NTNU

Norwegian University of
Science and Technology

A Parallel Implementation of Mortal Random Walkers in the Pore Network of a Sandstone

Per Mathias Nedrebø

Master of Science in Physics and Mathematics

Submission date: September 2008

Supervisor: Einar Rønquist, MATH

Co-supervisor: Jørn Amundsen, IDI

Ståle Fjeldstad, Numerical Rocks

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

Look at different parallelization approaches for a model simulating random walkers in a pore volume, implement and analyze them.

Assignment given: 21. April 2008
Supervisor: Einar Rønquist, MATH

A Parallel Implementation of Mortal Random Walkers in the Pore Network of a Sandstone

Mathias Nedrebø

TMA4910 Numerical Mathematics, Master Thesis
Department of Mathematical Sciences

NTNU, 2008

Mathias Nedrebø
Department of Mathematical Sciences
Norwegian University of Science and Technology
Trondheim
Norway

Printed September 22, 2008.

Typeset using LaTeX 2_ε.
Plots and figures created using Python and PSTricks.

Preface

This report is the result of the work done in the cource “TMA4910 Numerical Mathematics, Master Thesis”. The work has been carried out at the Department of Mathematical Sciences at the Norwegian University of Science and Technology (NTNU) in collaboration with Numerical Rocks.

I would like thank my supervisors, professor Einar Rønquist at the Department of Mathematical Sciences, and Senior Software Developer Ståle Fjeldstad at Numerical Rocks for guidance throughout the work on this project.

Mathias Nedrebø

Trondheim, September 22, 2008

Abstract

Simulations of the nuclear magnetic resonance relaxation method is an important part of a digital laboratory developed by Numerical Rocks. The laboratory is used to model petrophysical properties and simulating fluid flow in the pore scale of reservoir rocks.

The nuclear magnetic resonance relaxation method can be simulated on a computer using a method involving random walkers. This computer simulation can be parallelized to reduce computational time. The aim of this study has been to examine how overlapping boundaries affects speed-up and communication in a parallel simulation of random walkers. Several parallel algorithms have been proposed and implemented.

It was found that an overlapping partitioning of the problem is recommended, and that communication decreases exponentially with increasing overlap. However, increased overlap resulted only in a small negative impact on memory usage and speed-up.

Contents

Chapter 1

Introduction

Oil companies around the world are constantly performing oil and gas explorations, trying to find new oil fields to develop. One oil field can be populated with one or more strategically placed oil wells. The cost of a single well can be tens of millions of dollars for certain fields, it is therefore important to do proper investigations before launching development.

When the evaluation of quality of a potential oil well is carried out, several parameters are considered. Based on these parameters an estimate of the cost for development and production can be made. Based on these figures and the expected amount of extracted oil and gas an estimate of profit can be made. If these estimates are favourable an eventual development of the well can begin. One of the important factors in this evaluation process is the permeability of the reservoir. If the permeability is large enough the well is said to be exploitable. The percentage of oil in a reservoir that can be extracted increases with increasing permeability.

Usually the measurement of permeability is carried out in a conventional laboratory. In the laboratory, samples of sandstone taken from a potential well can be analysed. Permeability is a measure of the ease with which a fluid can pass through the pore spaces of a sandstone formation. Permeability can either be determined by empirical experiments or be estimated by the Nuclear Magnetic Resonance (NMR) relaxation method [1]. The empirical methods requires very high pressure to be applied to the sample for a long period of time. This renders the experiments impractical and expensive, and NMR relaxation is usually the preferred method when determining permeability of sandstone.

Numerical Rocks is a Norwegian based company trying to provide technology for fast and reliable prediction of reservoir rock properties without use of traditional laboratory testing. They develop a digital laboratory where digital (numerical) rock models can be analyzed. The digital rocks can for instance be acquired by Micro-CT scanning of a sample from the reservoir, or the model can be created from a thin rock section using Numerical Rocks e-Core technology. One of the experiments that can be carried out on this digital rock is simulations of the NMR relaxation method. The heart of this method is ran-

dom walkers [2], and a fast implementation of the random walk algorithm is essential to the overall performance of the simulation.

This report will examine the possibility to harvest the powers of several computers through parallel computing to speed up the random walk algorithm.

The report is sectioned in six chapters and two appendices. Chapter 1 is the introduction. The second chapter goes through the foundations and terminology used. The fields of random walks and parallel computing will be presented. Following in the next chapter is the problem formulation. Both the details of the digital rock model and a mathematical description of the random walkers used are given. The fourth chapter proposes various algorithms for the random walk method; both serial and parallel algorithms are given. Chapter 5 addresses implementations of the algorithms and performance testing of these programs. Finally, in the sixth chapter, a conclusion is made. The two appendices contains program code and testing results.

Chapter 2

Theory

2.1 Random Walks

2.1.1 History

The term *random walk* was first used in 1905 by the British statistician Karl Pearson when he posted a question in *Nature*. There he presented a model for mosquito infestation in a forest. The model described mosquito movements as displacements of length a at fixed time steps, but at a random angle. Given this model Pearson wondered what the mosquito distribution would be after n steps [3]. Lord Rayleigh had already solved a similar, but more general problem 25 yeas earlier and posted a solution to Pearson's problem just one week later.

A few yeas earlier, in 1900 the French mathematician Luis Bachelier published his doctor thesis *The Theory of Speculation* [4]. This is considered to be the first paper to use advanced mathematics in the study of finance. Bachelier discussed the use of Brownian motion to evaluate stock options. The ideas of Bachelier did not gain recognition until 73 years later when Burton Malkiel released his book *A Random Walk Down Wall Street* [5] and revived Bacheliers ideas. This book is now regarded a classic in theoretical finance.

The year 1905 was also the year when Albert Einstein published his paper *On the movement of small particles suspended in a stationary liquid demanded by the molecular-kinetic theory of heat* [6]. This was in a time with uncertainty concerning whether matter where continuum or discrete particles. By studying the Brownian motion of a dust particle i a microscope Einstein argued that atoms could be counted.

As history shows, the field of random walks is broad and applications of random walks can be seen in many scientific disciplines. It is reasonable to say that the scientific area of random walks was developed on the brink of the twentieth century. Now, a century later, random walks can be seen in a broad range of applications.

2.1.2 Applications

In thermodynamics, random walks can be used to model the propagation of heat. This can be done since the heat equation (also called diffusion equation) and the probability density function for Brownian motion is the same, and random walks is merely a discretized Brownian motion.

Random walks is often used to model different gambling problems. The *gambler's ruin* [7] problem is a simple example of this.

In polymer physics the formation of an ideal chain (the simplest model to describe a polymer) can be viewed as a random walk [8].

The decay of radiation from molecules in a sample analyzed by the nuclear magnetic resonance relaxation method can be simulated using random walks [9].

2.1.3 Random Walker

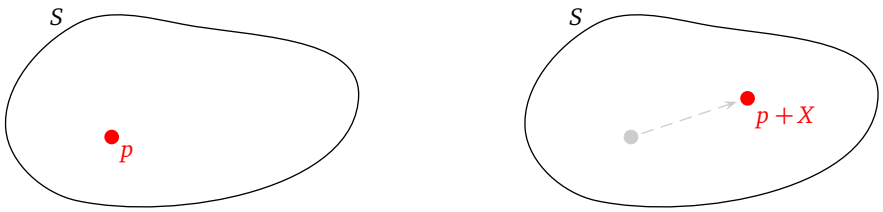
The particles participating in a random walk model are called *random walkers*, or simply walkers. A walker is associated with a location p in a space S and two random variables X and Y that are described by the probability density functions $f(x)$ and $g(y)$. The function $g(y)$ gives the probability density for the time interval between each step of the walker. The probability density of the walkers displacement at each step is given by the function $f(x)$. Both functions may depend on the position of the walker p or time t, or both.

$$\begin{aligned} &p \in S \\ &X \sim f(x) = f(x;p,t) \\ &Y \sim g(y) = g(y;p,t) \quad y > 0 \end{aligned} \tag{2.1}$$

For instance, a detailed model of the movement of a mosquito in a forest will depend on both p and t. Mosquitoes are more active during night, hence $g(y)$ should depend on t and favour smaller Y during night time. In addition, mosquitoes will most likely not fly hundreds of meters out on open water. Therefore, if a huge lake is located in the forest, $f(x)$ should depend on p and give X values that tend to guide the mosquitoes on shore. On the other hand we have Einsteins experiment, with a random walker (dust particle) in a homogeneous and time independent environment (air). Such problems have $f(x)$ and $g(y)$ that are both independent of p and t.

The locations of a random walker at any step can be given as a recursive function, with p_0 being the starting location of the walker and p_n being the location of the walker after n steps. This is described by (2.2) and it is illustrated in Figure 2.1.

$$\begin{aligned} &p_0 \in S \\ &p_n = p_{n-1} + X \in S \qquad n > 0 \\ &X \sim f(x;p_{n-1},t) \end{aligned} \tag{2.2}$$



(a) Location of the walker at time t.

(b) Location of the walker at time $t + Y$.

Figure 2.1: The movement of a random walker in a space S is governed by two random variables X and Y. The time for the next step is given by Y, and the walker displacement for that step is given by X.

The functions $f(x)$ and $g(y)$, and the space S depends on the problem at hand. The simplest possible random walk model describes a walker released at the origin of the one dimensional integer line $\mathbb{Z}$. The walker is only allowed to take unit steps left or right with equal probability and at intervals of unit time. This discrete model is given by the following system of equations:

$$\begin{aligned} S &= \mathbb{Z} \\ x_0 &= 0 \\ X \sim f(x) &= \begin{cases} \frac{1}{2} & \text{if} \quad x = \pm 1 \\ 0 & \text{else} \end{cases} \\ Y &= 1 \end{aligned} \tag{2.3}$$

A walker governed by (2.3) is shown in Figure 2.2.

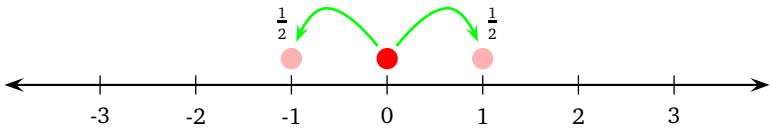


Figure 2.2: A one dimensional random walker located at the origin of the integer line. When the walker take its next step it will either go to -1 or 1, with equal probability.

2.1.4 Random Environment

The classical random walk problems, like the ones studied around year 1900 and the one given in (2.3), all consider particles in a fairly homogeneous media. However many problems involve highly irregular or chaotic media. Such media are called *random environments* [10].

For instance in polymer physics the structure of a polymer can be vastly affected by random impurities [11], and the ideal chain model is no longer valid.

Another random environment is the pore network of a sand stone. Diffusion in such a media may be modeled as random walks in a random environment. This is the kind of environment we will consider in the following chapters.

2.1.5 Mortality

A mortal walker is a walker that can cease to exist (or stop to walk) given some condition. A walker can for instance be assigned a death rate. This can be achieved by letting the walkers be killed with probability λ at each time step [12]. A second variation is to let the walker die when some predefined condition is met. For instance will the gambler in *gambler's ruin* stop to gamble when (or if) he looses all his money. If the gambler's fortune is viewed as a random walker on the positive half of the integer line, then the walker will die when it reaches the origin for the first time. This corresponds to the situation that the gambler has no more money to gamble with.

2.2 Random Numbers

The heart of all random walk methods is the generation of random steps. A basis for this process is the generation of one or more *random numbers*. These numbers can in turn be used to determine the time interval between steps, and the length and direction of steps. Random walks usually consists of a large amount of steps, and even the smallest bias in number generation may add up to significantly alter the final result. It is therefore important that the numbers used in a random walk method are truly random, or at least that they are random enough to not affect the results of the method.

When simulating random walks by the use of computers a huge amount of random numbers is needed. There are two different approaches for obtaining these numbers. The first approach is to get the numbers from some external source. The second one is to let the computer generate the numbers using an algorithm.

When using an external source *true random numbers*[1] can be found and used in the random walk. However, the process of finding these numbers is slow and expensive, and an algorithm generating random numbers is usually preferred.

[1] True random numbers can only be found by using a *non-deterministic* source to produce randomness. It is common to find truly random numbers by taking samples from a physical process that appear random at macroscopic scale. Examples of such processes are cosmic background noise or merely the toss of a fair coin.

When an algorithm creating random numbers is run by a computer a huge amount of numbers can be found at almost no cost. However, due to the deterministic nature of algorithms true random numbers can not be found, only *pseudo-random numbers* can be generated. Sooner or later the algorithm is going to repeat it selves, starting over again on the number sequence it has already generated. Hence it is important to choose a random number generator that produces a sufficient amount of random numbers, given the number of steps that the walkers will take.

2.3 Parallelization

2.3.1 Introduction

Ever since the appearance of the first computers, increased computing power has been the main driving force for further development of computer technology. Before the dawn of computers a scientist could perform approximately 10^2 calculations in an hour. Even early computers could do tens of thousands (10^4) of calculations. Once this immense computational power was available previously unsolvable problems could be solved.

Before 1952 only the 12 smallest Mersenne primes[2] were known, and the 12^{th} one being $2^{127}-1$ was discovered almost a century earlier by François Édouard Anatole Lucas. In 1952 five new and larger numbers were found by Raphael M. Robinson in under one year, by the use of a computer program. The largest one he found was $2^{2281}-1$[†]. Identifying and verifying the validity of those numbers would not have been possible without the aid of computers.

However, there will always appear new problems that require more computational power to be solved, or problems that can be solved more accurately if more computational power where available. This is the driving force behind the never ending strive in computer development. Intel Corporation co-founder Gordon E. Moore stated in 1965 that the number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially, doubling approximately every couple of years [13]. This has later been known as *"Moore's Law"* as the trend he observed have continued since then. History has also shown that the doubling happens even faster than every two years, and every 18^{th} month is now usually used when referring to the "law". Most computer resources, like processing speed and memory capacity follows Moore's Law quite closely [14], and hence the computational power of a computer follows this law.

While "Moore's Law" states that evolution of computers in fact happens extremely rapidly, people always tend to want more. According to Moore you

[2] A Mersenne number is a number of the form 2^n-1 and a Mersenne prime is a Mersenne number that also is a prime number.

