A Fast Adaptive Composite Grid Algorithm for Solving the Free–Space Poisson Problem on the Cell Broadband Engine

Daniel Ritter [1]

In the field of molecular dynamics Poisson's equation with open boundary conditions plays a major role. One issue that arises 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 this paper a method is discussed that uses hierarchical coarsened grids to overcome this problem. Special attention has to be paid to the discretization at the grid interfaces. A finite volume approach has been used for that. The resulting set of linear equations is solved using a fast adaptive composite grid algorithm. Emphasis is put on the implementation of the method on the STI Cell Broadband Engine, a modern multi core processor, that is powerful in floating point operations and memory bandwidth. Code optimization techniques are applied as well as parallelization of the code to get maximum performance on this processor. For validation of the performance test runs are executed and the runtime is analyzed in detail.

1 Introduction

Molecular dynamics (MD) are a classical application field for 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. MD methods can be improved in two directions: making algorithms faster without loss of accuracy or developing properties that reflect natural situations better without making algorithms slower. However, still the goal of computing ensembles at the edge of macroscopic level, is quite a challenge because the number of molecules in a milliliter of fluid is bigger by several orders of magnitude than the number of particles that can be simulated.
A common problem in MD is the solution of the potential equation in free space. Besides the high computational effort that is demanded for the simulation of PDEs in general, the infinite size of the domain forbids a direct numerical solution. Hence, the problem has to be modeled in a finite way. After an introduction of the governing equation with its boundary conditions section 2 discusses possible approaches.
While at the moment standard processors have two or four cores, one chip is available on the mass market that has nine ones in total: the Cell Broadband Engine (Cell/BE). Section 3 deals with considerations that are required to utilize all of its computational power, focusing on parallelization techniques. Besides that some special features that have to be considered are mentioned.
To draw a conclusion about the outcome of this effort, section 4 deals with test cases that were run on the Cell/BE. The presented results that can be taken again as improvements for future implementations. Finally, an outlook to those improvements is given and tasks for future research are shown.

2 Modeling Poisson's Equation with Open Boundary Conditions

This section introduces Poisson's equation with open boundary conditions and explains how they affect the numerical solution. An approach for dealing with the open b.c.s, finite hierarchical grid coarsening, is provided. Finally the particular multigrid method that is used to solve the posed problem is presented.

2.1 Poisson's Equation in Molecular Dynamics

Essentially in molecular dynamics Newton's second law is numerically solved for each particle. Therefore pairwise interactions have to be computed between all of them what takes $\mathcal{O}(N^2)$ operations for N particles. In the case of long–range interactions (e. g. gravitation, Coulomb's potential) none of these interactions can be neglected without making a notable error. Therefore another approach has to be chosen to calculate the interactions accurate with

[1]Department of Computer Science 10, University of Erlangen–Nuremberg, daniel.ritter@informatik.uni-erlangen.de

lower computational effort, ideally with $\mathcal{O}(N)$ operations. Basically there are two different ways for achieving that: Tree–based methods and grid–based ones, e. g. described in [10]. In the following, a grid–based method will be used. From potential theory it is known that the acting forces can be derived from a global potential Φ that is given as function of the particle positions. The governing equation for Φ is also known as Poisson's equation and given as

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

with the dielectric constant ε_0 and charge density $\rho(x)$ at position $x \in \mathbb{R}^3$. Δ is the Laplace operator. In case of point–shaped particles at positions $x_1, x_2, \ldots x_N$, $\rho(x)$ is a superposition of the Dirac impulses $\delta_i(x)$: $\rho(x) = \sum_{i=1}^{N} \delta(x - x_i)$. For numerical simulation this right–hand side is impractical to handle since it is not smooth and may cause big errors. The impulses are approximated usually by smooth functions with similar properties, e. g. Gaussian distributions or spline functions. Also more advanced smoothing methods like the Zenger correction described in [8] were developed for that purpose. From here, sufficient smoothness of the right–hand side is assumed and $f(x)$ will denote the smooth right–hand side.

2.2 Open Boundary Conditions

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 with long–range interactions. The charges of the molecules are inside the area, whereas the induced field is unbounded and its influence cannot be neglected without making a significant error. This problem can be formulated as

$$\begin{aligned} &\Delta\Phi(x) = f(x),\ x \in \mathbb{R}^3, \\ &\text{with } \Phi(x) \to 0 \text{ for } \|x\| \to \infty, \end{aligned} \tag{2}$$

with the restriction that $\text{supp}(f) \subset \Omega$ 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

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

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 subsection.

