FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

A Fast Multigrid Solver for Molecular Dynamics on the Cell Broadband Engine

Daniel Ritter

Master Thesis

A Fast Multigrid Solver for Molecular Dynamics on the Cell Broadband Engine

Daniel Ritter

Master Thesis

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Dipl.-Inf. Markus Stürmer
Bearbeitungszeitraum:	22.11.2007 – 22.05.2008

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 22. Mai 2008 .....

Abstract

In the field of molecular dynamics, Poisson's equation with open boundary conditions plays a major role. One issue that comes up, when this equation is solved numerically, is the infinite size of the domain. This prevents a direct solution, so that other concepts have to be considered. Within the scope of this thesis, a method is discussed that uses hierarchical coarsened grids to overcome this problem. Special attention has to be paid to the discretization on the grid interfaces. A finite volume approach has been used for that. The resulting set of linear equations is solved using an efficient method: a multigrid algorithm.
The introduced method is implemented on IBM's Cell processor, a modern multicore processor, that is powerful in floating point operations and memory bandwidth. Code optimization techniques are applied to get the maximum performance on this processor.
For a validation of the performance, test runs are performed, and the runtime is analyzed in detail.

Kurzfassung

Im Bereich der Molekulardynamik spielt die Poisson–Gleichung mit offenen Randbedingungen eine große Rolle. Hierbei ist die unbeschränkte Gebietsgröße bei der numerischen Lösung derselben ein Problem. Da dies eine direktes Lösen der Gleichung verhindert, müssen andere Möglichkeiten gefunden werden. In dieser Arbeit wird dazu eine Methode verwendet, die auf hierarchisch vergröberten Gittern basiert. Die Grenzflächen zwischen den einzelnen Gittern müssen getrennt behandelt werden. Zu diesem Zweck wird eine Finite–Volumen–Diskretisierung vorgestellt. Das entstehende lineare Gleichungssystem wird anschließend mit einem effizienten Mehrgitterverfahren gelöst.
Diese Methode wurde auf dem IBM Cell–Prozessor, einem modernen Mehrkernprozessor, implementiert. Seine Stärken sind hohe Leistungen bei Gleitpunktberechnungen und die hohe Speicherbandbreite. Der Programm–Code wurde mit verschiedenen Techniken optimiert, um die maximal mögliche Geschwindigkeit zu erreichen.
Um die Ergebnisse zu validieren, wurden Testläufe und anschließende Laufzeitanalysen durchgeführt.

Acknowledgements

I would like to thank Prof. Dr. Ulrich Rüde first of all, not only for supervising and supporting my master thesis, but also for being a motivating tutor throughout my master studies.
Furthermore, thanks to Markus Stürmer, who did a great job as my advisor, especially in giving programming hints.
I thank Matthias Bolten for providing code, inspiring this thesis and permitting access to the JUICE Cell cluster, and to Harald Köstler for correction and support in maths.
Some personal thanks go to my parents who persisted in supporting me during my studies.
Last, but not least, I thank my girlfriend Johanna, for correcting – especially English formulations –, supplying food and just for being there for me.

Contents

List of Figures

Chapter 1

Introduction

In recent years, computer simulation has become a major tool for a wide field of applications in science and industry. These range from material science, fluid dynamics and biotechnology to economics, medicine and microelectronics. Technology, like simulation, will be used in a commercial area only if the expected outcome is positive, i.e. if it helps to get the same output at lower cost or to increase the output at constant cost. For many application fields, the turning point was reached lately or will be reached in near future. The driving forces for this trend will be explained for the example of molecular dynamics.
Molecular dynamics — or N-body simulations — are a classical field within computer simulation. This results from two facts: There exists no analytical solution for the N-body problem for more than two particles, and the underlying mathematical model is simple — an 2nd order ODE for each particle. This problem can therefore be handled easily by a computer, involving numerical methods. Additionally, the same technique can be applied to systems of different orders of magnitude, ranging from galaxies to atoms. There is a number of computational issues, however, that had to be solved to use molecular dynamics as a universal tool and there is a number left of those to be solved in future. Nevertheless, these methods are quite popular in many disciplines nowadays. Three reasons for this effect are proposed:

- The applications themselves: Some of them treat problems that can only be fully understood, if they are inspected at molecular level. So the chemical properties and behavior of proteins, that result directly from their molecular structure, are required for the development of drugs against viral diseases, e.g. the HI virus. In material sciences, simulation is used to predict and optimize the properties of materials containing nano–particles. Previously, there was no idea of processes om this nano–scale size.

- Applied mathematics: The computer as a tool for mathematical simulation has fed back different requirements on mathematics again. So special and more efficient algorithms were developed, e.g. the multigrid for solving linear systems of equations that result from numerical treatment of partial differential equations (PDEs) or the fast Fourier transform (FFT) to compute the Fourier transform. In numerical mathematics, new methods should also consider the question of computational efficiency.

- Computer hardware: The speed of computers is doubling every two years — this has been valid for more than 40 years and it will also hold at least for the next 15 years, as hardware maufacturers claim. This is suggested to be the mean driving force, because it means that computations, possible on a some years old cluster, can be done on a standard workstation today. The latter, however, is cheaper by some orders of magnitude in purchase, energy consumption and maintenance, what makes simulations quite attractive. There is one more trend in hardware development that is described below.

Computer science is undergoing a dramatic change nowadays that has started a few years ago. It is the coming age of multicore processors that delivers both new opportunities and challenges for program developers. The idea of parallel programming, that only played a role in few parts of computing until then, has become a central concept for any application field: It is essential for a satisfying utilization of modern processors. Parallel (re)design of algorithms can be a demanding task for a programmer, who never thought about that before. While at the moment standard processors have two or four cores, a chip is available on the market that has nine ones: The Cell processor. Consider, that this number of cores will be standard in two or three years for PCs also. Therefore, it is a fascinating task to develop software for this processor, because concepts discovered here may be reused.

In the scope of this thesis, a multigrid method was brought onto the Cell processor, that solves Poisson's equation for a special set of boundary conditions. For this purpose, first of all, the basics of the Cell processor have to be understood. It has some unusual, new features, that are described in chapter (2). In this context, different programming issues and their influence on future programming models are discussed. An overview over available development tools on the CBE is given as well.
Chapter (3) deals with the role of Poisson's equation in molecular dynamics and with the open boundary conditions, that are imposed. This includes the derivation of the simulation model from mechanics and the formulation of the PDE, containing the boundary conditions. The used discretization model, which uses hierarchical coarsened grids, is explained. Special attention is also payed to the discretization on interfaces and to the features of the FAC multigrid method.
Implementation details and applied optimization techniques are illustrated in chapter (4) using a number of code examples. These range from general concepts to Cell–specific ones. One computational simplification, that speeds up the code, is also introduced.
Results of test runs are provided in chapter (5). The major performance measures are code runtime, achieved floating point performance and memory bandwidth. Besides the open-boundary conditions, also Dirichlet ones were used for Poisson's equation.
Finally, an conclusion on the reached goals is provided in chapter (6). Some ideas are listed that could further improve the performance of the code and suggestions are made for related subjects to further research.

Chapter 2

The Cell Architecture

The Cell Broadband Engine (CBE) is a promising new processor architecture. It was developed by IBM in cooperation with Sony and Toshiba and introduced in 2005. Devices with the Cell Processor include the Sony Playstation 3 and IBM Cell blade servers. It is of interest for scientific computing applications, since it is at the moment — ahead the Sun Niagara processor which has eight cores — the processor with the most cores on a chip that is available on a commercial system, in the case of the Playstation at low cost. From 2007, the second generation of the CBE is produced, the PowerXCell, and since May 2008, it is sold on the QS22 blade. It is an improved version, that can deal with more memory and has full pipeline support also for double precision floating–point operations.
The Cell BE is a heterogeneous multi-core architecture, consisting of one *Power Processor Element* (PPE), which can be considered as main processor, and eight co–processors, the so called *Synergistic Processor Elements* (SPEs). On the Playstation 3, one of the SPEs is deactivated, what may increase the output of the manufacturing process. Hence, six SPEs can be used for user applications on the PS3. One is always reserved by the hypervisor that controls the operating system and abstracts from parts of the hardware, that cannot be accessed directly. The PPE is a PowerPC core with 512 kB of Level2 cache. Each SPE consists of a *synergistic processor unit* (SPU) and a *memory flow controller* (MFC) that is scheduling data transfers between SPU and main memory: Each SPU has a local store of 256 kB besides its *synergistic execution unit*.
For computationally intensive applications, the SPEs are the highlight of the CBE, due to their high floating–point performance and their memory bandwidth, whereas the PPE is the part of the processor, that can handle control–intensive tasks better by far and is efficient at switching tasks. The theoretical peak performance of all the eight SPEs is about 204 Gflop/s with single precision and the maximum overall bandwidth of the Memory Interface Controller (MIC) which connects the PPE and the SPEs with the memory is 25.6 GiB/s. On chip there is also the *element interconnect bus* (EIB), that connects PPE, SPEs, MIC and the *Broadband Engine Interface* (BEI), which is the link to further I/O. The EIB has a theoretical maximum on–chip bandwidth of 204.8 GiB/s.
However, the high performance, which is possible with the CBE, doesn't come for free. The effort, that has to be done in programming, is higher than on a standard PC. Due to the reduced instruction set of the SPEs, a lot of work, that is automatically be done by many other modern CPUs has to be done by hand, namely the memory access: This has to be initiated by DMA transfers, scheduled via the MFC. From here, the main memory — the on–board RAM — is also denoted as memory, which is distinguished from the local store of the SPEs. Furthermore, the compiler for the SPE code is not so sophisticated as compilers for e.g. x86 processors nowadays — the current version 3.0 of the Cell SDK, released in September 2007, is the first one which is not classified as beta version by IBM.

This chapter does not only give an overview over the architecture of the CBE, but also describes the challenges in programming it and how to overcome them. The physical layout of a Cell

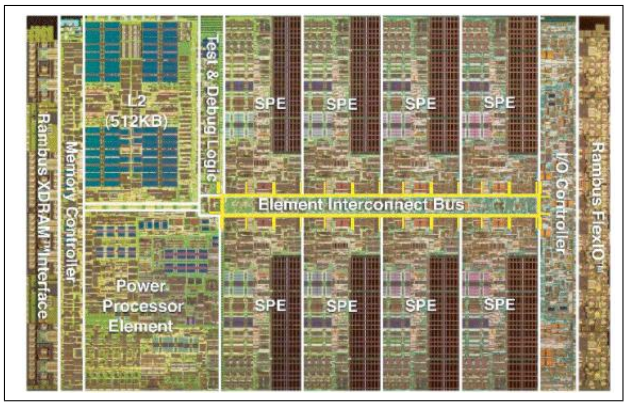

Figure 2.1: Layout of a Cell Die.[1]

die is shown in figure (2.1). The CBE contains 2.5 MiB of memory on the chip that is used as cache for the PPE and as local storage for the SPEs: Each SPE has 256 kB of local storage.

2.1 The Power and Memory Walls

The Cell processor was developed as a possible answer to two major problems, which came up in the last years (or even decades) and which will have dramatical impact on computers and computer science in the near and middle future.
The first one is the so–called power wall: Over decades, a big part of the speed gain in computers was done by increasing the clock rate, besides the fact, that the on–chip complexity was growing. But the power consumption grows quadratically with the clock rate, and the heat emission by the processor grows in the same order. In the consequence, cooling problems result from a higher clock rate as well as power consumption issues appear. The growth of the clock rate was the main reason for Moore's law to hold valid for the last 40 years, but nowadays there is only little reserve in clock rate left (for the next 40 years this strategy cannot work). It is quite sure, that the feature size on a chip will decrease at least for the next 10 to 15 years, what reduces space and power consumption requirements for a constant level of chip complexity. The idea behind new multicore architectures is to use this area and power for more cores on chip, usually two, three or four clones of the same chip. The cores may share the whole or parts of their cache hierarchy. The Cell is designed as a heterogenous system, where the SPEs have a reduced functionality only, but are optimized for speed in floating point operations and streaming of large data between memory and SPE. In contrast to many cache-based architectures, this concept does not prevent relatively long latencies. This limits the transfer rate, at least if small amounts of data are requested only per request.
The second challenge is the so-called memory wall. Different from the first problem, that can be overcome in principle by multicore processors, this is much harder to solve. It is caused by the different development in processor and memory speed over the last decades and is the main bottleneck for the speed of computationally intensive programs nowadays. The processor speed increased orders of magnitude faster than the memory speed in the past, i.e. the processor can manipulate data at a high rate, but the data is transferred to and from the memory at a relative low speed. This means, that the peak performance of the processor is hard to achieve, especially for programs that are memory intensive. Processor designers try

[1] Adapted from http://www-128.ibm.com/developerworks/power/library/pa-fpfeib/

to compensate this problem by using hierarchical caches on the chip, that are much faster in access, but also much smaller than the main memory. The programmer's influence on the utilization of the cache is indirect, i.e. the layout of data has to be optimized. The processor has to predict, which piece of data will be required in the near future, and prefetch this data into the cache. The CBE is completely different from that: The local storage of the SPEs have to be filled manually. So the paradigm for the design of data layout is giving the programmer full control over memory access, but also more responsibility for its efficiency.

2.2 The Power Processor Element

The power processor element is the main processor of the CBE. It is one 64–bit Power PC core (that is the reason, why a Linux system can easily be installed on the Playstation) and is a modern CPU, which is running at 3.2 GHz clock rate. A specific feature of the PPE, compared to other Power PC processors, is that it executes commands in order, so out–of–order execution is not implemented. For compensating that weakness, the PPE can handle two threads at a time – a kind of hyperthreading technology. It has full altivec support, that means, it has got a powerful SIMD engine for vector operations. The PPE also has 32 kB of Level 1 and 512 kB of Level 2 Cache. It is a quite average processor and regarded as single unit not too interesting for high performance scientific computing. It is clear that, for sufficient application speed, the utilization of the other processing units is crucial.
Due to its features, the PPE is the processor that runs the operating system. In parallel applications, one process runs on the PPE usually and does the setup and the scheduling of the SPE threads as well as I/O operations.

2.3 The Synergistic Processor Elements

The SPEs are the most revolutionary elements on the CBE. On–chip, there are 8 dedicated cores with a reduced functionality, which is optimized for memory and computationally intensive tasks. Their name *synergistic* is motivated by the fact, that the interplay between SPEs and PPE ideally leads to synergistic effects in performance, i.e. a significant increase in speed. The PPE is used in this context to span the SPE threads and to coordinate them, while the latter do the actual computation work. Each SPE consists of the processing unit itself and a memory flow controller. The processing unit includes 256 kB of local store. The local store is unprotected, i.e. treats data and code equally and is accessed untranslated by its SPE, which can access this memory only. The memory flow controller contains a DMA controller, the part that allows to load and store data: It can schedule DMA transfers between the main memory and the local store of the SPE. This is the only possibility for the SPE, to deal with data in the main memory, and is typically a new concept. The DMA commands are done asynchronously: Multiple commands can be queued in the MFC and executed in background, while a SPE thread can continue. Of course, the execution has to be stopped at the moment, the data is needed by the thread, but if the scheduling is done in a clever way, the memory latencies can be hidden effectively. Multiple buffering is one possible way for that.
The processing unit has its own instruction set, the *SPU Instruction Set Architecture*, that is SIMD capable in the following way: The SPE has 128 registers, each 128 bit wide. The floating-point unit, respective the two fixed-point units of each SPE, execute operations on these registers, which contain data vectors. These vectors may consist of fixed-point variables of 8, 16, 32 or 64 bit or floating-point ones of 32 or 64 bit length each. The operations on those are fully pipelined — except for the double precision floating point ones — and can therefore

be executed efficiently, if there are no branches in the code. Branching means for each misprediction, the whole pipeline has to be flushed, what makes branch intensive programs run slow on the SPEs. For using the SIMD capabilities, intrinsics can be used within the source code. Alternatively, also auto–vectorization can be done by the compiler, but this less effective.
The bandwidth of DMA transfers is highly depending on the alignment of data, both in the main memory and in the local store: Transfers are done within "cache lines" of 128 byte each: This means ideally, data blocks should start at an address that is a multiple of 128 and their lengths should also be multiple of 128 bytes. That restriction should be taken into account, when a high performance is demanded in applications.
The instruction set of the SPE cannot execute privileged code. This is the reason, why many system functions are not implemented on the SPEs natively. Some things like I/O using `fprintf(...)` are emulated by callbacks to the PPE and are available for debugging purposes at least. For execution on the SPEs, a program has to be developed, additional to the PPE main program, and to be compiled with a seperate SPE compiler. Afterwards, it can be embedded into the PPE binary or be loaded from the file system.