[†] As of September 22, 2008, 44 Mersenne Primes are known, the largest one being $2^{32582657}-1$

need to wait for more then eleven ($2^{11/2} < \mathbf{50}$) years if you need a computer that is 50 times a fast as the one you are using today. However, there is an alternative approach to the problem of computational speed. A construction company can not exchange their workers for workers that is 50 times more efficient, no matter how long they wait. So, what do they do? They hire more workers. The same can be done to increase the computational force, several computers can be told to cooperate on the same task. This is the essence of parallel computing, if you buy 50 computers you do not have to wait eleven years for the faster one. Unfortunately, just like on the construction site where 50 men need more supervision than one man, computers need special guidance if they are going to collaborate. The *serial* programs and algorithms that work on a single computer needs to be made *parallel* in a way such that each computer knows what part of the problem to work on.

In addition to the increased calculation power available through parallelization, there is a second and just as important advantage. The memory available to a parallel program is linearly dependent on the number of computers running the program. A lot of problems involve amounts of data that is larger than what a single computer can cope with. By distributing the data among computers so that each just holds a part of the problem in memory, they can handle problems of larger size than each could on their own.

2.3.2 Terminology

To avoid confusion the various terms used in this report when describing parallel computing, and the relation between these terms will be presented here.

Serial Computer A serial computer is a computer that is only capable of perform one task at a time. Serial computers can appear to do tasks simultaneously by rapidly alternating between them. This technique is called *multitasking*. However, multitasking has the drawback that each task is performed slower because it is frequently paused. Most older[3] personal computers are serial computers.

Parallel Computer A parallel computer is a computer that is capable of performing more than one task *simultaneously* without multitasking. They are categorized by how many simultaneous tasks they can perform. Modern personal computers can usually perform two to four tasks at the same time. Several serial or parallel computers can be connected together to construct a bigger parallel computer. Such parallel computers are often called *clusters* or *supercomputers*. Large clusters can perform thousands[4] of simultaneous tasks.

[3] The production of *parallel* personal computers started around year 2006.

[4] As of September 22, 2008 the worlds most powerful supercomputer can perform approximately hundred thousand simultaneous tasks (`http://www.top500.org`).

Computer When the term computer is used without further specifications it refers to a single physical computer, in contrast to a cluster. The individual computers that make up a cluster are called *nodes*.

Processor All computers contain at least one central processing unit (CPU). The CPU consists of one or more *cores*. It is the number of cores in a computer that limits the number of tasks that can be performed simultaneously. One core can carry out only one task at a time. When the term processor is used, it refers to the number of cores in the system, not the number of central processing units.

Process A process is the execution of program code on one processor. One processor can only run one process at a time, and one process can only be run by one processor at a time. Most programs are implemented to use just one process, and such programs will not be able to take advantage of parallel computers.

Serial Program A serial program is a program that spawn only one process.

Parallel Program A parallel program is a program that can spawn several processes. If the parallel program is started on a parallel computer the processes can run simultaneously. However if the parallel program is started on a serial computer the processes will be multitasked. Whenever the term process is used in relation to a parallel program, it is assumed that there is enough processors available to avoid multitasking. Most parallel programs, including the ones implemented in Chapter 5, ensures that the number of processes spawn matches the number of processors in the system it is run on.

2.3.3 Performance and speed-up

When evaluating a serial program we are interested in the estimated running time given a particular problem size n. Depending on the particular algorithm implemented, this might be a constant number or it might be an (exponential) function of n. The running time of a serial program is denoted $T_S(n)$.

Similarly, the running time of a parallel program depends on the problem size n, but in addition it depends on the number of processes p collaborating on the task. This function is called $T_P(n,p)$. The total running time of a parallel program is measured as the time from initialization of the program until *all* processes are finished.

Once more we return to the construction workers. If we say that one worker can build a car shed in two weeks (75 hours), then fifty workers can build fifty car sheds in two weeks. However, no one would expect fifty workers to build one car shed in 90 minutes. Paint would have to dry, people would have to wait for each others to move and some tasks may have to wait until others

are finished. They can for instance not start on the roof or walls before the framework is finished.

The same principles apply to parallel computing, if you have p processors you can *at best* hope for the program to run p times faster. That happens however only on simple and rare occasions, and usually a lower *speed-up* is achieved. The speed-up of a parallel program can be expressed as

$$S(n,p) = \frac{T_S(n)}{T_P(n,p)} \leq p \tag{2.4}$$

and the *efficiency* of a parallel program can be stated as

$$E(n,p) = \frac{S(n,p)}{p} \tag{2.5}$$

To see why the speed-up $S(n,p)$ is not always equal to p, and is in fact most likely less than p, we can have a closer look at $T_P(p,n)$. The time consumed when running a parallel program can be divided in three categories: calculation time, time spent on I/O[5] and time spent on communication between processes. This can be expressed by

$$T_P(n,p) = T_{P,\text{calc.}}(n,p) + T_{P,\text{I/O}}(n,p) + T_{P,\text{comm.}}(n,p) \tag{2.6}$$

The parallel program has to do the same total amount of calculations as the serial program. It also might have to do some additional calculations to figure out what part of the problem each process is responsible for.

$$T_{P,\text{calc.}}(n,p) \geq \frac{1}{p} T_{S,\text{calc.}}(n) \tag{2.7}$$

The same argument applies to time spent on I/O, a certain amount of data has to be read. At minimum each process needs to read a $1/p$-part of the data.

$$T_{P,\text{I/O}}(n,p) \geq \frac{1}{p} T_{S,\text{I/O}}(n) \tag{2.8}$$

The serial program obviously does not need to spend time on communication, since it is working by itself. However, processes in most parallel programs *will* have to communicate with each other to divide work, share information and synchronize tasks that depend on each other.

$$T_{P,\text{comm.}}(n,p) \geq \frac{1}{p} T_{S,\text{comm.}}(n) = 0 \tag{2.9}$$

[5] I/O or input/output refers to the communication between a program and the outer world, i.e. human interaction with the program or hard drive access

Combining the previous equations gives a lower bound for $T_P(n,p)$. However, the equality will only hold for trivial situations.

$$T_P(n,p) \geq \frac{1}{p}\left(T_{S,\text{calc.}}(n) + T_{S,\text{I/O}}(n) + T_{S,\text{comm.}}(n)\right) = \frac{1}{p}T_S(n) \tag{2.10}$$

By inserting (2.10) into (2.4) it can be verified that p is the maximum attainable speed-up. Unfortunately a *slow-down* ($S(n,p) < 1$) is also possible if housekeeping tasks are excessive.

$$0 < S(n,p) \leq p \tag{2.11}$$

If a speed-up larger than p is achieved, the serial implementation is most likely not optimal.

An alternative analysis of the parallel computational time $T_P(n,p)$ can be performed by dividing $T_S(n)$ in two fractions. The first fraction is the part of $T_S(n)$ that is perfectly parallelizable, this part is of size r. The remainder is of size $1-r$ and is the part of $T_S(n)$ that is strictly serial. This part can not be parallelized, meaning that the running time of that part of the problem is constant, and independent of the number of processors available.

$$T_P(n,p) = \frac{1}{p}rT_S + (1-r)T_S \tag{2.12}$$

Equation (2.12) can be used to compute the speed-up $S(n,p)$:

$$S(n,p) = \frac{T_S}{p^{-1}rT_S + (1-r)T_S} = \frac{1}{1-r+rp^{-1}} \tag{2.13}$$

By noting that rp^{-1} never becomes negative an upper bound for speed-up can be found:

$$S(n,p) < \frac{1}{1-r} \tag{2.14}$$

This equation is named Amdahl's Law [15] and it states that any given problem has an upper limit for speed-up, independent of p, if it contains a serial part ($r < 1$). For instance if ten percent of a problem has to be done serially then $r = 0.9$ and the maximum speed-up possible is $S(n,p) = 10$, no matter how many processes that are assigned to a task.

A common misinterpretation of Amdahl's Law is that massive parallelism is impossible. Fortunately that is not entirely true. The fraction of a problem $1-r$ that is strictly serial is usually inversely dependent on the problem size n. Amdahl simply states that for a *fixed* problem size there is an upper bound to $S(n,p)$, but in most real applications the reason for increasing p is to solve larger problems, i.e. larger n. The larger n will in turn result in an increased r and higher theoretical speed-up limit. However, it certainly is beneficial to have problems with high r even for small n.

When developing a parallel program it is important to keep in mind that both (2.11) and (2.14) limits the theoretical speed-up attainable.

Chapter 3

Problem

3.1 Overview

The sandstone permeability in an oil well is of great interest to oil companies. A well with low permeability is generally less profitable than a well with high permeability. One method used to measure permeability of sandstone is the Nuclear Magnetic Resonance (NMR) Relaxation method [16]. The NMR method is usually carried out in a laboratory. However, a popular alternative is to simulate the NMR process using computers [17].

To do NMR simulation digital sandstone models are required. A sandstone model consists of discrete sites located in a three dimensional cube. Each site is either solid or liquid. A cut plane from a digitalized sandstone sample of size $N^3 = 600^3$ is shown in Figure 3.1.

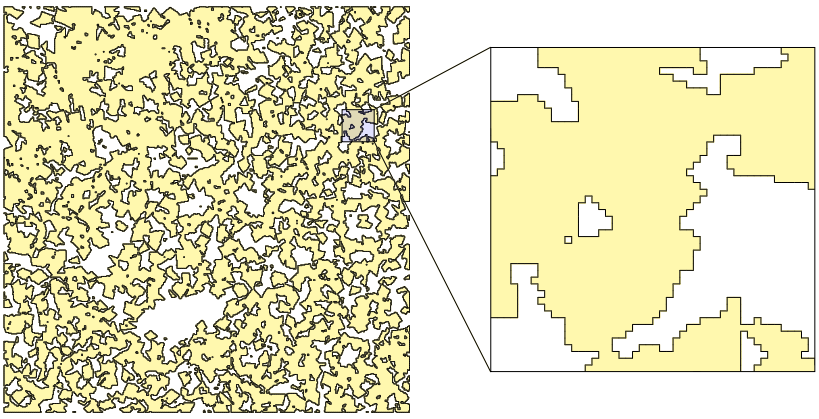
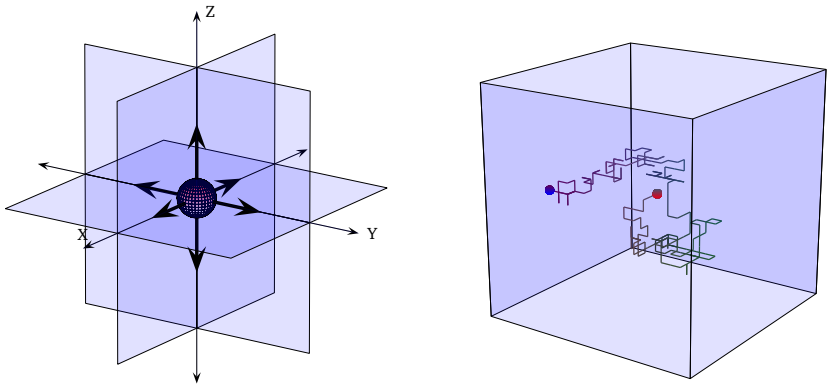


Figure 3.1: A two dimensional cut plane from a sandstone model of size 600^3. The yellow areas are solids, and the white areas in between the solids contain liquids. A small part of the sandstone is magnified and shown to the right to reveal the detailed structure of the discrete model.

By placing random walkers evenly spaced in the voids of the sandstone, NMR can be simulated. The walkers then move at random, and when a walker hit a wall it is either bounced back into the void with probability $1 - \lambda$ or it dies. In a physical context, the walkers correspond to atoms in the liquids filling the sandstone voids. The nuclei of these atoms are, during the NMR process, placed in a precessing state by an applied magnetic field. The number of atoms in a sample precessing can be found by measuring radiation. Due to diffusion, precessing atoms will move around and occasionally hit the solid walls of the voids. During these collisions there is a certain chance λ that atoms will cease to precess, and the total radiation from the sample will decrease. By recording the decreasing radiation from a given sample geologists can determine the permeability of the sandstone. It is the diffusion of precessing atoms that are simulated using random walkers, and dying walkers corresponds to atom nuclei ceasing to precess.

The random walkers used in our model can move in the six directions along the axes of $\mathbb{Z}^3$. The step length and the time interval between steps are fixed. Choice of direction however, is random and one of the six possible directions for movement is chosen with equal probability at each step. There is no correlation between steps. A walker with these properties is shown in Figure 3.2.



(a) The walker can move in six directions. It can move along the three axes of $\mathbb{Z}^3$, in both directions. Each direction is equally likely.

(b) A walker moving in a three dimensional, discrete and homogeneous box. The walkers were released at the center of the box.

Figure 3.2: A three dimensional random walker.

The resolution of the digital sandstone model is far more coarse-grained than atomic scale. Hence, each cell of the model will in reality contain lots of atoms. However, the digital model will only contain at maximum ~ 1 walkers in each cell. Collisions among walkers will therefore be disregarded in NMR simulations.

A sandstone sample is in reality just a small piece of a far larger system. It

is therefore natural to use periodic border conditions in the model.

3.2 Formal definition

The model we are looking at in this report can be described as a three-dimensional lattice A of size N^3. This is where the walkers shall move, and A is given by

$$A = \{(i,j,k) \ : \ 0 \leq i,j,k < N, \quad i,j,k \in \mathbb{N}\} \tag{3.1}$$

The lattice A is periodic in all three dimensions. This implies that walkers leaving the model on one side will re-enter on the opposite side:

$$A_{i,j,k} = A_{i+lN,j+mN,k+nN} \quad \forall \quad l,m,n \in \mathbb{Z} \tag{3.2}$$

The lattice sites of A can be divided into two groups. The site in the model that make up the voids of the sandstone are the sites that are *accessible* to walkers. These sites belongs to the first group, A_a. Sites in the solid part of the sandstone model belongs to A_i. These are the sites *inaccessible* to walkers. Each site in the model belongs to exactly one of the two sets. These properties can be summarized by the following equation:

$$\begin{aligned} A_a &\subseteq A \\ A_i &\subseteq A \\ A_a \cap A_i &= \varnothing \\ A_a \cup A_i &= A \end{aligned} \tag{3.3}$$

Given this partitioning a walker moving to a site in A_i will return to A_a with probability $1-\lambda$ and die with probability λ.

