Monte Carlo Simulation of Sintering on Multiprocessor Systems

Design, implementation and evaluation of microstructural storage and parallel execution for simulation of an atomic process

Jens R. Lind

Master of Science Thesis
Stockholm, Sweden 2004/2005

IMIT/LECS-2005-07

Monte Carlo Simulation of Sintering on Multiprocessor Systems

Design, implementation and evaluation of microstructural storage and parallel execution for simulation of an atomic process

Author: Jens R. Lind
Examiner: Vladimir Vlassov

Master of Science Thesis
Stockholm, Sweden 2004/2005

IMIT/LECS-2005-07

Abstract

As the availability and computational power of modern computer system increases, new fields of applications are made possible. One such field is the simulation of industrial processes. Simulating these processes allows for safer and cheaper research and development. But applications of real life simulations most often suffer great time and memory constraints. A metallurgy process called sintering, by which powders are formed into objects under high pressure and near melting point temperatures, has been accurately modelled as such a computer application based on the Monte Carlo technique.

The purpose of this master thesis project is to address the constraints of the sintering application and improve the memory and execution time performance. The latter is done by designing and implementing a parallel version of the application. The memory usage reduction is achieved by dividing the simulation data, resulting in fast and effective compression of locally sparse data. These subsets of the simulation data are then used as base for the parallelization strategy, allowing for them to be fairly independently scheduled among multiple processes.

An important aspect of this project was to improve the performance without making any changes to the underlying sintering algorithm, ensuring the simulation model remains accurate. This significant limitation and the design being based on two different, yet interlocked, fields of computer science, namely memory structure and parallelization, make this project an interesting and worthwhile master thesis.

Evaluations prove the parallel application to outperform the original version by a factor of ten while still maintaining the correctness of the underlying sintering simulation algorithm. The resulting application can simulate much larger models than the original version, but the time needed to complete simulations of any real life experiments is unfortunately still outside the scope of most multiprocessor systems.

Acknowledgements

First of all, I want to thank my supervisors Adam Postula and Peter Sutton at UQ for allowing me to come all the way to Australia. It has truly been a great journey and an endless string of memorable experiences. In regard to all the administrative work behind an overseas arrangement I want to thank Patrik Gärdenäs at KTH for his valuable help.

My supervisors deserve a second thank for all their support, feedback, comments and also for allowing me fairly free reins making me feel that this is my own project. My examiner Vladimir Vlassov at KTH I want to thank especially for taking time in the middle of the summer to sketch up a rough outline of a project plan and report contents. Those outlines proved to be crucial for the successful completion of this master thesis.

Furthermore, I want to thank John Xue at UQ and my father, Lennart Lind, for their comments on my project and especially the report.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

This report is the main document for my master thesis project at the School of Information Technology and Electrical Engineering (ITEE) which is a department within the University of Queensland (UQ). UQ is one of the major universities in Brisbane, Australia. The project is a mandatory, and final, step towards my master degree in Computer Science and Engineering at the Royal Institute of Technology (KTH).

This report deals with the simulation of the sintering process. Sintering is a metallurgical process forming objects of metal or ceramic powders by applying high pressure and temperatures below the powder's melting point over a long time. Accurate modelling of this process can be achieved by simulating movements of single atoms with the Monte Carlo type algorithm. Such a model has already been developed by Dr Roberta Sutton and Professor Graham Schaffer at the UQ Department of Mining, Minerals and Materials Engineering (MINMET) [29] but suffers from unacceptable run times if executed on a single workstation. A hardware accelerator to speed up computations is being designed by Dr Peter Sutton and Dr Adam Postula at ITEE. Some of their early work was done by Adam Postula together with David Abramson and Paul Logothetis [23].

This master thesis project investigates mapping of the already developed sintering simulation algorithm on a multiprocessor system and analyzes the achieved speedup. The memory model of the earlier algorithm is revised to allow for much larger data. The small datasets that the original version currently can handle is not enough to simulate real life sintering experiments, and as such can only be used for approximations.

1.1 Motivation

The current version of the sintering simulation algorithm is sequential and can therefore only be executed on a single processor. It suffers from very high simulation run times, even on small problems. The memory model of the algorithm does not scale well either and would not be reasonable when dealing with simulation of real life sintering. A revision of the algorithm and its memory model to allow for execution in parallel and on large scale problems should make it possible to within reasonable time and storage capacity simulate the sintering process on a multiprocessor system. The parallel algorithm would also prove a better evaluation basis for the hardware accelerator. Another motivating factor is that during the development of the parallel algorithm information useful to the design of the hardware accelerator might be found.

The original model was based on an experiment made by Alexander and Balluffi [29]. The original core algorithm, which should not be altered, was built to simulate that experiment. The experiment material was 128 μm diameter copper wire wound around a copper spool. In the computer this is represented by a model of at least four particles, with a radius of about $2.5 \cdot 10^5$ atoms each. Because the sequential program can not handle that amount of data, two smaller data sets were used where the particles had radii of 60 and 120 atoms. The correctness of the model was then proved by using an estimation based on results from the two different simulated model sizes

and the estimation was then compared to the actual results of Alexander and Balluffi's experiment. If a memory model is designed to allow for the parallel version to run simulations with data having the same size as the actual experiment, such a simulation would give much better results to use in comparison to the experimental result. Thus, based on that comparison, the core algorithm could be further fine tuned at MINMET (if needed).

The actual need to simulate sintering on computers is motivated by cost, as is often the case. "A sintering schedule is often set by trial and error techniques in industry" [11]. Thus, when experimenting with sintering a wide range of raw materials will be used. An accurate simulation model would mean that the trial and error phase could be run on the computer. Not only would the cost for buying raw materials be reduced but also the cost for running all the sintering machinery.

1.2 Goals and expected results

The goals are to create a parallel version of the sequential code that outperforms the latter when it comes to simulation times and problem sizes. This goal has to be met without any alterations to the underlying simulation algorithm. The parallel version should also scale well, i.e. an increase in the number of processors used for execution should result in a further decrease of simulation time for a fixed problem size. The parallel application should also be able to run with input data having particle sizes ranging from a few thousand atoms to several billion atoms.

It is expected that the parallel code will out perform the sequential version and that it will at least be able to simulate sintering within a system of four particles each having a radius of $4.0 \cdot 10^4$ atoms. Such a model has almost four billion atoms and it is expected that the rewrite of the storage scheme should put the memory usage for that model below two gigabytes.

It is also expected that the results from parallel simulations should be accurate and similar to the results from the original program within a reasonable margin of error. The error margin should, largely, be related to the randomness of the core algorithm.

1.3 Evaluation methods

First, when it comes to correctness between the original sequential code and the parallel version, the parallel application will generate output for a fairly small problem. A small problem is a model consisting of four particles each having between 30 and 150 atoms in radius. The average results from several parallel simulations can then be compared with the outputs of the sequential version when using input data consisting of identical models and the same random seeds (if necessary). For such small problems, it will also be possible to compare the results from the sequential and parallel version visually. Making sure that the end result has particles of roughly the same shape.

Second, when it comes to simulation speed, the parallel and sequential application will be run across the same multiprocessor system. The difference in CPU time used for varying small problem sets between the parallel and original program will be noted and compared. Because of the difference in the number of Monte Carlo steps

(MCS) between simulations of even the same sized models, the CPU time will be measured as time per MCS rather than the time needed to complete a full simulation. The parallel version will be run on larger inputs with varying number of threads, to measure the scalability.

Third, the new memory model can be measured based on the memory used compared to the old memory model. This will be done for a range of problem sizes. It is also interesting to see how the different parts of the memory model behave depending on how the model that is generated.

1.4 Thesis structure

The rest of this report is organized as follows. In section 2, a brief review of related work will be presented. Section 3 will describe the process of sintering and its uses. The original model and the resulting sequential algorithm will be presented in section 4. The design of the structure for a new memory model is described in section 5, also including various ways of initializing the data. In section 6, various methods of parallelization of the algorithm will be discussed. Section 6 also investigates all the bottlenecks of the algorithm which put an upper bound on the possible parallel performance. Section 7 will describe the implementation phase of the application with both the new memory model and the parallel code. The evaluation results will be analyzed in section 8 followed by conclusions and future work in section 9.

2. Related work

The research behind the algorithm design in this master thesis was based on related work from three major fields. First, the earlier work performed on the simulation model and the original program code. Second, articles on effective storage of sparse matrix based data, similar to the simulation data. Third, papers dealing with the design and implementation of parallel Monte Carlo algorithms.

2.1 Background

An important paper covering the original version of the sintering simulation program was written by Sutton *et al.* [29]. It gives a good view into the model and its background. Postula *et al.* [23] describes the first attempt to map the simulation program onto a specialized processor.

2.2 Sparse data storage

McKellar *et al.* [18] deals with pagination of large matrices, i.e. how the elements of a matrix should be mapped to pages in order to yield the minimum number of page faults. The partitioning of the matrices is done in either row storage or sub-matrix storage. Efficient storage of large matrices is also discussed by Sarawagi *et al.* [26]. The word chunking is used instead of pagination to indicate the division of an array into smaller storage units. The article also mentions reordering of the array to achieve faster access. The advantages of decomposition, in this case by using tiling, and compression of large data sets are briefly discussed by DeWitt *et al.* [8]. The data sets dealt with in the article are geographic imaging for a GIS database.

A sparse matrix is defined as containing only a small number of non-zero elements. The data dealt with in this report is not sparse, but large parts of the resulting array have the same value. Thus, for a subset of the array the contents can be seen as sparse. Ujaldon *et al.* ([30], [31]) investigates various techniques of storing sparse matrices in order to perform better in parallel. A more efficient lookup and a lower decomposition overhead are achieved at the cost of worse load-balance and locality. Lin *et al.* [17] further discusses various storage schemes, implementing three of them on a parallel machine with distributed memory. When computation activity is focused on only a relatively small region of a large matrix, that region can be reformatted into a sparse data structure according to work done by Cheung *et al.* [6]. The reformatting would allow for a more efficient computation. The workload of the algorithm this paper is concerned with is in fact a large array with the actual computation focusing only on a few smaller regions of the array.

Seamons *et al.* [27] deals with the handling large arrays with concerns of I/O performance. The article discusses chunking and compression and the combining of the two. It also briefly reports on two compression algorithm and having modified them to work on data already in memory.

2.3 Parallel Monte Carlo algorithms

Cvetanovic *et al.* [7] reports on the results from parallelization of various algorithms. The algorithm of interest for this report is Monte Carlo simulation. The article compares the implementation of three parallel strategies for Monte Carlo simulation. The three strategies were implemented on two different shared-memory systems. Parallelization of a Monte Carlo algorithm for simulating ion implanted particles is discussed by Hössinger *et al.* [15]. The algorithm is run on a cluster of computers using the MPI (Message Passing Interface) standard. The problem data is divided into sub-domains and each sub-domain is mapped to a slave process. A master process monitors the slaves' performance after the first time step and balances the load accordingly, i.e. more sub-domains are mapped to faster slaves. Communication between the slave processes will only be necessary when a particle leaves a slave's sub-domains.

Other than dealing with vectorization of a Monte Carlo algorithm for Electron-Gamma showers, Miura [19] also describes parallel implementation of the same algorithm. A shared stack or queue with particles is used, each process fetching work from the stack. This way load balancing is automatic. The article also describes the need of a parallel random number generator in order to receive the same results for the same problem. Beichl *et al.* [4] discusses the implementation of a parallel Monte Carlo algorithm for Molecular beam epitaxial growth for a two-dimensional model. In the implementation CMMD communication library is used for message passing. The model is divided into uniform sub-grids where one processor is assigned to each sub-grid. Most of the article deals with conflict resolution between neighbouring sub-grids, but it also mentions the importance of a parallel random number generator.

Another Monte Carlo algorithm, in this case dealing with the folding of large proteins, was parallized by Ripoll *et al.* [25]. A number of slave processes perform independent tasks, thus no communication is needed between the slaves. There may, at times, be fewer independent tasks available than the number of processors, leaving processors idle. Message passing between slaves and the master process is done through the iPSC/2 interface.

2.4 Summary

In this work, we propose chunking of the simulation data model into smaller *grids*. Sparse storage schemes are used on the contents of the grids when it is possible and necessary. There can be no one-to-one mapping of any of the related parallel Monte Carlo algorithms to this project. Instead, we propose using a shared circular queue from which the processes can fetch grids to perform work on. This technique is similar to the shared stack of particles described by Miura [19]. An important difference is that unlike the particles the grids can not be seen as independent. Therefore, the proposed parallel algorithm needs to be able to resolve grid conflicts, as discussed by Beichl *et al.* [4].

3. Sintering

The ISO definition of sintering is "The thermal treatment of a powder or compact at temperature below the melting point of the main constituent for the purpose of increasing its strength by bonding together of the particles." [9]

Sintering is mostly used with ceramic powders creating sanitary wares such as sinks, bathtubs and toilets. But in the following text we are interested in solid-sintering of "elemental" powders.

First we look at the underlying atomic process which makes sintering possible. Thereafter follows a description of the actual sintering process. At last we take a look at what products are currently being made by sintering.

3.1 Diffusion

Here follows a short presentation of diffusion based on the work of Shewmon [28] and Moran [22].

The reason to study diffusion is to learn how atoms move in solids. The diffusion of atoms is one of the most fundamental processes that control the rate at which many transformations occur. There are several different kinds of diffusion but in this report focus is put on self-diffusion. The reason for this is that the inputs that are considered are systems containing just one type of atom and self-diffusion is the process of movement of chemically identical atoms within a solid specimen.

Early work within the field of diffusion was conducted by Adolf Fick who in 1855 presented a paper where he derived two laws of diffusion. In the laws he introduces the concept of a diffusion coefficient which determines the rate with which elements move in a given solid by diffusion. The SI unit of the diffusion coefficient is square meters per second. The diffusion coefficient is sensitive to changes in temperature and varies between different elements.

The study of diffusion has later focused more on the actual atomic process, the movement of atoms. The diffusion coefficient can be related to the atoms' jump frequencies and jump distances. This naturally means that the jump frequency is sensitive to temperature changes. "Near melting point of many metals each atom changes sites roughly 10^8 times per second." [28]

Diffusion occurs to produce a decrease in Gibbs free energy. Every atom oscillates around its equilibrium position in the lattice. An atom with extra energy oscillates more violently than other atoms. If it has an adjacent vacant site, a neighbouring equilibrium position that is unoccupied, and if the oscillation is large enough it may move to that position. In order to perform the move the atom must also squeeze between two neighbouring atoms, as shown in Figure 1. This means that the two constraining atoms must simultaneously move apart. That movement will cause the entire lattice to dilate or expand temporarily. The lattice distortion sets a barrier to how often an atom can jump. In contrast to interstitial movement, where smaller atoms migrates by forcing their way between two larger atoms as show in Figure 2, the energy barrier is huge.

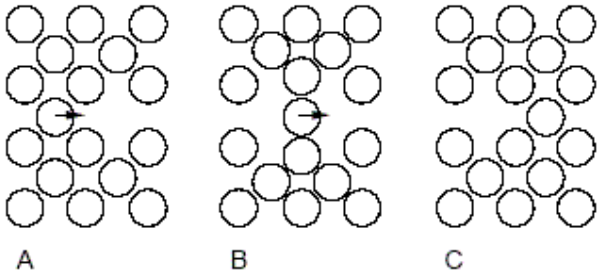


Figure 1: *An atom squeezing between two of its neighbours.* [5]

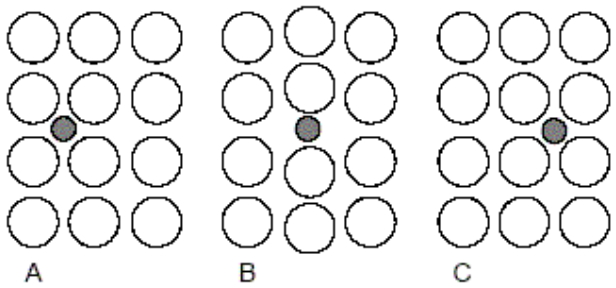


Figure 2: *A smaller atom squeezing between two larger atoms*. [5]

Regardless of how much extra energy an atom has it can not move unless there is a neighbouring vacancy site. The rate at which an atom is able to jump within a solid will, thus, clearly be determined by how often the atom encounters a vacancy and this in turn depends on the concentration of vacancies within the solid specimen. Both the probability of jumping and the concentration of vacancies vary with temperature.

An important observation can be made. The diffusion of atoms into vacant sites can just as well be thought of as the diffusion of vacancies onto atom sites. The difference is that a vacancy is always surrounded by sites which it can jump to (Figure 3). Although, a move must always involve an atom; a vacancy can not change place with another vacancy. The atomic jumps of the vacancies are completely random, thus are made in all directions and follow no particular pattern. Once a vacancy has exchanged place with an atom, where the atom actually made a jump into the vacant site, there is a higher probability that it will make a jump back to its previous position than anywhere else.

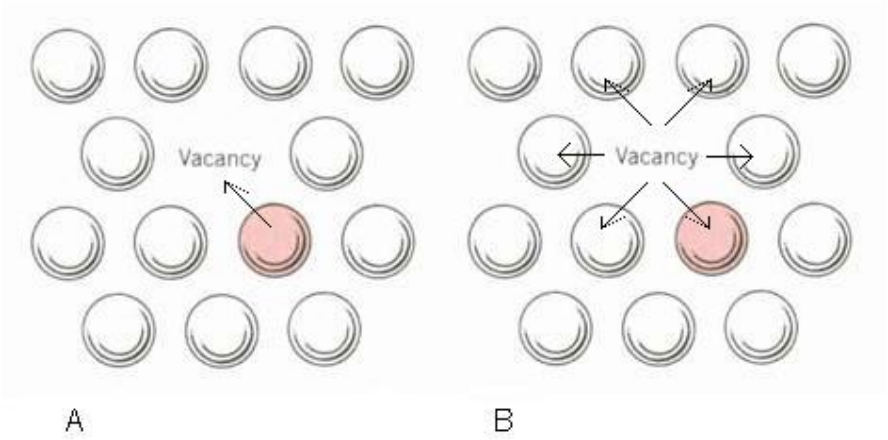


Figure 3: *(A) The atom may only move to the vacancy, (B) where as the vacancy may move in any direction.* [32]

It should be noted that defects within the solid specimen such as dislocations and grain boundaries have a higher rate of diffusion.

3.2 Sintering process

The information in this section is composed from books by German [11] and Kingery [16].

The ingredient of sintering is a powder (Figure 4), which most often has been compacted under pressure. The powder is then heated at a temperature below its melting point. This will cause the powder to bind together and form a solid substance. More exactly the sintering process provides energy to make the particles of the powder weld together.

Figure 4: *Very fine aluminium powder which can be used for sintering.* [12]

The result of the sintered powder can have a number of improved properties such as increased strength, hardness and conductivity. The only disadvantage is that the powder compact will shrink during the process.

The driving force of sintering is a reduction in the system free energy. The reduction is accomplished through diffusion, as described in previous section. Initially sintering will cause surface diffusion where necks are formed between particles as shown in Figure 5. The free energy of a particle can be related to its surface area. On an atomic state, the movement of atoms to form the neck is favourable because it reduces the net surface energy by decreasing the total surface area. During this stage there is no shrinkage. At a later stage there will be bulk diffusion as well as surface diffusion. During bulk diffusion, atoms within the particle may move (Figure 1) eventually causing vacancies to surface. The atom movement will start filling the pores within the powder and along the edge between the particles. Together, this movement and bulk diffusion make the substance become denser and shrink. In the end of the process the pores become isolated and the diffusion rates are extremely small. It is near to impossible to create an end result with 100% density through ordinary sintering.