2.4 Challenges in Cell Programming

When programming applications for a new kind of architecture like the CBE, there are different issues that come up during the implementation phase. Some of these issues are specific for the architecture, others are more general for a certain class of processors and some may even be characteristic for trends, that come up in future. The latter should be documented well and looked at in more detail, because they are concepts of stronger quality than the first ones. Understanding them can lead to new programming models or paradigms. While the first group of issues may be obsolete for any new hardware, the last may be of key importance for the next years in computer science.
For a typical application programmer with background on scientific computing and some experience in parallel programming, there are a number of new programming techniques required on the Cell processor. The following enumeration lists some major issues, that should be taken into account, when developing software for the Cell processor:

1. The strong dependency of memory performance on the alignment property: Alignment is required for both, the local store address and the effective (main memory) address. This requirement is specific for this processor and it can cause quite some trouble, if the memory layout is not considered from the beginning of the program design. This is a very strict alignment property compared to competing architectures, so it is improbable to become a major trend. However, the code will run, if this is not taken into account, but slower.

2. The efficient local store strategy is crucial for good performance on the CBE, since additional effort has to be made to hide memory latency. How that can be achieved – e.g. by using multiple buffers – will be explained in chapter (4). On the other hand, an advantage is that the programmer has full control over the local store, while cache prefetching is often a black box method and can only be optimized indirectly. Even if this problem is specific for the Cell processor, similar problems can come up in distributed memory systems, where communication is done between nodes with long latencies. To overcome these issues, similar solutions can be used.

3. The SPEs operate on 128 bit registers, which contain a vector of operands. These operands can be of different integer or floating–point data types. This means, that the

SPEs are using SIMD operations to get peak performance. Actually, the SPEs cannot directly process single operands, but emulate these operations using SIMD operations. Therefore, it is critical to use the SIMD units in programming, this is done via C intrinsics. SIMD registers and operations are a major trend on current and future processor architectures. On the Cell processor, those have to be activated by specific function calls, in future this job may be done — to a certain extend — by the optimization routines of a compiler.

4. In the first Cell generation, the pipelines of the SPEs do not fully support double precision (64 bit) floating point values, but only single precision (32 bit) ones. Hence, the performance decreases by a factor of about 14 when switching from single precision to double precision. This makes double precision operations almost useless. Even, though this characteristic is corrected in the second generation of the CBE, in other parallel computer hardware such as graphics boards, which play an increasing role in scientific computing, the same problem exists.

5. Multithreaded applications are the key to a good performance on a parallel computer system. On the CBE, typically one thread is created for each SPE to do the computational work and one PPE thread is used to do the synchronization. There are other data models possible for the Cell processor. Paradigms that have been set up over the last decades, can now be tested in real world applications on this and similar processors. Parallel programming is the biggest of the considered trends, e.g. standard PC hardware has already multicore processors today. However, the specific libraries that have to be used on the Cell are not the end of development, there may be more standardized interfaces to ensure portability of the code — at least within limits.

6. The heterogeneous architecture of the CBE makes it unique today. Other multicore processors on the market consist of clones of one core put on one die, with shared cache in addition. An Intel Core 2 Duo for instance consists of two fully equipped multi–purpose processors. The multi–purpose part of the Cell processor, the PPE, is relatively slow compared to those, but the specialized SPEs are way faster for their special tasks. It is vague, whether this is a trend for the future, at least in the field of high performance processors. For other areas of application, it will play a bigger role in the next years. AMD e.g. plans to include graphic functionality into CPUs in low budget PCs, the AMD Fusion. It is for sure a trend, to put more and diverging functionality onto the chip.

The solutions for Cell specific issues will also be specific and a number of its current shortcomes is extincted in its second generation. The compiler will improve in future versions and third–party tools, that make life easier for developers, will be introduced. However, the items classified as trends in the previous enumeration, will require concepts that are more general and — on a abstract level — transferrable to other architectures. E.g. ideas *how* multigrid methods can be parallelized are not dependent on a certain architecture. In chapter (4) strategies are proposed to overcome both classes of challenges.
For parallelization of codes, there are several models. These include SPE– and PPE–centered concepts, depending on which element has the main coordination task. In [6] a number of those is mentioned.

2.5 Development Environment and Toolkit

In developing computer programs — independent of the field of application — one issue to cope with is always to find suitable tools for enhanced program design. The meaning of effectiveness may change from one kind of software to the next one, though. In developing

software on standard PC hardware, a wide range of applications is available. Same holds for the Cell processor at a first glance, as it is simply an extended PowerPC, on which a lot of GNU/Linux tools can be executed. C code, that can be run on the SPEs, is different from standard code: A subset of the standard libraries are available only, i.e. an implementation of system libraries as the `stdio` library is missing or at least has no native support. On the other hand, there are SPE–specific intrinsics, that may be used within the code. Special programs have to developed to be run on the SPEs, therefore.
There are two compilers available for creating of SPE programs: the gcc, the GNU compiler developed by the open source community, and the xlc, provided and developed by IBM. The available programming languages include C, C++ and Fortran, which are de facto standard languages in scientific computing programs. For the SPE, a dedicated program has to be written, compiled and can be embedded into a PPE one, providing a frame for the SPE code. With the gcc comes the GNU debugger, which is a powerful tool for debugging PPE and SPE programs. It is not only available on terminal but can also be included into the Eclipse IDE for interactive debugging. For basic debugging purposes of the SPE program, also the C command `fprintf(...)` is available. This is executed by the SPE calling back the PPE. This interrupts its actual execution, reads the arguments, does the output and then continues the program execution. The callback is done by an interrupt request to the PPE, which evaluates the reason of this interrupt. This is a time–consuming operation and should only be used for debugging. Practically, this means to use macros in the C code for switching on and of the I/O functionality within the SPE program.
IBM also provides libraries for a wide field of applications on the Cell processor, e.g. vector, game and inter–process communication libraries for the SPEs and the PPE. An important one for parallel programs is the libspe2, which contains the functionality to set up threads on the SPE from a PPE program. The functions out of these libraries will be explained in chapter (4), where they appear in the code.
The CBE code can also be run in a simulator, for debugging or if no Cell-based system is available. The simulator is a quite interesting tool, that allows to load programs from the host system and gives the user full control over the memory contents and the threads at a time. The drawback of the simulator is, that it runs slow on a PC and therefore only makes sense for small problem sizes, for a first look at the program or for a clock cycle wise analysis of generated code.
One more method, which is lately available for the CBE, was tested in the scope of this thesis: The Visual Performance Analyzer (VPA), a visualization tool for multithreaded programs. It can help to identify bottlenecks and deadlocks in a parallel program and is an extension to the Eclipse IDE. When using it, events like synchronization points in the code or even user–specific events, can be traced both on the SPEs and the PPE. However, this program has great influence on the overall runtime and biases some of the commands, it actually shall trace. Hence, even if it is a powerful tool in theory, at the moment it is not very helpful in practice. One more problem is, that the created log files can grow to sizes of hundreds of MiB which makes the VPA slow and unhandy.

Chapter 3

The Poisson Equation with Open Boundary Conditions

Molecular dynamics (MD) are a classical application field of computer simulation. The underlying model is well-known, relatively simple and straightforward to implement. Nowadays it becomes feasible to simulate not only small systems, but also bigger setups of thousands or millions of particles. There has been a lot of work on improving MD methods in two main directions: either to make algorithms faster without loss of accuracy or, on the other hand, to develop properties that reflect natural situations better without making algorithms slower. However, still the goal of computing realistic ensembles, is not possible, because the number of molecules in the fluid is bigger by many orders of magnitude than the number of particles in the biggest computer simulations.
Within this chapter the basic model is developed from Newton's Second Law and including the discretization of that, using the Störmer–Verlet–method, before some background on short– and long–range potentials is delivered, that leads to the linked–cell algorithm and to Poisson's equation afterwards. Furthermore, it explains how a discretization of the Poisson equation is done, what the open boundary conditions are and how they affect the numerical solution of the problem, i.e. what special issues come up in a numerical treatment of them.
Several approaches for dealing with the open b.c.s are introduced and the finite hierarchical grid coarsening is looked at detailled, as it will be the used model.
Finally, the linear solver, a special version of a multigrid method is provided and a closer look is given to its specific features.

3.1 The Poisson Equation in Molecular Dynamics

In the field of molecular dynamics a major task is to predict the evolution of the state for a given set of N interacting particles. The state is defined as the position $x(t)$ of each particle and the first and second derivative of $x(t)$, velocity $v(t) = \dot{x}(t)$ and acceleration $a(t) = \dot{v}(t) = \ddot{x}(t)$ at time t. This state can also be used for the computation of other, global measures that describe the particle ensemble as a whole, such as its potential energy or its mean temperature. Due to the fact that the system is observed at molecular level, MD simulations can explain phenomena, that cannot be explained when using a method without consideration of molecular effects.

The interactions between N particles can be described by an ordinary differential equation of second order, which is derived directly from Newton's second law:

$$F_i(t) = m_i \cdot a_i(t), \tag{3.1}$$

with the acting force $F_i(t)$, mass m_i and acceleration $a_i(t)$ for each particle i at time t. When using this model, the particles are considered to be point charges. Using position $x_i(t)$ and velocity $v_i(t)$, the following system of equations is established:

$$\begin{aligned} \dot{x}_i(t) &= v_i(t) \\ \ddot{x}_i(t) = \dot{v}_i(t) &= a_i(t) = \frac{F_i(t)}{m_i}. \end{aligned} \tag{3.2}$$

For given initial conditions $x_i^0 := x_i(t_0)$, $v_i^0 := v_i(t_0)$ and $F_i^0 := F_i(t_0) \; \forall \; i \in 1, 2, \dots N$ at initial time t_0, a simulation of (3.2) can be done by numerical integration. In MD Störmer-Verlet methods are popular for time integration. Their properties fit the requirements on a numerical solver for MD quite good. The accuracy of global measures is more important than the accuracy of the single particle trajectory. This reflects the real situation: The trajectory of a particle is distorted by Brownian Motion. Therefore a method that is more accurate than a certain limit in means of the single trajectories is not useful in MD.
There are different Störmer-Verlet methods for MD, an introduction is given in [7]. The velocity Störmer-Verlet is a popular one as it computes velocity and position at the same time each. With time step Δt, equation (3.2) results in the following set of discretized equations using the velocity Störmer-Verlet-method:

$$\begin{aligned} x_i(t + \Delta t) &= x_i(t) + v_i(t)\Delta t + \frac{1}{2}\frac{F_i(t)}{m}\Delta t^2, \\ v_i(t + \Delta t) &= v_i(t) + \frac{F_i(t) + F_i(t + \Delta t)}{2m}\Delta t. \end{aligned} \tag{3.3}$$

Using these equations, the position and the velocity for the next time step $t + \Delta t$ from the previous time step t can be computed, if the acting force $F_i(t + \Delta t)$ is known. There are in principle two ways to compute $F_i(t + \Delta t)$:

- The first way is to consider the interactions pairwise between two particles i and j. These interactions are in general described as potential $\phi(d_{ij})$, which depends only on the distance $d_{ij} = |x_j(t) - x_i(t)|$ between particle i and j. Then the resulting force is given as

  $$F_ij(t) = \nabla\phi(r_{ij}). \tag{3.4}$$

  The drawback of this method is obviously the computational performance: The force, which is acting at particle i, is defined as $F_i = \sum_{i=1}^{N} F_{ij}$. This has to be done for each of the N particles, so the overall time complexity of the force computation is $\mathcal{O}(N^2)$. However, for setting the values of the forces, a time complexity of $O(N)$ is needed.

- The second way to compute the forces is to compute only a few of the interactions directly (i.e. by means of the first method) and treat the rest of the interactions in a different way, that depends on the character of the considered potential $\phi(d_{ij})$. The potential can either be classified as short–range or as long–range potential, depending on how fast it decreases with growing d_{ij}. If a potential $\phi(d_{ij})$ is decreasing faster than $\frac{1}{d_{ij}^D}$ for $d_{ij} \to \infty$ in a D-dimensional space, it is classified as short–range potential, otherwise as long–range potential. For a given $r_{\text{cut}} > 0$, let

  $$\hat{\phi}(d_{ij} := \begin{cases} \phi(d_{ij}) & \text{for } d_{ij} \leq r_{\text{cut}}, \\ 0 & \text{otherwise.} \end{cases} \tag{3.5}$$

It can be shown, that the error, that is made when replacing $\phi(d_{ij})$ by $\hat{\phi}(d_{ij})$, is bounded, if $\phi(d_{ij})$ is a short–range potential. This replacement is equivalent to cutting the influence radius of the potential. See [7], for details on the proof. There is a method to reduce the computational complexity, using $\hat{\phi}(d_{ij})$ for the computation of the acting forces: the linked–cell method. For this, the whole simulation domain is subdivided into cells (i.e. cubes with length $r \geq r_{\text{cut}}$). Each of these is represented by a linked list, that holds all the particle, which are in this cell at one moment. If only the particles in surrounding cells are used for the computation of the acting forces, while the influence of particles in other cells is neglected, then the computational complexity can be reduced to $\mathcal{O}(N)$, under the additional assumption, that the maximum number of particles in each cell is bounded.
If the considered potential is a long–range potential, the linked–cell method may not produce sufficient accurate results, since the error can be unbound. Therefore another method is needed. Basically, there are two ways of computing the forces with long-range potentials: Tree–based methods and grid–based methods. In the following, grid–based methods are explained in more detail, because they are the basis for the introduced proceeding.
The basic idea behind grid–based methods is the extension of the linked-cell algorithm for long–range potentials without changing its time complexity class, so the overall complexity still is $\mathcal{O}(N)$. This is done by splitting up the forces F_i in a near field F_i^{near} and a far field component F_i^{far}. The near field component is computed using the linked-cell method, whereas the far field component is considered in the computation using an *accumulated potential* $\Phi(x_1, x_2, \ldots, x_N)$, which depends on the current positions of the particles. The far field component of the force is given as $F_i^{\text{far}} = \nabla\Phi(x)$. Φ can be computed by solving Poisson's equation, which is given as

$$\Delta\Phi(x) = \frac{1}{\varepsilon_0}\rho(x), \tag{3.6}$$

with the dielectric constant ε_0, the charge density $\rho(x)$ and the Laplace operator Δ. (3.6) can be solved by a numerical method with sufficient accuracy under certain conditions, such as the smoothness of the right–hand side, respectively $\rho(x)$.
Let the *δ–distribution* be defined by the properties

$$\begin{aligned} &\delta(x) = 0 \text{ for } x \neq 0, \\ &\iiint_{\mathbb{R}^3} \delta(x) = 1. \end{aligned} \tag{3.7}$$

Since the partical have no spatial expansion, their charges are defined to be δ–distributions each and $\rho(x)$ is defined as sum of δ–distributions at positions x_i:

$$\rho(x) = \sum_{i=1}^{N} \delta(x - x_i). \tag{3.8}$$

$\rho(x)$ is obviously not smooth — not even continuous — in Ω: For this reason, solving (3.6) numerically, is very inaccurate. Another method has to be introduced to get a good approximation of the solution. One possibility is to exchange $\delta(x)$ by a similar, but smooth function, a *shield function* $\sigma(x)$. $\sigma(x)$ shall fulfill the following properties:

1. $\sigma(x)$ is smooth and differentiable,
2. $\sigma(x)$ has compact support,
3. $\sigma(x)$ is symmetric around 0 and

4. $\iiint_{\mathbb{R}^3} \sigma(x) = 1.$

Often a Gaussian function or a spline curve is considered for $\sigma(x)$. A Gaussian function does not fulfill condition 2, so it is cut off at a fixed radius. As it is decreasing very fast, this has only little impact on the correctness of the solution. The resulting right–hand side is defined as

$$\hat{\rho}(x) = \sum_{i=1}^{N} \sigma(x - x_i). \tag{3.9}$$

The solution of Poisson's equation should be done in optimal time complexity, which is $\mathcal{O}(N)$. For this complexity class, one possible solver is a *multigrid* method. The design of this solver strongly depends on the given boundary conditions. For practical reasons, it is assumed that the number of cells $M \propto N$, so that $\mathcal{O}(N) = \mathcal{O}(M)$ holds. It will be shown in the next sections, why a multigrid algorithm is the method of choice for open boundary problems.