Walkers should initially be evenly spaced in A_a. This is fulfilled by placing one walker at each site in A_a. Walkers are named by their starting position. A walker starting at A_{ijk} is named w_{ijk}. The position of a walker after a given number of steps is given by the recursive function p_n

$$\begin{aligned} p_0(w_{ijk}) &= (i,j,k) \\ p_{n+1}(w_{ijk}) &= p_n(w_{ijk}) + X \end{aligned} \tag{3.4}$$

where X is a random variable. Every walker should move to one of its six neighbouring sites in A at each time step. It is assumed that there is no drift or rotation, hence each of the six directions should be chosen by equal chance, with a probability of $\frac{1}{6}$. The random variable X determines the sequence of steps and is then given by:

$$X \sim f(x) = \begin{cases} \frac{1}{6} & \text{if} \quad x \in \{(\pm 1,0,0),(0,\pm 1,0),(0,0,\pm 1)\} \\ 0 & \text{else} \end{cases} \tag{3.5}$$

More than one walker can occupy the same coordinate point without interference between the walkers. If the site a walker moves to is in A_i the walker dies with probability λ. If the walker survives it is placed back to its last location in A_a. Walkers moving to a site in A_a simply continues its journey the next time step.

The simulations should continue until all walkers are dead, and the number of walkers still alive at each time step, should be recorded.

Chapter 4

Algorithms

4.1 Introduction

Various algorithms for the random walk problem stated in Chapter 3 will be proposed in this section. Relevant code describing the algorithms will be given, using a pseudo language. A sample algorithm demonstrating the structure of the pseudo code used is shown in Algorithm 1.

```
1: procedure ProcedureName(arg_1, arg_2, ···)
2:     for iterator assignments do
3:         if condition_1 then                          ▷ This is a comment
4:             var_1 ← var_2 + var_3          ▷ The arrow (←) means assignment
5:         else if condition_2 then
6:             var_1 ← ProcedureCall(arguments)
7:         else
8:             var_1 ← Null
9:     return return_value
```

Algorithm 1: Sample pseudo code. The code can consist of procedures with return values, procedure calls and control flow keywords.

The pseudo code may use verbose statements to describe an action, for instance in line 5 in Algorithm 2 the word **record** is used to indicate that we store the value $steps$, but we are not concerned with the details of how this is done.

A lattice A is provided by an *ASCII* text file and needs to be read from disc by the programs. At the end of execution results should be written to a single text file. However, the reading and writing of files are implementation dependent and the time spent on those tasks does not contribute significantly to the overall running time. I/O will therefore not be covered in this section.

The algorithms presented in the remainder of this section will somehow be provided with a three dimensional array A. It might be given as a procedure

argument or the algorithm may ask for it when needed. The algorithms release one walker in every site in A_a and terminates when all the walkers are dead.

4.1.1 Implementation language and libraries

All actual algorithm implementations are coded using the `C++` programming language [18] and the parallel implementations use Message Passing Interface (MPI)[1] for communication [15]. Performance of these implementations will be evaluated in Chapter 5.

4.2 Serial Algorithm

An algorithm solving the stated random walk problem in a serial manner is proposed in Algorithm 2. The algorithm iterates through each point in A and checks if that point is accessible to a walker. If it is accessible, a walker is released and the number of steps taken by the walker before it enters A_i and dies is recorded. When a walker released at one point is dead, the algorithm chooses a new point and releases a walker there. This process continues until there is no more points to release walkers from and the algorithm halts.

```
1:  procedure MAIN(A)
2:      for x in A do
3:          if x ∈ A_a then
4:              steps ← DOWALK(A, x)
5:              record steps
6:
7:  procedure DOWALK(A, x)
8:      steps ← 0
9:      repeat
10:         steps ← steps + 1
11:         x_next ← x + RANDOMSTEP()
12:         if x_next ∈ A_a then
13:             x ← x_next
14:     until walker dies
15:     return steps
```

Algorithm 2: Random Walk, Serial version. If the walker enters A_i and does not die the displacement is rejected, but it still counts as a step.

Algorithm 2 will be used as a reference algorithm when we later evaluate the speed-up of the parallel algorithms of this chapter.

[1] The official MPI (Message Passing Interface) standards documents can be found at `http://www.mpi-forum.org/`

4.3 Parallel Algorithm

When describing a parallel algorithm a few problems arise that are not present in serial algorithms. It is no longer sufficient with an algorithm that just instructs a single process. The algorithm needs to instruct a group of processes. It may even have to give each process customized instructions. This could be solved by writing one algorithm for each process, and for situations with just a few processes that approach could be feasible. However, when the number of processes grow large such an approach would be tedious. A second complication is the desire to be able to reuse an algorithm later and possibly with a different number of processes. That would be impossible with a one-to-one mapping between algorithm and process.

An common solution is to make *one* algorithm able to instruct *several* processes. This can be done by using conditional statements in the algorithm. An example algorithm using this approach is shown in Algorithm 3.

```
1:  id ← ParallelStart()                    ▷ Process gets an unique id ∈ [0, p)
2:
3:  if id = 0 then                          ▷ Processes can then do com-
4:      DoSomeWork( )                       ▷ pletely different work ...
5:  else if id = 1 then
6:      DoSomeOtherWork( )
7:                                          ▷ ... or they can do the same
8:  my_data ← FindMyProblemShare(id)        ▷ work, but work on different
9:  DoWork(my_data)                         ▷ data depending on their id
10:
11: Send(val_out, (id + 1))                 ▷ Communication with neighbours
12: val_in ← Receive(id − 1)
13:
14: ParallelEnd( )                          ▷ Wait until all other process are finished
```

Algorithm 3: Concepts of parallelization. The variable named p is the total number of processes executing the algorithm. The procedure FindProblemShare(id) returns a unique part of the problem to each process, and it ensures that the entire problem gets assigned to some process. How such a procedure can be implemented in a real application depends on the problem at hand.

The idea behind this approach is that each process executes the same algorithm as the others. However, the first thing that happens in the algorithm is the assignment of an unique id to each process, as shown in line 1 in Algorithm 3. Throughout the rest of the algorithm the process id is used to decide what work to do in each process. It is possible to assign specific tasks to specific processes, as shown in line 3, or different processes can work on different parts of a large problem as shown in line 8. The id can also be used to communicate with other processes. In line 11 all processes send a value to the process with higher id

and receive a value from the one with lower id. This particular communication technique is called ring communication, since it can be visualized as if all processes are organized in a ring and only communicating with the ones on their left and right hand side. Numerous other communication schemes can also be constructed using the process id's. In an actual ring communication implementation the process with $id = p - 1$ needs to send its value to the process with $id = 0$, since there is no process p. However, this is omitted in the algorithm to increase readability.

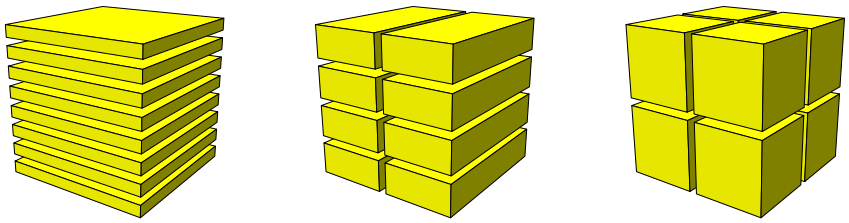
4.3.1 Implementation Factors

When parallelizing an algorithm several design decisions must be made. The first one is what to parallelize. One can either parallelize just work, or work and memory. The algorithmic design process can be simplified significantly if the problem does not require much memory and all processes can store the entire data set needed. Communication in such algorithms can usually be kept at a minimum. Unfortunately the random walk problem requires the three dimensional environment lattcie A to be stored and only small models would fit in the memory of a single computer. With the exception of Algorithm 4 in section 4.4 all algorithms presented will be *data-parallel*.

The second decision is how to divide the workload and distribute data. Given that the lattice A is three dimensional and cubic, there is three partition schemes that are natural to consider. The simplest one is a one dimensional *layered* partitioning as shown in Figure 4.1(a). The second option is a two dimensional partitioning, as shown in Figure 4.1(b). A third option is the three dimensional *cubic* partitioning, which is shown in Figure 4.1(c). Both the layered and the cubic approach have its advantages. In the layered approach each process has just two neighbours to communicate with, while processes in a 2D- or cubic partitioning has several. On the other hand, the surface-to-volume ratio is smallest in the cubic approach, resulting in less need for communication. Whenever a walker passes the surface of a process space it has to be transferred to a neighbouring process. This is the root of communication, and the smaller the surface of a process space is, the less communication is needed.

The one dimensional partitioning is not going to be used in any algorithms due to its very bad surface-to-volume ratio. The two dimensional partitioning is not going to be used due to its relatively bad surface-to-volume ratio in addition to processes having several neighbours. While the cubic partitioning requires the most advanced communication schemes, it is still chosen due to its superior surface-to-volume ratio. This is especially true when p grow large, as shown in Figure 4.2.

A third concern is what to do if some processes are assigned significantly more work than others. Given the nature of a random environment this is not an unlikely scenario, and in the extreme case one process might be assigned just points from A_a while an other only gets points from A_i. Naturally, the latter



(a) Layered (1D) partitioning (b) 2D partitioning (c) Cubic (3D) partitioning

Figure 4.1: Three alternatives for partitioning of the lattice A among eight processes.

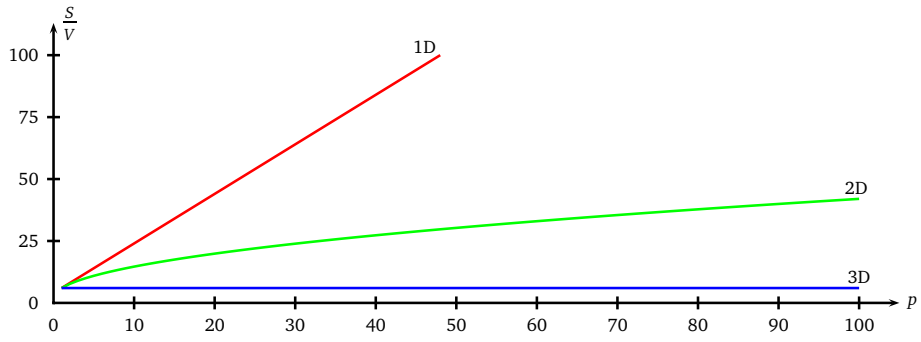


Figure 4.2: Surface-to-volume ($\frac{S}{V}$) ratio for the partitioning of a unit cube, using the three partition schemes shown in Figure 4.1 with different number of processes p.

process will not have any work to do at all and its calculation power will be wasted unless the algorithm can adapt to the situation and reassign work. This is the problem of *load balancing* and it will not be considered in this report.

4.4 Simple Parallel Version

This simplest possible parallel algorithm for the random walk problem is presented in Algorithm 4. The algorithm is parallel in work, but not in data, meaning that each of the p processes will only calculate the route of a fraction of the walkers, but all processes stores the entire walker environment lattice A in memory. The environment is partitioned in p cubic volumes, and each process is responsible for the walkers released in one of these cubes. This partitioning is shown in Figure 4.3. Since each process stores the entire environment A in memory, there will be no need to transfer walkers to other processes, even if a walker walks outside of its starting cube. Hence there will be no communication.

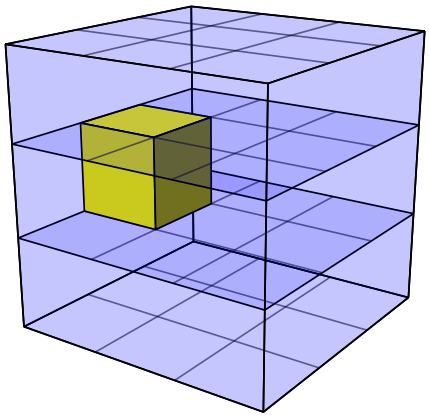


Figure 4.3: A cubical partition of work between 27 processes. The box shows the part of A where one particular process will be releasing walkers. Since all processes have access to the entire lattice A, they can continue to track walkers even when the walkers leave the box they were released in.

The choice of partitioning method is in fact irrelevant since no communication is needed. The cubic partitioning scheme is merely used here to demonstrate how it can be used, in a simplest possible context. Any partitioning method that assign an equal number of release points to each process could in fact be used in this algorithm.

Because of the cubic partitioning it is assumed in the algorithm that the number of processes p is a cubic number. It is also required that A, which is of size N^3, can be partitioned equally between the processes.

$$\begin{aligned} p &= d^3 \\ d &\in \mathbb{N} \\ \frac{N}{d} &\in \mathbb{N} \end{aligned} \tag{4.1}$$

```
1: procedure MAIN(A)
2:     id ← PARALLELSTART()
3:     i, j, k ← GETGRIDPOSITION(id, 3)
4:     for x in A_ijk do
5:         if x ∈ A_a then
6:             steps ← DOWALK(A, x)          ▷ See Algorithm 2, line 7
7:             record steps
```

Algorithm 4: Work-parallel random walk algorithm, with all data known to all processes. The procedure GETGRIDPOSITION(id, dim) is used to align all processes to a dim-dimensional grid as shown in Figure 4.3. The return values $i, j, k \in [0, d)$ are unique to each process. A_{ijk} is a part of A, defined by regarding A as a block matrix of size $d \times d \times d$.

The changes from Algorithm 2 to Algorithm 4 are small and not in the *hot spot*[2] of the algorithm. An implementation of Algorithm 4 is therefore expected to achieve a speed-up close to p, for all but the smallest problem sizes[3]. However, the implementation will not be very scalable, since every single process working on the problem needs to store all of A. Adding more processes will only reduce computation time, it will not increase the maximum problem size manageable.

The memory usage per process of a program implementing Algorithm 4 is

$$M_{SPV} = C \cdot N^3 \tag{4.2}$$

where C is a constant containing implementation details related to the storage of A.

4.5 Fully Parallel Version

While the algorithm in section 4.4 is parallel, it is not scalable. In this section an algorithm parallel in both work and memory will be presented. This algorithm will be fully scalable. The partitioning of the lattice A remains the same as in

[2] A hot spot is an area of the algorithm where the program implementing the algorithm will spend much of its time. Typical hot spots are the deepest nested loops of a program.

[3] If parallel programs are given to small problems to work on, will the time doing computations usually be less than the time doing housekeeping tasks. This is true even if there is no communication in the program.