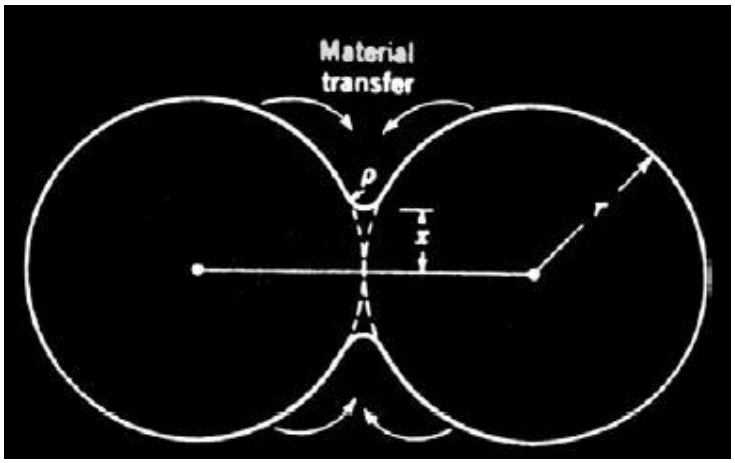


Figure 5: *Matter is transferred from the surface into the area between two particles, forming the neck.* [2]

There are several factors which control the sintering rate. The size of the particles in the powder is very important. As the particle size is decreased, the rate of sintering is increased. The sintering rate is, just as diffusion rate earlier, also strongly dependent on the temperature. Another factor that plays an important part in the result of the process is the sintering atmosphere.

There are various ways of measuring the substance progress during sintering. One method is to monitor the neck growth between particles or the density of the substance. It's also possible to measure the substance's shrinkage or free surface area.

3.3 Sintering products

Not until after the Second World War did products made from sintering become available at a larger scale. The arrival of these products was due to the progresses within the field of sintering made during the war. The driving bands for artillery were made of copper, but there was a shortage of that metal in the Western Europe. Other

solid metals, like iron or steel, prove to be poor replacements for copper because of their hardness. Eventually research within sintering of iron powder created a product which could replace the copper as part of the artillery driving bands. After the war the automobile industry became by far the biggest consumer of sintered parts and still is.

Most parts made by sintering are small, weighing around 10g. Some examples of the components currently being created are gears of various kinds and sizes (Figure 6), magnetic parts, electrical contacts, filters and metal working tools.

Figure 6: *Various gears made through sintering.* [20]

The text of this section was compiled from the book "*Powder metallurgy: the process and its products"* by Dowson [9].

4. Background

The background for the sintering simulation consists of the original experiments from which the model is derived. The content of the simulation data is individual sites within and around the particles. The data and the initialization of it are described after the original experiment. Last follows an introduction to the original application, its core simulation algorithm and all output generated during a simulation.

4.1 The simulation model

The simulation model described in this chapter was derived from an experiment dealing with copper wires, originally performed by Alexander and Balluffi [29]. The copper wires each has a diameter of 128 μm and are wound around a copper spool. They are then sintered at temperatures between 900˚C and 1000˚C for up to 600 hours. Cross sections of the wire are examined during different time intervals, the resulting images can be seen in Figure 7. These experimental results can be simulated by a series of close packed circles, which are of the exact same size and shape. Because of this only a subset of the problem needs to be simulated. The behaviour of four circles can be multiplied to create the same images as in Figure 7.

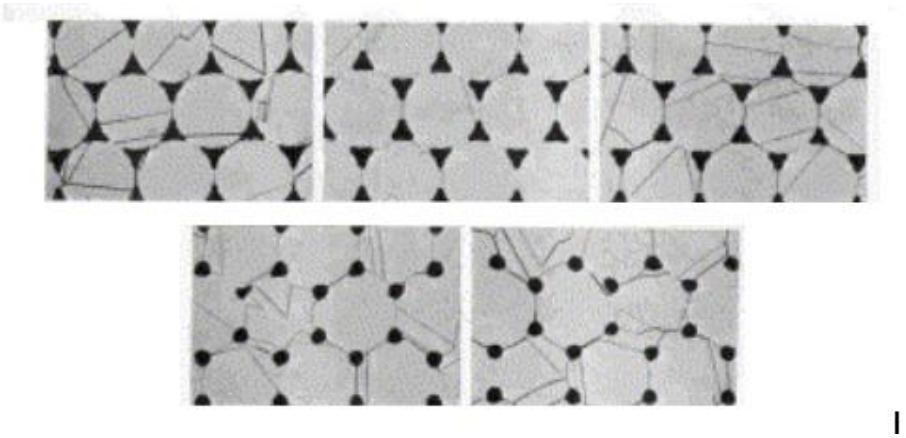


Figure 7: *An etched cross-section of 128 μm diameter copper wires after sintering at 900 ˚C for 600 hours.* [29]

4.2 The simulation data

The input is a representation of a number of closed packed circular particles. Currently the data generated can consist of four particles. The actual atoms and holes or vacancies of the data model are mapped onto a hexagonal grid, giving each atom six immediate neighbours as seen in Figure 8. This model is consistent with the atomic structure of Copper.

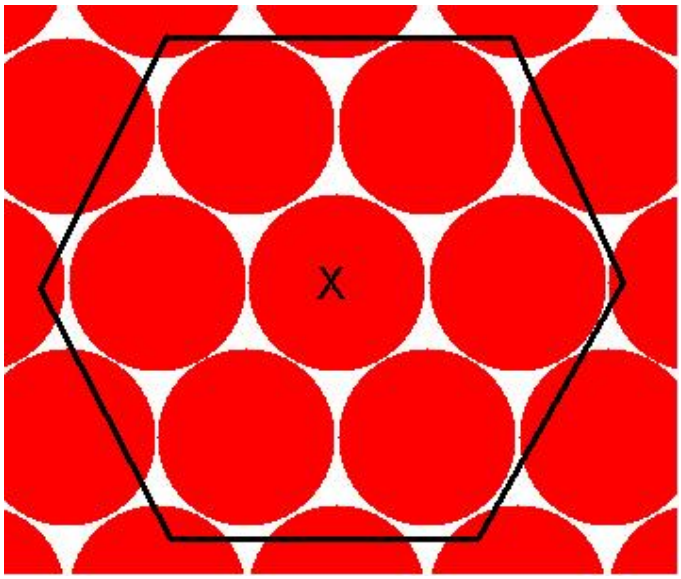


Figure 8: *The hexagonal atom pattern where the site marked by X has six immediate neighbours.*

Each site in the data corresponds to either an atom belonging to one of the particles or a type of vacancy. There are five types of vacancies which are of interest and they are surface vacancy, pore vacancy, pore surface vacancy, grain boundary vacancy or bulk vacancy. Which type a vacancy should be is determined by its neighbouring sites (Table 1 and Figure 9). Vacancies of the type free space are generally outside the range of interest for the simulation. The atom sites are represented by a number to identify which particle it belongs to. During the simulation an atom site may change particle, as will be discussed later.

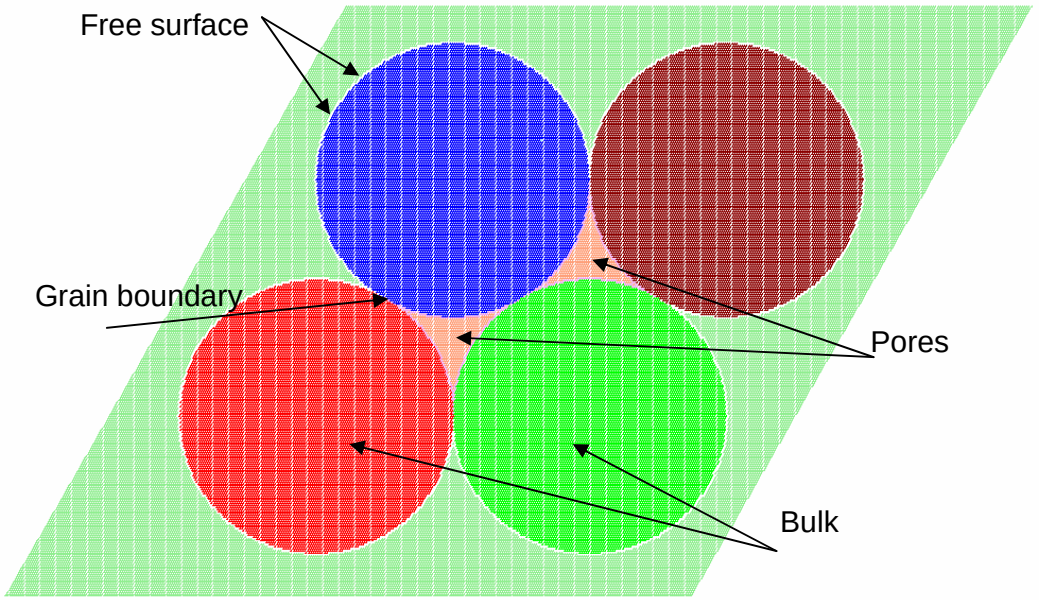


Figure 9: *The different areas within the model.*

Table 1: *List of the vacancy types and their characteristics (use* **Figure 9** *as reference).*

Vacancy type	**Neighbourhood**
Free space	Outside free surface
Grain boundary	At least two atoms from different particles and/or other grain boundary vacancies
Pore	Inside the pore but without any atoms as neighbours
Free surface	Between atoms from one particle and free space
Pore surface	Between atoms from one particle and pore
Bulk	Surrounded by atoms from one particle, may have other bulk vacancies as neighbour as well

The code that generates the data takes the number of particles, their radius in number of atoms and the sintering temperature. A two dimensional integer array is created and initially contains only free surface vacancies. The array maps to the hexagonal data as seen in Figure 10. The centre of each particle in the array is then calculated from the radius. By going through the array in row-major order and using the particles' centres each individual site is turned into a pore vacancy or an atom.

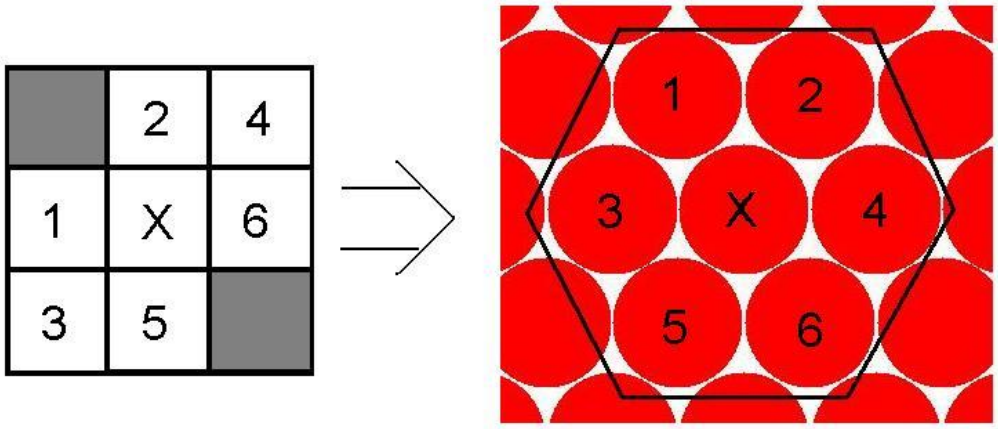


Figure 10: *The neighbourhood of a site (marked by X) in the array and when mapped to the hexagonal grid.*

There is a natural balance between how many bulk vacancies reside in the particles at a certain temperature and the particles' size. That balance is called the equilibrium concentration *ec* and is calculated from the total number of atoms *N* and the sintering temperature *T* (˚K) (Equation 1).

$$ec = 0.5 + N \cdot \exp\left[\frac{-1.1}{8.62 \cdot 10^{-5} \cdot T}\right] \qquad (1)$$

A number of sites containing atoms equalling the equilibrium concentration are randomly picked and turned into bulk vacancies. The array is now searched again and sites responding to surface vacancies or pore surface vacancies are correctly marked as such. To find those vacancies the neighbourhood of each vacant site is investigated. If there is at least one atom in the neighbourhood, the vacancy will be

turned into either a surface vacancy or a pore surface vacancy. The array is then saved to file.

The resulting data created does not contain any grain boundary vacancies. This type only appears during simulation when a vacancy becomes surrounded by atoms from two different particles.

4.3 The simulation algorithm

At the start of the algorithm the two dimensional data array, the matrix, is read from an input file and the initial *porosity* and *rugosity* coefficient are calculated. These values are updated throughout the simulation and occasionally written to files. The file can then be used to compare the simulation data with physical experimental data at different time intervals. These two values are discussed later in this section.

The algorithm used for simulating the sintering process of the copper wires takes advantage of the observation made in Section 3.1 by using vacancy diffusion. Instead of going through all atoms and determine if they should move, it only considers the vacancies with at least one atom in the neighbourhood. Therefore, before entering the main loop the matrix is read and a vacancy list is created.

Each step of the main loop starts by picking a site at random from the vacancy list and randomly choosing a direction to jump to. If there is no atom at that neighbouring site, the jump is disqualified. A special case that can also make the jump disallowed is if a grain boundary vacancy was moved perpendicular to the grain boundary. Another special case is when a move results in a bulk vacancy being created, that move is not allowed if the total number of bulk vacancies equals the equilibrium concentration.

The use of random numbers will be continuous occurrence throughout this section. Randomness for physical events is the most important aspect of the Monte Carlo technique, named so because of the random element of gambling and the many casinos in Monte Carlo. Therefore, an entire iteration through the vacancy list will be referred to as a Monte Carlo step (MCS).

It should be mentioned that an identical algorithm where the sites are picked sequentially from the vacancy list has been implemented as well. It showed similar results as the random version. Therefore, it would not be incorrect to consider vacancies in the same order every loop instead of randomly picking them from the list. Although no longer using randomness to select which vacancies to attempt to move, the Monte Carlo technique is still used to decide whether a selected vacancy should move or not, by applying random numbers.

In order for a vacancy and atom to change sites the atom has to have an energy equal to or above the activation energy. The probability that the atom has this energy is related to jump frequency which in turn is related to the diffusion coefficient. The algorithm assumes that all jumps involving a surface vacancy always has sufficient energy to perform the move. The probability for a jump involving a grain boundary vacancy or a bulk vacancy is then given in relation to the surface vacancy movement. For copper these probabilities are 10^{-4} for bulk diffusion and 0.6 for grain boundary diffusion. Thus, a random number is created for each bulk vacancy or grain boundary

vacancy move to simulate the jump frequency. If the probability is lower than the random number, the jump is disallowed.

Once a jump has been made, there is a chance that the movement will be reversed. The diffusion process strives to decrease the overall energy of the system and an atom's energy is determined by the number of neighbouring atoms. Thus, if the atom's jump increased the number of adjacent atoms, the atom is likely to remain in its new site. Whereas, a decrease in the number of nearest neighbours often causes the jump to be reversed. The probabilities based on the change in the number of adjacent atoms can be seen in Figure 11.

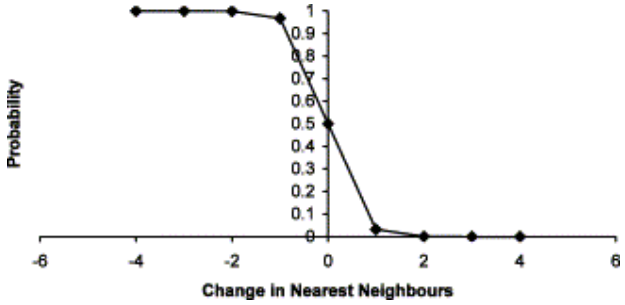


Figure 11: *Graph plotting the probability of an atom reversing a jump depending on the change in nearest neighbours.* [29]

If the move has been made there are a few things that will be checked. The moved atom may change particle type if it is surrounded by more atoms of another type. The neighbours of the moved atom are checked and the moved atom is set to the same type as the particle dominating the neighbourhood (Figure 12). There is also a chance that a moved atom caused a small pore area to get enclosed as shown in Figure 13 and the phenomenon is called a pinch off. This happens when two particle surfaces are close to each other and the atoms cause a bridge to form between the surfaces. The area, if small enough, will become part of the grain boundary. There is also the possibility of a pinch off where the bridge is of the same atom type. In that case the sites in the area become bulk vacancies and the equilibrium concentration might temporarily be overrun.

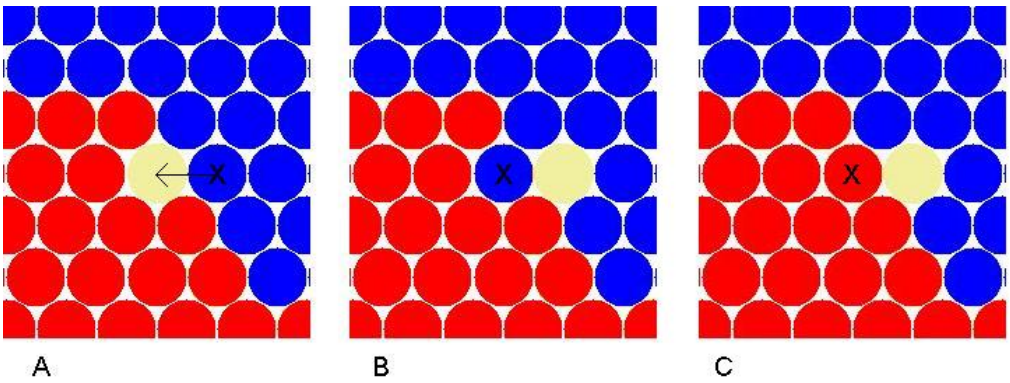


Figure 12: *(A) An atom moves into the grain boundary vacancy, (B) after the move the atom has one neighbour from its own particle and four neighbours of another particle, (C) thus it becomes part of that particle.*

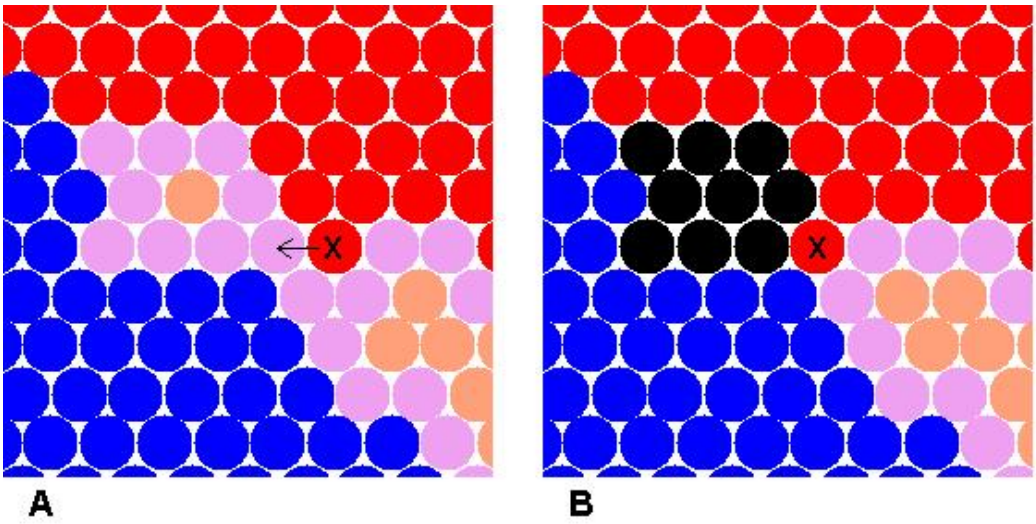


Figure 13: *(A) The atom marked X moves left and (B) blocks off a small part of the pore which becomes grain boundary vacancies.*

The moved vacancy will also need to be considered. If it is now part of a pore or the free area, its neighbouring vacant sites must also become part of the pore or the free area. And their neighbouring vacant sites must be freed as well. And so on.