3.2 Open Boundary Conditions

From here, the symbol f will denote the right–hand side, replacing $\hat{\rho}$, in order to consider more general cases also. One special class of boundary conditions in MD are the so–called open ones. These conditions correspond to the case, when particles are moving within a bounded area, but with long–range interactions. The charges of the molecules are inside the area, but the induced field is unbounded and its influence cannot be neglected, without making a significant error. This problem can be formulated as

$$\begin{aligned} &\Delta\Phi(x) = f(x),\ x \in \mathbb{R}^3, \\ &\text{with } \Phi(x) \to 0 \text{ for } \|x\| \to \infty, \end{aligned} \tag{3.10}$$

where $\text{supp}(f) \subset \Omega$ is a bounded subset of $\mathbb{R}^3$. This is a generalization of zero Dirichlet boundary conditions at infinity. An analog discrete formulation of this problem is given as

$$\begin{aligned} &\Delta_h\Phi(x) = f(x),\ x \in \{x | x = h \cdot z, z \in \mathbb{Z}^3\}, \\ &\text{with } \Phi(x) \to 0 \text{ for } \|x\| \to \infty, \end{aligned} \tag{3.11}$$

where $h > 0$ is the grid size of the discretization and Δ_h is a discrete Laplace operator. This discrete system still consists of an infinite number of equations. Therefore, further modifications are applied to make the problem computable on a finite machine. There are different approaches to this problem, which will be explained in the next section.

3.3 Overview over Previous Solvers

From the infinite discrete problem (3.11), there exist different ways to solve it numerically. An overview of some of them is given below:

1. Observe a finite subvolume Ω of $\mathbb{R}^3$ and develop a formula for setting the boundary conditions explicitly.

2. Solve the problem on an infinitely large hierarchically coarsened grid. For practical reasons, the process is stopped after a finite number of growing steps and the boundary conditions are set to zero on the coarsest grid.

3. Solve the problem on an finitely large hierarchically coarsened grid and set the boundary conditions explicitly on the coarsest grid — using an approximation of those values.

The first method was introduced by O.Buneman in [2] for the two-dimensional case. He solved the discrete problem 3.11 analytically, i.e. for the five point Laplace operator Δ_h he derived a recursive formula for Green's function G_h, fulfilling

$$\Delta_h G_h(x) = \delta_h, \tag{3.12}$$

with δ_h being a discrete δ–distribution. Later, Burkhart developed an asymptotic expansion of $G(x)$ for the 3-D case in [3]. This expansion can be expressed as

$$G_h(x) = \frac{1}{4\pi}[\frac{1}{\|x\|_2} + \frac{\eta_3}{\|x\|_2^3} + \frac{\eta_5}{\|x\|_2^5} + \dots], \tag{3.13}$$

with $\eta_i = \mathcal{O}(h^2)$, resulting in error order $\mathcal{O}(h^2)$, if only the first term is used. The main drawback of this method is the costly computation: Assume the domain as a cube with M unknowns. Then the number of grid points is $M^{\frac{1}{3}}$ in each dimension. Equation (3.13) has to be integrated, respectively summed up over the domain. That results in M evaluations of (3.13), for each of the $6 \cdot M^{\frac{2}{3}}$ boundary points, giving overall time complexity $\mathcal{O}(M^{\frac{5}{3}})$. The optimal time complexity of the whole algorithm should be $\mathcal{O}(M)$, therefore this method cannot be the way of choice. For the application of particle simulation, a multipole expansion can be used. This was introduced by Sutmann and Steffen in [4].
The second method was used by Washio and Oosterlee in [9]. They have chosen a grid extension rate α for each grid and have proven, that the results are accurate within order $\mathcal{O}(h^2)$ for infinite coarsening with extension rate $\alpha > 2^{\frac{2}{3}}$. However, for numerical results, the coarsening process is stopped after some steps and zero boundary conditions are imposed. An error estimation for the finite process is not given.
The third technique was introduced by M. Bolten in [1] and is a combination of the previous two possibilities: The coarsening of the grid is done for a finite number of steps and afterwards the boundary conditions on the coarsest grid are set, using rule similar to (3.13). Bolten proved for this method, that the error order is$\mathcal{O}(h^2)$. The whole proof can be found in [1]. This is the method used within the scope of this thesis and is described more detailled in the next section.

3.4 Used Model

1. Let the problem (3.10) be given on the domain $\Omega = [-\frac{1}{2}; \frac{1}{2}]^3$ and $\text{supp}(f) \subset \Omega$. Now, a numerical solution of this problem is the subject of interest. The problem is discretized on Ω as regular grid with grid size h in each direction and the Laplace operator Δ is discretized as 7–point stencil:

$$\Delta_h = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 1 & -6 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{3.14}$$

   The resulting problem is a special case of (3.11).

2. Let $\alpha \in (1; 2)$ be the extension rate. The grid is extended, using the following properties: The finest grid is defined as discretization of domain Ω_1 with mesh size h, where

$$\begin{aligned} \Omega_1 &:= \left[-\frac{\beta_1}{2}, -\frac{\beta_1}{2}\right]^3, \\ \beta_1 &\geq \alpha, \\ h_1 &:= h. \end{aligned} \tag{3.15}$$

This is only an enlargement of the original domain with same grid size. The extension is now done iteratively in combination with a grid coarsening up to level l, with the parameters

$$\begin{aligned} \Omega_k &:= \left[-\frac{\beta_k}{2}, -\frac{\beta_k}{2}\right]^3, \\ \beta_k &\geq \alpha^k, \\ h_k &:= 2^{(k-1)} \cdot h \end{aligned} \tag{3.16}$$

for $k \in \{2, 3, \ldots, l\}$. The β_k are used to have the same grid points on the finer and coarser grid. The discrete set of grid points at level k is defined as $\mathcal{G}_k := \{x \in \Omega_k | x = k \cdot z,\ z \in \mathbb{Z}\}$. Analog to the continuous problem, let $\delta\mathcal{G}_k := \mathcal{G}_k \cap \delta\Omega_k$ be the boundary of grid k in the discrete case. An example for the layout of am three–level extended 2D–grid is given in figure (3.1).

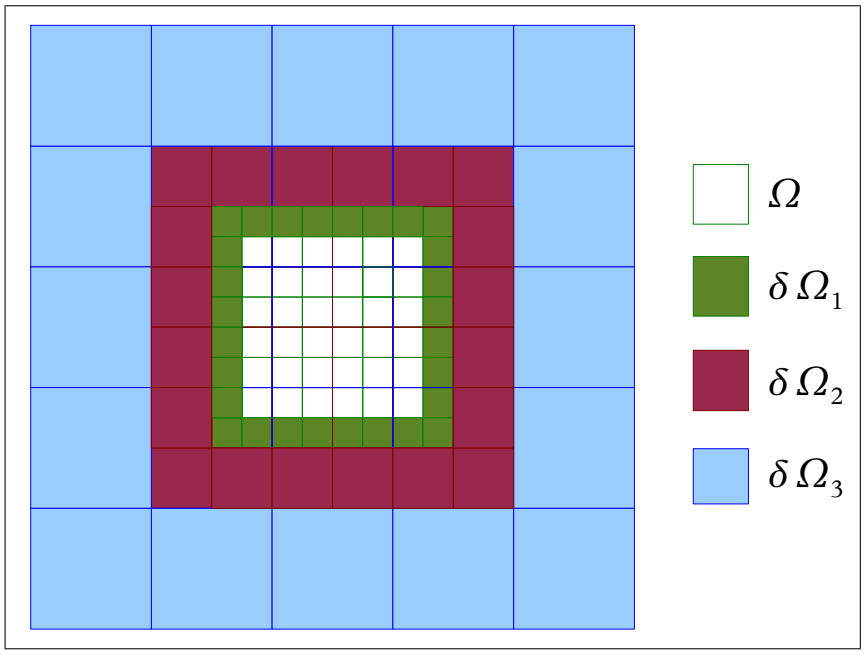

Figure 3.1: Structure of a Three–Level Extended 2D–Grid. Here, $\Omega_1 = \Omega \cup \delta\Omega_1$, $\Omega_2 = \Omega_1 \cup \delta\Omega_2$ and $\Omega_3 = \Omega_2 \cup \delta\Omega_3$.

3. After the lth extension of the grid, the boundary values are set explicitly on the boundary $\delta\Omega_l$ of Ω_l. For each boundary point $x_\delta \in \delta\Omega_l$ the potential is given as

$$\Phi(x_\delta) = \frac{1}{4\pi} \iiint_{\Omega} \frac{f(y)}{\|y - x_\delta\|_2} dy \tag{3.17}$$

for the continuous case. For the discrete problem, the potential for each boundary point $z_\delta \in \delta\mathcal{G}_l$ has to be computed via numerical integration. As only the value of $f(x)$ is known at the grid point, it is simply assumed for each grid point z, that the value of $f(z)$ is constant on a cube with length h symmetric around z. The potential can be computed in this case as

$$\Phi(z_\delta) = \frac{h^3}{4\pi} \sum_{z_{\text{fine}} \in \mathcal{G}_1} \frac{f(z_{\text{fine}})}{\|z_{\text{fine}} - z_\delta\|_2}. \tag{3.18}$$

4. The boundary conditions have to be distributed to the boundaries of the finer grids in a way that is conservative, i.e. a special discretization scheme has to be used at the interfaces between two grids. To ensure the conservation requirement, a finite volume discretization can be done. A 2D example for the structure of an interface is given in figure (3.2).

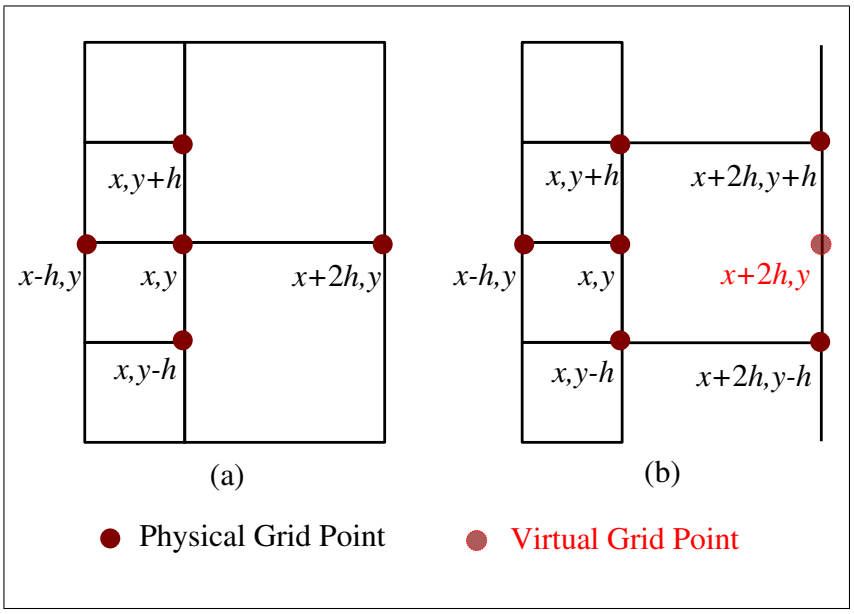

Figure 3.2: 2D Interface between Grids: (a) there is a corresponding coarse grid point to boundary point (x, y), (b) there is no corresponding coarse grid point for point (x, y), the value has to be interpolated from neighboring points on the coarse grid.

The discretization scheme is established by integrating Poisson's equation over a volume V and applying the Gaussian integral theorem:

$$\begin{aligned} \int_V \Delta\Phi(x)dx &= \int_V f(x)dx \\ &\Updownarrow \\ \oint_{\delta V} \nabla\Phi(s) \circ \vec{n} ds &= \int_V f(x)dx. \end{aligned} \tag{3.19}$$

This discretization scheme results in different special cases. If the volume is cubic, i.e. if the considered point is inside the grid, the finite volume discretization is identical to the 7–point Laplace operator. If the point is a boundary point, and there is a corresponding point on the coarse grid as in case (a) in figure (3.2), a modified finite difference scheme can be established. For reasons of simplicity, it is developed for the 1-D case using Taylor expansion below. The development in the other two dimensions is analog.

For an illustration, see figure (3.3). Let x_i be a grid point on the interface of grid k with i being the coordinate index, x_{i-1} the left neighbor of x_i on the same grid level and x_{i+2} the grid point on the next coarser grid $k+1$, which is the right neighbor of x_i. The Taylor expansions are given as

$$\Phi(x_{i+2}) = \Phi(x_i) + 2h\Phi'(x_i) + 2h^2\Phi''(x_i) + \frac{8h^3}{6}\Phi'''(x_i) + \mathcal{O}(h^4) \qquad \text{and} \tag{3.20}$$

$$\Phi(x_{i-1}) = \Phi(x_i) - h\Phi'(x_i) + \frac{h^2}{2}\Phi''(x_i) - \frac{h^3}{6}\Phi'''(x_i) + \mathcal{O}(h^4). \tag{3.21}$$

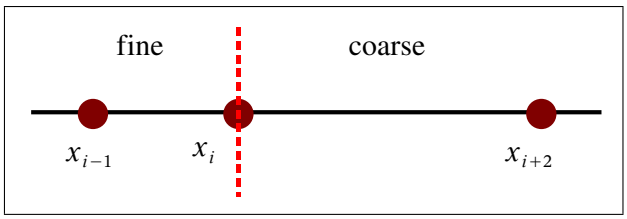

Figure 3.3: 1D Example of Interface: x_{i-1} and x_i are fine grid points, whereas x_{i+2} is a coarse grid pont, the red dashed line is the outer boundary of the fine grid.

Multiplying (3.21) with 2 and adding it to (3.20), this results in:

$$2\Phi(x_{i-1}) + \Phi(x_{i+2}) = 3\Phi(x_i) + 3h^2\Phi''(x_i) - h^3\Phi'''(x_i) + \mathcal{O}(h^4)$$

So,

$$\Phi''(x_i) = \frac{\frac{1}{3}\Phi(x_{i+2}) - \Phi(x_i) + \frac{2}{3}\Phi(x_{i-1})}{h^2} + \mathcal{O}(h) \tag{3.22}$$

is used as approximation for the second derivative of Φ at position x_i. For case (b) in (3.2), i.e. if there is no corresponding point on the coarse grid, the value of Φ on the virtual point has to be computed by linear interpolation of the above and below function values of Φ. The treatment of the interfaces can be implemented using a ghost layer as explained later in chapter (4).

M. Bolten has proven that this method has error order $\mathcal{O}(n^2)$. In principle, higher order could be achieved with enhanced discretization schemes. For solving the problem (3.10), a linear solver is required additionally. The structure of the problem implies that a multigrid method should be used for this purpose. This method can gain profit from the introduced grid hierarchy and will be subject to the next section.