Figure 4.3. And like before, each process is responsible for releasing walkers in one of the sub cubes of A. However each process does no longer store the entire lattice A, they only stores the part of A where they will be releasing walkers. This implies that the memory usage per process of a program implementing this algorithm will be

$$M_{FPV}(p) = \frac{M_{SPV}}{p} \tag{4.3}$$

where p is the number of processes and M_{SPV} is the memory usage per process of Algorithm 4.

Once a walker steps outside the domain of a process, the responsibility for that walker is transferred to the process responsible for the domain the walker enters. When a walker reaches the border of a domain, two things need to be done. First the responsible process needs to figure out which process the walker is heading for. Secondly it transfers relevant information to this neighbouring process. This information consists of walker position and the number of steps already taken by the walker. When the walker is sent away, the origin process forgets all about it and starts to process the next walker waiting in line. An algorithm describing the scheme outlined above is presented in Algorithm 5.

Due to the partitioning of the storage of lattice A between processes, far larger models can be analyzed using the improved algorithm presented here. By doubling number of processes, the maximum problem size doubles. In fact, the maximum manageable problem size will be p times that of Algorithm 4. Unfortunately this comes at a cost. Whenever a walker leaves a process additional work needs to be carried out to transfer walker information to the correct neighbour. It is therefore expected that the speed-up of Algorithm 5 is less than the speed-up of Algorithm 4.

4.6 Parallel Version With Overlapping Boundaries

If we focus on a walker located at the edge of one of the cubes in Algorithm 5 some interesting properties can be observed. Recall that each walker moves at random. Hence the walker close to the surface of the box will be likely to cross the surface, and thereby sent to the neighbour process. Once transferred to the neighbour the walker is still close to the surface and it may very well cross it again at the very next step or just a few steps later. Then, the walker needs to be transferred back to the original process again. Even though the cubic partitioning scheme minimized the surface-to-volume ratio, a large portion of the walkers are located near the cube surface. It is therefore inevitable that a large portion of walkers will jump forth and back between processes, creating large amounts of communication.

An algorithm that tries to overcome the problem of jumping walkers is presented in this section. The problem stated above is a result of the location of the entering point of walkers transferred from other processes. When the walker

```
1:  procedure MAIN( )
2:      id ← PARALLELSTART()
3:      i, j, k ← GETGRIDPOSITION(id, 3)
4:      A_ijk ← GETSUBPROBLEM(i, j, k)
5:      for x in A_ijk do
6:          if x ∈ A_a then
7:              steps ← DOWALK(A_ijk, x)
8:              if steps > 0 then                ▷ Check if walker has died or left
9:                  record steps
10:         PROCESSINCOMMINGWALKERS(A_ijk)
11:     while there is walkers left in A do      ▷ Wait for all walkers
12:         PROCESSINCOMMINGWALKERS(A_ijk)
13:
14: procedure PROCESSINCOMMINGWALKERS(A_ijk)
15:     while x, steps ← RECEIVEWALKER() do
16:         steps ← DOWALK(A_ijk, x, steps)
17:         if steps > 0 then                    ▷ Check if walker has died or left
18:             record steps
19:
20: procedure DOWALK(A_ijk, x, steps = 0)        ▷ steps defaults to 0 if not given
21:     repeat
22:         steps ← steps + 1
23:         x_next ← x + RANDOMSTEP()
24:         if x_next ∉ A_ijk then
25:             SENDWALKER(x_next, steps)        ▷ Send walker to neighbour process
26:             return 0
27:         if x_next ∈ A_a then
28:             x ← x_next
29:     until walker dies or has left
30:     return steps
```

Algorithm 5: A work- and data-parallel algorithm. A call to the procedure GETSUBPROBLEM(i, j, k) returns a three dimensional matrix A_{ijk}. This matrix contains the part of A corresponding to a cubic partitioning of A among p processes. Unlike in Algorithm 4 each process only stores this small part of A, allowing larger problems to be solved. The DOWALK() procedure is altered so that it stops *either* when a walker dies *or* when the walker leaves A_{ijk}. This procedure is also altered to take a third and optional argument called $steps$. Using this argument walks can be continued from a given step. The procedure PROCESSINCOMMINGWALKERS() looks for walkers entering A_{ijk} and lets them continue their walk in A_{ijk} with previously taken steps remembered. This procedure continues processing walkers until there is no more walker waiting to enter A_{ijk}. The last **while** loop in MAIN() ensures that the algorithm does not end until all walkers are dead.

is released by its new process it is just one step away from re-entering the old process. By introducing a buffer area between all processes, where any of the two can control the walker, communication will be reduced. This buffer can be constructed by extending the space assigned to each process by a fraction in all directions, as illustrated in Figure 4.4. Each process will still just release walkers in the same points as before, not in the entire extended space. Even the walkers earlier located at the edge of the process spaces will now be located some distance from the new edge. All walkers will therefore have to move at least some fixed non-zero distance before reaching areas not controlled by their parent processes. When a walker finally reaches the border of the process space, it will be transferred to the neighbour process, just as before. The entering point however, will no longer be on the edge of the new process space. The walker does now have to cross the entire buffer once more to return to its origin process. This will significantly reduce the need for communication between processes.

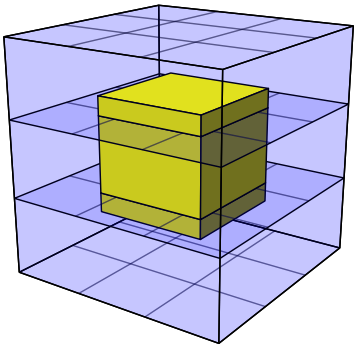
The drawback of this approach, is increased memory usage. The amount of additional memory needed depends on the size of the buffer, but it will by far be smaller than the memory usage of Algorithm 4. The exact memory usage per process for a program implementing this algorithm is given by

$$M_{PVOB}(p,o) = \left(\sqrt[3]{\frac{M_{SPV}}{p}} + 2 \cdot C \cdot o \right)^3 \tag{4.4}$$

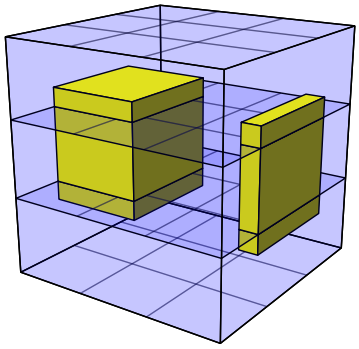
where p is the number of processes, o is the thickness of the space extension measured in lattices, M_{SPV} is the memory usage of Algorithm 4. The constant C contains implementation details related to the storage of A, this is the same constant as in equation (4.2). This equation behaves like $M_{SPV}p^{-1}$ for small o. Hence, the memory usage per process is close to that of Algorithm 5, and significantly less than that of Algorithm 4.

An algorithm describing the scheme outlined above is presented in Algorithm 6. Even though the algorithm does not seem much more complicated than Algorithm 5, the implementation of this algorithm will be more cumbersome due to the fact that two, four or even up to eight processes might store the same data. The extrema will happen in corners of the cubes, where eight process spaces meet. They will all overlap once the spaces are extended.

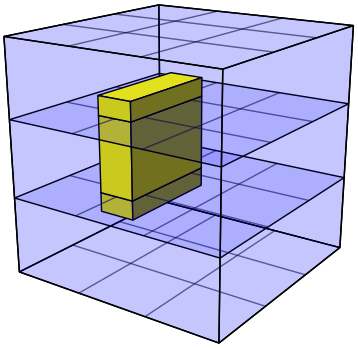
The speed-up of Algorithm 6 is expected to be in between the speed-up of Algorithm 5 and Algorithm 4. This is due to the fact that Algorithm 6 will generate less communication than Algorithm 5, but not as little as zero which is the case for Algorithm 4.



(a) The box located in the center of A, extended in all directions.



(b) The extended box might grow out of A. This part is then mapped to the opposite side of A due to the periodic border conditions.



(c) The part of A which is contained in both the boxes above. Walkers in this volume might be controlled by either of the two processes responsible for these two boxes.

Figure 4.4: Partitioning of the lattice A with overlapping boundaries. Compared to the box in Figure 4.3 the boxes here are expanded in all directions.

```
1:  procedure MAIN(overlap)
2:      id ← PARALLELSTART()
3:      i, j, k ← GETGRIDPOSITION(id, 3)
4:      A_ijko ← GETOVERLAPPINGSUBPROBLEM(i, j, k, overlap)
5:      for x in A_ijk do
6:          if x ∈ A_a then
7:              steps ← DOWALK(A_ijko, x)
8:              if steps > 0 then                ▷ Check if walker has died or left
9:                  record steps
10:         PROCESSINCOMMINGWALKERS(A_ijko)
11:     while there is walkers left in A do      ▷ Wait for all walkers
12:         PROCESSINCOMMINGWALKERS(A_ijko)
13:
14: procedure PROCESSINCOMMINGWALKERS(A_ijko)
15:     while x, steps ← RECEIVEWALKER() do
16:         steps ← DOWALK(A_ijko, x, steps)
17:         if steps > 0 then                    ▷ Check if walker has died or left
18:             record steps
19:
20: procedure DOWALK(A_ijko, x, steps = 0)
21:     repeat
22:         steps ← steps + 1
23:         x_next ← x + RANDOMSTEP()
24:         if x_next ∉ A_ijko then
25:             SENDWALKER(x_next, steps)        ▷ Send walker to neighbour process
26:             return 0
27:         if x_next ∈ A_a then
28:             x ← x_next
29:     until walker dies or has left
30:     return steps
```

Algorithm 6: Work- and data-parallel algorithm, with overlapping process boundaries. This algorithm is very similar to Algorithm 5. The main difference can be found in line 4 where the procedure GETOVERLAPPINGSUBPROBLEM() is called instead of GETSUBPROBLEM(). This procedure returns a matrix A_{ijko} which is similar to A_{ijk}, except that it also contains the points surrounding A_{ijk}. The number of extra points included in A_{ijko} is given by the procedure argument $overlap$. These extra points constructs the buffers between the processes. Throughout the algorithm the new extended matrix A_{ijko} is used in place of A_{ijk}. The only exception to this is in line 5 where A_{ijk} is still used. Here A_{ijk} is used since processes should not release walkers in the buffer.

Chapter 5

Implementation and Results

5.1 Implementing the Algorithms

All algorithms presented in the previous chapter have been implemented as computer programs. The implementations are written in the C++ programming language [18], and they are listed in Appendix A. The relation between algorithms in Chapter 4 and the code listings in Appendix A is shown in Table 5.1.

Table 5.1: Correlation between sections describing the algorithms and the appendix listing the source code implementations.

Section	Appendix
4.2 Serial Algorithm	A.1
4.4 Simple Parallel Version	A.2
4.5 Fully Parallel Version	A.3
4.6 Parallel Version With Overlapping Boundaries	A.4

5.2 Testing of the Programs

5.2.1 Testing Data

The various implementations in Table 5.1 have been tested with four or five different data sets. The data sets are all three dimensional discrete sandstone models, containing 80^3, 160^3, 300^3, 600^3 or 1000^3 data points arranged in a cubic grid. A cut plane of the second largest model is shown in Figure 3.1.

5.2.2 Testing Computers

The programs have been tested with two different hardware configurations. These configurations will in the following sections be labeled as SUN and IBM.

SUN The first setup used was a Sun-Fire-V490 server. This is a computer with four dual-thread UltraSPARC IV CPUs. This computer can run MPI jobs with up to eight simultaneous processes.

IBM The second setup used was a IBM P5 P575+ cluster with eight dual-core Power5+ CPUs per node. This computer can run MPI jobs with up to 864 simultaneous processes.

5.2.3 Testing Routines

The programs have been tested for total running time and for total amount of communication produced.

The running time is measured as the time from the start of the random walk, until all walkers are dead. The time used reading model data from file and initialization of MPI is not included. These steps are very dependent on implementation and platform. On some systems all processes can do I/O. Other systems may be restricted to one process doing all the I/O, and then that process is responsible for distribution of the data among the other processes. In a real world application the data may also already be loaded into memory by an earlier step in the computational pipeline. Hence the time consumed during I/O is disregarded.

Communication is measured as the total number of walkers leaving the processes throughout the entire execution of the program. This number can later be used to tell the rate at which the walkers leave the processes.

All timing simulations have been done at least three times consecutively to eliminate false results due to caching of data and other errors related to optimization done by the operating system. The two last results of the three simulations have been kept. If the two results differed significantly the test was executed again, now with five consecutively runs, where the last four was kept for later use. The results presented in this chapter are the mean values of the accepted runs.

5.3 Testing Results

5.3.1 Speed-up

When executing program A.4, which is the program with overlapping boundaries, the size of the overlap has been set to 10 points. This has been done for all the speed-up tests.

Results from the SUN computer

The timing results from the SUN computer are presented in Table 5.2 and the corresponding speed-ups are given in Table 5.3. Program A.4 and program A.2

have been executed using the full computer capacity of eight processes. The memory requirements for the data set containing 1000^3 grid points exceeded the memory capacity of the computer. Hence the tests are only conducted with the four smallest data sets.

Table 5.2: The number of seconds used by the random walk implementations when executed on the SUN machine. The tests have been executed for different data set sizes using eight processes for the parallel programs.

	Problem size			
Program	80^3	160^3	300^3	600^3
A.1	6	81	358	5021
A.2	1	16	71	719
A.4	1	9	43	745

Table 5.3: The speed-up achieved on the SUN machine when using eight processes in the parallel programs. The numbers are calculated from the entries of Table 5.2.

	Problem size			
Program	80^3	160^3	300^3	600^3
A.1	1.0	1.0	1.0	1.0
A.2	6.0	5.1	5.0	7.0
A.4	6.0	9.0	8.3	6.7

The running times for the smallest data set are so short that the two parallel programs appear similar. However, it is evident that parallel programs give a significant speedup compared to the serial version. A speed-up of five or higher is in fact seen in all tests. The speed-up of program A.2 is close to eight for the largest problem size. This is expected, as mentioned in section 4.4.

For the two medium sized problem sets an interesting effect can be seen in the results from program A.4. The speed-up for these two tests are larger than the number of processes used. This is most likely due to cache memory optimization [19]. The memory usage of each process executing program A.4 is considerably less than the memory usage of processes storing the entire problem in memory. Hence, a larger portion of process data will fit in cache, the cache will need to be updated less frequently and the memory bandwidth will have a smaller impact on execution speed.

The effect of cache optimization becomes smaller when the model size increases. The performance of program A.2 is slightly better than the performance of program A.4 for the largest problem size. This is expected, as mentioned in section 4.6, and it is also likely to hold for larger problem sizes.