Lastly, another special case considering grain boundary vacancy movement is annihilation. In real life experiments, annihilation occurs by particles collapsing in along the grain boundaries. Pore vacancies moving into the grain boundary and causing these collapses is what makes the particle shrink and the pore to eventually eliminate. In the algorithm annihilation happens when a moved vacancy is a grain boundary vacancy and it is simulated by an entire row of atom moving through the centre of mass of a particle, thus, effectively moving the grain boundary vacancy to the surface of the particle (Figure 14). The surface of the particle could either be actual free surface, pore surface or another grain boundary. If the vacancy ends up in another grain boundary, it will try to annihilate again. The move always takes place through the particle having the centre of mass closest to the vacancy. In case of an annihilated vacancy ending up in the grain boundary, it could annihilate back again through the same particle. Effectively ending in the same site it started. The chance of annihilation occurring is determined by the annihilation coefficient, which is a constant value set before simulation. Care must be taken when setting the coefficient; a too high value will cause distortions in the final particle structure.

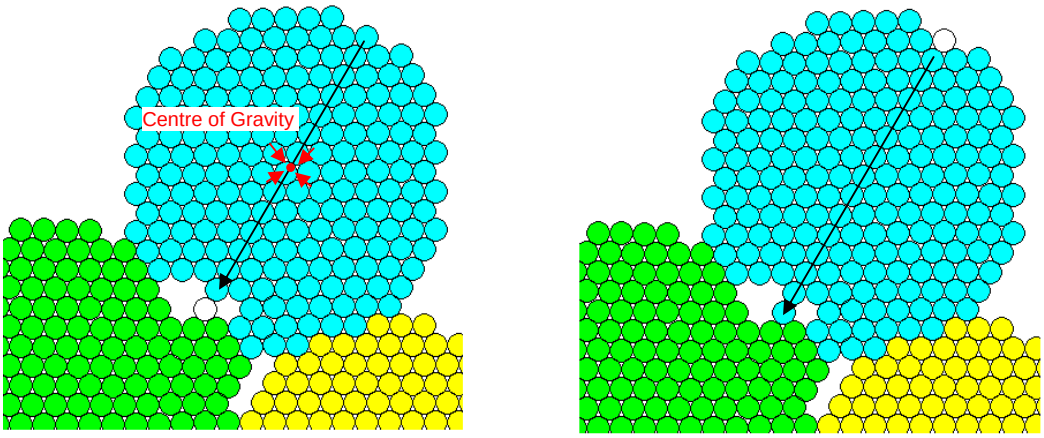


Figure 14: *An example of grain boundary annihilation.*

The line from an annihilating vacancy through the closest centre of mass to a surface is approximated by a modified Bresenham line algorithm [13]. It is modified to take into count the hexagonal grid rather than rectangular.

As mentioned in section 3, there are several ways of measuring sintering performance. In the algorithm the two features of interest are the pore size and the pore shape. The pore size, *porosity*, is defined as the ratio of the number of sites in the pore area N_p to the total number of sites in the substance N_{tot} (Equation 2). The pore shape is calculated as the curve formed by atoms on the pore surface N_s and the number of vacancies in the pore N_p. The *rugosity* is then the average pore shape for a given number of pores, N_{pores} (Equation 3).

$$Porosity = \frac{N_p}{N_{tot}} \tag{2}$$

$$Rugosity = \frac{N_s / N_{pores}}{2 \cdot \sqrt{\pi \cdot \left(N_p / N_{pores}\right)}} \tag{3}$$

When the *porosity* becomes zero, the sintering is complete. During the process five intermediate particle structures are saved. At the moment, those files can not be used to restart the simulation. Figure 15 shows how the particles evolve during the simulation.

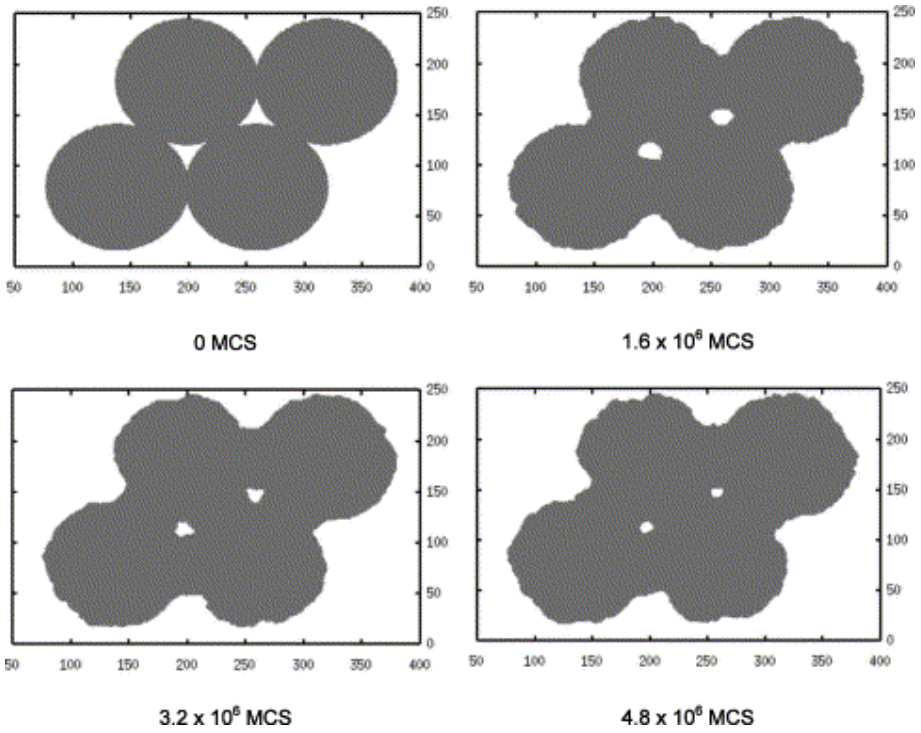


Figure 15: *This image shows how the micro structure changes during the simulation.* [29]

5. Memory model design

Before designing the parallel version of the algorithm the memory model has to be revised. First the current memory model needs to be introduced, which will make the need for revision obvious. An advantage of designing the memory model structure first is that its changes will then also work with an only slightly modified version of the original sequential code.

The individual atoms are arranged in a hexagonal pattern. This pattern is then mapped to a two dimensional array. Figure 16 shows four particles as their hexagonal pattern and how the same particles look in the matrix.

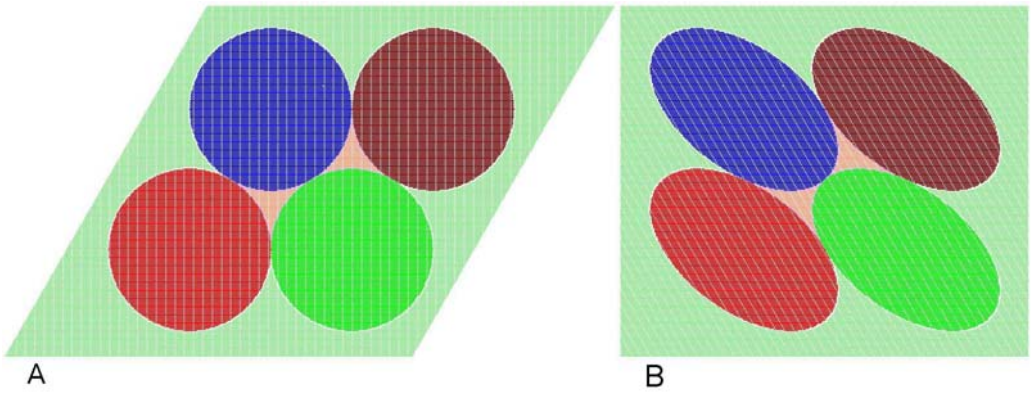


Figure 16: *(A) Image of four particles in the hexagon pattern and (B) the same particles as stored in a matrix.*

When the simulation finishing the pores of the particles will have completely disappeared and the resulting images are shown in Figure 17. From the figures it can be seen that during the simulation a large number of the elements in the array remain the same from the beginning to the end. The white outline around the particles (Figure 17) showing the sites that have been moved, every site outside that white outline has been of the same type through the entire simulation.

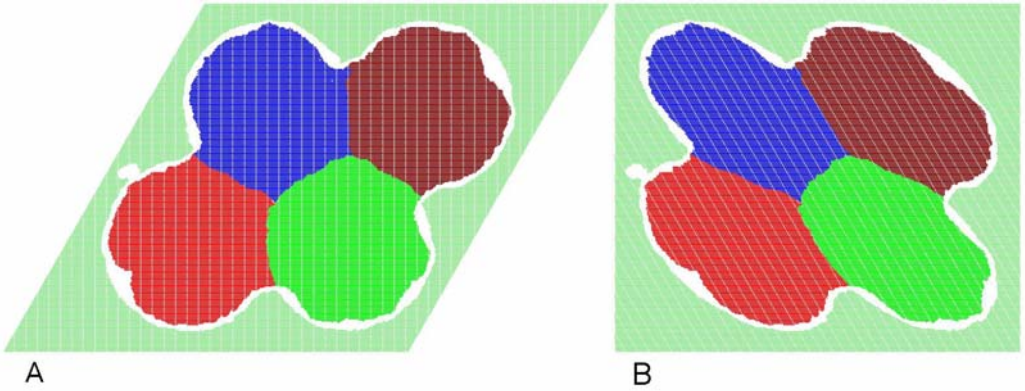


Figure 17: *(A) Image of four particles in hexagon pattern after the sintering is complete and (B) the same image as stored in the matrix.*

The two-dimensional array does not scale well with the problem size, nor does it treat the sites differently. A site is stored using the same amount of memory in the array regardless of type or if it has ever been updated. As shown in Figure 16 and Figure 17, a lot of the array will in fact be constant during the entire simulation. The fact that the experimental model of Alexander and Balluffi will use more than one *terabyte* of memory should make the need for a revision of the memory model obvious. But the revision is necessary for another reason as well. Because this project deals with a shared-memory system, the parallelization will result in several processes concurrently accessing the same model. If the model remains unchanged, the access to the two dimensional array will be restricted to one process only. That limitation is unacceptable for parallel execution.

The storage proposed is done by partitioning the particles into four different kinds of areas, called *grids*. It is expected that such a partitioning, chunking [26] or decomposition [8], of the model will be successful because of the large amounts of similar data in the model. The grid types, which are discussed more in detail below, are free, bulk, pore and surface. The grids all have the same dimension and the model is divided into a number of grids based on their size. The sites inside a grid determine which of the four types the grid is.

5.1 Grid types

In order for the partitioning of the model to successfully reduce the memory usage, the grids with sparse data must be found. "…significant savings in storage and computation can be achieved if this limited sparsity can be handled efficiently" [6]. In light of this, the different areas of sparse and dense data within the model cause the following four grid types to be named.

5.1.1 Bulk area

A bulk vacancy is a hole inside a particle. This type of vacancy has a very low probability of moving. In the case of copper at the sintering temperature the probability is ca. 10^{-4} relative to the surface diffusion [29]. By definition a bulk vacancy is only surrounded by atoms of the same kind and some times other bulk vacancies (Table 1). The bulk vacancies are completely randomly distributed inside a particle. The total number of bulk vacancies in all the particles is in the beginning of simulation equal to the equilibrium concentration. The concentration is dependant on the sintering temperature and the number of atoms in the model (Equation 1). Throughout the process the number of bulk holes should rarely exceed the equilibrium concentration; it is more likely the number will be lower than the concentration. There is always a chance that a bulk vacancy will eventually make it to the surface, thus becoming a surface vacancy. But there is also a chance that a surface vacancy will migrate into the particle, thus becoming a bulk vacancy. For a particle, the bulk vacancies can be considered to be non-zero data in a sparse array where the zeros represent the current atom type.

It should be noted that the equilibrium concentration is extremely small at sintering temperatures when compared to the number of atoms. Table 2 shows that equilibrium concentration puts the number of bulk holes just over a million when the total number of atoms is 50 billion. Because of this most of the bulk areas inside a particle will not actually contain any holes. Therefore, it might be more effective to design a global

storage scheme than a grid based one for the bulk areas. Another important thing to note is that because of the low number of bulk vacancies compared to atoms and their low probability of jumping the bulk data will rarely need to be updated. Even rarer are the occasions when a bulk vacancy reaches the surface or a new bulk vacancy is created. Thus, storage scheme for the bulk area doesn't need to be very fast when it comes to updating data, nor when making additions to or deletions from the data.

Table 2: *The particle radius effect on number of atoms and the equilibrium concentration, when dealing with a model of four circular particles at a sintering temperature of 1173 ˚K.*

Particle radius	Number of atoms	Equilibrium concentration
64	51,472	1
128	205,887	4
256	823,550	16
512	3,294,199	62
1024	13,176,795	248
2048	52,707,179	994
4096	210,828,714	3,974
8192	843,314,857	15,898
16384	3,373,259,426	63,589
32768	13,493,037,705	254,354
65536	53,972,150,818	1,017,413

A novel scheme has a global vector containing each bulk vacancy's location in the particle model. The vector has a fixed size equal to the equilibrium concentration, using the fact that the concentration is the maximum number of bulk vacancies allowed in the model. The bulk holes are sorted according to which particle it currently resides in and every particle has a pointer or index to its first bulk hole in the vector. If a bulk hole is moved, only its corresponding location data in the global vector needs to be updated. Should a bulk hole be eliminated, i.e. reach the surface, or a new bulk hole be created, the vector will need to be resorted and the particles' pointers or indexes to the vector will have to be updated accordingly.

Searching the global vector has the potential of becoming very time consuming and the access to the vector will have to be restricted in a multiprocessor environment. Therefore, it may be better from a time performance perspective to mark the bulk areas containing vacancies and treat those vacancies locally.

It is expected that bulk areas will generally appear in between 65% and 80% of the grids and the ratio should stay fairly constant during the simulation. This makes the name bulk even more suitable.

5.1.2 Pore area

The pore areas are created where two or three particles meet. These areas may contain both pore vacancies and grain boundary vacancies and up to three types of atoms. Throughout the simulation the number of pore areas will increase as sintering causes the particles to move closer to each other. Meaning, some surface areas within the pore will eventually become inhabited by more than one particle and thus be changed to pore areas. It should also be mentioned that there are pore areas where two or three particle surfaces meet but there are no holes between them. Such pore areas would

not be of any interest to the algorithm as there are no way the atoms can perform a jump. During the process more and more pore areas will end up having no vacancies as the sintering causes the holes between them to annihilate.

There are atoms that don't belong to the same particle throughout the entire simulation. If an atom makes a valid jump which causes it to have more neighbouring atoms of another particle than its own, it will become part of that particle (Figure 12). This means that the sites within a pore area are clustered together depending on their type. This also means that some pore areas will eventually become bulk areas as they end up being completely dominated by atoms from one particle. If sintering is done over a very long time grain growth can occur, which is when some particles grow larger at the cost of smaller particles. The atoms of the smaller particles will then eventually disappear into the larger ones.

As described earlier, part of the pore area might become victim of a pinch off. This is when atoms form a bridge between the surfaces of two particles. The area now enclosed by that bridge and the two surfaces of the particles will become a grain boundary, if small enough. This is something to keep in mind for these areas.

Since the data within a pore area consists of so many different types of atoms or vacancy sites it can not be considered sparse. But the fact that the same types of sites cluster together should be used in an effective storage scheme. Another important factor to consider is the pore areas that contain no vacancies, since these are of no interest to the simulation algorithm. For the pore areas with surface or grain boundary vacancies, the neighbouring types of each atom must be available in order to determine if a jumping atom should change particle type.

Because of the lack of sparse data and the need for access to most sites in a pore area a storage scheme where the data will be kept in a two dimensional array seems most reasonable. The array will have the same dimensions as the area and each element corresponds to a site in the pore area. If the area doesn't have any vacancies the data should be compressed, as it is of very little interest, in that state, to the sintering process. This does not require any particular compression algorithm. But since the data won't be read by the simulation process until, and if, a vacancy makes it way into the area; the algorithm should have a high compression ratio. Naturally, the algorithm should work on data already in memory and leave the compressed or decompressed data in memory as well, not having to store anything in files, as mentioned by Seamons *et al.* [27].

The fact that the pore areas is expected to populate less than 2% of the grids and that they will experience a lot of change, compared to bulk areas, there is little or no point to consider compressions for the model sizes dealt with in this report.

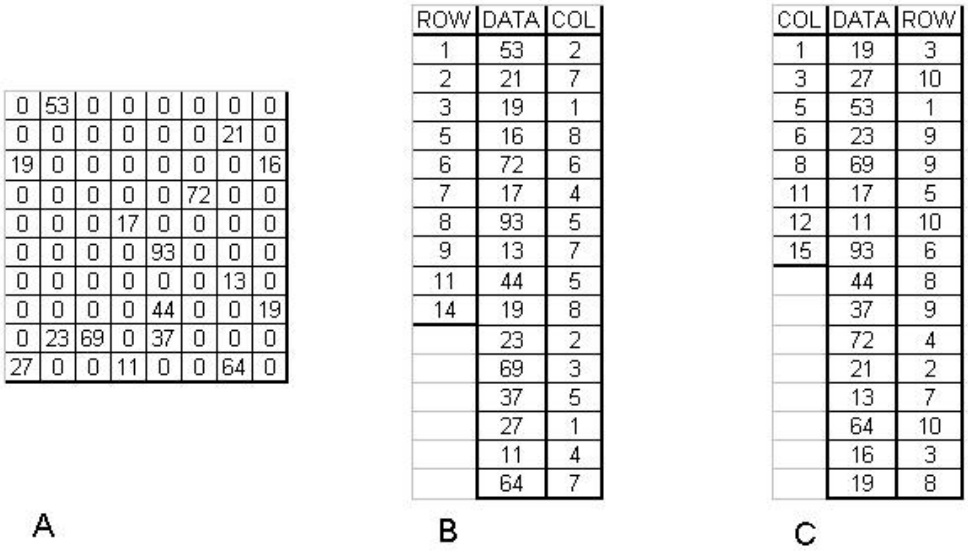
5.1.3 Surface area

Surface areas can appear in two places: outside the substance and in the pores. For both places a surface area consists of part of the surface from one, and only one, particle. The surface of a particle is made up by a barrier of surface vacancies between free vacancies and atoms of one type. When a surface vacancy and an atom change lattice sites the move is assumed to always have sufficient energy. Meaning, as a scaling factor, surface diffusion has a probability of 1 [29]. On the other hand, a

surface jump often ends up with the atom having fewer neighbours than in its previous lattice site, so there is a high probability of the jump being reversed. Because of this, bulk holes next to a surface atom play an important part when considering the reversing of a surface jump. Thus, when considering the memory model for any of these individual partitioning areas the interaction between them must also be part of the design.

One storage scheme treats the areas as sparse arrays only saving the actual surface vacancies. The atoms and free vacancies are in this case treated as zero valued elements. When a surface vacancy is picked for a jump, its neighbouring surface vacancies can be fetched from the data. But which of the neighbouring sites are actual surface atoms must be found in order to get the direction of the allowed jumps. There are two possible solutions to this considered here. The area could have a general direction stored with its data. The direction would then tell on what side of the surface vacancy barrier the atoms are positioned. A second solution would be to assume that the atoms reside in the direction toward the centre of mass for the current particle. During the simulation the direction data of the first solution might have to be updated and it would have to be calculated for new surface areas, whereas updating the centre of mass is already part of the algorithm.