3.5 Used Multigrid Solver

In the previous section, motivation was given for using a multigrid solver to solve 3.10. Within this section, the notation u is introduced for the unknowns to replace the function $\Phi(x)$. This is done to illustrate that the multigrid can also solve problems, apart from the previously considered one. To explain the used method, the problem

$$A \cdot u = f \tag{3.23}$$

is introduced, with the system matrix $A \in \mathbb{R}^{N\times N}$, the unknowns $u \in \mathbb{R}^N$ and the right-hand side $f \in \mathbb{R}^N$. Assume that (3.23) has a unique solution. The discrete problem (3.11) can be rewritten in this way — however, the matrix A is never set up explicitly, but only implicitly by applying the Laplace operator on the grid points. [10] gives a wide overview over different multigrid methods. While u denotes the exact solution, let $v \in \mathbb{R}^N$ be a computed one (which is never exact). A measure for the quality of the solution can be $e = u - v$, respectively the norm of e. This can never be achieved, since the vector u is unknown. There is another measure that is used in practice, the residual

$$r := f - A \cdot v. \tag{3.24}$$

Obviously $r = 0$ if and only if $v = u$ holds. Instead of solving (3.23) directly, the residual equation,

$$A \cdot e = r, \tag{3.25}$$

is solved and the previous solution v is corrected by e: $v^{(1)} = v + e$. Consider the error to be a superposition of low–frequency and high–frequency components. High–frequency parts are damped efficiently using a relaxation method, e.g. a Jacobi or Gauss–Seidel smoother. Low–frequency parts, though, are hardly damped by these ones. The idea behind multigrid is, that the frequency of the is doubled, if the mesh size of the grid is doubled. Applying some relaxation steps on the coarser grid will damp parts of the error, that are of lower frequency with respect to the fine grid — and could not be eliminated there. By introducing a number grids and applying the method recursively, the error can be damped significantly over all frequencies.
There are some requirements on the solution method given by the model and the boundary conditions described before. Hence, the multigrid algorithm has to be adapted carefully to the open boundary Poisson problem. In the proposed problem (3.10), the function Φ, which is identical to u here, has to be set on the coarsest grid for the boundary conditions — see step 3 of the previous section. Many multigrid methods only restrict the residual equation (3.25) on the coarser grids, i.e. the original problem (3.23) is only given on the finest grid. The fast adaptive composite method (FAC), introduced in chapter 9 of [10], is one possibility to avoid this problem. This algorithm restricts (3.23) and can therefore solve (3.10). The algorithm can be written as algorithm 3.1.

Algorithm 3.1 FAC$(k, k_{\max}, \nu_{\text{pre}}, \nu_{\text{pre}}, h)$.
(Δ_h is the discrete Laplace operator with mesh size h, $\hat{I}_k^{k+1}$ is the restriction operator and I_{k+1}^k is the prolongation operator.)

---

if $(k = k_{\max})$ **then**
  smooth u_k $100(\nu_{\text{pre}} + \nu_{\text{post}})$ times
  return
else
  smooth u_k for ν_{pre} times
  compute residual $r_k \leftarrow f_k - \Delta_h u_k$
  restrict residual $r_{k+1} \leftarrow \hat{I}_k^{k+1} r_k$
  restrict $u_{k+1} \leftarrow \begin{cases} \hat{I}_k^{k+1} u_k & u_{k+1} \in \mathcal{G}_k \\ u_{k+1} & \text{otherwise} \end{cases}$
  restrict $f_{k+1} \leftarrow \begin{cases} r_{k+1} + \Delta_{k+1} u_{k+1} & u_{k+1} \in \mathcal{G}_k \\ f_{k+1} & \text{otherwise} \end{cases}$
  call FAC$(k+1, k_{\max}, \nu_{\text{pre}}, \nu_{\text{pre}}, 2 * h)$
  compute error $e_{k+1} \leftarrow \Delta_{k+1}^{-1} f_{k+1} - u_{k+1}$
  prolongate $e_k \leftarrow I_{k+1}^k e_{k+1}$
  correct $u_k \leftarrow u_k + e_k$
  smooth u_k for ν_{post} times
end if

Special about the FAC is the way the restriction is done: Consider two grids only from now, fine and coarse one. The residual r_k is computed on the fine grid by subtracting the discrete Laplacian $\Delta_h u_k$ from the right–hand side. Then, the restriction operator is used to restrict r_k to the next finer grid, r_{k+1}. This — like all the restrictions in the FAC — is only possible for the points, which are elements of both the fine and the coarse grid. Additionally, the function u_k is also restricted using $\hat{I}_k^{k+1}$ to u_{k+1}. The right–hand side on the coarser grid is given by adding the residual on the coarse grid to $\Delta_{k+1} u_{k+1}$ for all the points, that are elements of the fine one. In the first iteration u_{k+1} and f_{k+1} are equal zero for all points outside of the fine grid, but for sequent v-cycles, the computed values from the step before are

kept. After solving the problem on the fine grid, the correction term e_{k+1} is computed and prolongated to back to the fine grid, where the solution is corrected by e_k.

Some more parameters have to be chosen for the multigrid method: kernels for restriction and prolongation as well as a smoother. In the proposed solver a underrelaxing Jacobi smoother, is used for smoothing. A red–black Gauss–Seidel smoother would be more efficient, but would be difficult to adapt for a higher–order method that can be introduced in future work. Pre- and Postsmoothing are done by the Jacobi algorithm as well as the solution of the problem on the coarsest grid. The restriction is done by a direct injection, i.e. every second value on the fine grid is copied to the corresponding point on the coarse grid. For prolongation, trilinear interpolation is performed to compute the values on the fine grid.

Chapter 4

Implementation Details and Performance Optimization

In chapter (2), the features Cell processor were described in detail. This chapter shows, how the implementation of a multigrid method is done and how the code can be tuned on the CBE, apart from numerical issues that where treated in the chapter before. Which techniques are used for increasing the performance of the code, depends on the algorithm and has to be considered early in the development process. The methods can be transferred to other programs for the Cell processor and, furthermore, some of the introduced concepts can also have impact on performance, when applied on other architectures. In addition to performance increasing methods, which do not have effects on computed values, the last section within this chapter gives an idea for a computational simplification of the model, which changes values numerically, but increases the performance of the algorithm significantly.

4.1 General Remarks

Before introducing the special programming techniques used in the proposed program, some hints shall be given on the data layout on the SPUs and general programming issues on the Cell. For instance, the data type used to store the grid contents is always `vector float`, i.e. a 128 bit vector, containing 4 32 bit float variables. These are always processed as one unit by the SPU. For arithmetic operations, intrinsics are used:

- `vector float c = spu_add(vector float a, vector float b)` for elementwise addition,
- `vector float c = spu_mul(vector float a, vector float b)` for elementwise multiplication,
- `vector float c = spu_sub(vector float a, vector float b)` for elementwise subtraction,
- `vector float c = spu_div(vector float a, vector float b)` for elementwise division and
- `vector float d = spu_madd(vector float a, vector float b, vector float c)` for a fused multiply–add–operation, i.e. $d = a \cdot (b + c)$.

Besides those arithmetic intrinsics, one more SPU intrinsic is used quite often in the source code, the intrinsic

`vector char d = spu_shuffle(vector char a, vector char b, vector char c).`

This function shuffles vectors `a` and `b` bytewise arbitrarily according to pattern vector `c`: each byte of `c` determines, which value is taken for the same byte in the result vector `d`. The first half–byte selects from which vector (0 for vector `a` or 1 for vector `b`) the value is copied, whereas the second half–byte selects the position within this vector the value is copied from. If e.g. `c` is set to

```
vector char c = {0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15,
0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b};
```

the result vector will in its first 4 bytes contain the 4 last bytes of `a` and in the remaining 12 bytes contain the first 12 bytes of `b`. An overview over all SPU intrinsics is given in [6], pp. 64 – 75.

4.1.1 Linewise Processing

The 3-D data arrays for the grids are processed linewise in the following manner: C is a language where arrays are stored in row major order, i.e. the last index of a D-dimensional array is the one, which has subsequent elements in memory. So it should be the one, which is increased in the innermost loop, when iterating over all elements of the array. In the developed program, the convention for 3-D arrays is that the first index denotes the position in x–direction, the second one in y–direction and the third one in z–direction. So, the third index is iterated in the innermost loop of all functions. Linewise processing means therefore iteration over z while fixing y and x. Data is always stored linewise with consequences due to the SXU unit, that executes the SIMD commands: If the total length of the array in z–direction is not a multiple of 4, dummy elements have to be inserted at the end of each line since the SPU can only process a vector of 4 floats at a time. Even more dummy elements may be necessary, if proper alignment is required, see chapter (2)
The programmer has to initiate all load/store operations. For this purpose, the SPU intrinsics `mfc_get(...)`, `mfc_put(...)`, `mfc_set_tag_mask(...)` and `mfc_read_tag_status_all()` are used.

1. `mfc_get(target_ls, source_ea, length, tag, tid, rid)` is used to copy `length` bytes from the effective address (main memory) `tsource_ea` to the local store address `target_ls` of the SPE. The `tag` is an identifier for the mfc transfer and can be a number between 0 and 31 as 32 channels are available. `tid` and `gid` are unused in this context and set to zero.

2. Analog `mfc_get(source_ls, target_ea, length, tag, tid, rid)` is used to copy data from the local store of the SPU to the main memory. A group tag is also specified here.

3. `mfc_set_tag_mask(channel)` sets the tag to the value(s) that should be checked with the next `mfc_read_tag_status_all()` command. The tag id, that was set by the MFC request before, corresponds to the bit with the same number. If e.g. tag id 5 was used, the channel for this transfer is `1<<5`= 32. If for more than one tag id is waited, the channel numbers can be combined by disjunction.

4. `mfc_read_tag_status_all()` waits, until all MFC requests with the previously set tag identifiers are completed.

The effective addresses may be 32 or 64 bit addresses, depending on which mode is used by the PPE program, while the local store addresses are 32 bit pointers – only 19 bit are used for addressing within the local store, actually. In the multigrid program the, 64 bit addressing mode is activated, so the effective addresses have data type `unsigned long long`, whereas the local store addresses have the C pointer type (denoted by `*`) in the SPE program.

4.1.2 Local Operators

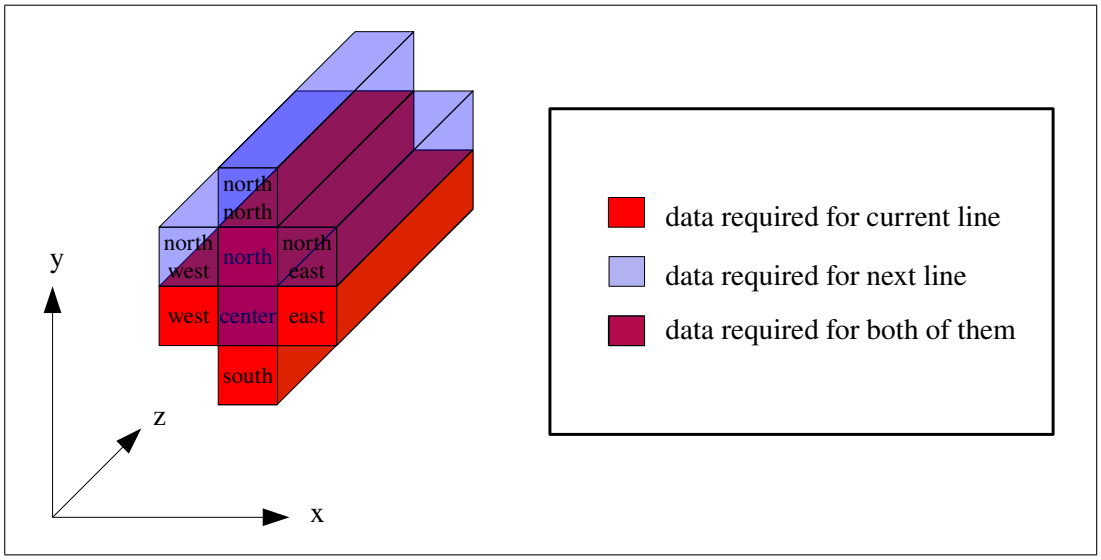

Figure 4.1: Lines of Domain required for Jacobi Stencil

The multigrid algorithm can be parallelized easily, because the operators that are applied onto the data are local operators. That includes the Jacobi operator, which is used as smoother, the Laplace operator, which calculates the residual, the prolongation and the restriction operator. Local means here, that those operators have only dependencies on a small area within the whole domain and their impact is also restricted to a small area. Assuming linewise processing, like mentioned in the previous subsection, also neighboring lines have to be loaded for every operation on the domain. This is shown in figure (4.1) for the Jacobi operator: Besides the center line also the north, south, west and east data lines are required. Furthermore, the two purple lines can be reused, when processing the next one, so that only the three blue ones (north–north, north–west and north–east) have been loaded from memory. This technique is explained in the next section in more detail.

4.2 Double Buffering

Like mentioned in chapter (2), the SPUs have no caches to hide latencies between main memory and local store. Therefore the programmer has to develop a strategy to achieve that. One possibility for this purpose is to use multiple buffering for all load and store operations, more precisely all `mfc_get(...)`and `mfc_put(...)` commands by using two or more buffers in the local store. While the first buffer is filled, the SPU can already work on the data in the second one and vice versa. In SPU programs this can be achieved in several steps as shown below exemplary for the Jacobi method. The store operations are left out within this example for reasons of simplicity:

1. The buffer itself, the variable `locBuf` is declared. To make things easier, also pointers are declare, which are used to point to the active buffer line in each case, those are pointers to the local store (32 bit). The `unsigned long long` variables are actually pointers to the addresses of the required arrays in main memory (64 bit). The `linesize` variable is the length of the line that is loaded/stored at a time.

```
    //the big buffer for data lines (SPU local store)
          vector float locBuf[12][64] __attribute__ ((aligned(128)));

    //pointers for convenient access to f and u (SPU local store)
5         vector float * ucenter, * ueast, * uwest, * unorth,
                       * usouth, *u1north,* u1east, * u1west,
                       * datf, * datf1, * utmp, * utmp1;

    //pointers to data (in main memory)
10        unsigned long long globData; // u
          unsigned long long globF;    // f
          unsigned long long globTmp; // utmp

    //size of a line
15        int lineSize = z * sizeof(vector float);

    //initialize the buffers..
          ucenter = locBuf[0];
          ueast = locBuf[1];
20        uwest = locBuf[2];
          unorth = locBuf[3];
          usouth = locBuf[4];
          u1north = locBuf[5];
          u1east = locBuf[6];
25        u1west = locBuf[7];
          datf = locBuf[8];
          datf1 = locBuf[9];
          dattmp = locBuf[10];
                dattmp1 = locBuf[11];
```

2. The data required for the Jacobi steps in the first line is loaded via `mfc_get(...)` commands. In this first step there are six commands used. This is the only piece of data, that is waited for immediately using the `mfc_read_tag_status_all()` command. In the inner loop, `mfc_get(...)` commands are executed nonblocking and only four calls are needed since data can be reused from the previous step.

```
    for (int it = 0; it<2; it++) {


      for(int i = start; i < end; i++) {
//load all the data needed for the first line
        mfc_get(ucenter, globData+((i*y*z)+z)*sizeof(vector float), lineSize, 1, 0, 0);
        mfc_get(ueast, globData+(((i+1)*y*z)+z)*sizeof(vector float), lineSize, 1, 0, 0);
        mfc_get(uwest, globData+(((i−1)*y*z)+z)*sizeof(vector float), lineSize, 1, 0, 0);
        mfc_get(unorth, globData+((i*y*z)+2*z)*sizeof(vector float), lineSize, 1, 0, 0);
        mfc_get(usouth, globData+(i*y*z)*sizeof(vector float), lineSize, 1, 0, 0);
        mfc_get(datf, globF+((i*y*z)+z)*sizeof(vector float), lineSize, 1, 0, 0);

//wait until first line is loaded
        mfc_write_tag_mask(1<<1);
        mfc_read_tag_status_all();

        for(int j = 1; j < y−1; j++) {

//now load the data for the next line, but do not wait...
          mfc_get(u1north, globData+((i*y*z)+((j+2)*z))*sizeof(vector float), lineSize, 2, 0, 0);
          mfc_get(u1east, globData+((i+1)*y*z+(j+1)*z)*sizeof(vector float), lineSize, 2, 0, 0);
          mfc_get(u1west, globData+((i−1)*y*z+(j+1)*z)*sizeof(vector float), lineSize, 2, 0, 0);
          mfc_get(datf1, globF+(i*y*z+(j+1)*z)*sizeof(vector float), lineSize, 2, 0, 0);
```

3. Afterwards, the Jacobi operation is executed for all elements in this line. This part is left out in the code example since it has no influence on the load/store operations.