However, the memory usage of program A.2 is 6.6 times larger than that of program A.4 as given by equation 4.4. Hence, program A.4 should still be the preferred choice for problems of this or large size.

Results from the IBM computer

The timing results from tests executed on the IBM computer using eight processes are presented in Table 5.4 and the corresponding speed-ups are given in Table 5.5.

Table 5.4: The number of seconds used by the random walk implementations when executed on the IBM machine. The tests have been executed for different data set sizes using eight processes for the parallel programs.

	Problem size				
Program	80^3	160^3	300^3	600^3	1000^3
A.1	1	23	95	1596	2800
A.2	1	12	50	1008	1518
A.4	1	9	42	1021	1382

Table 5.5: The speed-up achieved on the IBM machine when using eight processes in the parallel programs. The numbers are calculated from the entries of Table 5.4.

	Problem size				
Program	80^3	160^3	300^3	600^3	1000^3
A.1	1.0	1.0	1.0	1.0	1.0
A.2	1.0	1.9	1.9	1.6	1.8
A.4	1.0	2.6	2.3	1.6	2.0

From the tests of the serial program A.1 it is clear that the performance of the IBM computer is greater than the performance of SUN. The running times for the smallest data set are all to short to draw any conclusions about speed-up. Unfortunately and unexpectedly only small speed-ups are seen also for the larger problem sets. None of the tests show speed-ups even close to eight, as one would expect. This behaviour is likely to be caused by an unfortunate combination of communication implementation in the programs and hardware configuration. However, time constrains have not allowed sufficient investigation to locate the source of the error.

Due to the inexplicable results further test with larger amount of processes have not been conducted.

5.3.2 Communication

Special tests of program A.4 have been carried out to measure what impact the overlap thickness has on communication. When the program has been executed the number of walkers leaving the processes has been recorded. This experiment has been done for overlap thickness in the range from 0 to 40, and for data sets of size 80^3, 160^3, 300^3 and 600^3. The tests have been repeated with 8, 64, and 125 processes. The data material from these tests is quite comprehensive and it is listed in section B.1 in Appendix B. Semi-logarithmic plots of the data is shown in Figure 5.1, Figure 5.2 and Figure 5.3.

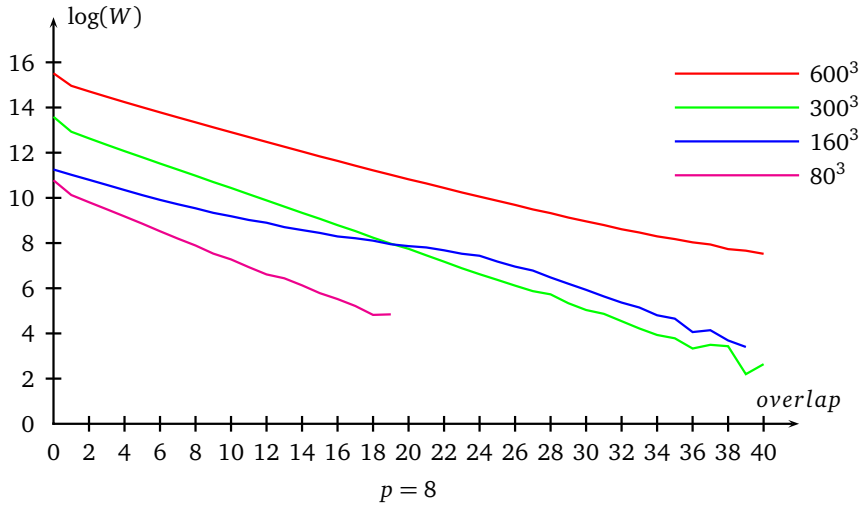


Figure 5.1: Number of walkers W sent between processes for different lattice sizes and different overlaps. The lattice A is partitioned between eight processes.

The plots show lines that are straight or close to straight for all data sets and for all process numbers. This implies that communication between processes decreases exponentially with increasing overlap thickness.

By looking at some entries in the tables the effect of this exponential relation can be clearly seen. Table B.2 shows the data from the test with 64 processes. The memory usage per process for the largest data set is given by

$$M_0 = \frac{(C \cdot 600)^3}{64} \tag{5.1}$$

if there is no overlap. C is a constant depending on data representation in the implementation, and it will be canceled out in the following equations. By increasing the overlap to 10 the memory usage for each process is given by (4.4) and equals

$$M_{10} = \left(\sqrt[3]{M_0} + 2 \cdot C \cdot 10 \right)^3 \approx 1.46 M_0 \tag{5.2}$$

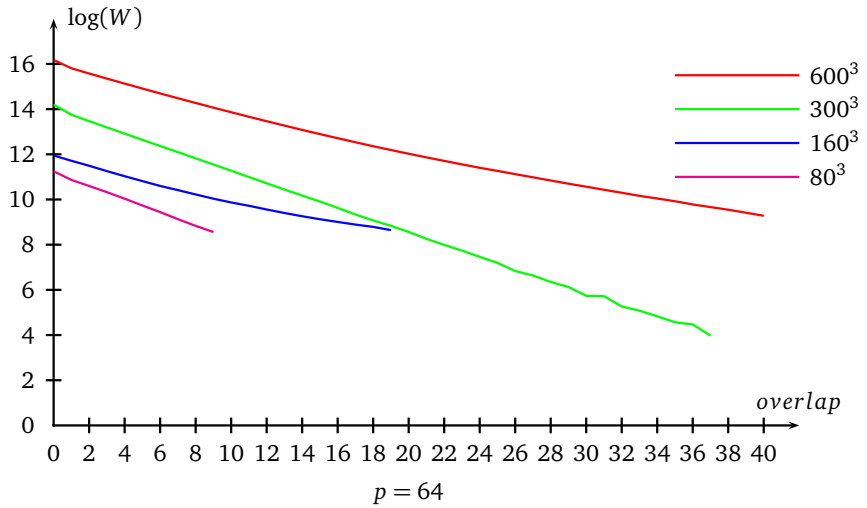


Figure 5.2: Number of walkers W sent between processes for different lattice sizes and different overlaps. The lattice A is partitioned between 64 processes.

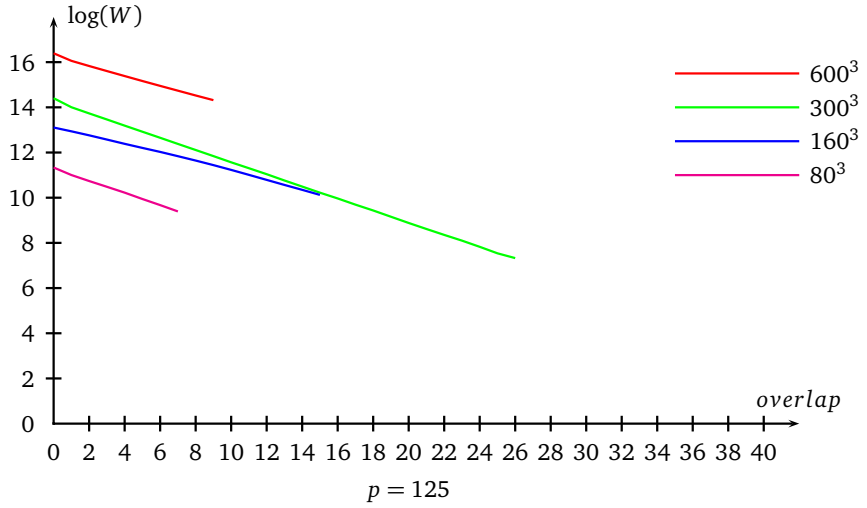


Figure 5.3: Number of walkers W sent between processes for different lattice sizes and different overlaps. The lattice A is partitioned between 125 processes.

That is an increased memory usage to only 1.5 of the original amount. At the same time the communication decreases significantly, from 10590048 transferred walkers to 1047001 transferred walkers. This equals a drop in communication to less than $1/10$ of the original communication. If one allows a memory increase to $3M_0$ the communication decreases to $1/410$ of the original communication. Even if one allows an increase in memory to $3M_0$ there is still only need for $1/21$ of the total model to be stored in each process. It is clear that communication decreases far more rapidly than memory is increasing.

By inspection of the tables in section B.1 in Appendix B it is evident that for large models it is possible to entirely eliminate communication. This can be done by increasing the overlap sufficiently, and still only a fraction of the model would have to be stored in each process.

Chapter 6

Conclusion

In this project several parallel algorithms for the random walk method have been proposed. The parallel algorithms were designed to minimize communication between processes executing implementations of the algorithm, while keeping memory usage close to the theoretical minimum. This was done by dividing the model data evenly among processes, and in addition each process was given access to all data in a layer "close" to itselves. The amount of additional data added to a process was measured in terms of the thickness of the data extension layer. A serial algorithm for the random walk problem have also been given. This serial algorithm has been used as a reference when evaluating the parallel algorithms.

All algorithms have been implemented as `C++` programs, and the parallel programs have been benchmarked against the serial program. The results from these tests indicates that the parallel programs achieve speed-ups close to the theoretical maximum. Complications occured when testing the programs on a larger system and thus further tests were not carried out.

Communication needs for the parallel algorithms were also examined. The amount of communication between processes was recorded for a large range of problem sizes and varying thickness of the data extension layer. The communication was found to decrease exponentially with increasing layer thickness, independent of the number of processes involved in the program execution. The increased memory usage due to the extra data stored in each process was modest for small overlaps.

The possibility to completely remove communication from the parallel programs was also discussed, and this can be accomplished by making the overlap sufficiently large. Communication will then either be zero, or it will be below some predefined tolerance level and one can discard the few walkers trying to leave. Implementation of the parallel algorithms can be simplified significantly by elimination of communication. The total running time of a program will also decrease if communication can be eliminated.

Continued studies of problems related to parallelization of the random walk model could consist of making a more robust prototype of the parallel program

and run tests for a larger number of processes than done in this report. A second interesting problem to be explored further is the properties of communication free programs. The thickness of a layer needed to eliminate communication will depend on the pore structure of the model. A model with large pores will need a larger overlap to stop all walkers from leaving. Development of a relation between these two properties could be done in a continued study.

Appendix A

Code

The following sections list the program implementations of the algorithms proposed in Chapter 4. The programming language used is C++.

A.1 Serial

```
001:  #include <cstdlib>
002:  #include <ctime>
003:  #include <iostream>
004:  #include <fstream>
005:  #include <string>
006:
007:  #define MORTALITY 0.1
008:
009:  using namespace std;
010:
011:  unsigned short int *read_model(const char* file_name,
012:                                 int *dx, int *dy, int *dz)
013:  {
014:    ifstream fs(file_name);
015:    string s;
016:    int dim;
017:
018:    // Skip two lines
019:    getline(fs, s);
020:    getline(fs, s);
021:
022:    // Read model dimension
023:    fs >> *dx;
024:    fs >> *dy;
025:    fs >> *dz;
026:    dim = *dx * *dy * *dz;
027:
028:    // Skip two lines
029:    getline(fs, s);
030:    getline(fs, s);
031:
032:    // Allocate three dimensional array
033:    unsigned short int *A = new unsigned short int[*dx * *dy * *dz];
034:
035:    // Read model
036:    int v;
037:    for(int z=0; z<*dz; z++)
```

```
038:      for(int y=0; y<*dy; y++)
039:        for(int x=0; x<*dx; x++)
040:          {
041:            fs >> v;
042:            A[z * *dy * *dx + y * *dx + x] = (unsigned short int)v;
043:          }
044:
045:    return A;
046:  }
047:
048:  int random_walk(unsigned short int *A,
049:                  unsigned int x, unsigned int y, unsigned int z,
050:                  int dx, int dy, int dz)
051:  {
052:    int steps = 0, dir, axis;
053:    bool died = false;
054:
055:    // Release one walkers if (x,y,z) is a non-solid point
056:    // and follow the walker until it dies
057:    if(!A[z*dy*dx+y*dx+x])
058:      while(!died)
059:        {
060:          dir = (rand() % 2) * 2 - 1;
061:          axis = rand() % 3;
062:          switch(axis)
063:            {
064:            case 0:
065:              x = (x + dir) % dx;
066:              break;
067:            case 1:
068:              y = (y + dir) % dy;
069:              break;
070:            case 2:
071:              z = (z + dir) % dz;
072:              break;
073:            }
074:          // Check if walker hits wall
075:          if(A[z*dy*dx+y*dx+x])
076:            {
077:              // Check if walker dies
078:              if(rand() < RAND_MAX * MORTALITY)
079:                died = true;
080:              else // If the walker dont die it is bounced back,
081:                switch(axis)  // return it to previous location
082:                  {
083:                  case 0:
084:                    x = (x + dx - dir) % dx;
085:                    break;
086:                  case 1:
087:                    y = (y + dy - dir) % dy;
088:                    break;
089:                  case 2:
090:                    z = (z + dz - dir) % dz;
091:                    break;
092:                  }
093:            }
094:          else
095:            steps++;
096:        }
097:    return steps;
098:  }
099:
```

```
100:  int main(int argc, const char* argv[])
101:  {
102:    time_t start_t, stop_t;
103:
104:    // Place a seed for the random function
105:    srand(time(0));
106:
107:    // Read the model into an array
108:    if(argc < 2)
109:      {
110:        cerr << "Usage: " << argv[0] << " model" << endl;
111:        return(-1);
112:      }
113:
114:    // Read model data from file
115:    // and print the time used
116:    time(&start_t);
117:    int dx, dy, dz;
118:    unsigned short int *A = read_model(argv[1], &dx, &dy, &dz);
119:    printf("Problem size:\t%dx%dx%d\n", dx, dy, dz);
120:    time(&stop_t);
121:
122:    printf("I/O time:\t%f\n", difftime(stop_t, start_t));
123:
124:    // Time the total random walk duration
125:    time(&start_t);
126:    for(int z=0; z<dz; z++)
127:      for(int y=0; y<dy; y++)
128:          for(int x=0; x<dx; x++)
129:            random_walk(A, x, y, z, dx, dy, dz);
130:    time(&stop_t);
131:
132:    printf("Walk time:\t%f\n", difftime(stop_t, start_t));
133:
134:    return(0);
135:  }
```