The surface vacancies can be stored using CRS (Compressed Row Storage) or CCS (Compressed Column Storage) as described in [30] and [31]. Both of the schemes use three vectors *ROW*, *COL* and *DATA*. In CRS, the non-zero elements store their data and column index in the *DATA* and *COL* vector respectively. These two vectors will have the same size. The *ROW* vector marks the beginning for each row in the *DATA* and *COL* vectors. CCS works as CRS but with rows and columns interchanged. The result of the two schemes applied to a sparse matrix can be viewed in Figure 18. Because the data stored are only surface vacancies, there is no need to have a *DATA* vector for the surface areas. And instead of using two separate vectors a special buffer, as described in CFS scheme from [17], can be used.



A

0	53	0	0	0	0	0	0
0	0	0	0	0	0	21	0
19	0	0	0	0	0	0	16
0	0	0	0	0	72	0	0
0	0	0	17	0	0	0	0
0	0	0	0	93	0	0	0
0	0	0	0	0	0	13	0
0	0	0	0	44	0	0	19
0	23	69	0	37	0	0	0
27	0	0	11	0	0	64	0

B

ROW	DATA	COL
1	53	2
2	21	7
3	19	1
5	16	8
6	72	6
7	17	4
8	93	5
9	13	7
11	44	5
14	19	8
	23	2
	69	3
	37	5
	27	1
	11	4
	64	7

C

COL	DATA	ROW
1	19	3
3	27	10
5	53	1
6	23	9
8	69	9
11	17	5
12	11	10
15	93	6
	44	8
	37	9
	72	4
	21	2
	13	7
	64	10
	16	3
	19	8

Figure 18: *(A) a sparse matrix and the resulting vectors when encoded with CRS (B) and CCS (C).* [31]

A further improvement is to allow both CRS and CCS compression depending on the data in the surface area. The choice is fairly simple, if the surface of an area has more vacancies located in different rows than columns, the CRS scheme should be used. And the opposite is true for CCS. This will allow for faster access to a specific element in the array. An example of this can be seen in Figure 19 where the search needs to investigate six vector elements for CRS encoding but only three for CCS encoding. In surface areas, where the vacancies are evenly spread among the rows and columns, one scheme can be set as dominant or the neighbouring surface areas can be checked and a scheme is selected from that information. In order to allow two different compression schemes each surface area must record its selected scheme. If a surface vacancy jumps into a bulk area or free area, the area should be transformed into a surface area with the same compression scheme as the area from which the surface vacancy originated.

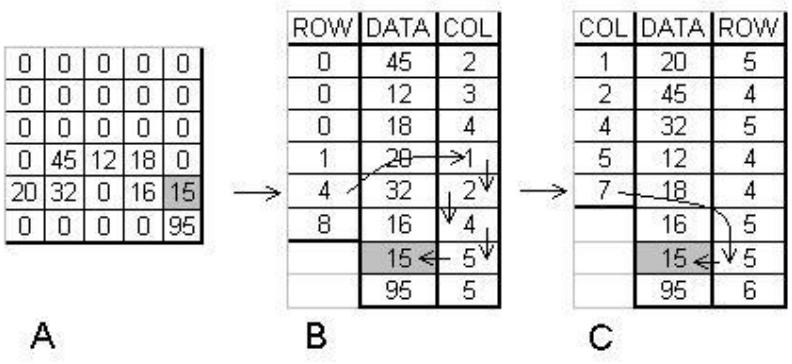


Figure 19: *The search path to find element at (5,5) in a matrix (A) for CRS encoding (B) and CCS encoding (C).*

A second storage scheme, which is similar to the previous one, stores both the surface vacancies and the surface atoms. The increased amount of memory needed for this scheme is the price for always knowing the valid direction a surface vacancy can move. In this case the *DATA* vector of the CRS and CCS schemes would have to be used in order to decide if the element is a surface vacancy or an atom. A possible second solution would be to have two *ROW* and *COL* vectors or two CFS buffers, one for vacancies and one for atoms.

Grids of surface type are expected to make up roughly 5% of the total number of grids. For small model, any surface area compressions will have little influence on the memory usage, but as larger models are needed the same compressions might become essential.

5.1.4 Free areas

There will be areas where there are no surface vacancies or atoms, mainly outside the substance but also inside the pore. Because sintering causes the substance to shrink, the free areas outside the substance will never be of much interest to the algorithm. The free areas inside the pore will, on the other hand, become surface areas and eventually pore areas as the pore continues to shrinks. Free areas will not contain any data and there is therefore no need to design a memory model for them. They are of

interest in the next section, when designing the partitioning of the entire particle model.

The free areas go from having each site stored in the two dimensional array to being treated as empty. The memory gained here is enormous, especially considering the fact that the free areas are expected to take up as much as 25% of the grids and that most of those grids will remain unaffected by the simulation algorithm (Figure 16 and Figure 17).

5.2 The grid structure

In this scheme the two dimensional array of the original memory model is divided into several equally sized grids, as in submatrix storage [18]. Each grid will be of one of the types described in the previous section. An example of this can be seen in Figure 20. As the simulation progresses the grids may change area type as movements occurs inside them.

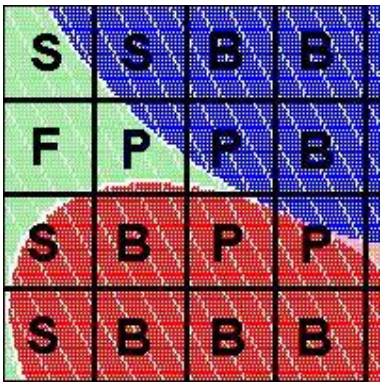


Figure 20: *The grid pattern as applied to part of a model. S marks a surface area, B a bulk area, P a pore area and, finally, F a free area.*

Care must be taken when deciding the size of the grids. Smaller grids will be fast to read and update, but may experience a low compression ratio and have more transactions where an atom moves across a grid's border forcing the neighbouring grid to be read and updated as well. Larger grids may cause an overall worse compression rate because grids have to be initialized with just a few sites of interest, such as a large grid having just a few surface vacancies in a corner. But within an actual large grid the compression rate will be higher, although updating the data may be slower than for a small grid. The number of grids will also be important in the later stage when the algorithm is parallelized. There must be enough of interesting grids, i.e. grids with vacancies, to keep each process occupied without trying to read or write the same grids.

In light of all these variables, selecting the optimal grid size in advance can be difficult. The size may be optimal for the starting model, but as the particles shrink the size might not be as good. Optimally setting the size is made even harder by the fact that the shrinkage of particles is random, and varies between two runs of the algorithm. Therefore, focus should be put on ensuring a high degree of parallelisation. The grid size should be set so that the number of processor can run the code at the same time with the minimum amount of pauses due to the same grid being accessed.

For the four particles, as shown in Figure 16, which each have a radius of 60 atoms the resulting array has 300 rows and 302 columns. A fixed sized rectangular grid for this array will never fit perfectly. Instead the dimension will be increased slightly to allow for the perfect fit. The new columns or rows will on the other hand just contain free space vacancies that are never even considered by the sintering algorithm.

To store the grids the Offset Vector scheme (OV) [6] can be used, where the grids are numbered as shown in Figure 21 and stored in an one dimensional array. Cheung *et al.* [6] give two equations (Equation 4 and Equation 5) for OV when representing an M x N sparse matrix. An array (V_f) contains the offsets of non-zero elements and the second array (V_v) holds the data values for the elements. The elements in the sparse array are located at $x(a,b)$.

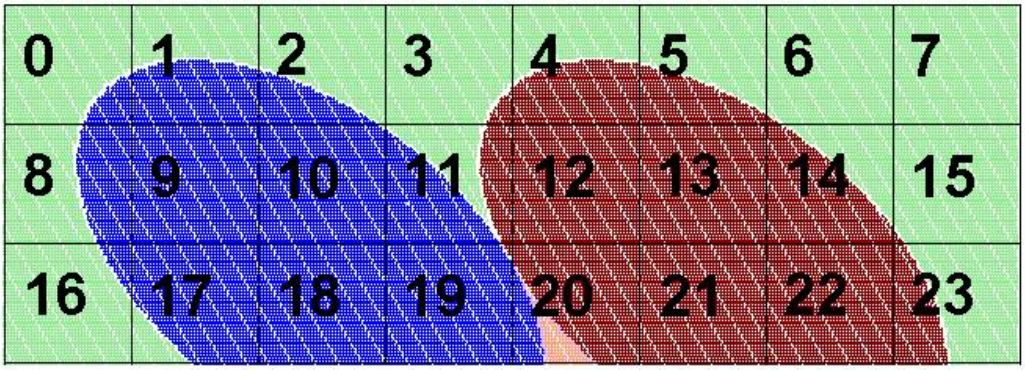


Figure 21: *The grid indexes*

$$V_v(i) = x\left(V_f(i)/N, V_f(i) \bmod N\right) \quad (4)$$

$$V_f(i) = a \cdot N + b \quad (5)$$

For this particle model the array V_f contains the grid numbers and the array V_v contains the corresponding grid data. From a hole or atom at (x,y) in the particle model the responding grid (a,b) can be found from Equation 6. The variables gx and gy define the grid's dimensions. The location, in the particle model, of grid $V_f(i)$'s top left corner (x,y) is found by Equation 7 which is derived from Equation 4 and Equation 6. N is the total number of grids along the width of the two dimensional array for the original particle model.

$$(a,b) = \left(\lfloor x/gx \rfloor, \lfloor y/gy \rfloor\right) \quad (6)$$

$$(x,y) = \left(\left(V_f(i)/N\right) \cdot gx, \left(V_f(i) \bmod N\right) \cdot gy\right) \quad (7)$$

A slightly modified Offset Vector scheme has only a V_v array the same size as the number of grids and no V_f array. So, the result from Equation 5 instead gives the index for V_v directly, as seen in Equation 8. In this case all grids will be in memory, even the ones only containing free space. But finding the data for a grid will no longer involve having to search V_f for the grid number.

$$V_f(i) = i \quad (8)$$

5.3 Data initiation

There are a few points of interest when creating the grid and initializing its data. Firstly, there are only two sites that are of interest when initializing the grid model. That is the surface vacancies and the bulk vacancies. These are the ones that are later moved in the algorithm. But it is still important to know where in relation to these vacancies the particles and pores are located. Secondly, the total number of atoms will need to be calculated in order to determine the equilibrium concentration. A number of bulk vacancies equalling this concentration must then be added inside the particles. Thirdly, the result must be written to file. This means that actual time of the simulation is not bound to the time spent initializing data. It also means that the data initializations phase only needs to be run once, unless a different bulk vacancy distribution or another particle radius is needed.

The original initialization algorithm (section 4.2) investigates each individual site of the model, first when adding the atoms and then when finding the surface vacancies. Not only is this ineffective when the model becomes large, but this also means a lot of access to each grid. Even though the data initialization only needs to be run once for a particular problem size, the grid model will need a more efficient algorithm. Especially when considering the fact that the Alexander and Balluffi model, with $2.5 \cdot 10^5$ atoms in each particle radius, results in more than 800 billion individual sites.

One way to initialize the data would be to consider each grid and their sites. This means creating a small matrix which is a subset of the original model. Use the same initialization algorithm on all sites of that smaller matrix and then compress and save them before reading next grid. The grid might then need to be read again if a bulk vacancy is being added to it. Several grids might actually have to be read again if a bulk vacancy is trying to insert itself at the boundary of a grid. This is because all the neighbouring sites of a bulk vacancy must be checked so that none of them belong to the particle surface. Saving the grids could be done to file directly in order to decrease the memory usage. This algorithm would reduce the grid accesses, but still result in all sites having to be individually investigated.

Another way to initialize the data is to only consider the surface of each particle. The particles in the model being created are circular (Figure 16). Thus, creating a circle in the hexagonal grid and then transforming the surface of that circle onto the matrix model will give the actual particle surface. Another way to get the particle surface is to draw the ellipse directly in the matrix. In both cases adding one to the radius of the shape gives the surface vacancies enclosing the particle. To speed up the creation of the surface, the symmetry of the circle and ellipse equations can be used, *Bresenham's Circle Algorithm* [13].

A third option can be designed which has the advantage of independence from the geometry of the particles. In this case, the starting point is an atom on the surface of a particle. The neighbours of that atom are investigated and surfaces added to the data. If another atom is found, that atom's neighbourhood is investigated. The algorithm keeps adding surfaces until an atom found is the same as the starting point. This *surface walker algorithm* is repeated for each particle.

All these three schemes only check the surface sites, instead of every site in the model. The surface areas become initialized as surface sites are added to their grids. If a surface is added to an area where there is already surfaces belonging to other particles, then that area is a pore grid. The other grids are easily initialized. They are either free or bulk areas, so if a site of one of the remaining grids is an atom, then the grid is a bulk area, otherwise it is a free area. The number of sites investigated will thus be a lot less than the total number of sites in the model.

5.4 Summary

The two dimensional model is divided into a number of fixed size grids, a technique called chunking by Sarawagi *et al.* [26] or decomposition by DeWitt *et al.* [8]. The grids allow for locally sparse data [6] within the model to be taken advantage of, especially for grids of type bulk or free. Together, the two types are expected to represent 85% to 95% of the total grid population and therefore, because of their high degree of compression, the memory used compared to the original model should be reduced by at least a factor of 5. The other two grid types, clearly not expected to be the majority, are the ones that are important to the actual sintering algorithm. The grids of type pore contain different atoms and all kinds of vacancies and surface type grids contain surface vacancies and free vacancies. The latter type can treat the surface vacancies as sparse data which, when compressed, is expected to have a noticeable memory usage reduction on larger sized models.

6. Algorithm design

The parallelization is based on shared-memory system and asynchronous execution. Meaning that all processes share the same main memory and each process executes at its own rate, so called "*lightweight processes*" [3]. Another assumption made when designing the parallel algorithm is that the model is divided into grids, as described in previous section. The actual storage model of the grids will not be of importance to the algorithm. This will ensure that several different grid based storage schemes can be implemented.

Later in this section various designs for a parallel version of the code will be considered. But first the sequential code receives a face lift. The face lift consists of a few modifications that will ensure better performance both for sequential and parallel execution. The code after the update will still contain some bottlenecks that set an upper bound on the performance of any parallel algorithm. These limitations will be identified and discussed.

6.1 Updating the sequential code

Before starting with the parallel design there are a few short comings in the original code that should be addressed. The problems are updating the vacancy list, validating a suggested move, special error detection code, the pore pinch algorithm and improving file input and output. Already after these changes the simulation execution time is expected to be much shorter, resulting in a very good platform to start the parallel design from.

6.1.1 Vacancy list

One factor that makes the original simulation program slow is that it fails to update the vacancy list correctly. Whenever a surface vacancy is moved from the surface of a particle it should no longer be a surface vacancy and it should no longer reside on the vacancy list. The white outline surrounding the particles in Figure 17 consists of surface vacancies. The reason the white area is larger than the immediate surface of the particles stems from the fact that the vacancy list is never purged.

A change in the algorithm is merited. An additional check should be added for each move that is performed; where every surface vacancy in the moved vacancy's neighbourhood should be examined. If one is found without any neighbouring atoms, it should be freed from the vacancy list and set as free vacancy, no longer a surface vacancy.

Since a move consists of a vacancy changing place with an atom, failing to address this matter will cause the algorithm to consider vacancies that has no valid direction to move. They will be completely surrounded by other vacancies. And as the simulation progresses the vacancy list will keep growing. For a large model this means that eventually a very small percentage of the vacancy list have a potentially valid move.

Another beneficial effect of keeping the number of vacancies up to date is that at every intermediate stop where the rugosity and porosity is calculated will be faster. In the original model the entire model is examined and each site corresponding to a

surface vacancy has its neighbourhood checked. The rugosity and porosity computations results in an inspection of all sites and an additional inspection of six sites per each surface vacancy found. A correct update to the vacancy list however means that the computation can be done without consulting the model. The performance gain is two-fold by allowing the calculating process to complete faster and by allowing the other processes to keep accessing and working on the model instead of having to wait for the computations to be done.

6.1.2 Validating move

If a moving vacancy has picked its random direction to a site where an atom resides, the move is made. After the site swap the probability of the move being made is checked and also the probability of the move being reversed. If the move is invalidated the sites are swapped back. Thus, two writes to the model is made for a failed move.

A fairly small update to the algorithm is necessary where a move is not performed until it has been completely validated. This means there would be no writes to the model for a failed move.

The importance of reducing the number of writes to the model when dealing with grids and parallel processes is two-fold. Firstly, a write will force a read grid to be saved. Saving a grid is time consuming when using any kind of compression. There may also be cases where a compressed grid can be read without decompressing it. A write on the other hand always needs a decompressed grid. Secondly, in parallel a grid may be read by any number of processes at the same time, but a writer must have exclusive access. This is called the Readers/Writers Problem [3].

6.1.3 Special error detection code

Error detection code already exists in the original program, but there is need for more detection in order to assure the correct execution of the parallel version. Therefore, in the interest of simulation time, the code should be further specialized to allow the program to run in a debug mode or a fast mode. In the fast mode all the error detection will be disabled, which should give a greatly improved run time.

Another specialized error detection code that should be added to the program is an override of the systems memory allocation function. The override should keep track of how much memory has been allocated, to what part of the address space and from where in the code. It should also keep track of freeing of the memory. At the end of execution all memory allocations and de-allocations should be accounted for. If any memory has not been freed, the part of the code that caused the memory leak can be found. Finding and removing memory leaks should greatly improve performance by both allowing for more memory being available and fewer allocations to be made. In fast execution mode this special detection code should also be turned off.

6.1.4 Pore pinch

Also mentioned in section 4.3, it is possible that a moved atom caused a small pore area to get enclosed as shown in Figure 13. This is something that is checked for each move. And for each moved atom it does in fact perform a pinch off evaluation for all its surface vacancy neighbours, regardless if the surface neighbours are actually connected. This is fairly ineffective because if one surface neighbour has been

evaluated for pinch off and the search concluded that there was no pinch off, any other surface vacancy within that same region will lead to the same conclusion. In fact, for a pinch off to be possible the moved atom must have separated its neighbouring vacancies into at least two different regions. Only when a part of the pore splits because of a bridge of atoms can a pinch off be possible. The last atom to complete such a bridge is the only one that can cause the pinch off and that atom is also the only one that will have separated its neighbouring vacancies into two regions.

Determining if surface vacancies in an atom's immediate neighbourhood are connected can be done at a low computational cost. The algorithm works by grouping surface vacancies together as shown in (Figure 22). It starts with the first surface vacancy found in the neighbourhood and then circles around the atom, all following vacancies are added to the group until an atom is found. If any other vacancies are found after the atom(s), they will belong to the second group. There can be at most three groups of surface vacancies, group being a fairly misguiding term since they'd only contain one vacancy each. It takes three atoms to separate three vacancies making up for the total of six sites in the neighbourhood. Lastly, if a second or a third group is being formed, but the algorithm returns to the first vacancy without having encountered any atoms, then that second or third group should belong to the first group (as seen in part C of Figure 22). When the all groups have been found, the pore pinch computation is only run once for each group instead of once for each surface vacancy. And further more, the pore pinch computation is only run if at least two groups have been found.