4. After the Jacobi operator has been applied, the `mfc_read_tag_status_all()` command is used to wait for the last load operations. Finally, the pointers are swapped.

```
                    mfc_write_tag_mask(1<<2);
                    mfc_read_tag_status_all();

//and swap the pointers
                    tmpPtr = ucenter;
                    ucenter = unorth;
                    unorth = u1north;
                    u1north = usouth;
                    usouth = tmpPtr;

                    tmpPtr2 = ueast;
                    ueast = u1east;
                    u1east = tmpPtr2;

                    tmpPtr3 = uwest;
                    uwest = u1west;
                    u1west = tmpPtr3;

                    tmpPtr2 = datf1;
                    datf1 = datf;
                    datf = tmpPtr2;

                }
              }
```

4.3 Ghost Layers at Interfaces

While operations on one grid are straightforward to implement and run quite efficiently, the interfaces, that have to be considered as described in section (3.4), make the code quite inefficient and inflexible. Additional data has to be loaded from the next coarser grid, whenever boundary elements are accessed. This holds for each Jacobi iteration and the computation of the residual, so special versions have to be written of these program parts. Introducing a ghost layer around each grid is an elegant way to overcome this issue. Hence, the grid size is extended by two elements in each dimension, one each at beginning and end of it. These are virtual points, since they do not correspond to a physical point on the domain and the function values at this points are set so, that they produce the same results as values from the coarser grid and the special discretization on the boundary would produce. Only one function has to be written, which imposes this ghost layer, while the functions for Jacobi and residual computation are unchanged — i.e. no special treatment of the boundary has to be done within those functions. On the other hand, the algorithm can be adapted to other boundary conditions with relatively low effort when using a ghost layer. However, the function which is setting the ghost layer values can become quite complicated, but the rest of the functions will stay easier and better readable.

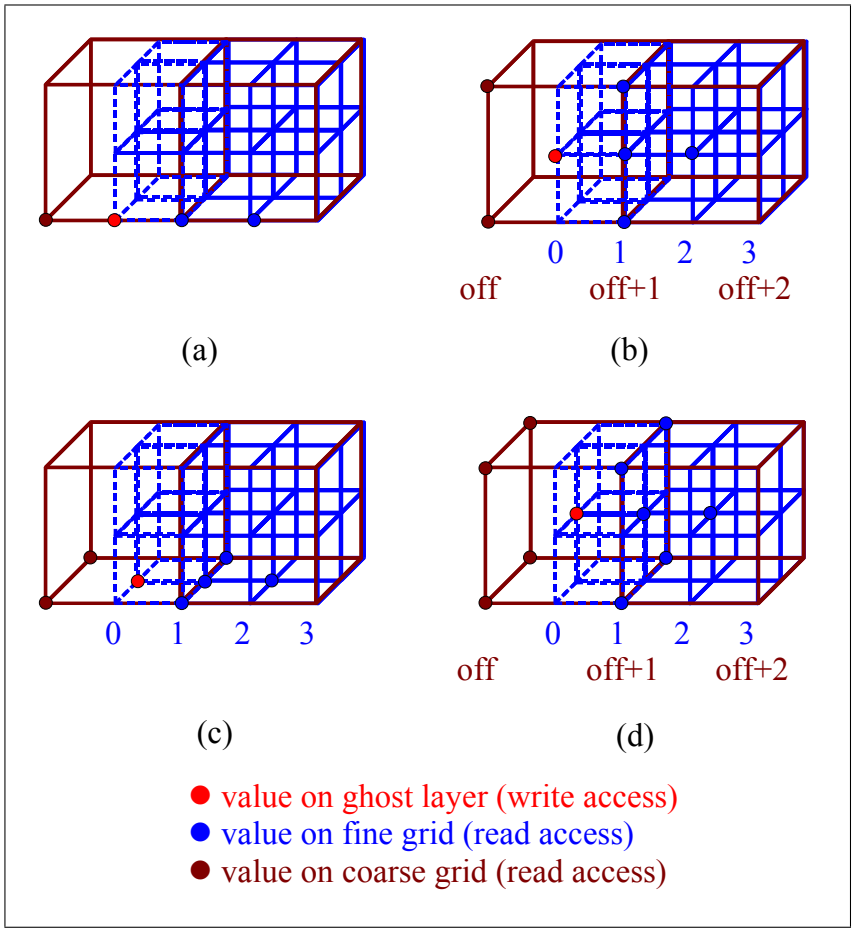

Figure 4.2: Data Dependencies Ghost Layer

Figure (4.2) shows, how the ghost layer is treated in the algorithm and how two grids are connected. The fine one is blue and the layer with index 0 is the ghost layer. The result on

the opposite side respective in the other spatial dimensions can be considered analog. Four cases occur which have to be treated different. This results from the fact, that there is no corresponding coarse grid point for every fine grid one (cases (b) – (d)) and the value has to be interpolated. Linear interpolation is applied for case (b) and (c) and bilinear one for case (d). The light red dot shows, for which ghost layer point the value is computed in each case and the blue respective dark red ones give the data dependencies for this ghost layer value. To avoid `if`–statements in all computation kernel for ghost layer computation, four neighboring ghost layer points, precisely cases (a) – (d) are set in one inner iteration, then the indices are increased by 2 each. Again, coarse and fine grid lines are double buffered to hide memory latencies and speed up the code. Altogether, 20 buffers are in use.

The major drawback of this technique is, that the performance behavior is differing from interface to interface. While a whole line can always be processed in the top, bottom, left and right interfaces, for the front and back interface, only one vector has to be loaded from each line and same holds for the storing. On the other hand, the number of `mfc_get(...)/mfc_put(...)` operations is roughly squared in the latter interfaces, compared to those in the other ones. This causes great differences in the computation time for the interfaces.

4.4 Multi-Threading and Domain Decomposition

The high level of on–chip parallelism is the most interesting feature of the CBE. Utilizing these parallel structures is essential for good performance. This section describes how a multigrid algorithm can be parallelized and what programming concepts are given to the programmer on the Cell.

The number of operations per point is known at compile time, in the proposed multigrid algorithm and the computational load is distributed equally over the inner points of the grid. Therefore, no load balancing is required, but the load can be distributed to all SPUs easily. Since only local operators are used within the multigrid algorithm, static domain decomposition into pieces of roughly same size should be sufficient in terms of gained performance, while straightforward to implement. Of course, synchronization points have to be established after each Jacobi iteration — because for the next iteration values are required from other SPUs – and after each coarsening of the grid — because space partitioning can change here.

Up to this point, the considerations have been general for a parallel program and do not depend on the platform, that is given: Parts have been identified, that can be parallelized, an idea (domain decomposition) has come up and some limits (synchronization points) have been found. The critical question is, how these ideas can be implemented into a Cell program.

The framework for each application on the CBE is a PPE program, that is used for initialization, thread creation and cleanup afterwards. Threads can be created on the PPE using the POSIX thread library (pthreads). These PPE threads give control to one SPE and then sleep until control returns from the SPE. For this purpose, one SPE context has to be created for each utilized SPE. A binary image containing the SPE program has to be loaded, then the execution of this program with optional parameters is triggered. The PPE thread is sent to sleep until the SPE program is finished. The thread creation is shown on the next page.

```
    //context pointer for each thread
    spe_context_ptr_t ctx[NUM_SPU];

    //program handler for each thread
 5  spe_program_handle_t * program;

    //Pthread objects
    pthread_t pthr[NUM_SPU];

10  [...]

    //open program image
    if ((program = spe_image_open("./spu/maincell"))==NULL) {
      fprintf(stderr, "Could not open prog maincell\n");
15    exit(1);
    }

    //create all the SPU threads
    for (thr = 0; thr < NUM_SPU; thr++){
20    fprintf(stderr, "CREATE THREAD %d!\n", thr);

    //create context
      ctx[thr] = spe_context_create(SPE_EVENTS_ENABLE,NULL);
      if (!ctx[thr]) {
25      fprintf(stderr, "Could not create context %d!", thr);
        exit(2);
      }

    //create Pthread
30    if(pthread_create(&pthr[thr], NULL, &call_jac_spe, &(jacArgs[thr]))) {
        fprintf(stderr, "Could not create thread %d!", thr);
        exit(3);
      }
     }
```

A wrapper function (`call_jac_spe(...)`) is used for passing arguments from the PPE to the SPE program: Like any C-program, also the SPE one contains a `main` function. This accepts three variables of type `unsigned long long` as arguments:

- `unsigned long long spe_id`: unique identifier for the SPE thread respective its corresponding PPE thread.
- `unsigned long long argp`: pointer to an argument for the SPE program. In the case of the multigrid algorithm, a struct is referenced here with all necessary informations about the grid sizes and locations and interprocess communication variables.
- `unsigned long long envp`: environment pointer of the program. Here, an extra argument can be specified, this pointer is not required and therefore is set to `NULL` in the multigrid program.

The function `call_jac_spe` writes as:

```
void call_jac_spe(void * arg) {
//entry address of spu program
  unsigned int entry = SPE_DEFAULT_ENTRY;
//Infos on error...
  spe_stop_info_t stop_info;
//Pointer to spu program arguments
  struct JacArg * argPtr = (struct JacArg *) arg;
//try to load program into SPU
  if (spe_program_load(ctx[argPtr->id], program)) {
    fprintf(stderr, "THREAD %d: Error loading spu program", argPtr->id);
    exit(3);
  }
//run the context with arguments...
  spe_context_run(ctx[argPtr->id], &entry, 0, argPtr, NULL, &stop_info);
//after execution of spu program, exit thread!
  pthread_exit(0);
  return;
}
```

The argument pointer, that is used here, points to a `struct JacArg` which is defined below. Recall that addresses in the main memory are of type `unsigned long long`. The `__attribute__ ((aligned (xx)))` statement forces `xx`–byte alignment for the declared variable. For dynamic memory allocation, the function `memalign(...)` is replaces `malloc(...)`. The `struct common` includes the required communication variables, that will be discussed after the code example.

```
struct JacArg : public alignator<64> {
  //relaxation parameter of jacobi
  float omega __attribute__ ((aligned (16)));
  //Data for IPC
  struct Common com __attribute__ ((aligned (64)));
  //ID of the process....
  int id __attribute__ ((aligned (16)));
  //number of mglevels!
  unsigned int levels __attribute__ ((aligned (16)));
  //vectors for size of grids, length = levels
  unsigned long long xLev __attribute__ ((aligned (16))); //int *
  unsigned long long yLev __attribute__ ((aligned (16))); //int *
  unsigned long long zLev __attribute__ ((aligned (16))); //int *
  unsigned long long hLev __attribute__ ((aligned (16))); //float*
  //vectors of offsets in each dimension for each level...
  unsigned long long xOff __attribute__ ((aligned (16))); //int *
  unsigned long long yOff __attribute__ ((aligned (16))); //int *
  unsigned long long zOff __attribute__ ((aligned (16))); //int *
  //vector start, end for domain decomposition for each level
  //length = levels
  unsigned long long start __attribute__ ((aligned (16)));
  unsigned long long end __attribute__ ((aligned (16)));
  //Grid ** grAll all the grids, length = 5*levels
  unsigned long long grAll __attribute__ ((aligned (16))); //Grid **
  //float * fp pointer for sum over rhs
  unsigned long long fp __attribute__ ((aligned (16))); //float *
};
```

Even though different binaries are generated for the PPE and SPE programs, parts of the code are identical to ensure that the data types as `struct JacArg` are the same and to define some global macros.
For synchronization, in general inter–process communication has to be used, e.g. shared variables or semaphores. On the Cell processor, the library *libsync* provides a wide range of communication methods. The completion variables offer a functionality that is quite easy to understand and sufficiently powerful for the required process synchronization. A completion variable `comp` has to be initialized with the command `init_completion(comp)`. If the command `wait_for_completion(comp)` is called, the calling process waits until another process executes `complete(comp)` to start running again. For a full synchronization of a number of SPE threads, code can be constructed easily. Besides completion variables, also one atomic counter variable is needed to implement synchronization. Atomic variables are provided by data type `atomic_ea_t` in libsync and are accessed using atomic functions, e.g. `atomic_dec_return(...)`, which decrements a counter and returns the previous value atomically or `atomic_set(...)`, that assigns a new value to the variable atomically. With the completion variables `ppuwait, spuwait[2]` and the atomic counter `thrun`, initialized with the total thread number, the SPU code for a synchronization point is written as:

```
//How many threads are still running + decrement thrun
int threadsleft = atomic_dec_return(thrun);
//If I am the last one...
        if (threadsleft == 1) {
                complete(ppuwait);
        }

// wait until
        wait_for_completion(spuwait[waitidx]);
        waitidx ^= 1;
```

On the PPU, the counterpart of the SPU code has to be implemented to perform synchronization. For synchronization, the same data types are also available on on the PPE. The PPE code is given by:

```
//wait until ppuwait is completed
 wait_for_completion(ppuwait);
//init ppuwait again
 init_completion(ppuwait);
//reset thrun
 atomic_set(thrun, NUM_SPU);
//init next spuwait
 init_completion(spuwait[waitidx^1]);
//complete current spuwait
 complete_all(spuwait[waitidx]);
 waitidx ^=1;
```

More detailled information on the synchronization library — besides other libraries on the CBE is found in [5].

4.5 Vectorization of the Code

For performant execution of numerical kernels, those have to be parallelized using the SXU. This is done explicitely using the data type `vector float` and the intrinsics `spu_add(...)`, `spu_mul(...)` etc. The SIMDization of the operations should be done completely and no branching should occur in the innermost loop, since branching interrupts the pipeline and causes drastic decreases in performance.

```
// SIMD jacobi iterator for vector float uctr
//the result is written to the vector utmp
inline void vec_weight(vector float & uctr, vector float & unorth, vector float & usouth,
                       vector float & uwest, vector float & ueast,
                       vector float & f, vector float &utmp,
                       vector float &om6, vector float &om1, vector float &h) {

//Vectors for intermediate results
        vector float front, back, tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;

//vectors for scrambling the elements before and after uctr
        vector unsigned char sh_fr = (vector unsigned char)
                                     {0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
                                      0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b};
        vector unsigned char sh_ba = (vector unsigned char)
                                     {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
                                      0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13};

//scramble vectors before and after uctr
        front = spu_shuffle(*((&uctr)-1), uctr, sh_fr);
        back = spu_shuffle(uctr, *((&uctr)+1), sh_ba);

//compute the Jacobi weighting for point uctr
        tmp0 = spu_mul(h, f);
        tmp1 = spu_add(usouth, unorth);
        tmp2 = spu_add(uwest, ueast);
        tmp3 = spu_add(back, front);
        tmp6 = spu_mul(om1, uctr);
        tmp4 = spu_add(tmp0, tmp1);
        tmp5 = spu_add(tmp2, tmp3);
        tmp7 = spu_add(tmp4, tmp5);
        tmp0 = spu_mul(om6, tmp7);

//write it to utmp!
        utmp = spu_add(tmp0, tmp6);
        return;

}
```

All the numerical kernels (Jacobi, Laplace, restriction and prolongation) were SIMDized within the program code. Due to the problem structure, it can happen that array dimensions are not multiples of 4. In that case, the last vector in each line contains one or more entries, which are simply ignored. Furthermore, the offset from one grid to the next may not be a multiple of 4. This plays a role in the injection method: Values injected on the coarse grid have to be shifted within the `vector float`. To allow vectorized operations nevertheless, special shuffling of the result vectors has to be implemented as shown on the next page.

```
//determine shuffle vectors depending on offset
 switch (zoff%4) {
   case 0 : sel_vec = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
5            sel_vec2 = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
            break;
   case 1 : sel_vec = {0x1c, 0x1d, 0x1e, 0x1f, 0x00, 0x01, 0x02, 0x03,
                       0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b};
10           sel_vec2 = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                       0x08, 0x09, 0x0a, 0x0b, 0x1c, 0x1d, 0x1e, 0x1f};
            break;
   case 2 : sel_vec = {0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
                       0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07};
15           sel_vec2 = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                       0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f};
            break;
   case 3 : sel_vec = {0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
                       0x1c, 0x1d, 0x1e, 0x1f, 0x00, 0x01, 0x02, 0x03};
20           sel_vec2 = {0x00, 0x01, 0x02, 0x03, 0x14, 0x15, 0x16, 0x17,
                       0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f};
            break;
}
[...]
25  for(int i = start; i < end; i+=2){
  [...]
    for(int j = 3; j < ySize-2; j+=2) {
//loading
      [...]
30        for(int k = 0; k < zSize-1; k+=2){
//injection and laplacian on coarse grid (SIMD versions!)
            vec_inject(uf[k], uf[k+1], tmp_c[1]);
            vec_inject(uf[k+2], uf[k+3], tmp_c[2]);
            vec_inject(uf_n[k], uf_n[k+1], tmp_n);
35          vec_inject(uf_w[k], uf_w[k+1], tmp_w);
            vec_inject(uf_e[k], uf_e[k+1], tmp_e);
            vec_inject(uf_s[k], uf_s[k+1], tmp_s);
            vec_inject(df[k], df[k+1], tmp_d);
            vec_inject(ff[k], ff[k+1], tmp_f);
40          vec_laplace(tmp_c[1], tmp_n, tmp_s, tmp_w,
                        tmp_e, tmp_d, tmp_f, vec_h);
//shuffling according to zoff%4
            tmp_f_sh = spu_shuffle(tmp_f, tmp_fn, sel_vec);
            tmp_fn = spu_shuffle(ZEROVEC, tmp_f, sel_vec2);
45          fc[(k+1)/2+zoff/4] = tmp_f_sh;
            tmp_c_sh= spu_shuffle(tmp_c[1], tmp_cn, sel_vec);
            tmp_cn = spu_shuffle(ZEROVEC, tmp_c[1], sel_vec2);
            uc[(k+1)/2+zoff/4] = tmp_c_sh;
            tmp_d_sh= spu_shuffle(tmp_d, tmp_dn, sel_vec);
50          tmp_dn = spu_shuffle(ZEROVEC, tmp_d, sel_vec2);
            dc[(k+1)/2+zoff/4] = tmp_d_sh;
            vec_inject(uf[k], uf[k+1], tmp_c[0]);
          }
//shuffling last element
55      [...]
        }
  [...]
  }
```