A.2 Simple Parallel

```
001:  #include <cstdlib>
002:  #include <fstream>
003:  #include <string>
004:  #include <unistd.h>
005:  #include "mpi.h"
006:
007:  #define NDIMS 3
008:  #define MORTALITY 0.1
009:
010:  using namespace std;
011:
012:  MPI_Status status;
013:  time_t start_t, stop_t;
014:
015:  void MPI_Exit(int e)
016:  {
017:    MPI_Finalize();
018:    exit(e);
019:  }
020:
```

```
021:  unsigned short int *read_model(const char *file_name, int *mod_dim)
022:  {
023:    // Read model from file
024:    //
025:    ifstream fs(file_name);
026:    string s;
027:
028:    // Skip two lines
029:    getline(fs, s);
030:    getline(fs, s);
031:
032:    // Read model dimension
033:    fs >> mod_dim[0];
034:    fs >> mod_dim[1];
035:    fs >> mod_dim[2];
036:
037:    int mod_size = mod_dim[0]*mod_dim[1]*mod_dim[2];
038:    unsigned short int *A = new unsigned short int[mod_size];
039:
040:    // Skip two lines
041:    getline(fs, s);
042:    getline(fs, s);
043:
044:    for(int z=0; z<mod_dim[2]; z++)
045:      for(int y=0; y<mod_dim[1]; y++)
046:        for(int x=0; x<mod_dim[0]; x++)
047:          {
048:            fs >> A[z*mod_dim[1]*mod_dim[0]+y*mod_dim[0]+x];
049:          }
050:
051:    return A;
052:  }
053:
054:  void time_start(int rank)
055:  {
056:    // Synchronize processes
057:    // and start the timer
058:    //
059:    MPI_Barrier(MPI_COMM_WORLD);
060:    if(rank == 0)
061:      {
062:        time(&start_t);
063:      }
064:  }
065:
066:  void time_stop(int rank, const string &label)
067:  {
068:    // Synchronize processes, stop the timer
069:    // and print final time and label
070:    //
071:    MPI_Barrier(MPI_COMM_WORLD);
072:    if(rank == 0)
073:      {
074:        time(&stop_t);
075:        printf("%s:\t%f\n", label.c_str(), difftime(stop_t, start_t));
076:      }
077:  }
078:
079:  int random_walk(unsigned short int *A,
080:                  unsigned int x, unsigned int y, unsigned int z,
081:                  int dx, int dy, int dz)
082:  {
```

```
083:    int steps = 0, dir, axis;
084:    bool died = false;
085:
086:    // Release one walkers if (x,y,z) is a non-solid point
087:    // and follow the walker until it dies
088:    if(!A[z*dy*dx+y*dx+x])
089:      while(!died)
090:        {
091:          dir = (rand() % 2) * 2 - 1;
092:          axis = rand() % 3;
093:          switch(axis)
094:            {
095:            case 0:
096:              x = (x + dir) % dx;
097:              break;
098:            case 1:
099:              y = (y + dir) % dy;
100:             break;
101:           case 2:
102:             z = (z + dir) % dz;
103:             break;
104:           }
105:         // Check if walker hits wall
106:         if(A[z*dy*dx+y*dx+x])
107:           {
108:             // Check if walker dies
109:             if(rand() < RAND_MAX * MORTALITY)
110:               died = true;
111:             else // If the walker dont die it is bounced back,
112:               switch(axis)  // return it to previous location
113:                 {
114:                 case 0:
115:                   x = (x + dx - dir) % dx;
116:                   break;
117:                 case 1:
118:                   y = (y + dy - dir) % dy;
119:                   break;
120:                 case 2:
121:                   z = (z + dz - dir) % dz;
122:                   break;
123:                 }
124:           }
125:         else
126:           steps++;
127:       }
128:   return steps;
129: }
130:
131: int main(int argc, char *argv[])
132: {
133:   int rank;
134:   int procs;
135:
136:   // Size of cube grid
137:   int cube_dim[NDIMS] = {0};
138:   int cube_wrap[NDIMS] = {1};
139:
140:   // Position in cube
141:   int cube_pos[NDIMS];
142:
143:   // Cube communicator
144:   MPI_Comm cube_comm;
```

```
145:
146:    // Size of model
147:    int mod_dim[NDIMS];
148:
149:    // Model
150:    unsigned short int *A;
151:
152:    // Initialize MPI
153:    MPI_Init(&argc, &argv);
154:    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
155:    MPI_Comm_size(MPI_COMM_WORLD, &procs);
156:
157:    // Check if model is provided, if not
158:    // exit and write "Usage:" message
159:    if(argc < 2)
160:      {
161:        if(rank == 0)
162:          fprintf(stderr, "Usage: %s model\n", argv[0]);
163:        MPI_Exit(-1);
164:      }
165:
166:    // Allign processes to grid
167:    MPI_Dims_create(procs, NDIMS, cube_dim);
168:    // if(rank == 0)
169:    //   {
170:    //     fprintf(stdout, "Processes arranged in grid of size %dx%dx%d\n",
171:    //          cube_dim[0], cube_dim[1], cube_dim[2]);
172:    //   }
173:    // and create cubic communicator
174:    MPI_Cart_create(MPI_COMM_WORLD, NDIMS, cube_dim, cube_wrap, 1, &cube_comm);
175:    // and find position in grid
176:    MPI_Cart_coords(cube_comm, rank, NDIMS, cube_pos);
177:
178:    // Place a seed for the random function
179:    srand(time(0)+rank);
180:
181:    // Start timing of I/O
182:    time_start(rank);
183:
184:    // Only P0: Read and distribute model
185:    if(rank == 0)
186:      {
187:        A = read_model(argv[1], mod_dim);
188:      }
189:
190:    MPI_Bcast(mod_dim, NDIMS, MPI_INT, 0, MPI_COMM_WORLD);
191:    int mod_size = mod_dim[0]*mod_dim[1]*mod_dim[2];
192:
193:    if(rank != 0)
194:      {
195:        A = new unsigned short int[mod_size];
196:      }
197:
198:    // Receive model from P0
199:    MPI_Bcast(A, mod_size, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);
200:    if(rank == 0)
201:      printf("Problem size:\t%dx%dx%d\n", mod_dim[0], mod_dim[1], mod_dim[2]);
202:
203:    // Stop I/O timer, and write time used
204:    time_stop(rank, "I/O time");
205:
206:    // Time the total random walk duration
```

```
207:    time_start(rank);
208:
209:    // Do the walking
210:    unsigned int x1 = cube_pos[0]       * mod_dim[0] / cube_dim[0];
211:    unsigned int x2 = (cube_pos[0] + 1) * mod_dim[0] / cube_dim[0];
212:    unsigned int y1 = cube_pos[1]       * mod_dim[1] / cube_dim[1];
213:    unsigned int y2 = (cube_pos[1] + 1) * mod_dim[1] / cube_dim[1];
214:    unsigned int z1 = cube_pos[2]       * mod_dim[2] / cube_dim[2];
215:    unsigned int z2 = (cube_pos[2] + 1) * mod_dim[2] / cube_dim[2];
216:    for(unsigned int z=z1; z<z2; z++)
217:      for(unsigned int y=y1; y<y2; y++)
218:        for(unsigned int x=x1; x<x2; x++)
219:          random_walk(A, x, y, z, mod_dim[0], mod_dim[1], mod_dim[2]);
220:
221:    // Stop the random walk timer, and print result
222:    time_stop(rank, "Walk time");
223:
224:    MPI_Exit(0);
225:  }
```

A.3 Fully Parallel

```
001:  #include <cstdlib>
002:  #include <fstream>
003:  #include <string>
004:  #include <unistd.h>
005:  #include "mpi.h"
006:
007:  #define NDIMS 3
008:  #define MORTALITY 0.1
009:
010:  using namespace std;
011:
012:  void random_walk(signed short int *, int, int, int, int,
013:                   int *, int *, int, MPI_Comm);
014:
015:  MPI_Status status;
016:  MPI_Request request;
017:  int flag;
018:  time_t start_t, stop_t;
019:  int problem_size = 0;
020:  int leaving = 0;
021:
022:  void MPI_Exit(int e)
023:  {
024:    MPI_Finalize();
025:    exit(e);
026:  }
027:
028:  void process_incomming_walkers(signed short int *A, int *proc_dim,
029:                                 int *cube_dim, int rank, MPI_Comm cube_comm)
030:  {
031:    // Look for walkers entering the process space
032:    // and start a random walk from the entering point
033:    //
034:    int source;
035:    int sourcev[NDIMS] = {0};
036:    int w_data[4];
037:    MPI_Cart_coords(cube_comm, rank, NDIMS, sourcev);
```

```
038:
039:    // Loop until no walkers is found waiting
040:    bool finished;
041:    do
042:      {
043:        finished = true;
044:        // Look in all six directions
045:        for(int axis = 0; axis <= 3; axis++)
046:          for(int dir = -1; axis <= 1; axis+=2)
047:            {
048:              sourcev[axis] += dir;
049:              // Find id of process
050:              MPI_Cart_rank(cube_comm, sourcev, &source);
051:              sourcev[axis] -= dir;
052:              // Check if any walkers are waiting
053:              MPI_Iprobe(source, 0, cube_comm, &flag, &status);
054:              if(flag)
055:                {
056:                  // If walkers are waiting, receive data form
057:                  // neighbour pocess and let the walker
058:                  // continue its journy in this process
059:                  MPI_Irecv(w_data, 4, MPI_INT, source, 0, cube_comm, &request);
060:                  random_walk(A, w_data[0], w_data[1], w_data[2], w_data[3],
061:                              proc_dim, cube_dim, rank, cube_comm);
062:                  finished = false;
063:                }
064:            }
065:      }
066:    while(!finished);
067:  }
068:
069:  void random_walk(signed short int *A, int x, int y, int z, int s,
070:                   int *proc_dim, int *cube_dim, int rank, MPI_Comm cube_comm)
071:  {
072:    short int dir, axis;
073:    int w_data[4];
074:    int destv[NDIMS] = {0};
075:    int dest;
076:    int dx = proc_dim[0];
077:    int dy = proc_dim[1];
078:    int dz = proc_dim[2];
079:    int xn = x;
080:    int yn = y;
081:    int zn = z;
082:    int steps = s;
083:    bool sent = false;
084:    bool died = false;
085:
086:    // Release one walkers if (x,y,z) is a non-solid point
087:    // and follow the walker until it dies or is sent
088:    // to an neighbour
089:    if(!A[zn*dy*dx+yn*dx+xn])
090:      while(!sent and !died)
091:        {
092:          dir = (rand() % 2) * 2 - 1;
093:          steps++;
094:          axis = rand() % 3;
095:          switch(axis)
096:            {
097:            case 0:
098:              xn = xn + dir;
099:              if(xn < 0 || xn >= dx)
```

```
100:                sent = true;
101:              break;
102:            case 1:
103:              yn = yn + dir;
104:              if(yn < 0 || yn >= dy)
105:                sent = true;
106:              break;
107:            case 2:
108:              zn = zn + dir;
109:              if(zn < 0 || zn >= dz)
110:                sent = true;
111:              break;
112:            }
113:          if(sent) // Send walker to neighbour process
114:            {
115:              MPI_Cart_coords(cube_comm, rank, NDIMS, destv);
116:              destv[axis] = (destv[axis] + cube_dim[axis] + dir) % cube_dim[axis];
117:              MPI_Cart_rank(cube_comm, destv, &dest);
118:              w_data[0] = (xn + dx) % dx;
119:              w_data[1] = (yn + dy) % dy;
120:              w_data[2] = (zn + dz) % dz;
121:              w_data[3] = steps;
122:              process_incomming_walkers(A, proc_dim, cube_dim, rank, cube_comm);
123:              MPI_Isend(w_data, 4, MPI_INT, dest, 0, cube_comm, &request);
124:              leaving++;
125:              process_incomming_walkers(A, proc_dim, cube_dim, rank, cube_comm);
126:            }
127:          // Check if walker hits wall
128:          if(!sent && A[zn*dy*dx+yn*dx+xn])
129:            {
130:              // Check if walker dies
131:              if(rand() < RAND_MAX * MORTALITY)
132:                died = true;
133:              else // If the walker dont die, return it to previous location
134:                switch(axis)
135:                  {
136:                  case 0:
137:                    xn = (xn + dx - dir) % dx;
138:                    break;
139:                  case 1:
140:                    yn = (yn + dy - dir) % dy;
141:                    break;
142:                  case 2:
143:                    zn = (zn + dz - dir) % dz;
144:                    break;
145:                  }
146:            }
147:        }
148:  }
149:
150:  signed short int *read_model(const char* file_name, int p, int* proc_dim,
151:                               int* cube_dim, int procs, MPI_Comm cube_comm)
152:  {
153:    // Read part of model from file
154:    //
155:    ifstream fs(file_name);
156:    string s;
157:
158:    // Skip two lines
159:    getline(fs, s);
160:    getline(fs, s);
161:
```

```
162:    // Read model dimension
163:    int dx, dy, dz;
164:    fs >> dx;
165:    fs >> dy;
166:    fs >> dz;
167:    if(p == 0)
168:      printf("Problem size:\t%dx%dx%d\n", dx, dy, dz);
169:    problem_size = dx * dy * dz;
170:    proc_dim[0] = dx / cube_dim[0];
171:    proc_dim[1] = dy / cube_dim[1];
172:    proc_dim[2] = dz / cube_dim[2];
173:
174:    int proc_size = proc_dim[0]*proc_dim[1]*proc_dim[2];
175:    signed short int *A = new signed short int[proc_size];
176:
177:    int dest[NDIMS] = {0};
178:    signed short int v;
179:
180:    // Seek to start of file and skip four lines
181:    fs.seekg(0, ios::beg);
182:    getline(fs, s);getline(fs, s);getline(fs, s);getline(fs, s);
183:    MPI_Cart_coords(cube_comm, p, NDIMS, dest);
184:    for(int z=0; z<dz; z++)
185:      for(int y=0; y<dy; y++)
186:        for(int x=0; x<dx; x++)
187:          {
188:            fs >> v;
189:            // Map data from file to correct location in A
190:            if(dest[0]*proc_dim[0] <= x && x < (dest[0]+1)*proc_dim[0]
191:               &&
192:               dest[1]*proc_dim[1] <= y && y < (dest[1]+1)*proc_dim[1]
193:               &&
194:               dest[2]*proc_dim[2] <= z && z < (dest[2]+1)*proc_dim[2])
195:              {
196:                A[(z%proc_dim[2])*proc_dim[1]*proc_dim[0] +
197:                  (y%proc_dim[1])*proc_dim[0] +
198:                  (x%proc_dim[0])]
199:                  = v;
200:              }
201:          }
202:    return A;
203:  }
204:
205:  void time_start(int rank)
206:  {
207:    // Synchronize processes
208:    // and start the timer
209:    //
210:    MPI_Barrier(MPI_COMM_WORLD);
211:    if(rank == 0)
212:      {
213:        time(&start_t);
214:      }
215:  }
216:
217:  void time_stop(int rank, const char* label)
218:  {
219:    // Synchronize processes, stop the timer
220:    // and print final time and label
221:    //
222:    MPI_Barrier(MPI_COMM_WORLD);
223:    if(rank == 0)
```

```
224:     {
225:       time(&stop_t);
226:       printf("%s:\t%f\n", label, difftime(stop_t, start_t));
227:     }
228: }
229:
230: int main(int argc, char* argv[])
231: {
232:   // Read the model into an array
233:   int rank;
234:   int procs;
235:   int cube_dim[NDIMS] = {0};
236:   int cube_wrap[NDIMS] = {1};
237:   MPI_Comm cube_comm;
238:
239:   // Initialize MPI
240:   MPI_Init(&argc, &argv);
241:   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
242:   MPI_Comm_size(MPI_COMM_WORLD, &procs);
243:
244:   // Check if model is provided, if not
245:   // exit and write "Usage:" message
246:   if(argc < 2)
247:     {
248:       if(rank == 0)
249:         fprintf(stderr, "Usage: %s model\n", argv[0]);
250:       MPI_Exit(-1);
251:     }
252:
253:   // Allign processes to grid
254:   MPI_Dims_create(procs, NDIMS, cube_dim);
255:   // if(rank == 0)
256:   //   {
257:   //     fprintf(stdout, "Processes aranged in grid of size %dx%dx%d\n",
258:   //          cube_dim[0], cube_dim[1], cube_dim[2]);
259:   //   }
260:   // and create cubic communicator
261:   MPI_Cart_create(MPI_COMM_WORLD, NDIMS, cube_dim, cube_wrap, 1, &cube_comm);
262:
263:   // Place a seed for the random function
264:   srand(time(0)+rank);
265:
266:   // Size of sub model
267:   int proc_dim[NDIMS];
268:
269:   signed short int *A;
270:
271:   // Start timing of I/O
272:   time_start(rank);
273:
274:   // Each process: read part of model
275:   A = read_model(argv[1], rank, proc_dim, cube_dim, procs, cube_comm);
276:
277:   // Stop I/O timer, and write time used
278:   time_stop(rank, "I/O time");
279:
280:   // Time the total random walk duration
281:   time_start(rank);
282:
283:   // Do the walking
284:   unsigned int dx = proc_dim[0];
285:   unsigned int dy = proc_dim[1];
```

```
286:    unsigned int dz = proc_dim[2];
287:    for(unsigned int z=0; z<dz; z++)
288:      for(unsigned int y=0; y<dy; y++)
289:        for(unsigned int x=0; x<dx; x++)
290:          random_walk(A, x, y, z, 0, proc_dim, cube_dim, rank, cube_comm);
291:
292:    // Stop the random walk timer, and print result
293:    time_stop(rank, "Walker time");
294:
295:    // Sum the number of walkers leaving each process
296:    // and print the total to screen
297:    int leaving_tot = 0;
298:    MPI_Reduce(&leaving, &leaving_tot, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
299:    if(rank == 0)
300:      {
301:        printf("Leaving:\t%d/%d -> %.2f%s\n",
302:               leaving_tot, problem_size,
303:               100.0 * leaving_tot / problem_size, "%");
304:      }
305:    MPI_Exit(0);
306:  }
```

