemakkelijker hoopt te kunnen winnen. Het is ook gebruikelijk om de productie van een reservoir te simuleren met behulp van grote computer-programma's. Deze zogenaamde reservoir simulatoren zijn gebaseerd op partiële differentiaalvergelijkingen die de stroming van vloeistoffen en gas door poreuze media beschrijven. Het injectie-productie mechanisme kan weergegeven worden in een vereenvoudigd modelprobleem, dat uitgaat van een 2-dimensionaal reservoir, met een injectie-put en een productie-put. Er vindt dan stroming plaats van de injectie-put naar de productie-put.

De reservoir simulator die in dit proefschrift als uitgangspunt genomen is, discretiseert de vergelijkingen met behulp van een gemengde eindige elementen methode. Adaptieve roosterverfijning wordt toegepast, teneinde het scheidingsvlak van water en olie (in het eenvoudigste geval) efficiënt te volgen. De oplossmethode staat bekend als het IMPES-schema, wat bestaat uit een impliciet en een expliciet gedeelte. Het impliciete gedeelte (berekenen van druk en totale snelheid) is het meest rekenintensieve gedeelte van de simulator. Een multigrid methode wordt gebruikt om het stelsel lineaire vergelijkingen in het impliciete gedeelte efficiënt op te lossen. Het gebruik van adaptieve roosterverfijning maakt vectorisatie van het algoritme slechts beperkt mogelijk. Parallel rekenen is echter een andere mogelijkheid om tot snelheidswinst te komen.

In dit proefschrift wordt verslag gedaan van onderzoek naar de parallelle eigenschappen van de bovenbeschreven reservoir simulator, met als doel te komen tot een efficiënte implementatie op parallelle computers. Alvorens de reservoir simulator zelf te beschouwen, is onderzocht in hoeverre minder complexe problemen (nl. lineaire band-stelsels vergelijkingen) met behulp van een parallelle computer efficiënt kunnen worden opgelost. Gebleken is dat methoden die het stelsel opdelen (partitie-methodes) het meest bruikbaar zijn, hoewel vooral de communicatie tussen de verschillende
processoren van de parallelle computer zeer nadelige gevolgen voor de speed-up kan hebben. Het blijkt mogelijk te zijn betrouwbare modellen voor de hoeveelheid communicatie, naast modellen voor de hoeveelheid rekenwerk, op te stellen, zodat speed-up factoren te voorspellen zijn.

Na een korte beschrijving van de reservoir simulator wordt ingegaan op de parallele methode, die toegepast gaat worden. Vanwege onze gerichtheid op parallelle computers met een beperkt aantal processoren wordt gekozen voor een domein decompositie methode, die het rekengebied opsplits in een aantal subdomeinen, die aan de processoren toegekend worden. Dit leidt tot interne randen tussen de subdomeinen, waarvan de beschrijving (datastructuur) gecompliceerd wordt door de lokale verfijning. Deze lokale verfijning veroorzaakt tevens een steeds wisselende werklust voor de processoren. Teneinde de totale werklust zo goed mogelijk te verdelen, wordt de subdomeinverdeling, indien nodig, aangepast. Ook de behandeling van grove roosters tijdens het multigrid proces legt bepaalde beperkingen op aan de domein decompositie.

Actuele implementatie van het parallelle algoritme, voor zowel het expliciete als het impliciete (multigrid) gedeelte heeft plaatsgevonden op local memory en shared memory computers (NCube/four en Alliant FX/4). Net als voor het parallel oplossen van stelsels vergelijkingen is het ook nu mogelijk modellen op te stellen voor rekenwerk en communicatie-werk. Met behulp van deze modellen kunnen speed-up factoren voor meer processoren en grotere problemen betrouwbaar voorspeld worden. De snelheidswinsten ten opzichte van het snelste sequentiële algoritme blijken, zowel experimenteel als theoretisch volgens deze modellen, vrij groot te zijn. Voor testproblemen van een redelijke grootte liggen de efficiency-waarden boven de 75%.
CURRICULUM VITAE