4.6 Computational Simplifications on the Code

Additional to the previous optimization techniques, that enhance the performance of the algorithm, but do not change the numerics of it, an advanced version of the code has been developed, that may produce slightly different numerical results: Imposing the boundary values on the coarsest grid, i.e. step (3) in chapter (3.4), was identified as the bottleneck of the algorithm and took more than 90% of the computational time.
Revisiting equation (3.17) and the derived one (3.18), that is

$$\Phi(z_\delta) = \frac{h^3}{4\pi} \sum_{z_{\text{fine}} \in \mathcal{G}_1} \frac{f(z_{\text{fine}})}{\|z_{\text{fine}} - z_\delta\|_2},$$

it is obvious that the most costly term — in means of computational time — is the computation of $\|z_{\text{fine}} - z_\delta\|_2$, because this requires one square root operation. Even worse, this term is set up for every cell in the finest grid, i.e. N times per boundary point z_δ.
However, the diameter of the finest grid is 1 by definition and the diameter of the coarsest, i.e. of the lth one, is under the assumption $\beta > 2^{\frac{2}{3}}$, which was made for stability reasons, given as $d_l > 2^{\frac{2}{3}l}$.
The idea for the simplification is that, for sufficiently big l, the diameter of the finest grid becomes negligible with respect to d_l and therefore in (3.18), $\|z_{\text{fine}} - z_\delta\|_2$ can be replaced by $\|z_\delta\|_2$ with only little influence on the error. So, $\|z_\delta\|_2$ can be factored out, what results in

$$\Phi(z_\delta) = \frac{h^3}{4\pi \cdot \|z_\delta\|_2} \sum_{z_{\text{fine}} \in \mathcal{G}_1} f(z_{\text{fine}}). \tag{4.1}$$

For future research, numerical analysis of the error imposed by this simplification should be done as well as further approaches for simplifications. In the tested problem, this simplification did not affect the overall result of the computation. This may also be due to the symmetric layout of the model problem, though.

Chapter 5

Tests and Results

After the implementation of the multigrid method, the pivotal question is, what the outcome of the applied effort in terms of computational performance is. In order to find out that, performance tests were executed on both the Playstation 3 and the Cell Blade QS20. In principle, the processors are identical on both, so the same code can be executed with two exceptions: The number of available SPEs is 6 on the Playstation, but 8 on the QS20, and the overall memory size of the Playstation is 256 MiB, while the QS20 has got 1 GiB. This restricts the maximal problem size on the PS3, i.e. bigger grid sizes are only tested on the QS20.
In more detail, one QS20 blade contains two Cell processors, which have 512 MiB of main memory available each. The theoretical bandwidth is 25.6 GiB/s for each of the Cell processors, whereas this peak performance can hardly be achieved in practice. If an aplication uses more than 512 MiB of memory, the threads are distributed by random over the 2 CBEs and the full memory can be used. This means, that the double memory bandwidth is available. The default thread and memory assignment can be overridden using the `numactl` command.
A first measure is the runtime of the code. This can be measured using the UNIX command `time` and can give a first hint on the overall performance. However, it is not very accurate, since times for initialization and thread creation are included. Therefore, the program is profiled in more detail, using in–code timing commands for its parallelized parts. Bottlenecks can be identified this way. Furthermore, the relative speed of the method is calculated from that, i.e. the achieved memory bandwidth and floating point performance.
In chapter (2) the strong alignment requirement of the Cell was mentioned. In the tests it was also measured how the alignment affects the memory bandwidth.

5.1 Scalability

In parallel programs, two ways of scaling are distinguished:

- *Strong scaling* means the speedup that can be achieved if the parallel program at fixed problem size is run on an increasing number of processing units. With ideal parallelization, the computing time t_P for P processing units calculates as $t_P = \frac{t_1}{P}$. Due to the interprocess communication, that grows with P, this speedup can hardly be achieved in practice. This limit is also known as communication overhead.

- *Weak scaling* means that the speedup of a program that can be achieved, when the problem size is increased as well as the number of processing units, so that the computational work for each processor stays constant. Ideally, the processing time is also constant, independent from the number of processors. However, in reality the increase

of performance drops with growing problem size in most applications. In this weak scaling, though, even superlinear scaling effects are possible, because for growing sizes, loop overheads play an less important role.

Both classes of scaling will be looked at in the performed tests, since both the problem size and the number of computing nodes is variating. One graph in a diagram denotes an experiment with constant problem size for different numbers of SPEs used, so the strong scaling can be directly read from each graph. Weak scaling can be observed, if runtimes out of two different curves are taken into account: If the size in each dimension is doubled in a test case, the resulting problem size is multiplied by eight: The eight–threaded bigger test case can be compared to the single–threaded smaller one. If the weak scaling is good, the runtimes should be identical.
In reality, scaling is also limited by the parts of the code that have not been parallelized. These include in case of the multigrid algorithm:

- The initialization routines with all the required memory allocation and the setup of the problem,
- setup of the communication variables, thread creation and loading of the SPU programs,
- computation of the error norms and
- output of the computed solution.

For the smallest considered grid size (17^3), only the first two items need more than 0.01 seconds, whereas the total runtime is around 0.06 seconds. For bigger sizes, the allocation may take several seconds as well due to swapping effects.

5.2 Wall-Clock Runtime

The wall clock time that is spend for program execution is taken as first performance measure. This is done using the UNIX shell command `time`. Simple timing cannot answer the question, where in the algorithm most time is spend, but it is an easy way to get a first impression of the program performance. Time spend in sequential parts of the code is always part of the measured time. The timing is averaged over ten program executions, while each one of those executes one vcycle.

5.2.1 Runtime Tests for Open Boundary Conditions

Tests were performed with open boundary conditions for 16^3, 32^3 and 64^3 cells on the Playstation 3 and with 16^3, 32^3, 64^3 and 96^3 cells on the QS20. On the PS3, 1–6 threads were run, on the QS20 8 ones. In a multigrid algorithm, the most computational work is done on the finest grids, since they have the greatest size. When using some finite boundary conditions, e.g. Dirichlet ones, the number of grid points halves in each dimension with each grid level. This is not the case for open boundary conditions. The used model, discussed in section (3.4) causes an internal size on the finest grid that is twice as big as the external one and decreases slower with each grid level than in the Dirichlet case: Table (5.1) shows, what the eight biggest grid sizes are for different external grid sizes.

Ext. Size	# Grid Levels	Eight finest Grid Sizes									Mem. [MB]
16^3	8	35	35	35	23	19	19	15	11		8
32^3	12	67	67	39	35	35	23	19	19	...	26
64^3	16	131	131	131	71	67	67	39	35	...	159
96^3	20	195	195	103	99	55	51	51	31	...	504

Table 5.1: Overview over Grid Sizes, Levels and Memory Requirements for Open B.C.s

The first three grid sizes are always identical: So the computational work is the same for each of the three finest grids. Additionally, in table (5.1), the memory requirements for each size are provided. Clearly, the biggest grid size cannot be run on the PS3 due to a lack of main memory.

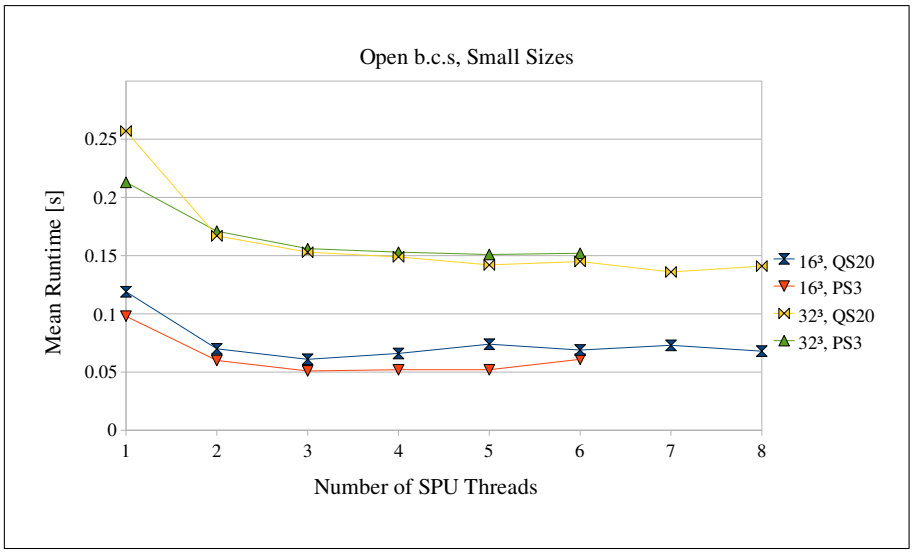

Figure 5.1: Runtime over Thread number for Problem Sizes 16^3 and 32^3 with Open B.C.s

In figure (5.1) and (5.2), runtime results for different problem sizes are displayed in seconds. It can be seen at a first glance, that there is a scaling effect when increasing the number of threads. The scaling works only within limits, though: It is very strong, when increasing the threads from 1 to 2, but for more than 3 threads, it is hardly notable.
A comparison between the smaller problem sizes (figure (5.1)) and the bigger ones (figure (5.2)) shows, that the speedup gained by utilizing more of the SPEs, is better for bigger sizes. For those, a slight effect can be measured for upto 5 SPEs, whereas there is no such effect for the first ones. This is not surprising, since loop overheads and synchronization overheads play a bigger role for small array sizes. Recall, that the problem size increases by a factor of eight if the number of points in each dimension is doubled. The runtime, however, increases by a factor of roughly 2, from size 16^3 to 32^3, i.e. this runtime is determined by the sequential initialization time. when increasing the problem size further, from 32^3 to 64^3, this factor grows to about $1.6s/0.254s = 6$, i.e. at this problem size, the overall time spent for actual solving is more important here.
Three reasons are identified for the the limitation of the overall speedup to less than 4 SPEs: The communication overhead for more threads, the growing influence of the sequential code parts and the limitation of memory bandwidth (if all the available bandwidth is used by three SPEs, then using more does not make much sense). For parts of the code, the memory bandwidth is the limiting factor, actually, as will be explained in the next section.

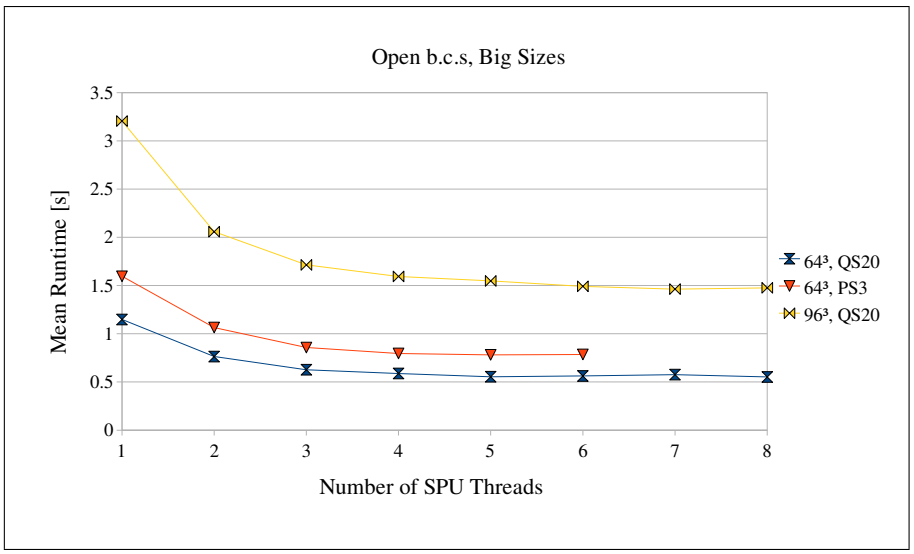

Figure 5.2: Runtime over Thread number for Problem Sizes 64^3 and 96^3 with Open B.C.s

Another effect is visible in the speed comparison between PS3 and QS20: The PS3 is quite competitive in terms of speed, upto the 32^3 problem, but significantly slower on the 64^3 problem as the memory requirement is near the maximal available memory here, like shown in table (5.1). When dealing with this problem size on the PS 3, data has to be swapped out to disk to free memory for the multigrid program. This decreases the overall performance. The QS20 can gain no advantage from its additional SPEs for this test.

5.2.2 Runtime Tests for Dirichlet Boundary Conditions

For Dirichlet boundary conditions, tests were executed for the problem sizes 64^3, 128^3, 192^3 and 256^3. The two biggest grid sizes could only be run on the QS20, because of the required memory. Different to the open boundary test case, the size is halved in each coarsening step for Dirichlet ones. The number of levels is therefore chosen so, that the coarsest level is size 9 resp. 13 (for 192^3 only). The number of grid levels and the memory requirements are displayed in table (5.2).

Size	# Grid Levels	Mem. [MB]
32^3	3	7
64^3	4	12
128^3	5	60
192^3	5	193
256^3	6	447

Table 5.2: Overview over Memory Requirements for Dirichlet B.C.s

The maximal available numbers of SPEs were used again both on the PS3 and on the QS20. The figures (5.3) and (5.4) show the overall runtime for Dirichlet boundary conditions. The measured runtimes are, like in the previous section, averaged over 10 runs of one vcycle each.

Obviously, the scaling effect that can be achieved for Dirichlet b.c.s is much weaker than for open ones. The speedup improves with growing size, though, and the performance of the PS3 is slightly below that of the QS20. For grid sizes 64^3 and 128^3, shown in figure (5.3),

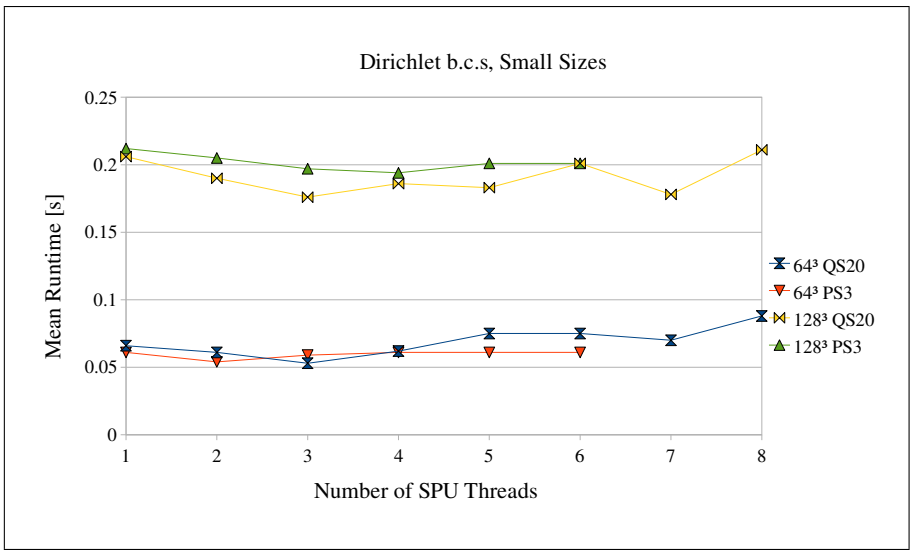

Figure 5.3: Runtime over Thread number for Problem Sizes 64^3 and 128^3 with Dirichlet B.C.s

there is hardly any scaling effect, the performance is even dropping slightly for more than 4 SPEs. For the bigger grid sizes in (5.4), the scaling effect can at least be seen, but it is also quite slow. Even here is no effect for a thread number of more than 3. The weaker scaling, compared to the open b.c. test case, can be explained by the fact, that the number of levels is smaller in case of the Dirichlet ones and the grid size is halving with every level. The domain decomposition is more effective for bigger grids and the number of those is relatively smaller, when Dirichlet b.c.s are imposed. I.e., a smaller part of the program is parallelized effectively, and the rest is not. With increasing size, the efficiency of the parallelization gets better, what can be seen in figure (5.4). However, the introduced parallelization method was adapted for the open b.c.s.