A.4 Fully Parallel With Overlapping Boundaries

```
001:  #include <cstdlib>
002:  #include <fstream>
003:  #include <string>
004:  #include <unistd.h>
005:  #include "mpi.h"
006:
007:  #define NDIMS 3
008:  #define MORTALITY 0.1
009:
010:  using namespace std;
011:
012:  void random_walk(signed short int *, int, int, int, int,
013:                   int *, int *, int, MPI_Comm);
014:
015:  MPI_Status status;
016:  MPI_Request request;
017:  int flag;
018:  time_t start_t, stop_t;
019:
020:  // The size of the overlap
021:  int overlap = 0;
022:
023:  int problem_size = 0;
024:  int leaving = 0;
025:
026:  void MPI_Exit(int e)
027:  {
028:    MPI_Finalize();
029:    exit(e);
030:  }
031:
032:  void process_incomming_walkers(signed short int *A, int *proc_dim,
033:                                 int *cube_dim, int rank, MPI_Comm cube_comm)
034:  {
035:    // Look for walkers entering the process space
```

```
036:    // and start a random walk from the entering point
037:    //
038:    int source;
039:    int sourcev[NDIMS] = {0};
040:    int w_data[4];
041:    MPI_Cart_coords(cube_comm, rank, NDIMS, sourcev);
042:
043:    // Loop until no walkers is found waiting
044:    bool finished;
045:    do
046:      {
047:        finished = true;
048:        // Look in all six directions
049:        for(int axis = 0; axis <= 3; axis++)
050:          for(int dir = -1; axis <= 1; axis+=2)
051:            {
052:              sourcev[axis] += dir;
053:              // Find id of process
054:              MPI_Cart_rank(cube_comm, sourcev, &source);
055:              sourcev[axis] -= dir;
056:              // Check if any walkers are waiting
057:              MPI_Iprobe(source, 0, cube_comm, &flag, &status);
058:              if(flag)
059:                {
060:                  // If walkers are waiting, receive data form
061:                  // neighbour pocess and let the walker
062:                  // continue its journy in this process
063:                  MPI_Irecv(w_data, 4, MPI_INT, source, 0, cube_comm, &request);
064:                  random_walk(A, w_data[0], w_data[1], w_data[2], w_data[3],
065:                              proc_dim, cube_dim, rank, cube_comm);
066:                  finished = false;
067:                }
068:            }
069:      }
070:    while(!finished);
071:  }
072:
073:  void random_walk(signed short int *A, int x, int y, int z, int s,
074:                   int *proc_dim, int *cube_dim, int rank, MPI_Comm cube_comm)
075:  {
076:    short int dir, axis;
077:    int w_data[4];
078:    int destv[NDIMS] = {0};
079:    int dest;
080:    int dx = proc_dim[0];
081:    int dy = proc_dim[1];
082:    int dz = proc_dim[2];
083:    int xn = x;
084:    int yn = y;
085:    int zn = z;
086:    int steps = s;
087:    bool sent = false;
088:    bool died = false;
089:
090:    // Release one walkers if (x,y,z) is a non-solid point
091:    // and follow the walker until it dies or is sent
092:    // to an neighbour
093:    if(!A[zn*dy*dx+yn*dx+xn])
094:      while(!sent and !died)
095:        {
096:          dir = (rand() % 2) * 2 - 1;
097:          steps++;
```

```
098:          axis = rand() % 3;
099:          switch(axis)
100:            {
101:            case 0:
102:              xn = xn + dir;
103:              if(xn < 0 || xn >= dx)
104:                sent = true;
105:              break;
106:            case 1:
107:              yn = yn + dir;
108:              if(yn < 0 || yn >= dy)
109:                sent = true;
110:              break;
111:            case 2:
112:              zn = zn + dir;
113:              if(zn < 0 || zn >= dz)
114:                sent = true;
115:              break;
116:            }
117:          if(sent) // Send walker to neighbour process
118:            {
119:              MPI_Cart_coords(cube_comm, rank, NDIMS, destv);
120:              destv[axis] = (destv[axis] + cube_dim[axis] + dir)
121:                            % cube_dim[axis];
122:              MPI_Cart_rank(cube_comm, destv, &dest);
123:              w_data[0] = (xn + dx + 2 * dir * overlap) % dx;
124:              w_data[1] = (yn + dy + 2 * dir * overlap) % dy;
125:              w_data[2] = (zn + dz + 2 * dir * overlap) % dz;
126:              w_data[3] = steps;
127:              process_incomming_walkers(A, proc_dim, cube_dim, rank, cube_comm);
128:              MPI_Isend(w_data, 4, MPI_INT, dest, 0, cube_comm, &request);
129:              leaving++;
130:              process_incomming_walkers(A, proc_dim, cube_dim, rank, cube_comm);
131:            }
132:          // Check if walker hits wall
133:          if(!sent && A[zn*dy*dx+yn*dx+xn])
134:            {
135:              // Check if walker dies
136:              if(rand() < RAND_MAX * MORTALITY)
137:                died = true;
138:              else // If the walker dont die, return it to previous location
139:                switch(axis)
140:                  {
141:                  case 0:
142:                    xn = (xn + dx - dir) % dx;
143:                    break;
144:                  case 1:
145:                    yn = (yn + dy - dir) % dy;
146:                    break;
147:                  case 2:
148:                    zn = (zn + dz - dir) % dz;
149:                    break;
150:                  }
151:            }
152:        }
153:    }
154:
155:    signed short int *read_model(const char* file_name, int rank, int* proc_dim,
156:                                 int* cube_dim, int procs, MPI_Comm cube_comm)
157:    {
158:      // Read part of model from file
159:      //
```

```
160:    ifstream fs(file_name);
161:    string s;
162:
163:    // Skip two lines
164:    getline(fs, s);
165:    getline(fs, s);
166:
167:    // Read model dimension
168:    int dx, dy, dz;
169:    fs >> dx;
170:    fs >> dy;
171:    fs >> dz;
172:    if(rank == 0)
173:      printf("Problem size:\t%dx%dx%d\n", dx, dy, dz);
174:    problem_size = dx * dy * dz;
175:    int pdx = dx / cube_dim[0];
176:    int pdy = dy / cube_dim[1];
177:    int pdz = dz / cube_dim[2];
178:    proc_dim[0] = pdx + overlap * 2;
179:    proc_dim[1] = pdy + overlap * 2;
180:    proc_dim[2] = pdz + overlap * 2;
181:
182:    int proc_size = proc_dim[0]*proc_dim[1]*proc_dim[2];
183:    signed short int *A = new signed short int[proc_size];
184:
185:    int dest[NDIMS] = {0};
186:    signed short int v;
187:
188:    int p = rank;
189:    // Seek to start of file and skip four lines
190:    fs.seekg(0, ios::beg);
191:    getline(fs, s);getline(fs, s);getline(fs, s);getline(fs, s);
192:    MPI_Cart_coords(cube_comm, p, NDIMS, dest);
193:    for(int z=0; z<dz; z++)
194:      for(int y=0; y<dy; y++)
195:        for(int x=0; x<dx; x++)
196:          {
197:            fs >> v;
198:            //
199:            // Map data from file to correct location in A,
200:            // including the shifting of data due to the overlap
201:            //
202:            if((((dest[0]*pdx - overlap) <= x
203:                 &&
204:                 x < ((dest[0]+1)*pdx + overlap))
205:                ||
206:                ((dest[0]*pdx + dx - overlap) <= x
207:                 &&
208:                 x < ((dest[0]+1)*pdx + dx + overlap))
209:                ||
210:                ((dest[0]*pdx - dx - overlap) <= x
211:                 &&
212:                 x < ((dest[0]+1)*pdx - dx + overlap)))
213:               &&
214:               (((dest[1]*pdy - overlap) <= y
215:                 &&
216:                 y < ((dest[1]+1)*pdy + overlap))
217:                ||
218:                ((dest[1]*pdy + dy - overlap) <= y
219:                 &&
220:                 y < ((dest[1]+1)*pdy + dy + overlap))
221:                ||
```

```
222:                ((dest[1]*pdy - dy - overlap) <= y
223:                 &&
224:                 y < ((dest[1]+1)*pdy - dy + overlap)))
225:               &&
226:               (((dest[2]*pdz - overlap) <= z
227:                 &&
228:                 z < ((dest[2]+1)*pdz + overlap))
229:                ||
230:                ((dest[2]*pdz + dz - overlap) <= z
231:                 &&
232:                 z < ((dest[2]+1)*pdz + dz + overlap))
233:                ||
234:                ((dest[2]*pdz - dz - overlap) <= z
235:                 &&
236:                 z < ((dest[2]+1)*pdz - dz + overlap))))
237:              {
238:                A[((z + dz - (dest[2]*pdz - overlap)) % dz) * proc_dim[1]
239:                                                           * proc_dim[0] +
240:                  ((y + dy - (dest[1]*pdy - overlap)) % dy) * proc_dim[0] +
241:                  ((x + dx - (dest[0]*pdx - overlap)) % dx)]
242:                  = v;
243:              }
244:          }
245:    return A;
246:  }
247:
248:  void time_start(int rank)
249:  {
250:    // Synchronize processes
251:    // and start the timer
252:    //
253:    MPI_Barrier(MPI_COMM_WORLD);
254:    if(rank == 0)
255:      {
256:        time(&start_t);
257:      }
258:  }
259:
260:  void time_stop(int rank, const char* label)
261:  {
262:    // Synchronize processes, stop the timer
263:    // and print final time and label
264:    //
265:    MPI_Barrier(MPI_COMM_WORLD);
266:    if(rank == 0)
267:      {
268:        time(&stop_t);
269:        printf("%s:\t%f\n", label, difftime(stop_t, start_t));
270:      }
271:  }
272:
273:  int main(int argc, char* argv[])
274:  {
275:    // Read the model into an array
276:    int rank;
277:    int procs;
278:    int cube_dim[NDIMS] = {0};
279:    int cube_wrap[NDIMS] = {1};
280:    MPI_Comm cube_comm;
281:
282:    MPI_Init(&argc, &argv);
283:    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
284:    MPI_Comm_size(MPI_COMM_WORLD, &procs);
285:
286:    // Check if model and overlap is provided,
287:    // if not exit and write "Usage:" message
288:    if(argc < 3)
289:      {
290:        if(rank == 0)
291:          fprintf(stderr, "Usage: %s model overlap\n", argv[0]);
292:        MPI_Exit(-1);
293:      }
294:
295:    // We need at least two processes,
296:    // exit if only one process is started
297:    if(procs < 2)
298:      {
299:        fprintf(stderr, "We need at least two processes\n");
300:        MPI_Exit(-1);
301:      }
302:
303:    // Allign processes to grid
304:    MPI_Dims_create(procs, NDIMS, cube_dim);
305:    // if(rank == 0)
306:    //   {
307:    //     fprintf(stdout, "Processes aranged in grid of size %dx%dx%d\n",
308:    //          cube_dim[0], cube_dim[1], cube_dim[2]);
309:    //   }
310:    // and create cubic communicator
311:    MPI_Cart_create(MPI_COMM_WORLD, NDIMS, cube_dim, cube_wrap, 1, &cube_comm);
312:
313:    // Place a seed for the random function
314:    srand(time(0)+rank);
315:
316:    // Read overlap, given on command line
317:    overlap = atoi(argv[2]);
318:    if(rank == 0)
319:      printf("Overlap:\t%d\n", overlap);
320:
321:    // Size of sub model
322:    int proc_dim[NDIMS];
323:    signed short int *A;
324:
325:    // Start timing of I/O
326:    time_start(rank);
327:
328:    // Each process: read part of model
329:    A = read_model(argv[1], rank, proc_dim, cube_dim, procs, cube_comm);
330:
331:    // Stop I/O timer, and write time used
332:    time_stop(rank, "I/O time");
333:
334:    // Time the total random walk duration
335:    time_start(rank);
336:
337:    // Do the walking
338:    unsigned int dx = proc_dim[0];
339:    unsigned int dy = proc_dim[1];
340:    unsigned int dz = proc_dim[2];
341:    for(unsigned int z=overlap; z<dz-overlap; z++)
342:      for(unsigned int y=overlap; y<dy-overlap; y++)
343:        for(unsigned int x=overlap; x<dx-overlap; x++)
344:          random_walk(A, x, y, z, 0, proc_dim, cube_dim, rank, cube_comm);
345:
```

```
346:    // Stop the random walk timer, and print result
347:    time_stop(rank, "Walker time");
348:
349:    // Sum the number of walkers leaving each process
350:    // and print the total to screen
351:    int leaving_tot = 0;
352:    MPI_Reduce(&leaving, &leaving_tot, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
353:    if(rank == 0)
354:      {
355:        printf("Leaving:\t%d/%d -> %.2f%s\n",
356:               leaving_tot, problem_size,
357:               100.0 * leaving_tot / problem_size, "%");
358:      }
359:    MPI_Exit(0);
360:  }
```