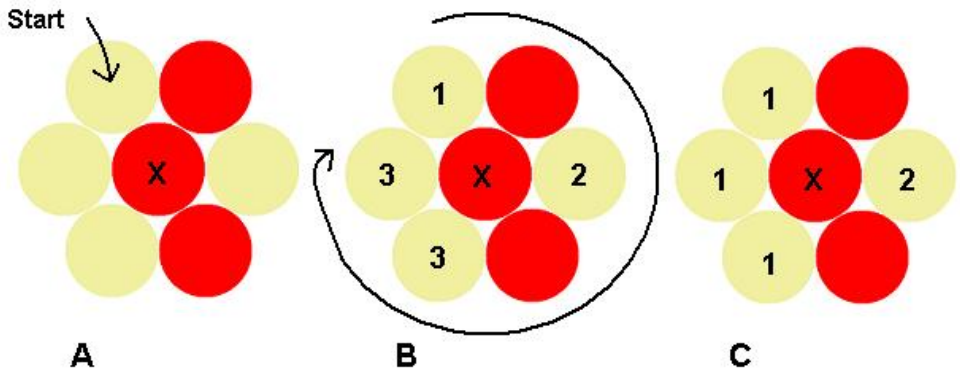


Figure 22: *(A) The neighbourhood of atom marked by X is investigated from the start point (B) in clock wise motion adding unconnected vacancies to different groups and (C) at last connecting the end of the search to the start.*

6.1.5 File I/O

The last issue before considering the parallelization of the original code is the file input and output. The data initialization stage writes a file that can subsequentially be read by the main simulation algorithm. But the algorithm uses a different format when writing to file, which means that the intermediate models that are written to file can not be read by the simulation algorithm.

The simulation and data initialization algorithms would benefit of common file interface. This would ensure that a model written to file can always be read back into the simulation.

There are environmental events, such as a system crash, which might cause sintering simulation to exit. For long execution times it is important to be able to continue the simulation from last saved model instead of restarting it completely. This change alone can mean hours saved for any simulation. Other benefits achieved through a common file interface is that changes to how the model should be stored on disc only has to be made in one part of the code.

6.2 Limitations

Another stop before dealing with the parallel design is to identify the limitation of concurrency. The obvious bottleneck is that several processes need to repeatedly access the same model. But there are also other global data to consider, which will be discussed first in this section. There are two parts of the simulation algorithm which can substantially limit the parallel performance, namely vacancy annihilation and pore pinch. The problem with vacancy annihilation and pore pinch is largely a result of the model access limitation, but they still merit a separate investigation.

6.2.1 Grid access

As mentioned earlier the global data that is accessed most frequently is the model. Not only must access to individual grids be synchronized, but also to the entire model structure in order to assure access to more than one grid at the same time. For instance, when a moving vacancy is in the corner of a grid, three grids, other than the one containing the moving atom, must be read (Figure 23). The move can only be validated if the process has been granted access to all those three grids, not just one or two of them. Another process might own access to one of those grids, thus this process must wait until access is granted to it. It is expected that the access to the model will be the biggest limitation to parallel performance.

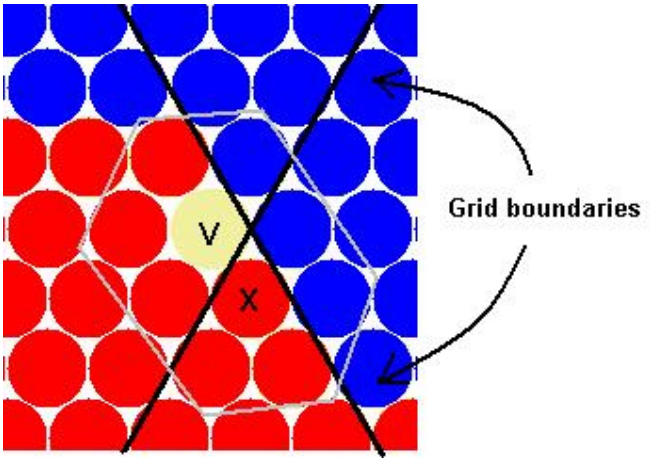


Figure 23: *When atom (X) makes a jump to vacancy (V), inspecting the neighbourhood of the two sites will cause four different grids to be read.*

One idea to reduce this problem would be to allow for several processes to read a grid at the same time. Only when writing will they need exclusive access. The problem with this strategy is that when a process wants to write to a grid that is shared, it must either wait for the other process to finish reading the grid or pre-empt them. Regardless of whether the process waits for or pre-empts the grid, all the other processes potential moves must be invalidated. Thus, a lot of work is lost.

A far better strategy should be to minimize the risk of several processes sharing the same grids. This is done by choosing a good grid selection algorithm and ensuring that as few as possible of the neighbouring grids is needed at any given time. It is also important to have a total number of grids large enough to guarantee close enough independent execution for all processes. This will mean modifying the grid dimensions for the same problem when increasing and, to a lesser extent, decreasing the number of processes to be used. A large number of grids should have a big impact on the degree of concurrency, but may greatly reduce memory performance. There is a trade-off to be made there.

6.2.2 Global variables

Other global data that must be synchronized between processes is the centre of mass for each particle and the counter for each different site type in the model. The code should be altered to ensure that writes to these global variables are bunched together. That way the synchronization is kept at a minimum, but will still be a noticeable limitation.

The last group of global data is the file pointers. The files are rarely accessed and it is expected that the synchronization pose a neglect able limitation.

6.2.3 Vacancy annihilation

As described in section 4.3 grain boundary vacancy movement may cause annihilation. Annihilation is simulated by an entire row of atom moving through the centre of mass of a particle, thus, effectively moving the grain boundary vacancy to the surface of the particle. The process will have to write to a grid that is on the opposite side of the particle, but before finding that grid it must access other grids inside the particle when calculating the Bresenham line [13]. If one of those grids is being accessed by another process, the annihilation will have to wait.

Waiting poses two problems. Firstly, it limits the parallel performance if the wait causes the process to stop execution. Secondly, it may cause error of correctness, since during the wait the surface where the grain boundary vacancy should end up might be altered and the centre of mass used to approximate the line may move.

A solution to this problem is to allow the process to interrupt the other process that has access to the grid through which the annihilation should take place. The interruption could either lead to the process in charge of the annihilation to take over the access to the grid or the other process may take over execution of the annihilation. The first interruption scheme is fairly simple, but it would force the other process to pre-empt any work being done to the grid. The second scheme is similar to how work is passed between processes in a distributed system [15]. It is more desirable because none of the processes will have to delay, but demands a larger part of the annihilation

algorithm to be rewritten. The major part of the rewrite is to allow the events at either side of particle to be executed separately.

6.2.4 Pore pinch revisited

There are more problems with the pore pinch algorithm when dealing with the grid model and parallelization. Since the pore pinch algorithm is based on a *depth-first search* [1], it has a tendency to follow vacancies in a straight line. This leads to a high probability of leaving the current grid and having to write to neighbouring grids. The write is caused by the flagging of sites to mark them as visited. A vacancy analyzed by pore pinch will be stored as a temporary type to ensure that it is not investigated again during the same pore pinch calculation. Writing to a neighbouring grid forces that thread to gain exclusive access to it and it might also result in the grid having to be restored.

To lower the probability of a pinch pore algorithm to leave the current grid a *breadth-first search* [1] should be used instead of a *depth-first search*. A fairly simple example of the difference can be seen in Figure 24 where the *depth-first search* moves over the boundary of the grid whereas the *breadth-first search* stays within the grid. Another benefit of *breadth-first search* is that there is no longer any need for a recursive function within the algorithm. A recursive function causes many jumps in a code and also a lot of stack variables to be allocated, none of which are good traits. The width first search only needs some way to store which sites to investigate for each depth; luckily the original search algorithm already has a position vector which can be used for this. The similarities of the two different implementations can be seen in Figure 25.

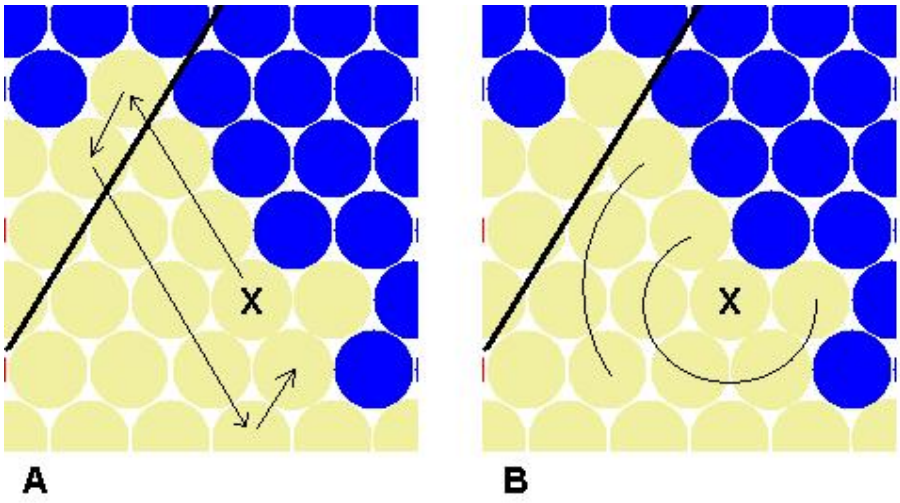


Figure 24: *The pore pinch algorithm searching for an area of 9 vacancies as (A) depth-first and (B) breadht-first.*

```
depth_first( site ):
ADD site TO list
LOOP each neighbour
  IF neighbour IS vacancy
  AND neighbour NOT IN list
    depth_first( site )
  END IF
END LOOP
```

```
Breadth_first( site ):
ADD site TO list
LOOP all NEW sites IN list
  LOOP each neighbour
    IF neighbour IS vacancy
    AND neighbour NOT IN list
      ADD (NEW) site TO list
    END IF
  END LOOP
END LOOP
```

Figure 25: *Pseudo code of depth-first search and breadth-first search versions of the pore pinch algorithm.*

The combination of dealing with the two issues, checking pinch off for every surface vacancy in neighbourhood (as described in section 6.1.4) and having to write to neighbouring grids, is expected to have a very positive effect on the parallel performance. The time spent for each move is much shorter, that alone being a reason for decreased run time, and in turn there will be less and shorter periods of exclusive access to grids.

6.3 The parallel algorithm

After considering the improvements to the sequential part of the program and also the factors causing limitation to the possible achievement in performance, the design phase of the parallel code can start. The possible parallelization is governed by the different independent parts of the code that will allow for concurrent computation. Whenever two or more processors are performing work on dependent parts of the code, one must be granted exclusive access to that particular part. Allowing for mutual access to such a part of the code would mean the execution risks ending up in an inconsistent state.

The main simulation code centres on iterating through all moveable vacancies and validating potential moves through the Monte Carlo technique. The parallel tasks to be assigned to different processes are therefore independent vacancies or grids. The vacancy, or a vacancy in a grid, attempts to move. Independence between the tasks assures that several moves can be considered, validated and performed in parallel. Parallelization of the program can then be cut down into two major fields: the model and the vacancy list.

The model makes up the case of how the grids or vacancies should be distributed among the processes, in order to minimize the potential dependent moves. Load balancing becomes an issue here as well. One process might receive more work than other, forcing the other processes to wait. The distribution will be able to balance the tasks by either static or dynamic partitioning.

A cleverly designed vacancy list can further assure independence of vacancies within a grid and the neighbouring sites necessary to validate any potential move. Thus, allowing for a higher degree of concurrency.

Last, we have to consider how to ensure that dependent tasks are performed by one and only one process at a time. But also that other processes waiting for access to critical section of the code will have to wait as short time as possible.

6.3.1 Task selection

Section 6.2.1 describes the need of a good grid selection algorithm and even if the individual tasks distributed among the processes are the movement of single vacancies, grid usage must be effective and the work must be balanced between the processes. This might be the single most important issue of the parallelization. Therefore, a number of various schemes are considered in this section.

One grid selection strategy would be to divide the grids among the processes. Either uniformly [4] or using dynamic load balancing [15]. Regardless of which of these two schemes is used the execution of all processes has to wait for the slowest one at each Monte Carlo step (MCS). A process may not start over on its part of the model until all other processes have completed their step. For a uniform distribution this would mean that a process assigned to grids with few vacancies will spend most its time waiting for other processes. A dynamic load balancing scheme would ensure that such a process received grids from the slower processes, but at the cost of the added balancing computation and complexity, which might have to be performed by a separate scheduler process. There is still need for synchronization between the processes as a move might potentially cross the boundary between two processes' areas. One way to deal with the boundary crossing would be to allow the neighbouring processes to share the boundary strip, like a no-man's-land [4]. Since a move taking place at the boundary would only need to know a depth of two sites across the boundary (Figure 26), the strip would be small. Another solution would be to use data request and data deliver messages that interrupts the neighbouring process causing it to act on the message [15]. The messages would either tell the process to retrieve site data or to execute part of a move (Figure 27).

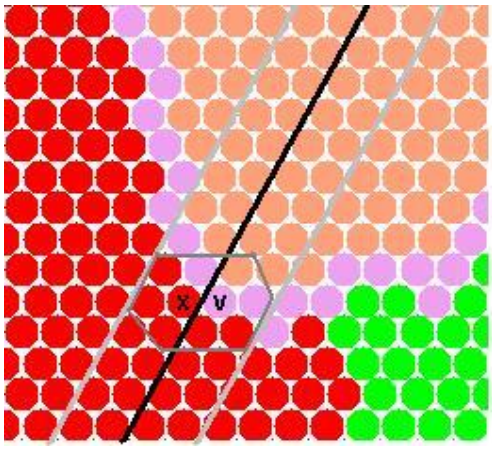


Figure 26: *The neighbourhood of any two sites on direct opposite sides of a partion boundary are limited to a strip of four sites in breadth with the actual border in the centre.*

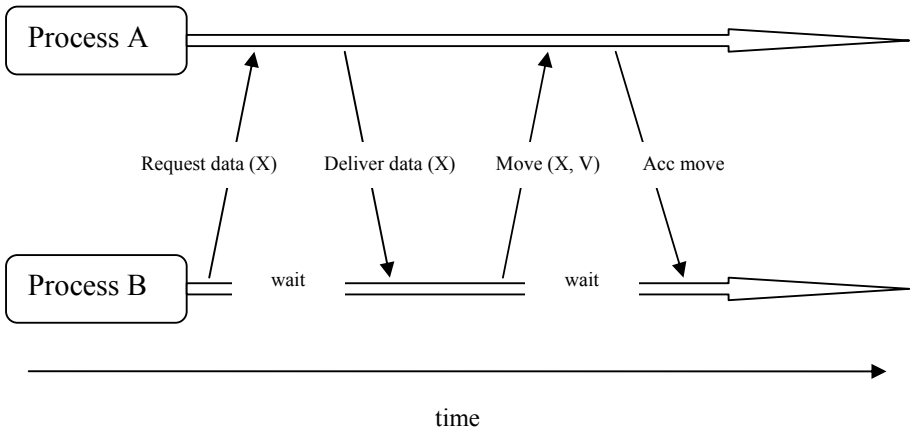


Figure 27: *Message passing between process A and B when dealing with the move of atom X to vacancy site V (***Figure 26***).*

The two strategies seem more usable for distributed processors because each process only needs to know a subset of the problem. So, whenever a move is performed the other processes only need to be informed of any updates to the global variables, their grids will still be valid. This greatly reduces the amount of data transferred compare to allowing any process to run any given grid.

Another strategy uses a master process which chooses guaranteed independent grids and send them to slave processes [25]. An independent grid would have all its nine neighbouring grids free and also, in the case of a grain boundary, the grids of any potential annihilation path need to be free. The advantage is that there is no need for synchronization between the processes. But there may not be enough independent grids to keep all processes busy and the time spent finding independent grids might cause processes to have to wait for work.

Many grids contain no vacancies and those that do tend to be neighbours, following the surface of the particle. In light of this, finding independent grids and still allow for scaleable performance when adding more processes might be hard, if not impossible.

A stepping stone before the next strategy is for the processes to fetch individual moves [7]. A master process would select a vacancy and a random direction then pass that on to a slave process to validate the move. The move could then be performed by either the master process or the slave process. Allowing the master process to perform the move means that the slave process is only reading data, thus it doesn't need to be synchronized. If two moves are done using the same data, i.e. the same neighbouring sites, the master would select one of them and discard the other.

It is expected that this scheme would perform very poorly since the moves aren't guaranteed to be within the grid the slave last worked on. So, there will be a lot of grids read over and over again for each Monte Carlo step. On the other hand the scheme is easy to implement since it is roughly equivalent to the sequential code.

The last strategy is based on the processes fetching grids to perform work on. The grids reside on a common queue [19]. One Monte Carlo step is finished when every

grid has been fetched from the queue. The queue will then be made ready for the next step. Effectively, this means that the queue in question should be circular. In order to make the selected grids as independent as possible, the grid indexes need to be scrambled across the queue. This could either be done randomly or statically by for instance taking every third index and skipping a couple of rows (Figure 28). The reason for suggesting every third index is because for each highlighted grid in Figure 28A the neighbouring nine grids are different from any other highlighted grid's neighbours. Using every second index would result in shared neighbourhoods and using every fourth index would have the selection returning to the first line of grids earlier and thus result in more potential grid locks. A third option would be to reorder conflicting grids positions in the queue. Whenever a grid selected from the queue is not able to be run it would be moved to another position. Eventually the grids would then be ordered optimally within the queue, a kind of dynamic load balancing.

0	1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43
44	45	46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63	64	65
66	67	68	69	70	71	72	73	74	75	76
77	78	79	80	81	82	83	84	85	86	87
88	89	90	91	92	93	94	95	96	97	98

A

0	1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43
44	45	46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63	64	65
66	67	68	69	70	71	72	73	74	75	76
77	78	79	80	81	82	83	84	85	86	87
88	89	90	91	92	93	94	95	96	97	98

B

Figure 28: *The grid selection in order to reduce potential grid locks. (A) First run starting on first row and picking every third grid on every third row. (B) And in second run starting from the second index on the first row. And so on, until all grids have been considered.*

Further improvements can be made to the queue by using a Sliding Window algorithm. Sliding window is used for network protocols [14]. The general idea, shown in Figure 29, is that a sender has a window it can use to store several data messages. As long as that window is not full it may keep sending messages. If the window becomes full it must wait for the receiver to catch up and acknowledge some of the outstanding data messages. For the grid model the window size is the same as the total number of grids and the frames being sent are the actual grids. The grids are ordered on the transmission queue according to the grid selection algorithm being used. If a grid is being run and another process tries to access it, i.e. the process wants to run the grid for the next Monte Carlo step, the process has to wait. As long as the end of the window isn't reached the processes can keep running grids. The advantage

of this is that the processes won't have to stop at the end of every Monte Carlo step. This means that the processes will spend less time waiting and they will also be spread further throughout the model.

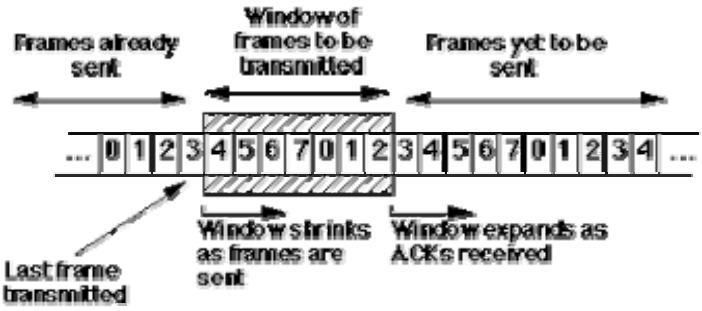


Figure 29: *The Sliding Window data transfer protocol.* [24]

This strategy is only effectively useable for a shared-memory system, because any given grid could during the simulation be used by all processes. If used in distributed system, every process would have to be able to access any grid, thus resulting in many and large network transmissions. But as a shared-memory strategy it is very interesting. Firstly, there are only worker processes, which both decrease the complexity and assure better CPU usage. Secondly, work balancing is done automatically with faster processes picking more grids from the queue than the slower ones. The only real limitation would be the sliding window. The window must have enough grids to ensure that it doesn't keep getting filled up, i.e. the model must be split into enough grids to keep the processes busy (as mentioned in Section 6.2.1).