2.3 Previous Solvers

From the infinite discrete problem (3) there exist different ways to solve it numerically:

1. Observe a finite subvolume Ω of $\mathbb{R}^3$ and develop an approximation for setting the boundary conditions explicitly. This method was introduced by O. Buneman in [2] for the two-dimensional case. He solved the discrete problem (3) analytically, i. e. for the five point Laplace operator Δ_h. Later, R. Burkhart extended the model for the 3D case in [3]. For the application of particle simulation, a multipole expansion can be used. This was introduced by Sutmann and Steffen in [4]. The main drawback of this method is again the computational complexity: For N unknowns, the overall time complexity calculates as $\mathcal{O}(N^{\frac{5}{3}})$. However, the optimal one of the whole algorithm should be $\mathcal{O}(N)$.

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. This was used by Washio and Oosterlee in [12]. They have chosen a grid extension rate α for each grid and have proved that the results are accurate within order $\mathcal{O}(h^2)$ for infinite coarsening with extension rate $\alpha > 2^{\frac{2}{3}}$. Though, for finite coarsening an error estimation is not given.

In the following, a model that is a combination of both approaches will be considered.

2.4 Used Model

Problem (3) will be solved on an finitely large hierarchically coarsened grid with setting the boundary conditions explicitly on using an approximation formula. This technique was introduced by M. Bolten in [1] and is a combination of the previous two possibilities. The advantages of both are combined: Bolten proved that the error is of order $\mathcal{O}(h^2)$ for this method. If the number of boundary points of the coarsest grid is considered to be constant the computational complexity is $\mathcal{O}(N)$. In more detail the model looks as follows: Let the problem (2) be given with $\Omega = [-\frac{1}{2}; \frac{1}{2}]^3$ and 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: $\Delta_h = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 1 & -6 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$. The resulting problem is a special case of (3).

1. 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

$$\begin{aligned} \Omega_1 &:= \left[-\frac{\beta_1}{2}, -\frac{\beta_1}{2}\right]^3, \\ \beta_1 &\geq \alpha, \\ h_1 &:= h. \end{aligned} \tag{4}$$

   This is only an enlargement of the original domain with same grid size.

2. The extension is now done iteratively in combination with a grid coarsening up to level l with parameters

$$\begin{aligned} \Omega_k &:= \left[-\frac{\beta_k}{2}, -\frac{\beta_k}{2}\right]^3, \\ \beta_k &\geq \alpha^k, \\ h_k &:= 2^{(k-1)}h \end{aligned} \tag{5}$$

   for $k \in \{2, 3, \ldots, l\}$. The β_k are used to fix corresponding grid points on grid levels $(l-1)$ and l. 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 an three–level extended 2D–grid is given in figure 1 (1).

3. After the lth extension of the grid, the boundary values are set explicitly on the boundary $\delta\mathcal{G}_l$. For each boundary point $x_\delta \in \delta\mathcal{G}_l$ the potential is given as

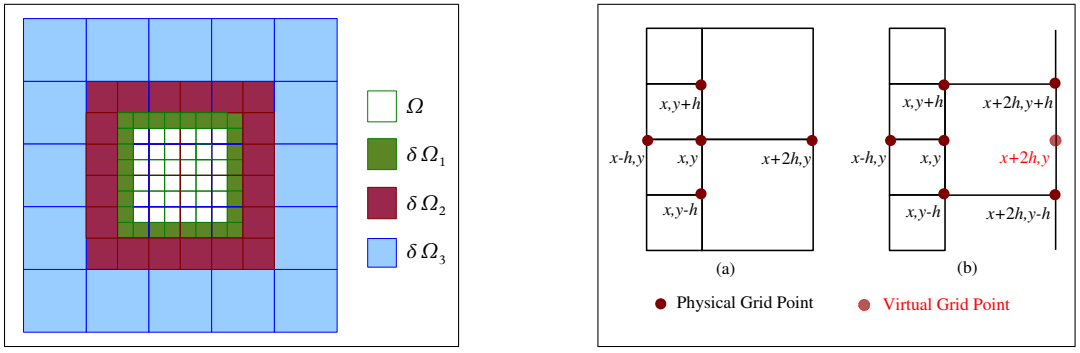
$$\Phi(x_\delta) = \frac{1}{4\pi} \iiint_{\Omega_1} \frac{f(y)}{\|y - x_\delta\|_2} dy \tag{6}$$

   for the continuous case. In the discrete problem the potential for each boundary point $z_\delta \in \delta\mathcal{G}_l$ has to be computed via numerical integration:

$$\Phi(z_\delta) = \frac{h^3}{4\pi} \sum_{z_{\text{fine}} \in \mathcal{G}_1} \frac{f(z_{\text{fine}})}{\|z_{\text{fine}} - z_\delta\|_2}. \tag{7}$$