Again, the performance on QS20 and PS3 is about the same for the small grid sizes, but due

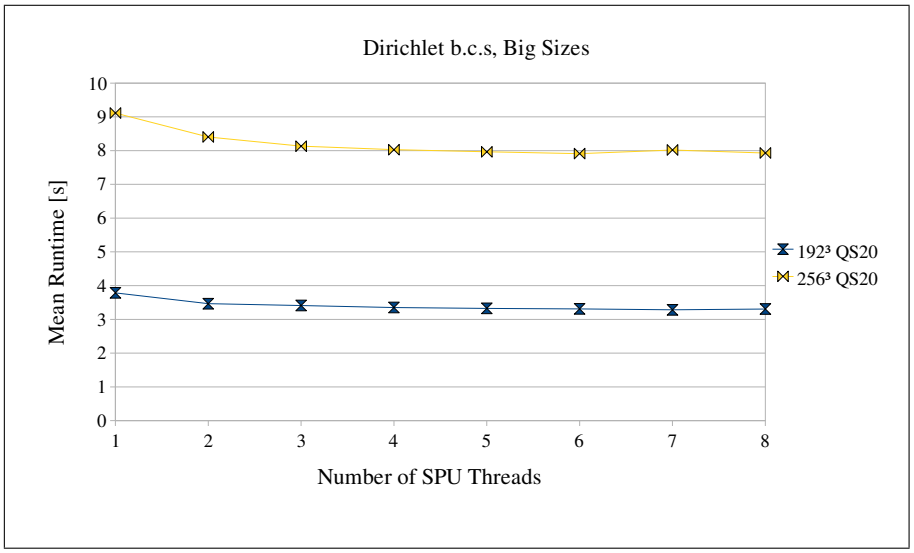

Figure 5.4: Runtime over Thread number for Problem Sizes 192^3 and 256^3 with Dirichlet B.C.s

to the memory requirements shown in table (5.2), the two big test cases can only be run on the QS20 blade. For the 128^3 problem, the PS3 is slightly slower again.

5.3 Detailled Runtime Analysis

Besides the measurements of the overall runtime, also a more detailled scheduling was done for the SPE code. In this section, the runtime is analyzed examplary for the Jacobi Solver and a conclusion is drawn for the limits of performance. This is done by comparing the achieved performance measures memory bandwidth and floating point operations per second to the theoretical peak values on the CBE. Therefore, these measures are computed from the runtime.
To illustrate how great the influence of proper alignment is on the Cell processor, the detailled runtime analysis was done for three different cases: For arrays that are not necessarily aligned with 128 byte, for arrays that are aligned with 128 byte and for 128 byte aligned arrays, forcing interleaved memory strategy, which should deliver twice the memory bandwidth on the QS20. Precisely, the alignment property means, that each piece of data, loaded from the main memory (or stored into it) is aligned with 128 byte in the main memory and the local store. These data pieces are all lines of all grids that are used. While the used buffers in local store are aligned by 128 byte, this does not necessarily hold for the lines of the grids in the main memory unless the number of elements is a multiple of 32 (usually, the size is greater by 1, because powers of 2 denote the number of *cells*).
Interleaving is a technique, that spreads subsequent logical addresses to several physical memory banks, so the refreshing time, each of those banks takes after an access, is hidden, when large subsequent data blocks are transferred between processor and main memory. If interleaving is forced on the QS20, it enables both memory busses for the executed program and therefore doubles the bandwidth that is possible, 50 GiB/s in theory, and about 40 GiB/s in practice.
For the interleaving case, both Cell processors on the blade are used, while in the other cases only one of them is used, since the required memory is less than 512 MiB, which is available for each one.

5.3.1 Unaligned Arrays

In contrast to the timing in the previous section, the detailled measuring is done with in-code commands using the `gettimeofday()` C function. The measurements are done in microseconds, though the accuracy of the function is limited by the Linux operation systems. Time is measured at every synchronization point in the code, to trace the single parts of the code exactly. In chapter (4), the optimization techniques for the code were explained. They were applied to all the kernels, i.e. injection, interpolation and the smoother, so similar performance is expected for all of them and the performance of the smoother is taken into account only. The measurements in this chapters were all done on the IBM QS20, since the performance differences to the PS3 are considered to be negligible (according to the experiments of the previous section).

An overview over the timing of one Jacobi iteration is given in figure (5.5) for different problem sizes. This figure has a logarithmic scale to show all the times at one glance. In this case, the actual grid sizes are considered and the times are averaged over 4 runs each. It is shown clearly, that the scaling of the smoother is better compared to the overall scaling in the previous chapter. This is easy to understand, because the smoother is fully parallelized and should behave better than the average. However, the best relative effect is achieved by increasing the thread number from 1 to 2, whereas beyond 4 threads, the effect is considerably

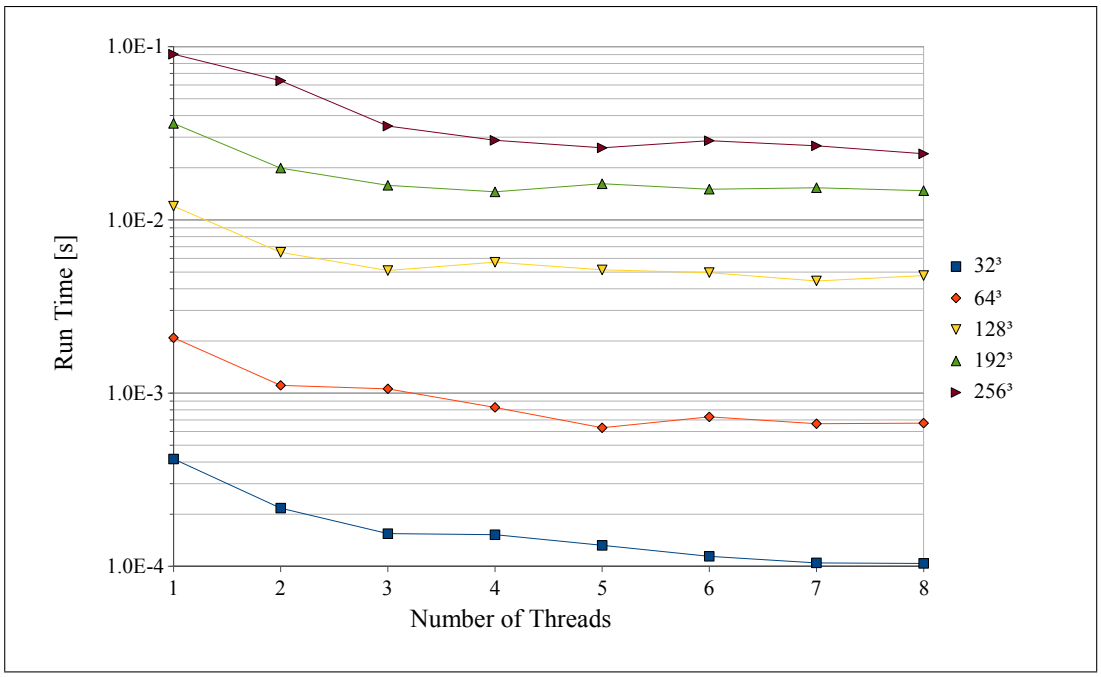

Figure 5.5: Overview Timing per Jacobi Iteration

low. This results from the increased synchronization effort that has to be made and to the fact that the load is not completely balanced: The domain decomposition is done in x–dimension only, i.e. in slices. One or more threads can have one slice more to compute, which takes more time. A third reason, that will be considered in the following subsections, can be the limited memory bandwidth or floating-point performance of the CBE.

Overall, the picture of the runs is inconsistent. In terms of weak scaling, only little parallel effects are reached. The runtime for 128^3 with 8 threads is 4 times higher than that one for 64^3 points with one thread. For the next bigger sizes 64^3 and 128^3 the analog factor between the speeds is still greater than 2, even though it would be 1 ideally.

The floating-point performance of the Jacobi solver can be computed easily: for every inner grid point a fixed number of floating point operations has to be done: 10 operations per point. So, the overall numbers of operations per Jacobi iteration calculates as $\#\text{op} = (\text{size} - 1)^3 \cdot 10$, where size is the number of cells in one dimension. The floating point performance P_{flop} is given for the elapsed time t as

$$P_{\text{flop}} = \frac{\#\text{op}}{t}.$$

This has been computed for the runtimes and is visualized in (5.6) with the left y–axis. The maximal achieved performance is 7 Gflop/s, whereas around 200 Gflop/s are possible. This means, the peak performance reached in the Jacobi solver is below 4% of the theoretical one. This low value can therefore be excluded as limiting factor.
The floating point performance is not the limiting factor for performance and so a second look is given to the memory bandwidth, that is utilized by the Jacobi smoother, in the next paragraph.
The memory bandwidth of the Jacobi solver is calculated analog to the floating-point performance in the previous subsection. For this purpose, the number of memory transfers for each inner grid point is set up as follows: For each Jacobi step, 7 neighboring values are needed

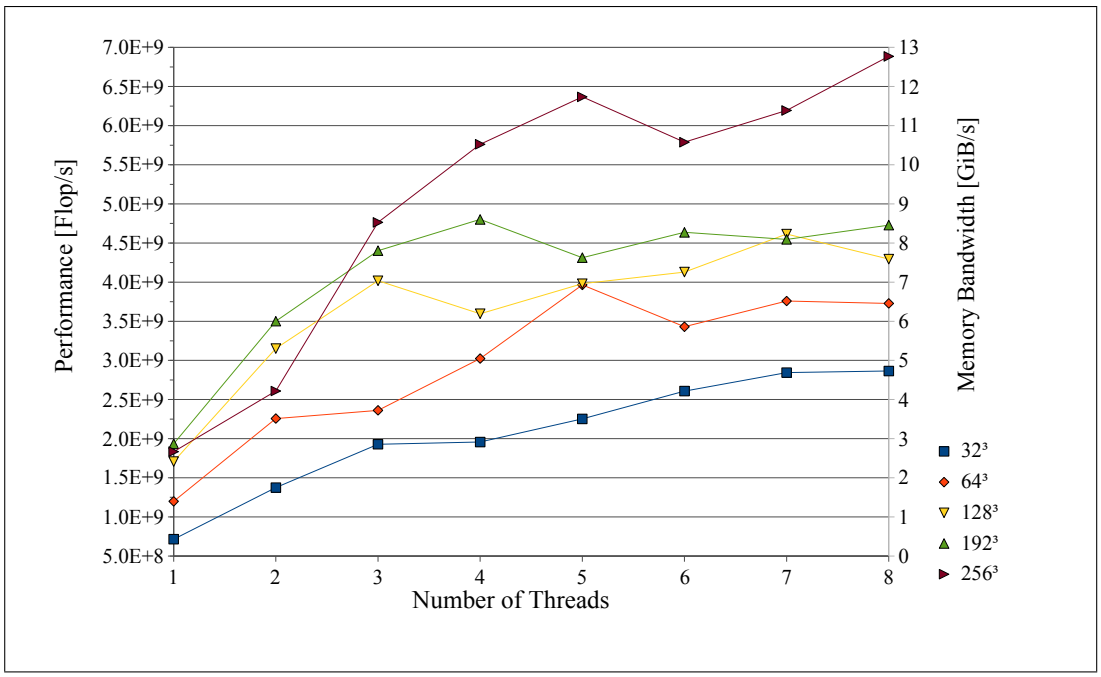

Figure 5.6: Floating Point and Memory Performance of Jacobi Smoother

from the left-hand side with 3 of those in the same line (front, center, back), that is considered to be loaded already into the local store (this line was required by previous steps, see also figure (4.1)). Additionally, the line, which lies in the south with respect to the considered grid point, is in the local store already. From the 7 required values, 4 are already loaded at the moment, when the grid point is accessed. This means that three values for the function (north, east, west) and one value for the right hand side have to be loaded from main memory. One value is saved to the result array, so altogether 5 load/store operations are executed for each inner grid point. As each value is a 32 bit floating point number, an amount of 20 byte has to be transferred per inner grid point. The transferred bytes can therefore be calculated as $\#\text{mem} = (\text{size} - 1)^3 \cdot 20$. The memory performance calculates for runtime t as

$$P_{\text{mem}} = \frac{\#\text{mem}}{t}.$$

The corresponding graph is contained in figure (5.6), regarding the right hand y–axis (the bandwidth scaling is identical to that of the floating point performance in the used performance model). The theoretical peak performance is 25.6 GiB/s, whereas the achieved one is maximal 12.8 GiB/s.
Even this maximal bandwidth can be reached only for the biggest tested grid size and a sufficient number of threads. The performance drops significantly in all the other cases. The scaling of the performance, depending on the number of threads, is good for 2 and 3 threads, where the performance is doubled and tripled, but is not so effective for more than 4 threads. At least, performance is not dropping for these cases. The memory bandwidth is the limiting factor of the speed for the Jacobi solver.

5.3.2 Aligned Arrays

A test run was performed for properly aligned arrays, i.e. for the grid size equal to a power of 2. The runtime results for this run are given in figure (5.7). At a first glance, the figure shows, that the speed can be increased significantly, if the alignment property is taken into

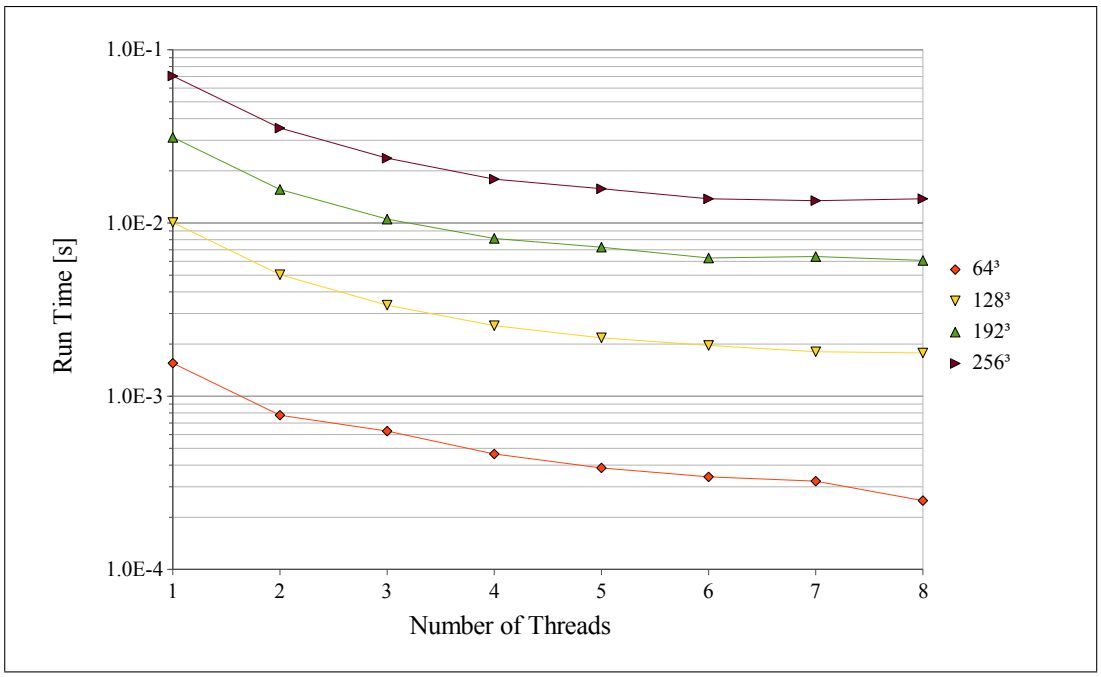

Figure 5.7: Runtime Jacobi Solver with 128 byte Alignment

account. In contrast to figure (5.5), the runtimes are lower by almost 50% for all cases.
The scaling is almost ideal for the test cases with 1 to 6 threads each, independent of the problem size. For 7 and 8 threads, not much additional speed is gained, though. The distances between the graphs in figure (5.7) are almost identical for each order of magnitude due to the logarithmic scale. This shows, that the scaling is quite predictable for proper alignment.
For analyzing the weak scaling, the runtime for size 64^3 (128^3) with one thread is compared to that for size 128^3 (256^3) and eight ones: The ratio is 1.14 (1.37), which is a much better performance for weak scaling than in the unaligned case.
For a better understanding, also the floating point performance and the memory bandwidth for aligned arrays are shown in figure (5.8). Analog to the previous subsection, the floating point performance is not the limiting factor, the maximal achieved one is around 6.5% of the theoretical peak performance. The more interesting measure is the memory bandwidth again. The highest measured memory bandwidth within this test case was 22.7 GiB/s, which is 88% of the theoretical peak. Other measurements, e.g. in [8] affirm, that this is the maximum reachable bandwidth, to be reachable for scientific computation codes on the CBE.