6.3.2 Vacancy list(s)

Once the task selection routine is good, there is further room to improve the independence of the tasks. The sites involved in validating a potential move are limited to the two sites changing position's neighbourhoods, as highlighted by the hexagon in Figure 26. The two sites are next to each other which add up in a total of ten sites that need to be investigated. Once the move has been validated and if it is to take place, more sites will become involved in the process. But focusing on just validating the moves and their high degree of locality should make it possible to greatly decrease the amount of data a process needs to have exclusive access to.

Let's first review the case of using a global vacancy list as in the original simulation code. In that case the entire grid selection process becomes useless. Of course, the list could be sorted and sequential access to it rather than random access would allow for a rather crude method of controlling which grids are used. The time needed to resort the list for every update and also the complexity of ensuring not only grouping of the vacancies belonging to the same grid but also keeping the processes from using neighbouring grids is appalling. Another disadvantage of the global vacancy list is that further synchronization would be needed to ensure exclusive access when updating it.

Therefore in order to utilize both an effective grid selection method and take advantage of the locality of data when validating any given move, the design of the vacancy list has to be changed.

The first improvement is the use of grid local vacancy lists. Thus a process will stay inside the same grid for all the vacancies of that grid. But even when each grid has its own vacancy list, the potential grids that is needed to validate a sequence of moves within the list is all the immediate eight neighbouring grids. Either every move will result in its neighbourhood being investigated and the necessary grids being locked or the process will have to keep all the eight grids locked (Figure 30), in order to ensure that no validated move has to wait to be performed. In the first case a lot of time is spent determining the grids needed to validate a move and synchronizing grid access. In the second case the number of grids in the model has to be large, and needs to grow significally with each process added to the simulation and even with a fair amount of grids in the model the risk of grid lock is unacceptable.

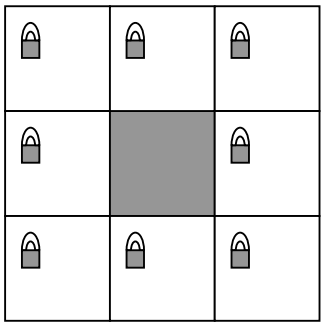


Figure 30: *The highlighted grid's neighbourhood, where all grids have to be locked.*

To really take advantage of the data locality in the validation process and reduce need for synchronization the vacancy list is divided into four sections (Figure 31). When using this shy version of *divide-and-conquer* [1], the dealing with one subset of the vacancy list results in only the current grid and three other grids to be locked. That is the three grids needed for the move of a vacancy in the corner of the grid (Figure 23). Each process will always maintain exclusive access to four grids. Not only is this a great improvement compared to having nine grids locked down, but also is the time having exclusive access to neighbouring grids reduce, since the grids needed for each subset are different.

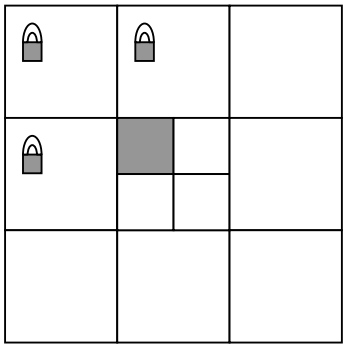


Figure 31: *A grid divided into four subsets, and the three neighbouring grids needed to be locked by the highlighted subset.*

An even further improvement can be drawn from the fact that it is only the two sites closest to the grid boundary that needs the neighbouring grid for validation (Figure 26). Using more of the *divide-and-conquer* approach, the vacancy list is divided into eight subsets, as shown in Figure 32. Only the corner subsets force the process to gain exclusive access to three neighbouring grids. The central part will in fact only require the current grid to be locked. The disadvantage of eight vacancy lists compared to four is the increased complexity, resulting in more updates when moving a vacancy and either higher amount of overhead saved per grid or more time needed when reading a grid to create the lists.

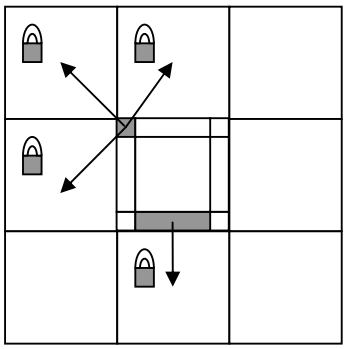


Figure 32: *The locked neighbours for the two types of highlighted border subsets; the arrows indicating which locked grid belongs to which subset. The centre subset does not need any neighbouring grids locked.*

6.3.3 Grid locks

Now that the grid and vacancy methods have been examined in order to find good parallel behaviour, it still leaves the possibility that two processes might want to access the same grid at the same time. Whereas the previous two sections dealt with minimizing the risk of a grid lock situation, this section will show various ways of handling processes when the situation arise.

Firstly, when dealing with a grid a process must gain exclusive access to it. The grid locking mechanism must therefore satisfy the following four properties [3]:

1. **Mutual Exclusion**. At most one process at a time is executing the grid.
2. **Absence of Deadlock**. If two or more processes are trying to access the same grid, at least one will succeed.
3. **Absence of Unnecessary Delay**. If a grid is free, a process is not prevented from accessing it.
4. **Eventual Entry**. A process attempting to gain access to a grid will eventually succeed.

Each grid will have a lock and the process owning a lock will have exclusive access to that grid. If another process wants to access a grid that is locked, a grid lock situation occurs which must be resolved.

Depending on the status of the process different strategies are possible for solving a grid lock situation, but the easiest and most robust solution is waiting. The process will be paused and put on a list to wait for access to the grid. When the grid is unlocked, a process from the list is granted access to the grid. Most often the processes will be picked from the list in a FIFO order (*first-in, first-out*), i.e. the list is a queue. But there are some cases where a process needs to ensure access to more than one grid at the same time. In that case, locking one grid at a time might cause a deadlock. For instance, if two process need access to the same two grids to work on their vacancy lists, they might gain access to one grid each and would then have to wait forever for the second grid. This deadlock situation can be resolved by using the *spin lock* strategy. A process sits in a tight loop examining all locks of the grids it needs. Only when all the locks are free does it attempt to gain access to them.

If the process only needs access to the grid in order to perform vacancy annihilation, the access is brief compared to a full Monte Carlo run of the vacancies in the grid. In that case the process should not be forced to wait for the access. Either the access to the grid could be taken by the process, pausing the process that currently owns the grid's lock, so called *pre-emption*. Or the work could be passed on to the owning process. Allowing work to be passed on demands rewriting parts of the code to find separate entities. Passing on work might also cause the model to temporarily be in an incorrect state. For example, one process passes on the moving of the annihilating vacancy to another process, but performs the atom movement in the grain boundary. At that stage, before the other process finalizes the annihilation movement, the model would temporarily have more atoms than at the start, which is wrong. *Pre-emption* on the other hand, might result in the process losing access to its grid and that might force it to cancel whatever it was working on. It will also cause the process to have to re-read the grid once it has regained access to it. The time lost in this fashion will have to be recaptured by the process that *pre-empts* the lock instead of waiting.

Another situation where putting process to sleep until the grid is freed might not be the best option, is when there is other work that can be done. Discarding the validation or performing of a move because of a grid lock is unacceptable, since it would mean lost work and conflict with the Monte Carlo algorithm. But there are places where a process would be able to find similar but non-conflicting work to do. For instance, if a grid has several vacancy lists, as described in previous section, the only demand on the process is to evaluate each list once. The order of evaluation is not important. So, if evaluation of a list would result in a wait because one or more of

the needed neighbouring grids are locked, another vacancy list could be selected. The work on the locking vacancy list would be done later; hopefully by then the necessary grids will be free.

6.4 Summary

After analyzing the flaws of the original version and the limitation of possible concurrency, the parallel algorithm was decided to use a grid based approach. The processes tasks are to consider one grid at a time and iterate through the vacancies, attempting to move them. The grids reside on a shared list, a so called "*bag of tasks*" [3], from which the process fetch their tasks. This allows for dynamic load balancing, with faster processes fetching more work, as long as there is more work. To further increase the independence of the tasks, the grids can be distributed randomly, statically or dynamically on the shared list. The grids will also be iterated through in subsections, allowing for fewer neighbouring grids to potentially be involved in validating a move. Together, the *bag of task* and working on subsets of the grid are expected to allow for a high degree of concurrency and should scale well if there are enough grids and access to other dependent part of the code, such as global variables, is not too extensive.

7. Implementation

For clarity, the implementation description has been divided into three sections. The first section deals with implementations to the simulation code that had nothing to do with the parallel algorithm. The next section deals with the implementation of the memory model. As such, the two first section's changes were implemented without having to deal with the problems that come with concurrent execution. The parallel implementation and its four phases are described in the third, and final, section.

7.1 Sequential algorithm updates

There were several suggested modifications to the original simulation code made in the previous section. But before making any additions, the entire code was restructured. The restructuring phase played a vital role in reading and understanding the code, and to clear the path towards future modifications. It also allowed for a clearer plan as to what can and cannot be done, the limitation of keeping the core simulation algorithm intact. The restructuring further ensured a separation of different logical part of the code, so that modification could be kept fairly local.

Once the code had been restructured the main objective was to implement error detection instructions. Any addition, regardless of how small, must still maintain the correctness of the simulation process. By the addition of instructions to guard against failure the transitions of the code from one version to another was kept much more smoothly.

At last the implementations suggested during the design phase were implemented. Updating the vacancy list proved to be a matter of surprisingly few modifications and resulted in making the code look more logical and easier to follow. A disadvantage is that after the vacancy list had been altered, the original code and the new improved sequential version behave differently for the same random seed. This forces the code to completely rely on the detection instructions for correctness.

Another improvement, that caused better performance than expected, was to validate a move without actually performing it. The performance boost became clearer after profiling of the original code. The profile proved that reverting an invalidated move was in fact responsible for roughly 1% of the total execution time (see Appendix A).

The restructuring phase had unified the data initialization and simulation codes model variables making the implementation of abstract file read and write routines much easier.

In total, the updates made to the original version altered roughly 5% of the code.

The last part of the sequential implementation phase was to improve the data initialization algorithm. The three different methods, mentioned in Section 5.3, that only consider the surface of each particle were implemented. Of the three algorithms only the last one, the *surface walker algorithm*, was correct. This due to the fact that the particles weren't exact circles in the hexagon pattern and that adding one to the radius wasn't enough to find the correct number of surface vacancies.

7.2 Memory model implementation

Next implementation phase was to alter the memory model. This was done before creating the parallel version, since the memory model should behave the same for both parallel and sequential execution. Thus, the correctness of the new memory model could be proven without having added the complexity of parallel processing. In this phase there were two very different versions created. The first version was based on minimizing the total memory usage, which later proved to cause the execution time to be unacceptable. Therefore, a second version was implemented, which was based more on speed rather than low memory usage.

Before talking about the two different memory model versions there are some changes to the code that they have in common. This is, perhaps quite obviously, how data is read from and written to the model. The original code directly uses the two dimensional model array, but for grids there are a few things that must be determined before anything can be done. The correct grid must be loaded into memory and it must also be uncompressed. If a new grid has to be loaded into memory, the grid it replaces might have to be saved and compressed to ensure that any change to it is accounted for.

In both of the versions the main part of the model consists of a vector containing pointers to each grid's structures. The vector fulfils Equation 8 in Section 5.2 allowing for easy access to any grid. Because there are a lot of grid accesses having to search the vector for a grid (Equation 5) proved to consume too much time. The actual grid structure contains all the important data, such as the type of the grid (section 5.1) and pointer to grid contents.

The bulk and free areas contain the same sites. In the case of a free grid it is either all free space or pore sites whereas the bulk grid contains only atoms of one particle. Therefore there is no need to save any contents along with these grids. Instead a flag is used to indicate which site type the grid of type free or bulk contains. There is the special odd bulk grid which will have one or more bulk holes, in that case the locations of the holes will be stored on a list.

The first version involves two major parts. First, the vacancy list is created every time a grid is selected for execution. This is separate from the grid accessing mechanism, since a grid that is being access for anything but execution doesn't need any vacancy list. Second, surface areas are compressed with the CRS and CCS schemes. This proved, as expected, to be extremely complex. Especially determining which side of a surface belongs to the particle.

In the second version the compression of surface grids was completely discarded and each grid had its vacancy list saved with it. The vacancy list was first set at a fixed size, causing fairly poor memory usage. But varied size lists, on the other hand, resulted in a lot of allocations and deallocations. By allocating the vacancy lists in chunks, instead of for one vacancy at a time, the number of memory calls were greatly reduce. In fact, the chunking was enough to bridge the allocation time gap between the statically and dynamically sized lists.

The bulk and free areas are treated identically in the two versions, except for the bulk vacancies. In the first version, a separate list has to be saved along with the bulk grids containing the bulk holes location. This is solved by using the contents pointer of the grid structure as a pointer to the list, but it demands special code when handling the bulk areas with vacancies. On the other hand, there is no need to construct a vacancy list every time a bulk grid of this type is accessed, because the list can be used directly. In the second version there's no need for special treatment, the bulk vacancy list is just stored as any normal vacancy list. This is something that shows another benefit of saving the vacancy list with the grids.

There are a few more bits and pieces that are similar for both versions and deserve some mentioning. When reading from a free or bulk area grid, there is no need to decompress the data into a matrix. Both free and bulk areas only contain one type of site, except for the odd bulk holes which can easily be checked without decompression. Unfortunately, in the first version the surface areas must always be decompressed, even just to read them. Writing to a grid of type free or bulk area, or surface area in the first version, will force the grid to be decompressed into a matrix, and that matrix would need to be allocated. To reduce the number of system calls for memory a pool of sub matrices is created. When writing to an area that would normally cause a matrix to be allocated, the needed memory is instead fetched from the pool. Only if the pool is empty will the system be asked to allocate more memory.

As mentioned earlier, because of the memory usage being of lesser importance than the execution time, the second model version was chosen for the parallel implementation.

7.3 Parallel algorithm implementation

The completely updated sequential version of the code was a good base for the parallel versions. The description of the implementation is in the second part of this section. First follows a brief background of the pthreads library which was used to allow parallelization of the code.

7.3.1 Pthreads library

The pthreads library defines a standard set of C routines for multithreaded programming. Pthreads is short for Portable Operating System Interface (POSIX) threads which is an IEEE standard. PASC (Portable Application Standards Committee) [21] is the group that created and continuously develops the POSIX standards. The library ensures thread management and synchronization which is portable between different systems.

From now on the worker processes of the parallel sintering simulation algorithm will be referred to as threads.

7.3.2 The parallel implementation phase

The first phase of the parallel implementation consisted of putting threads in charge of the Monte Carlo steps. The necessary parts had to be taken from the main program and put into the thread's work load. The implementation was based on the last strategy of the design in section 6.3.1. I.e. the work load consisted of each thread picking grids for execution from a shared queue. The grids were then subjected to the

same treatment as the entire model is in the original program. Vacancies within the grid are randomly selected and a move is attempted in a random direction. The reason for implementing this strategy is because it is the most suitable for a shared-memory system, also the queue automatically ensures load balancing between the threads and reading and writing grids are kept at a minimum.

As expected, forcing the threads to wait at the end of Monte Carlo step took a substantial amount of time. Not only did the waiting threads cause the added time, but also the fact that all the threads started picking grids off the queue at the same time and at the same place. Therefore, the sliding window idea of section 6.3.1 was implemented resulting in the threads never having to stop at end of the MCS. This puts a lower bound on the number of grids; there must be enough grids so that the threads don't hit the end of the sliding window. If the number of grids is over that bound gaining access to the shared queue is the only time limiting factor left when ensuring that threads have grids to work on. This factor puts an upper bound on the number of grids. Using too many grids will cause an increased number of accesses to the shared queue, and thus, more situations where a thread has to wait. Using the sliding window meant that a status indicator had to be added to all grids. If a grid has its status set as running, another thread may not try to pick it from the list, nor may a thread run any other grid in its immediate neighbourhood.

Moving on to the actual shared queue, it was first simply sorted by grid index. This meant that the threads would pick grids next to each other, resulting in many grid locks. Of the three queue arrangements suggested in section 6.3.1, the static sorting described in Figure 28 was deemed best. The random approach was not much better than the first grid index sorted implementation. Two neighbouring grids could still randomly be placed next to each other in the queue, and cause a grid lock situation. The dynamic sorting approach is, on the other hand, near to perfect but was discarded because of the time spent updating the queue.

The second implementation phase investigated the further reduction of grid lock situations based on the design ideas of section 6.3.2. Since the grid selection strategy chosen already forced the code to commit to grid local vacancy lists instead of a global vacancy list, that design issue was no longer of interest. The locking of nine neighbouring grids, as seen in Figure 30, proved to cause a lot of grid lock situation, even with the improved grid selection algorithm in place. Therefore the vacancy list was mapped into four squares, each representing a corner of the grid Figure 31. The Monte Carlo algorithm no longer randomly picked vacancies within the grid, but instead was run separately for each vacancy list square. The change from a thread locking nine neighbouring grids down to three neighbouring grids proved to nearly eliminate all grid lock situations. In fact, the elimination is effective enough that there is no need to consider breaking up the grids into smaller parts.

From the first phase, any grid lock situation was resolved by pausing the thread that failed to obtain a lock, and waiting for it to be unlocked. The reordering of vacancy list sequence and vacancy annihilation unlocking mechanism described in section 6.3.3 were now considered. Trying another vacancy list, when a grid lock situation has arisen, only made the program run slower. This is due to the fact that the division of the grid into four parts already nearly eliminated all such grid lock situation. Also, the grid lock situation is resolved quickly, since the thread owning the needed lock

only needs to perform work on a quarter of the grid before unlocking the neighbourhood. The complexity of adding any of the vacancy annihilation grid lock solutions was, in combination with the low number of annihilation that occurs, grounds for not trying to implement the mechanisms. Also, when following the annihilation line through the bulk of a particle, those grids should be free since they contain no vacancies. Therefore the only problem is a grid locked at the opposite surface of the annihilating grain boundary vacancy. If the grid is locked because of being a neighbour to a running grid, the grid lock situation will be resolved quickly. The real issue is when the grid causing the lock situation is an actual running grid. But, because of the relatively few annihilations performed during a Monte Carlo step, implementing any of the grid lock mechanisms is expected to give very limited improvements (if any).

Before the third phase the code went through an extensive peer review. Generally, the third phase then involved the code receiving several face lifts in the shape of extended commentary, added type definitions and structures, and more constants. This ensured that the code is easier to understand and modify. Most importantly, the third phase removed all, up to this date, known bugs.

The fourth, and last, phase was perhaps the most important phase. Synchronization between the threads had to be reduced; this meant identifying the parts of the code resulting in bottlenecks. Once those parts were identified, they had to be remade as best possible to allow for better concurrency. Because of the limitation of not being allowed to alter the core simulation algorithm, this was fairly difficult.

In the end, the implementation resulted in doubling the amount of code found in the updated original version (Section 7.1), leaving almost only the Monte Carlo validation of a move unaltered.

8. Evaluation

This chapter describes evaluation of the proposed parallel version of the simulation algorithm. The chapter consists of four sections. The first section contains a brief description of the tools used for evaluation. The second part deals with the correctness of the new parallel version of the simulation algorithm. The correctness is based on the original algorithm data and the model's microstructure. In the third section the new memory model is evaluated. The memory needed for different sized models and grid dimensions is analyzed and also compared with the memory demand in the original version. Finally, performance evaluation of the parallel simulation program is presented in the last section. Not only will the execution time compared to the original version be compared, but other important factors such as performance for different model, grid sizes and number of threads. These factors show the scalability of the program.

8.1 Tools

GNU gprof [10] is one important tool that was used to better understand which parts of the program are critical for increased performance. Gprof is used to profile programs and an example of such a profile can be seen in Appendix A.