4. The boundary condition are distributed from $\mathcal{G}_k$ to $\delta\mathcal{G}_{k-1}$ in a conservative way, i. e.a special scheme is used at the interfaces between two grids. To ensure the conservation requirement, a conservative finite volume discretization is done. A 2D example for the structure of an interface is given in figure 1 (2).

In principle higher error order than $\mathcal{O}(h^2)$ could be achieved with enhanced discretization schemes. For solving the discretized PDE, a linear solver is required. The structure of the problem enforces the implementation of a multigrid method since this method can gain profit from the introduced grid hierarchy while keeping the overall time complexity limited to $\mathcal{O}(N)$.



(1) (2)

Figure 1: (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 $\Omega_3 = \Omega_2 \cup \delta\Omega_3$. (2) 2D Interface between Grids: A corresponding coarse grid point to boundary point (x, y) exists (a), does not exist (b). In the latter case the value at the virtual grid point $(x + 2h, y)$ has to be interpolated from surrounding coarse grid points

2.5 Used Multigrid Solver

An appropriate multigrid algorithm has to be adapted to the open–boundary Poisson problem: The function Φ has to be set on the coarsest grid for the boundary points — see step 3 of the previous subsection. As a consequence, a full approximation scheme (FAS) was used since this method works with a restriction of the original equation. The method of choice is a variation of a FAS that already takes into account the coarsening structure: The fast adaptive composite grid method (FAC), introduced in chapter 9 of [13]. The FAC method will be run in several V-cycles. Special about the FAC is the applied restriction: Consider two grids, the fine at level $k-1$ and the coarse at level k. The residual r_{k-1} is computed as $r_{k-1} = f_{k-1} - \Delta_{h_{k-1}}\Phi_{k-1}$. It is then restricted to grid level k as $r_k = \hat{I}^{h_k}_{h_{k-1}} r_{k-1}$ for all grid points which are elements of both grids. Additionally, the function Φ_{k-1} is also restricted as $\Phi_k = \hat{I}^{h_k}_{h_{k-1}} \Phi_{k-1}$. The right–hand side on the coarser grid is calculated as $\Delta_{h_k}\Phi_k + r_k$ for all points, that are elements of the fine grid. In the prolongation step a correction term e_k is prolongated back to the fine grid as $e_{k-1} = I^{h_{k-1}}_{h_k} e_k$. Then new value for Φ_{k-1} is calculated as $\Phi_{k-1} + e_{k-1}$.
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 under–relaxing Jacobi smoother is used for pre– and post–smoothing as well as for solving problem on the coarsest grid. The restriction is done by a direct injection, while for prolongation trilinear interpolation is performed.

3 Implementation and Performance Optimization

Within this section first the features of the Cell/BE are described briefly. This is required since the optimal development of software for the Cell/BE highly depends on understanding the underlying hardware. Furthermore, it shows how the parallel implementation of a multigrid method is done and what paradigms were used within the development process.

3.1 Architectural Overview over the Cell Broadband Engine

The first implementation of the CBEA, the so-called Cell Broadband Engine, is used e. g. in the Sony Playstation™ 3 game console and IBM's QS20 and QS21 blades. Its organization is depicted in Fig. 2 [5, 6]: The backbone of the chip is a fast ring bus—the Element Interconnect Bus (EIB)—connecting all units on the chip and providing a throughput of up to 204.8 GFlop/s in total when running at 3.2 GHz. A PowerPC–based general purpose core—the Power Processor Element (PPE)—is primarily used to run the operating system and control execution, but has only moderate performance compared with other general purpose cores. The Memory Interface Controller (MIC)

can deliver data with up to 25.6 GB/s from Rambus XDR memory and the Broadband Engine Interface (BEI) provides fast access to I/O devices or a coherent connection to other Cell processors. The computational power resides in eight Synergistic Processor Elements (SPEs), simple but very powerful co-processors consisting of three components: Synergistic Execution Unit (SXU), Local Storage (LS), and Memory Flow Controller (MFC).