In contrast to the previous results, where maximum performance was only reached for the biggest grid size 256^3, in the aligned case the performance of the smaller grid sizes 128^3 and 192^3 is almost identical. Only for the 64^3 grid, the performance drops by roughly 20 percent. Independent of the problem size, the performance scales almost linear upto 6 threads, until the peak in memory bandwidth is reached. For the small test case it is even better for 7 and 8 hreads also, since this peak is not reached.
Altogether, the alignment can be identified as crucial factor for high–performance programs on the CBE and the bandwidth is the limiting factor for the proposed multigrid method. This could be overcome by developing a more advanced load/store strategy, that minimizes memory access.

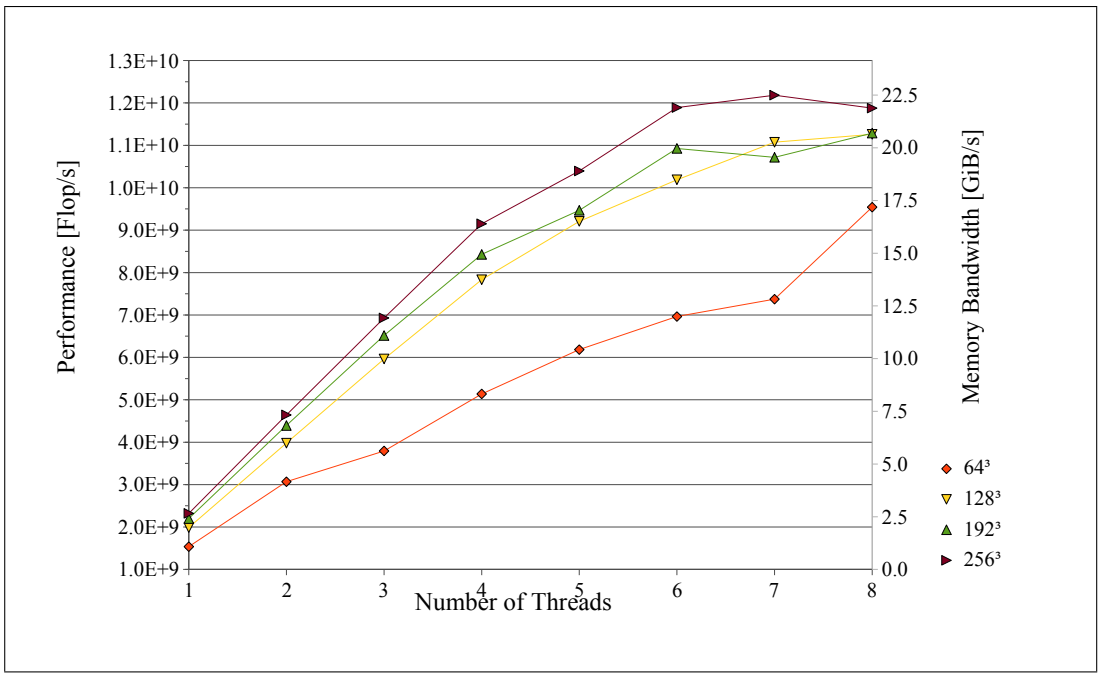

Figure 5.8: Floating Point and Memory Performance of Jacobi Smoother with 128 byte Alignment

5.3.3 Interleaved Memory

Finally, measurements were made with properly aligned arrays (compare the previous subsection) and additional memory interleaving. This means, the main memory is accessed in a way that allows higher data transfer rates: Consecutive memory addresses are spread over several memory banks. In case of the CBE, one 128 byte "cache line" is mapped to one memory bank each. On the IBM QS20, double memory bandwidth is possible in theory, if interleaving is used.
Figure (5.9) displayes the runtimes of the code, for 128 byte aligned arrays and activated interleaving. It is obvious, that the performance can be further increased by interleaving: The runtimes of the program are shorter compared to the aligned arrays only. While the performance of the single–threaded code is almost the same in both cases, interleaving improves the scaling for a large number of threads, so a scaling effect can still be reached for even 7 and 8 threads. E.g. for size 256^3, the strong scaling results in exactly double speed for 2 threads, 3.95 times higher for 4 threads and 5.62 times higher one for 8 threads. These values are even better than for the test case without interleaving.
The comparison of the weak scaling analog to the previous sections results in the ratio 0.94 for 128^3 points in eight threads / 64^3 points in one thread and 1.24 points 256^3 in eight threads / 128^3 points in one thread what is significantly better than for the aligned arrays without interleaving. In the first case, even a superlinear speedup is achieved.
To get a deeper impression of the performance, in figure (5.10) the floating point and memory performance are analyzed. Like in the previous test cases, the floating point operations do not limit the performance. The maximal one is 14.3 Gflop/s, i.e. 7% of the theoretical peak performance. The more interesting measure is again the memory bandwidth. The behavior is similar to that in the previous case: The utilized bandwidth is scaling almost linear with the number of threads, in this case upto eight ones, which is better. Like for the aligned arrays, the memory performance is in the same order of magnitude for the grid sizes 128^3, 192^3 and 256^3, being only lower for 64^3.

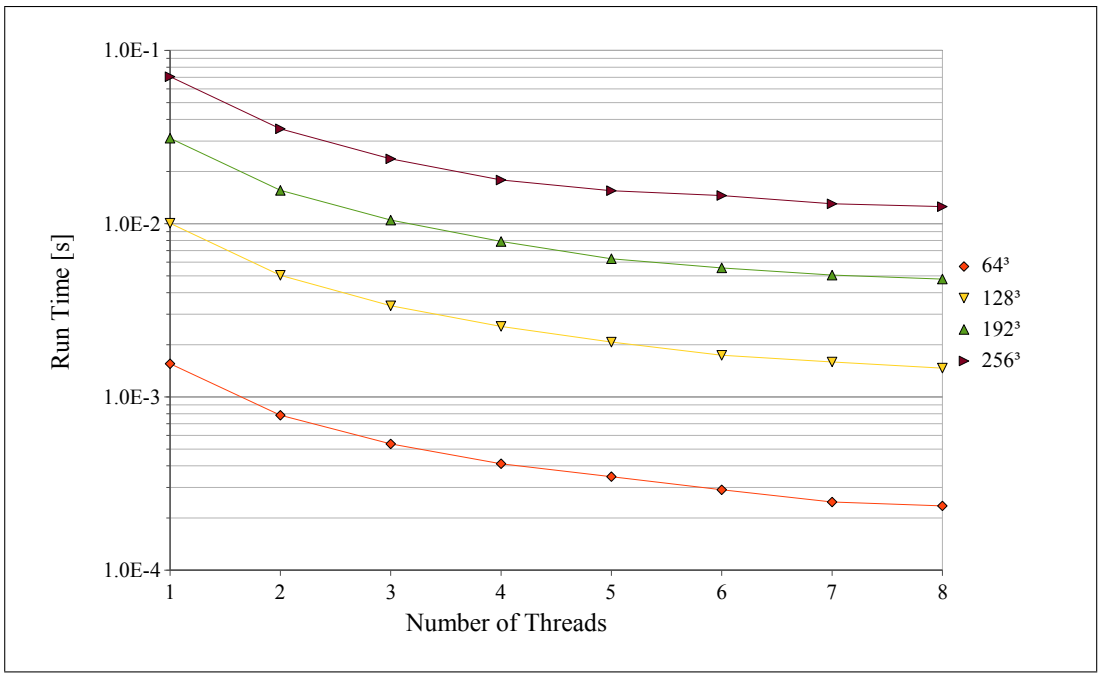

Figure 5.9: Runtime Jacobi Solver with Interleaving

The maximal achieved memory bandwidth is 26.7 GiB/s, which is better than in the previous subsection, but the performance is relatively low since around 50 GiB are possible with interleaving in theory. However, interleaving gives additional performance on the QS20.

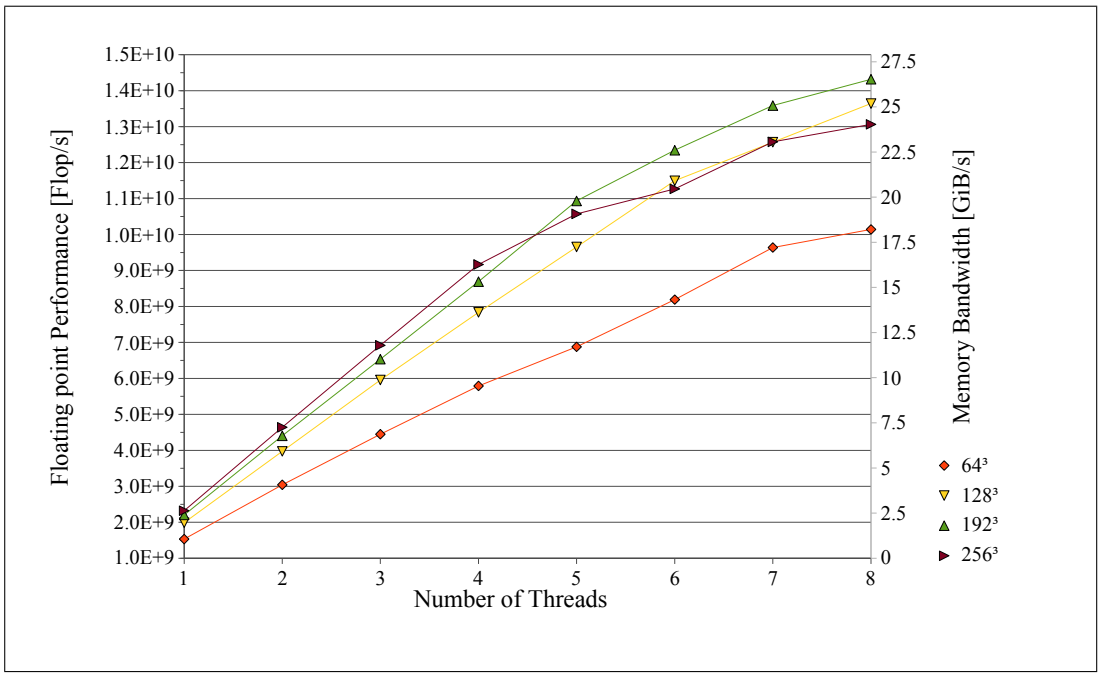

Figure 5.10: Floating Point and Memory Performance of Jacobi Smoother with Interleaving

Chapter 6

Conclusion

In the scope of this thesis, a parallel multigrid algorithm was developed and implemented on the Cell Broadband Engine. This work included an adaption of data sets and structures to the specific requirements given by this platform, a look into the mathematical basis of the proposed problem, the implementation of the program on the CBE, especially the SPEs and finally the performance analysis of this program. The result that was achieved is ambivalent, however. It has been shown, that the limits of the CBE can be reached by using advanced programming techniques, but at the prize of relatively high programming effort. On the other hand, there are topics left for future research and ideas for possible improvements that evolved during working on this thesis.
Regarding the success that has been made, the following goals were reached:

- For Poisson's equation with open–boundary conditions, a solution method was developed that uses hierarchical grid coarsening for an explicit setting of the boundary conditions. Therefore, the method introduced by M. Bolten, was used and some computational simplifications were applied on that. The discretization at the interfaces was integrated into a FAC multigrid solver.

- For multi–threaded implementation, the introduced optimization techniques were used. The code was optimized to run on the SPEs of the Cell processor.

- Tests were performed with the Cell processor, on a Playstation 3 as well as on an IBM QS20 blade, in order to find out, what performance can be gained. The results of those tests show, that the maximal available memory bandwidth can be reached for proper alignment. The developed code runs a hundred times faster, compared to single–threaded PC code without any optimizations (reference implementation by M. Bolten).

There is quite some space for further improvements left, though, in both the mathematical and the implementation part of the proposed problem. This includes the following issues:

- A higher–order method for the solution of the Poisson equation could be derived. For this purpose, an adequate discretization has to be found, especially for the interfaces.

- The algorithm could further be optimized in means of speed and memory requirements. This could be done by exploiting data redundancies. The kernels could be more adaptive, so that a linewise, slicewise or volumewise processing is chosen, depending on the problem size. Furthermore, the interprocess communication may be optimized, so more advanced blocking techniques could be used.

- The runtime performance can be compared to other parallel codes, optionally in combination with analyzing the programming effort / performance outcome ratio, in order

to get an impression, how difficult it is to achieve the same performance on competing processors.

- Since the IBM QS22, an improved version of the CBE, is available now, that does not share some of the mentioned issues (single precision only, memory restriction), tests could be executed on that, for comparison.
- The solver could be tested within a MD framework to analyze the performance of that. Therefore some SPEs may run the solver, while others run the linked–cell part of the code.

Altogether, the Cell processor is a next generation microprocessor, that is available today and has big potential for scientific computations. Its main drawback, at the moment, is the high effort, that is needed for performant programming. There may be more tools to help the developer in future. But it is also sure that the complexity, a programmer has to deal with is higher for parallel applications compared to sequential ones.

Bibliography

[1] M. Bolten. Hierarchical grid coarsening for the solution of the poisson equation in free space. *Electronic Transactions on Numerical Analysis*, 29:70–80, 2008.

[2] O. Buneman. Analytic inversion of the five-point poisson operator. *Journal of Computational Physics*, (8):500–505, 1971.

[3] R. H. Burkhart. Asymptotic expansion of the free-space green's function for the discrete 3-d poisson equation. *SIAM Journal on Scientific Computing*, 18(4):1142–1162, 1997.

[4] G. Sutmann, and B. Steffen. A particle-particle particle-multigrid method for long-range interactions in molecular simulations. *Computer Physics Communications*, 169:343–346, July 2005.

[5] IBM Corporation, Rochester MN, USA. *Example Library API Reference, Version 3.0*, 2007.

[6] IBM Corporation, Rochester MN, USA. *Programming Tutorial, Software Development Kit for Multicore Acceleration, Version 3.0*, 2007.

[7] M. Griebel, S. Knabek, S. Zumbusch and S. Caglar. *Numerische Simulation in der Molekulardynamik*. Springer, 2003.

[8] Markus Stürmer, Gerhard Wellein, Georg Hager, Harald Köstler and Ulrich Rüde. Challenges and potentials of emerging multicore architectures. In *High Performance Computing in Science and Engineering Munich 2007: HLRB/KONWIHR Results and Review Workshop 2007*. Springer. Accepted for publication.

[9] T. Washio and C. Oosterlee. Error analysis for a potential problem on locally refined grids. *Numerische Mathematik*, 86(3):539–563, 2000.

[10] William K. Briggs, Van Emden Henson and Steve F. McCormick. *A Multigrid Tutorial – 2nd Ed.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.