The UNIX command time is not a tool but a shell script. It measures the real life time a program spent executing, and also the CPU time used by the program in both user and system mode. It is important to note that for parallel execution the CPU time will be the sum of each individual threads CPU usage.

8.2 Correctness

Assessing the correctness of the new parallel version is perhaps the most important part of the evaluation. It must have maintained the same basic sintering model as the original program had. To ensure that the sintering simulation is still accurate, the visual result and the data outputs of the original and the parallel version is compared.

The visual evaluation consists of making sure that the particles roughly have the same shape at the end of the simulation. The data output is the porosity and rugosity of the model, as described in Section 4.3, for different Monte Carlo steps. These values are compared between the two program versions for each time interval.

Table 3: *The two different models used for evaluation.*

Model	Particles	Radius	Atoms	Vacancies
Small	4	64	59374	1607
Large	4	128	237619	3316

It should be noted that because of the limitations in the original simulation program the particle models used for evaluation are small. The comparisons are based on the two memory models listed in Table 3.

8.2.1 Visual

The visualization is based on an in-built graphics generator of the original program. Because of the changes in the memory model, the graphics generator is not part of the parallel version. Instead, a converter was written which transform a file containing the new type of model to the old type. The resulting file can then be run with the original program and its graphics generator.

The converter also allows the two different versions to run with the exact same input. And it makes possible comparisons between the new and the original data initializations methods. Based on such comparisons, the two shape based data initialization algorithm were proven incorrect, as described in the last part of section 7.1.

The simulation generated images from two different sized models can be seen in Figure 33 and Figure 34. Each figure has two images: one from the original version and one from the parallel version. The outline of the particles in both figures is almost identical between the two versions' images. There is also a similarity between the outline of the particles between the two figures. This is important, since the particle mass should strive to achieve a form which has a low surface energy. Inside the shape, each individual particle is slightly different in the images. This is due to the randomness of the atom movements in the algorithm, especially along the border between the particles. Because the end results of the simulations are similar and in parts identical, the microstructure and its behaviour from the original version is maintained in the parallel version.

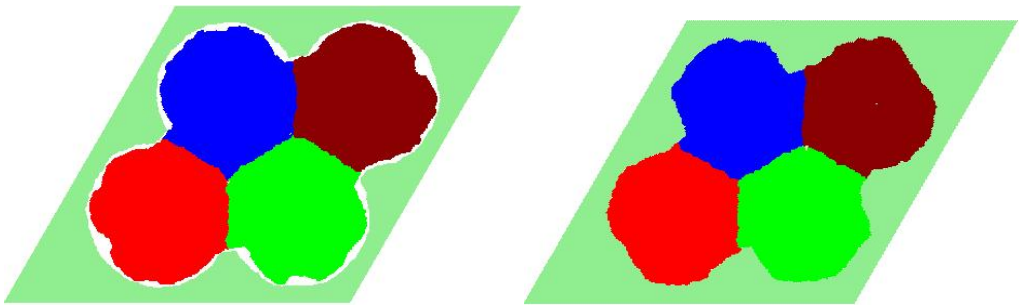


Figure 33: *The particle structure after successful simulation of the small model (*Table 3*) as generated by the original program (left) and the parallel program (right).*

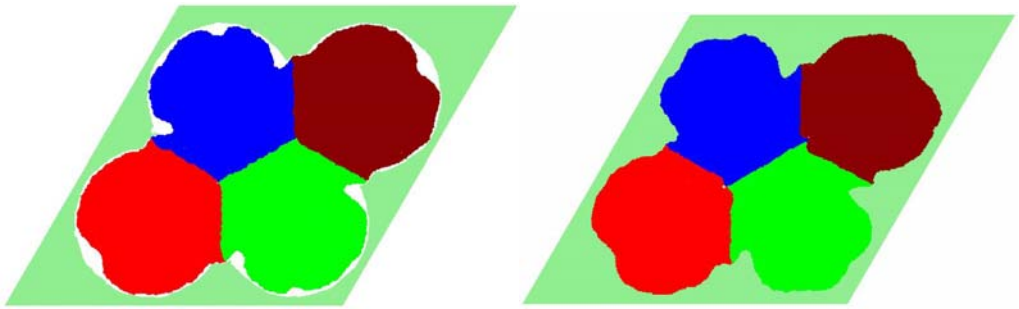


Figure 34: *The particle structure after successful simulation of the large model (*Table 3*) as generated by the original program (left) and the parallel program (right).*

There is one difference between the images, which is not part of the actual particle mass, and that is the white outline that can be seen around the particles of the original version. This phenomenon is due to the incorrect update of surface vacancies and it is described in section 6.1.1. Thus, it has no importance concerning the correctness of the parallel version.

8.2.2 Porosity and rugosity

The visualization has proven that the sintering algorithm is maintained correctly in the parallel version, at least at the very end of the simulation. To prove that it also behaves correctly during the course of the simulation the output data will need to be compared. The porosity and rugosity is calculated at different time intervals of a sintering simulation. The underlying equations which the calculations are based on are described in section 4.3. Even between simulation runs the values at each interval is different, depending on both the input model and the randomness of the algorithm. Therefore to get a more accurate comparison the average from 10 and 5 different simulations is used.

Figure 35 shows the results gained when using the small model (Table 3). As can be seen both the porosity and rugosity values are similar. That is especially true for the first two million Monte Carlo steps where the absolute error is within 0.05 (Figure 36). At the last part of the simulation process the parallel program diverges to zero at a much faster rate than the original program. This is to be expected, because the changes to the update of the vacancy list described in section 6.1.1. The parallel code will have a very small list in the end, and therefore quickly annihilate the last pore vacancies. The original code, on the other hand, has a list the size of all the surface vacancies found during the entire process, therefore it takes time before it randomly finds and removes all pore vacancies.

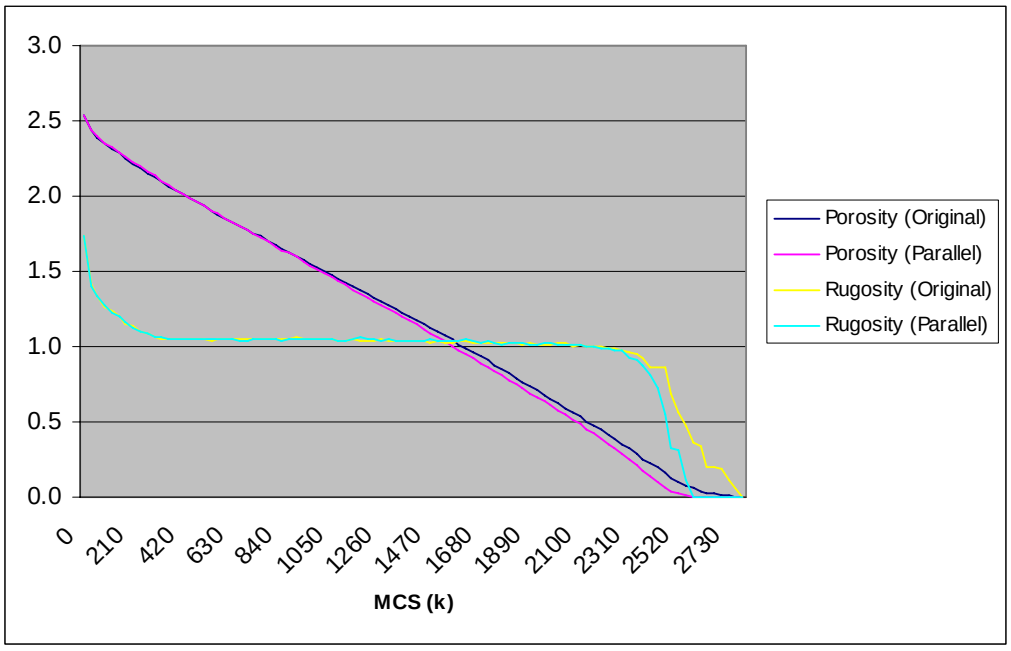


Figure 35: *Plotting the average porosity and rugosity from 10 simulations of both the original and parallel sintering program using the small model (*Table 3*).*

The reduction in Monte Carlo step due to the correct update of the vacancy list becomes more noticeable when moving on to larger models. In Figure 37, the parallel version completes the simulation about two million Monte Carlo steps earlier than the original version. During the earlier parts of the simulations, when the vacancy lists are still roughly the same size, both the porosity and rugosity values are within a neglect able margin of error. Thus, for the problem sizes that the original code can handle, the parallel version behaves correctly.

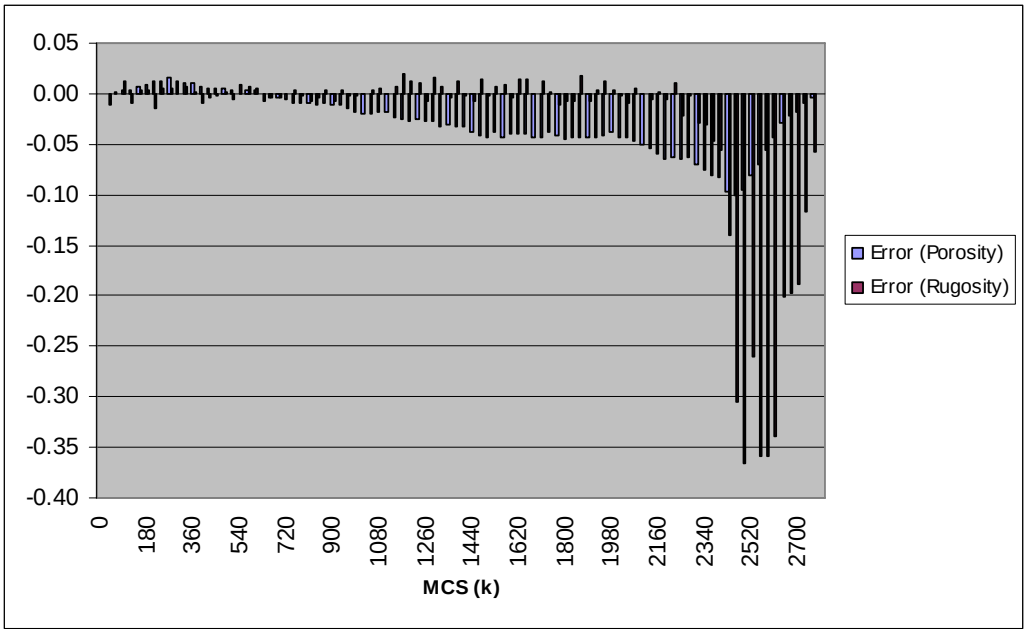


Figure 36: *Plot of the porosity and rugosity difference between the original and parallel simulations for each Monte Carlo step.*

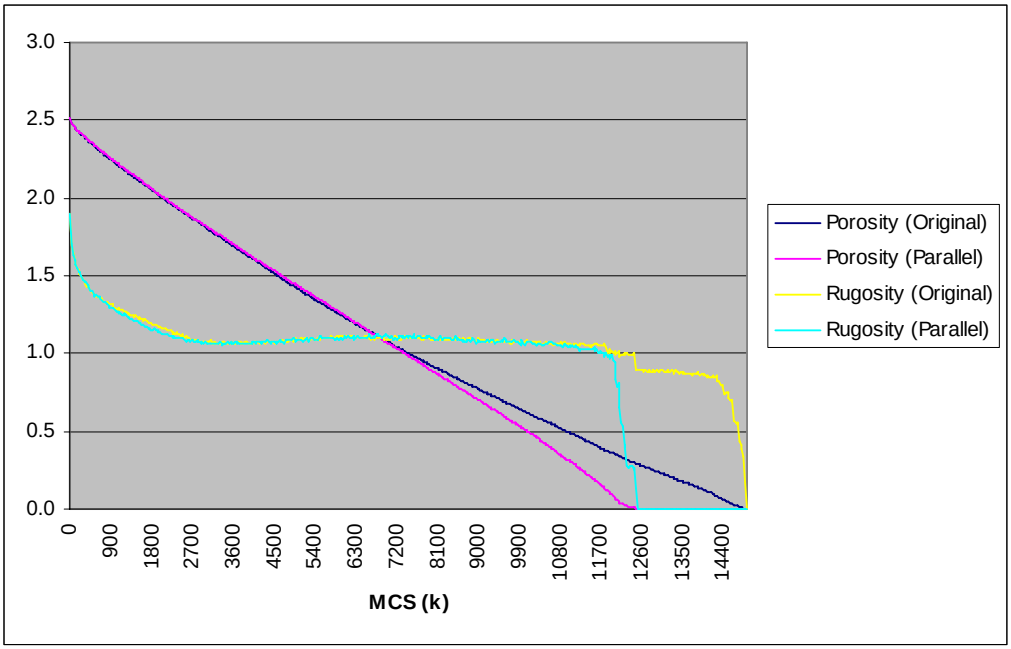


Figure 37: *Plotting the average porosity and rugosity from five simulations of both the original and parallel sintering program using the large model (*Table 3*).*

8.3 Memory usage

Evaluation of the memory usage is not limited to the small sized models of Table 3. Instead, the original program's memory usage can be derived from the model size, which is based on the particle radius. There will be no need to run the original data initiation algorithm, which will allow for larger model sizes than it can really handle. It is also relevant to evaluate the memory usage of the new memory model based on different grid dimensions.

For the small sized models, that the original simulation code can handle, the new memory model doesn't offer much improvement (Figure 38). But already at a radius of 512 atoms the difference starts to show. At a point between the radius of 2048 and 4096 the original model ends up being larger than one gigabyte. When the new memory model passes the 100 megabyte mark, the original model has already gone beyond 100 gigabytes. This proves the grid based model to be extremely successful, even though many of the compression schemes considered during the design were discarded (Section 7.2).

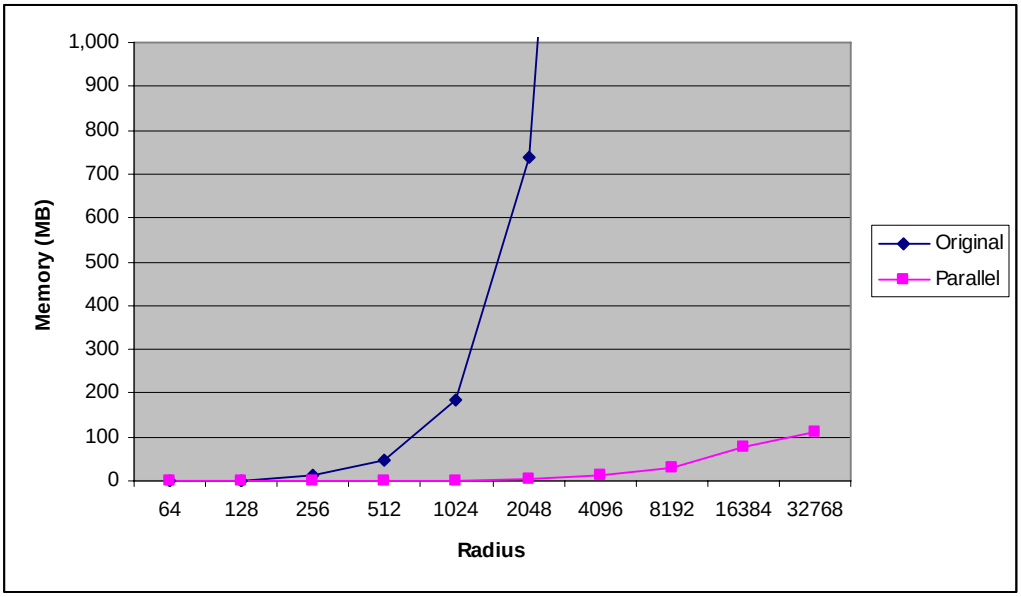


Figure 38: *Memory used by the original and parallel version for models with four particles.*

The grid dimensions play an important part in the size of the model. The compression achieved by the partitioning varies but more important is the difference in overhead size for keeping all the grids stored in a list. The influence of different memory structure on the total memory usage for grid dimensions can be seen in Figure 39.

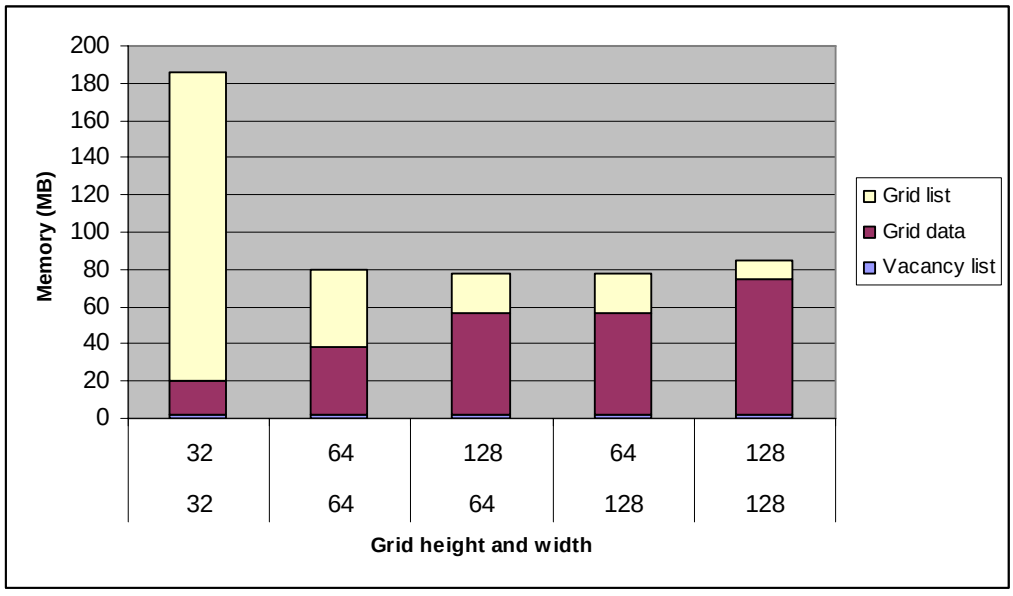


Figure 39: *The memory needed for a model with four particles and 16384 atoms in radius; the memory needed is broken up into the grid list, grid data and vacancy list.*

The vacancy list plays a very small role, because it is dynamic instead of static (section 7.2). Since the chunks that make up the list are relatively small (set to four vacancies per allocation in Figure 39), its size is dependant on the number of vacancies in the model, rather than the grid dimensions. But the vacancy list is also

part of the overhead data. Each grid has a structure containing pointers to four vacancy lists and four vacancy counters. Four of each because of the division of the grid into squares as described in section 6.3.2. The overhead is also due to saving the type of grid (section 5.1), a pointer to the grid's contents and the status of the grid. The status is whether the grid is being executed by a thread or not (section 7.3.2). In total, the overhead is 32 bytes for any type of grid. The last part of Figure 39, the grid data, is the memory needed to store the contents of each grid. Only grids of type pore and surface add to that number. A smaller grid dimension means fewer grids of those types, and also that less memory is needed to store the contents of those grids.

8.4 Parallel performance

Because of the fact that the number of Monte Carlo steps varies between each simulation run, and also varies greatly between the original and parallel version, performance measurements should not be based on execution time of a simulation. In order to achieve a common ground between the original and parallel programs the comparisons are based on CPU time spent per MCS. Furthermore, the measurement is done on the first million Monte Carlo steps, because then the original and parallel algorithms have roughly the same workload. This is based on the fact that over a million MCS the parallel version has a noticeable lower number of vacancies in its workload made apparent by a smaller porosity value (Figure 35).