Appendix B

Test results

B.1 Communication

The complete tables of results from the communication testing is given in Table B.1, Table B.2 and Table B.3. The maximum overlap is limited by the size of the data set. The overlap has an upper limit given by

$$\text{overlap} < \frac{1}{2}\sqrt[3]{\frac{N}{p}} \tag{B.1}$$

where N is the problem size and p is the number of processes used. Any overlap larger than this value would in fact cover the entire data set, and testing has only been conducted for overlaps smaller than this limit.

Table B.1: Communication between processes when executing program A.4 on a system with eight processes.

	Problem size			
Overlap	80^3 (512000)	160^3 (4096000)	300^3 (27000000)	600^3 (216000000)
0	48013 (9.3775%)	77312 (1.8875%)	795978 (2.9481%)	5473463 (2.5340%)
1	25021 (4.8869%)	61265 (1.4957%)	411862 (1.5254%)	3143646 (1.4554%)
2	18170 (3.5488%)	49049 (1.1975%)	306957 (1.1369%)	2456120 (1.1371%)
3	13303 (2.5982%)	39124 (0.9552%)	230573 (0.8540%)	1934149 (0.8954%)
4	9694 (1.8934%)	31155 (0.7606%)	174291 (0.6455%)	1528358 (0.7076%)
5	7036 (1.3742%)	24872 (0.6072%)	132803 (0.4919%)	1216153 (0.5630%)
6	5028 (0.9820%)	20149 (0.4919%)	100624 (0.3727%)	970531 (0.4493%)
7	3636 (0.7102%)	16611 (0.4055%)	77006 (0.2852%)	776844 (0.3597%)
8	2676 (0.5227%)	13889 (0.3391%)	58807 (0.2178%)	624358 (0.2891%)
9	1869 (0.3650%)	11355 (0.2772%)	44403 (0.1645%)	500291 (0.2316%)
10	1446 (0.2824%)	9777 (0.2387%)	34182 (0.1266%)	403108 (0.1866%)
11	1027 (0.2006%)	8281 (0.2022%)	25891 (0.0959%)	325072 (0.1505%)
12	744 (0.1453%)	7332 (0.1790%)	19711 (0.0730%)	262343 (0.1215%)
13	627 (0.1225%)	6036 (0.1474%)	14970 (0.0554%)	211962 (0.0981%)
14	459 (0.0896%)	5300 (0.1294%)	11363 (0.0421%)	171329 (0.0793%)
15	325 (0.0635%)	4671 (0.1140%)	8731 (0.0323%)	138094 (0.0639%)
16	250 (0.0488%)	3994 (0.0975%)	6587 (0.0244%)	113192 (0.0524%)
17	184 (0.0359%)	3692 (0.0901%)	5072 (0.0188%)	91534 (0.0424%)
18	124 (0.0242%)	3317 (0.0810%)	3783 (0.0140%)	74490 (0.0345%)
19	127 (0.0248%)	2830 (0.0691%)	2900 (0.0107%)	61191 (0.0283%)
20		2604 (0.0636%)	2312 (0.0086%)	50086 (0.0232%)
21		2455 (0.0599%)	1732 (0.0064%)	41712 (0.0193%)
22		2163 (0.0528%)	1308 (0.0048%)	34224 (0.0158%)
23		1858 (0.0454%)	978 (0.0036%)	27999 (0.0130%)
24		1700 (0.0415%)	751 (0.0028%)	23281 (0.0108%)
25		1316 (0.0321%)	585 (0.0022%)	19384 (0.0090%)
26		1051 (0.0257%)	453 (0.0017%)	16134 (0.0075%)
27		883 (0.0216%)	355 (0.0013%)	13212 (0.0061%)
28		650 (0.0159%)	308 (0.0011%)	11220 (0.0052%)
29		493 (0.0120%)	208 (0.0008%)	9218 (0.0043%)
30		376 (0.0092%)	154 (0.0006%)	7791 (0.0036%)
31		281 (0.0069%)	130 (0.0005%)	6616 (0.0031%)
32		214 (0.0052%)	94 (0.0003%)	5490 (0.0025%)
33		172 (0.0042%)	68 (0.0003%)	4758 (0.0022%)
34		122 (0.0030%)	51 (0.0002%)	4011 (0.0019%)
35		105 (0.0026%)	44 (0.0002%)	3557 (0.0016%)
36		58 (0.0014%)	28 (0.0001%)	3076 (0.0014%)
37		63 (0.0015%)	33 (0.0001%)	2801 (0.0013%)
38		40 (0.0010%)	31 (0.0001%)	2281 (0.0011%)
39		30 (0.0007%)	9 (0.0000%)	2128 (0.0010%)
40			14 (0.0000%)	1852 (0.0009%)

Table B.2: Communication between processes when executing program A.4 on a system with 64 processes.

	Problem size			
Overlap	80^3 (512000)	160^3 (4096000)	300^3 (27000000)	600^3 (216000000)
0	76750 (14.9902%)	154199 (3.7646%)	1465915 (5.4293%)	10590048 (4.9028%)
1	52347 (10.2240%)	122181 (2.9829%)	936832 (3.4697%)	7317830 (3.3879%)
2	40019 (7.8162%)	97999 (2.3926%)	704102 (2.6078%)	5814601 (2.6919%)
3	30436 (5.9445%)	77270 (1.8865%)	534411 (1.9793%)	4636341 (2.1465%)
4	22813 (4.4557%)	61739 (1.5073%)	405402 (1.5015%)	3716090 (1.7204%)
5	16863 (3.2936%)	49689 (1.2131%)	307285 (1.1381%)	2985679 (1.3823%)
6	12595 (2.4600%)	40126 (0.9796%)	234534 (0.8686%)	2405027 (1.1134%)
7	9213 (1.7994%)	33325 (0.8136%)	178889 (0.6626%)	1943981 (0.9000%)
8	6847 (1.3373%)	27526 (0.6720%)	135875 (0.5032%)	1576867 (0.7300%)
9	5210 (1.0176%)	22829 (0.5573%)	103362 (0.3828%)	1282151 (0.5936%)
10		19293 (0.4710%)	78527 (0.2908%)	1047001 (0.4847%)
11		16670 (0.4070%)	59407 (0.2200%)	856175 (0.3964%)
12		14148 (0.3454%)	45033 (0.1668%)	703306 (0.3256%)
13		12115 (0.2958%)	34134 (0.1264%)	579859 (0.2685%)
14		10495 (0.2562%)	26121 (0.0967%)	478462 (0.2215%)
15		9195 (0.2245%)	20072 (0.0743%)	397660 (0.1841%)
16		8149 (0.1990%)	15147 (0.0561%)	331753 (0.1536%)
17		7257 (0.1772%)	11307 (0.0419%)	277483 (0.1285%)
18		6547 (0.1598%)	8687 (0.0322%)	232373 (0.1076%)
19		5682 (0.1387%)	6854 (0.0254%)	196749 (0.0911%)
20			5229 (0.0194%)	166742 (0.0772%)
21			3858 (0.0143%)	141960 (0.0657%)
22			2944 (0.0109%)	121591 (0.0563%)
23			2291 (0.0085%)	104050 (0.0482%)
24			1738 (0.0064%)	89600 (0.0415%)
25			1329 (0.0049%)	77969 (0.0361%)
26			927 (0.0034%)	67545 (0.0313%)
27			762 (0.0028%)	58745 (0.0272%)
28			570 (0.0021%)	50964 (0.0236%)
29			457 (0.0017%)	44292 (0.0205%)
30			309 (0.0011%)	38764 (0.0179%)
31			305 (0.0011%)	33850 (0.0157%)
32			193 (0.0007%)	29658 (0.0137%)
33			160 (0.0006%)	25836 (0.0120%)
34			125 (0.0005%)	23008 (0.0107%)
35			96 (0.0004%)	20389 (0.0094%)
36			87 (0.0003%)	17661 (0.0082%)
37			53 (0.0002%)	15731 (0.0073%)
38				13977 (0.0065%)
39				12256 (0.0057%)
40				10707 (0.0050%)

Table B.3: Communication between processes when executing program A.4 on a system with 125 processes.

	Problem size			
Overlap	80^3 (512000)	160^3 (4096000)	300^3 (27000000)	600^3 (216000000)
0	83806 (16.3684%)	492083 (12.0137%)	1801414 (6.6719%)	13159339 (6.0923%)
1	60118 (11.7418%)	415830 (10.1521%)	1207739 (4.4731%)	9415959 (4.3592%)
2	46072 (8.9984%)	348271 (8.5027%)	919241 (3.4046%)	7488474 (3.4669%)
3	35706 (6.9738%)	288824 (7.0514%)	702770 (2.6029%)	5991320 (2.7738%)
4	27567 (5.3842%)	239079 (5.8369%)	535101 (1.9819%)	4802112 (2.2232%)
5	20793 (4.0611%)	199564 (4.8722%)	408751 (1.5139%)	3855379 (1.7849%)
6	15878 (3.1012%)	167198 (4.0820%)	312389 (1.1570%)	3108114 (1.4389%)
7	11992 (2.3422%)	138340 (3.3774%)	238099 (0.8818%)	2513961 (1.1639%)
8		114057 (2.7846%)	181967 (0.6740%)	2032523 (0.9410%)
9		93207 (2.2756%)	138903 (0.5145%)	1652877 (0.7652%)
10		75419 (1.8413%)	105503 (0.3908%)	
11		60550 (1.4783%)	81230 (0.3009%)	
12		48471 (1.1834%)	62371 (0.2310%)	
13		38783 (0.9469%)	47200 (0.1748%)	
14		31209 (0.7619%)	36064 (0.1336%)	
15		24941 (0.6089%)	27516 (0.1019%)	
16			21308 (0.0789%)	
17			16202 (0.0600%)	
18			12513 (0.0463%)	
19			9486 (0.0351%)	
20			7197 (0.0267%)	
21			5514 (0.0204%)	
22			4247 (0.0157%)	
23			3295 (0.0122%)	
24			2505 (0.0093%)	
25			1879 (0.0070%)	
26			1520 (0.0056%)	

References

[1] William P. Rothwell and H. J. Vinegar. Petrophysical applications of NMR imaging. *Appl. Opt.*, 1985.

[2] M. Leibig. Random walks and NMR measurements in porous media. *Journal of Physics A: Mathematical and General*, 26(14):3349–3367, 1993.

[3] Barry D. Hughes. *Random Walks and Random Environments: Random Walks*, volume 1. Oxford Science Publications, 1995.

[4] Louis Bachelier. *Théorie de la spéculation*. Gauthier-Villars, 1900.

[5] Burton Malkiel. *A Random Walk Down Wall Street*. W. W. Northon & Company, 1973.

[6] Albert Einstein. On the movement of small particles suspended in a stationary liquid demanded by the molecular-kinetic theory of heat. *Ann d. Phys*, 1905.

[7] J. Laurie Snell Charles M. Grinstead. *Introduction to Probability: Second Revised Edition*. American Mathematical Society, 1997.

[8] Pierre-Gilles de Gennes. *Scaling Concepts in Polymer Physics*. Cornell University Press, 1979.

[9] J. J. Tessier, K. J. Packer, J.-F. Thovert, and P. M. Adler. NMR measurements and numerical simulation of fluid transport in porous solids. *AIChE Journal*, 1997.

[10] Barry D. Hughes. *Random Walks and Random Environments: Random Environments*, volume 2. Oxford Science Publications, 1995.

[11] F. Comets and N. Yoshida. Brownian directed polymers in random environment, 2003.

[12] Gregory F. Lawler. *Intersections of Random Walks (Probability and its Applications)*. Birkhäuser Boston, 1996.

[13] Intel Corporation. Excerpts from A Conversation with Gordon Moore: Moore's Law.

[14] Ethan Mollick. Establishing Moore's Law. *IEEE Annals of the History of Computing*, 2006.

[15] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publisher, Inc, 1997.

[16] Michael Jerosch-herold and Hans Thomann. Permeability determination from NMR relaxation measurements for fluids in porous media, February 1995.

[17] Irwan Hidajat, Mohit Singh, Josh Cooper, and Kishore K. Mohanty. Permeability of Porous Media from Simulated NMR Response. *Transport in Porous Media*, 2002.

[18] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1985.

[19] Jim Handy. *The Cache Memory Book, Second Edition (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 1998.