The parallel performance evaluation was run on two different multiprocessor systems (Table 4). *Agave* is slightly faster than *Altix64*, but it is a shared system where the number of CPUs and memory available cannot be guaranteed. *Altix64* has the advantage of ensuring that all pre-specified resources (such as CPUs and memory) are available throughout the entire program execution. Unfortunately, *Altix64* is heavily used, and it is therefore difficult to run applications needing too many resources. Also, *Altix64* suffered from many forced reboots and system failures due to power loss. Because of this *Altix64* was mainly used for the correctness evaluations in Section 8.2.

Table 4: *List of the two available multiprocessor systems.*

	Agave	Altix64
CPUs	12	64
CPU type	SUN UltraSPARC III	Intel Itanium II
Clock speed	N/A	900 MHZ
RAM memory	24 GB	64 GB
System bus	2.9 GB/s	N/A
Operating System	SUN Solaris 9	SGI ProPac 2.4

As can be seen in Figure 40, the parallel version greatly outperforms the original simulation program. Even at small models the difference is noticeable, although that is mainly because of the sequential based changes made in the parallel version (Section 7.1). Since the measurement is only taken from the first million MCS, the original version can be run with larger models than it normally can handle. For those models, the original program would in reality not ever be able to finish the sintering simulation. The reason for this is ones again the failure to correctly update the vacancy list (Section 6.1.1). Models with particle radii over 250 atoms result in the

original version to eventually have such a big vacancy list, that the actual moveable vacancies are almost never randomly selected. Thus, the simulation will never finish within a reasonable amount of time. It is for the larger models that the parallelization of the algorithm enhances the performance, rather than being a result of the sequential based changes.

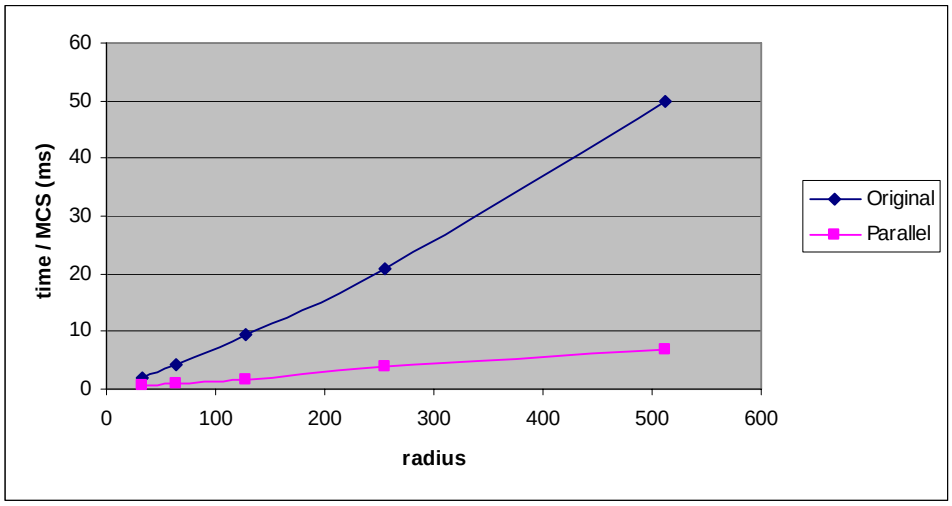


Figure 40: *The performance of the original and parallel versions of the sintering simulation, measured in time per Monte Carlo step.*

Section 7.3.2 mentions an expected upper and lower bound on the parallel performance based on the number of grids in the model and the number of threads used. The Figure 41 shows this expectation to be true where the optimal number of grids for five or more threads is around 1800. When there are more than 1800 grids, the high number of accesses to the common grid list forces the threads to wait for each other. When there are less than 1800 grids, the threads will have to wait because of either reaching the end of the sliding window or grid lock situations.

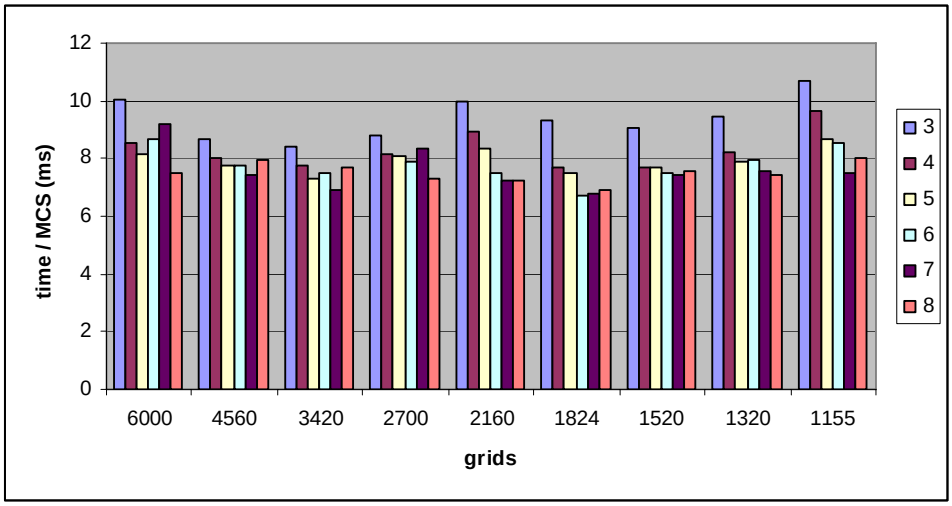


Figure 41: *Performance of the parallel sintering simulation on a model with 512 atoms in each particle radius, for a varied number of grids and threads.*

The sequential changes making up the bulk of the parallel program's performance improvements over the original version when dealing with smaller models can be seen in Figure 42. Unlike in Figure 41, the time per MCS stays fairly constant regardless of the number of threads. The sequential performance is still dependant on the number of grids. This is due to the fact that more grids means lower load times, whereas less and larger grids means a fewer number of grid accesses.

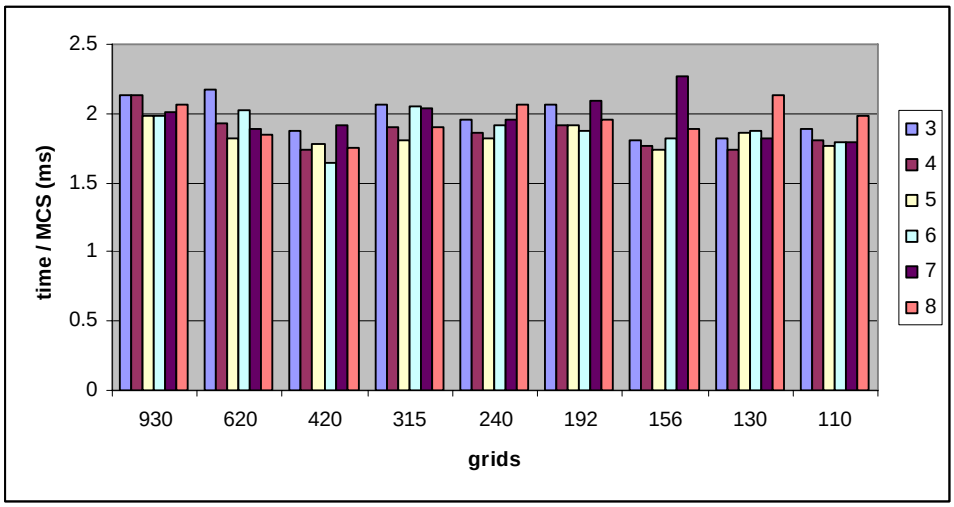


Figure 42: *Same as in* Figure 41*, but the model now has 128 atoms in each particle radius.*

It should be noted that in both Figure 41 and Figure 42 the time per MCS is subject to a slight degree of error because of the randomness in the workload. The figures still paint the general pictures of which grid and thread settings may be optimal for the two radiuses. Another factor to bear in mind is that the figures were based on results from simulations on *Agave* (Table 4) and therefore it cannot be guaranteed that eight, or even seven, CPUs were available during the entire execution of the simulation.

The expected result called for the parallel sintering application to be able to run a model with 40000 atoms in each particle radius. The memory model has shown to be capable of handling such a model. Unfortunately, the time and resource constraint sets it beyond the scope of this report. Assuming that the parallel time per MCS is linear in Figure 40 the Equation 9 can be fitted to the data, where r is the radius.

$$\frac{T}{MCS} = 1.34 \cdot 10^{-2} \cdot r + 0.116 \qquad (9)$$

A rough estimation of the number of MCS needed can be calculated with Equation 10 which was derived by Sutton *et al.* [29].

$$MCS = 2.7003 \cdot 10^{5} - 19999 \cdot r + 488.71 \cdot r^{2} + 0.83045 \cdot r^{3} \qquad (10)$$

The total time needed on *Agave*, using between six and eight processors, is then estimated by Equation 11.

$$T = \frac{T}{MCS} \cdot MCS \tag{11}$$

For a model having 40000 atoms in each particle radius, the estimated time and MCS needed are $4.6\,s/step$ and $5.4 \cdot 10^{13}\,steps$. The total time is an unacceptable $2.5 \cdot 10^{14}\,s$ or nearly eight million years. But this is estimations based on the *Agave* system with not even 8 processors available. There are other factors to consider as well. Equation 10 is based on the original version, whereas Figure 35 and Figure 37 show that the parallel version greatly reduces the number of MCS. Also, the time needed for each MCS should not be treated as a constant. As the number of vacancies in the model decrease, so will the time per MCS. Meaning that in the estimation the maximum time per MCS is used, because it is measured from the first million MCS of the simulation execution.

9. Conclusions and future work

In this master thesis studies of sintering, diffusion, sparse data storage and parallel implementation of Monte Carlo algorithms were combined to improved performance of a sequential sintering simulation application. A grid-based solution was used to break up the simulation model, i.e. all individual sites within four particles and their immediate surrounding. The division of the model allowed for locally sparse data to be used which resulted in greatly reduced memory usage (Figure 38). Parallelization was achieved by allowing processes perform concurrent Monte Carlo steps on separate, independent grids. Working on subsets of the grids and using a good grid selection strategy resulted in a high degree of independence among the grids. Evaluation proved the speed-up gained through parallel execution to be a factor of 10 (Figure 40) and the simulation time decreases when adding more processors (Figure 41).

9.1 Conclusions

As always when dealing with simulations of a real life phenomenon it is important to understand the background and underlying processes. In this case diffusion is the underlying process that makes sintering possible. Perhaps the most serious aspect of this project was to ensure that the core of the original sequential simulation algorithm was kept intact through the transaction to a parallel program. Regardless of how much performance can be gained by alteration of the algorithm, the sintering process must remain the same. As such, the real life aspect of the algorithm is a severe limitation to the evolution of the simulation program, but it also sets an effective block on the project scope. In the end, performance gain through parallelization was possible without the core of the simulation algorithm being altered. Thus, the single most important goal of this project was reached.

The employment of a grid based division of the particle model used for the simulation proved to result in an extremely good memory usage reduction. It easily meets the goal of being able to handle billions of atoms and in fact, the expected memory usage was beaten by more than a factor of ten. This proves just how efficient the *divide-and-conquer* strategy is. The design also gives several ideas concerning further memory reduction, in case future simulation will demand millions of atoms in the particle radius.

Although the memory model met with the expected results, the parallel performance did not. The available system resources and time for this project were not enough to run any large models, nor were they enough to give a good estimation of the scalability. Scalability could only be proven for up to six or seven processors, where it did give an increased performance for each added processor, which was one of the goals.

The choice of system is always important for any parallel algorithm. Monte Carlo algorithms generally perform well on a distributed scheme, and therefore this report, and its shared memory implementation, should be treated as a stepping stone between the sequential and full parallel version of the sintering simulation. This is to be concluded from the fact that a distributed memory scheme on a network of processors more readily allows scalability beyond 64 CPUs, as will be needed for larger models.

9.2 Future work

One important aspect to implement into the algorithm is the possibility to have particles of different kinds of atoms. There are several new things that needs considering when simulating this. Firstly, the diffusion rate might differ between atom types and, secondly, a particle may only allow a certain concentration of different atom types. This is something that is being worked on at MINMET.

Another part of future work is to allow for particles of different sizes and different shapes. This would obviously mean a complete rewrite of the data initialization phase which currently heavily depends on the circular form of the particles. The difference in sizes would only amount to a change in determining the centre of each particle. The actual simulation algorithm would work just as well with any size or shape of particles, if the data is initialized correctly.

As mentioned in the conclusion, ones the problem size becomes really large, it is expected that the need for process power will expand beyond the realms of current shared memory architecture. In the future, an attempt should be made to adapt the algorithm to a distributed architecture. This project was from the beginning set out to be an implementation of the sintering code to a cluster of computer, i.e. a network of processors and distributed memory. But since the network was not set up on time, shared memory architecture had to be used instead.

10. References

1. Akl, S. G., *The design and analysis of parallel algorithms*. 1989: Prentice-Hall International Editions. ISBN: 0-13-200073-3.
2. Alford, N., *Lecture Course in Materials*, http://www.eeie.sbu.ac.uk/research/pem/Materials%20Lectures/Chapter%206%20%20CERAMICS.pdf. 08-12-2004.
3. Andrews, G. R., *Foundations of multithreaded, parallel, and distributed programming*. 2000: Addison Wesley Longman Inc. ISBN: 0-201-35752-6.
4. Beichl, I. M., Teng, Y. A., Blue, J. L. *Parallel Monte Carlo simulation of MBE growth*. in *9th International parallel processing symposium*. 1995. Santa barbara.
5. Cerezo, A., *Mechanisms of migration*, http://www.materials.ox.ac.uk/teaching/diffusion/DiffusionLecture1.pdf. 08/12/2004.
6. Cheung, A. L., Reeves, A. P. *Sparse data representation for a data-parallel computation*. in *Scalable High Performance Computing Conference*. 1992. Williamsburg, VA USA.
7. Cvetanovic, Z., Freedman, E. G., Nofsinger, C. *Efficient decomposition and performance of parallel PDE, FFT, Monte Carlo simulations, simplex, and sparse solvers*. in *Conference on High Performance Networking and Computing*. 1990: New York.
8. DeWitt, D. J., Kabra, N., Luo, J., Patel, J. M., Yu, J. *Client-server paradise*. in *20th International Conference on Very Large Data Bases*. 1994. Santiago, Chile: Morgan Kaufmann Publishers Inc.
9. Dowson, G., *Powder metallurgy: the process and its products*. 1st ed. 1990, Bristol: A. Hilger. ISBN: 0-85-274006-9.
10. Fenlason, J., *GNU gprof*, http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html. 08/12/2004.
11. German, R. M., *Powder metallurgy science*. 1st ed. 1984, Princeton, N.J.: Metal Powder Industries Federation. ISBN: 0-91-840460-6.
12. Gray, T., *13 Aluminum*, http://www.theodoregray.com/PeriodicTable/Elements/013/index.s7.html. 08/12/2004.
13. Hearn, D., Baker, M. P., *Computer graphics*. 1986: Prentice-Hall International. 352. ISBN: 0-13-165598-1.
14. Hercog, D., *Generalised sliding window protocol*, in *Electronics Letters Online*. 2002.
15. Hössinger, A., Langer, E., Selberherr, S., *Parallelization of a Monte Carlo ion implantation simulator*. IEEE transactions on computer-aided design of integrated circuits and systems, 2000. **19**(5): p. 560-567.
16. Kingery, W. D., *Introduction to ceramics*. 2nd ed. 1976, New York: Wiley. ISBN: 0-47-147860-1.
17. Lin, C., Chung, Y., Liu, J. *Data distribution schemes of sparse arrays on distributed memory multicomputers*. in *International Conference on Parallel Processing Workshops*. 2002.
18. McKellar, A. C., Coffman Jr, E. G., *Organizing matrices and matrix operations for paged memory systems*. Communications of the ACM, 1969. **12**(3): p. 153 - 165.

19. Miura, K. *Vectorization and parallelization of transport Monte Carlo simulation codes*. in *Winter Simulation Conference*. 1990. New Orleans.
20. Moran, C., *Sintering - My Industry - Applications & Equipment*, http://us.lindegas.com/international/web/lg/us/likelgus.nsf/DocByAlias/nav_met_heat_sin. 08/12/2004.
21. PASC, *PASC Information Server*, http://www.pasc.org/. 21/12/04.
22. Porter, D. A., Easterling, K. E., *Phase transformations in metals and alloys*. 2nd ed. 1992, London: Chapman & Hall. ISBN: 0-41-245030-5.
23. Postula, A., Abramson, D., Logothetis, P. *The design of a specialised processor for the simulation of sintering*. in *the 22nd EUROMICRO Conference*. 1996. Prague.
24. Rice, P., *INT32DC Data Communications Lecture #9*, http://ironbark.bendigo.latrobe.edu.au/courses/bcomp/c202/2003/L09/L09.html. 08/12/2004.
25. Ripoll, D. R., Thomas, S. J. *A parallel Monte Carlo search algorithm for conformational analysis of proteins*. in *Supercomputing '90*. 1990. New York.
26. Sarawagi, S., Stonebraker, M. *Efficient organization of large multidimensional arrays*. in *Tenth International Conference on Data Engineering*. 1994: IEEE Computer Society.
27. Seamons, K. E., Winslett, M., *A Data Management Approach for Handling Large Compressed Arrays in High Performance Computing.* Frontiers of Massively Parallel Computation, 1995: p. 119 - 128.
28. Shewmon, P. G., *Diffusion in solids*. 2nd ed. 1989, Warrendale, Pa.: Minerals Metals & Materials Society. ISBN: 0-87-339105-5.
29. Sutton, R. A., Schaffer, G. B., *An atomistic simulation of solid state sintering using Monte Carlo methods.* Materials Science and Engineering A, 2002. **335**: p. 253-259.
30. Ujaldon, M., Zapata, E. L., Sharma, S. D., Saltz, J., *Parallelization Techniques for Sparse Matrix Applications.* Journal of parallel and distribution computing, 1996.
31. Ujaldon, M., Zapata, E. L., Sharma, S. D., Saltz, J. *Experimental evaluation of efficient sparse matrix distribuations*. in *10th international conference on Supercomputing*. 1996. Philadelphia, United States: ACM Press.
32. University, K., *Diffusion mechanisms*, http://www.eng.ku.ac.th/~mat/MatDB/MatDB/source/Proc/kinetics/vacancy/vacancy.htm. 08-12-2004.

11. Abbreviations

IEEE	Institute of Electrical and Electronics Engineers
ITEE	School of Information Technology and Electric Engineering
KTH	Royal Institute of Technology
MCS	Monte Carlo Step
MINMET	Department of Mining, Minerals and Materials Engineering
MPI	Message Passing Interface
PASC	Portable Application Standards Committee
POSIX	Portable Operating System Interface
UQ	University of Queensland

12. Appendix

A. Profiling results

This appendix contains the profiling results of the original program run on an input with the large model (Table 3) as input. The profiling was done with the gprof tool.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 65.70  10944.25 10944.25                             main
 20.18  14306.01  3361.76                             nn
 10.02  15974.73  1668.72                             count_ones
  1.67  16252.33   277.60                             chk_pore_pinch
  1.30  16469.01   216.68                             grain
  1.03  16641.38   172.37                             move_atom_back
  0.07  16652.76    11.38                             chk_atom_type
  0.03  16657.29     4.53                             chk_vac
  0.00  16657.35     0.06                             rugos
  0.00  16657.39     0.04                             vacancy_annihilation
  0.00  16657.42     0.03                             find_vac_and_atoms
  0.00  16657.43     0.01                             find_next_pixel
  0.00  16657.43     0.00      482     0.00     0.00  sqrt
  0.00  16657.43     0.00        9     0.00     0.00  tanh
  0.00  16657.43     0.00        8     0.00     0.00  expm1
  0.00  16657.43     0.00        5     0.00     0.00  exp

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
```