The SXU is a custom Single Instruction Multiple Data (SIMD) only vector engine with a set of 128 128-bit-wide registers and two pipelines. It operates on 256 kB of its own LS, a very fast, low-latency memory. SXU and LS constitute the Synergistic Processor Unit (SPU), which has a dedicated interface unit, connecting it to the outside world: the primary use of the MFC is to asynchronously copy data between LS and main memory or the LS of other SPEs using Direct Memory Access (DMA). It also provides communication channels to the PPE or other SPEs and is utilized by the PPE to control execution of the associated SPU. Each SPE can be seen as a very simple computer performing its own program, but dependent on and controlled by the PPE.

The Cell/BE is able to perform 204.8 GFlop/s using fused–multiply–adds in single precision (not counting the abilities of the PPE), but is limited regarding double precision. Only six SPEs are available under Linux running as a guest system on the Sony Playstation™ 3, what reduces the maximum performance there accordingly to 153.6 GFlop/s. The newer PowerXCell 8i [7], used in IBM's QS22 blades, differs from the older Cell/BE by SPEs with higher performance in double precision (12.8 instead of 1.8 GFlop/s each) and a converter that allows connecting DDR2 memory to the MIC.

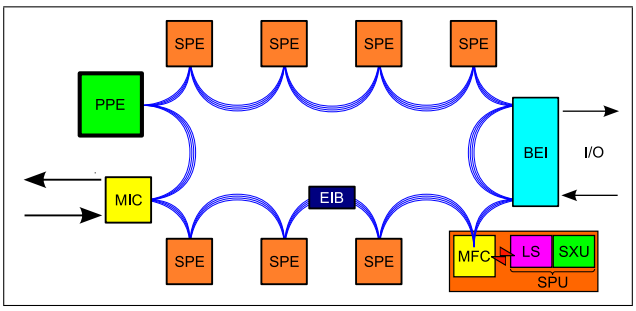


Figure 2: Schematic view of the STI Cell Broadband Engine.

While standard PowerPC software and compilers can be executed on the PPE's computation unit (the PowerPC Processor Unit, PPU), software must be adapted to take advantage of the SPEs, whose SXUs use their own instruction set. The basic approach to write CBEA-enhanced software is to separately implement the parts running on PPE and the SPEs, where libraries and language extensions help in issuing and waiting for DMA transfers, and doing the communication and synchronization between the different agents.

3.2 Parallelization approach

The multigrid method that has been introduced in section 2 shall run at maximal speed on the Cell/BE. The key technique for gaining speed is to parallelize the algorithm on the SPEs and it shall be the main focus within the rest of this section. This SPE–level parallelization uses multi–threading via *POSIX threads*, while in–thread parallelization is done via the *SIMD* capabilities of the Cell/BE. Additional effort is taken to speed up the execution of the computational kernels. For understanding the parallelization approach a good starting point are the used data structures.

Data Layout and Line–wise Processing The underlying data structure for the grids are simply 3D arrays. In C arrays are stored in row major order, i.e. the last index of a D-dimensional array is the one with subsequent elements in memory. So it should be the one to be 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. Line–wise processing means therefore iteration over z while fixing y and x. Since the SXU executes only SIMD commands, the total length of the array in z–direction has to be a multiple

of 4—eventually dummy elements have to be inserted at the end of each line. To fulfill alignment properties even more dummy elements may be necessary.

Domain Decomposition The paradigm of line–wise processing leads directly to the chosen approach for distributing data to several SPEs: Domain decomposition. It is assumed that one line of data fits into the local store of an SPE. This is no restriction for the proposed test cases; on the Playstation™ 3 as well as on the QS20 the grid size is limited by the available main memory. The data is distributed line by line to the SPEs. For reasons of simplicity this distribution is done in blocks a priori. This static work distribution is sufficient for the multigrid algorithm since the number of operations per grid point is not only known before, but also constant over all the grid points.

Double Buffering On the Cell/BE, the programmer has to initiate all load and store operations. For this purpose, the SPU intrinsics `mfc_get(...)`, `mfc_put(...)` are used to activate the DMA transfer.

1. `mfc_get(target_ls, source_ea, length, tag, ...)` 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 (32 available data channels).

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

These DMA transfers have a certain latency that is relatively high and therefore will slow down the program execution. An elegant way to hide the latencies is to use a double buffering scheduling strategy: For each data line that is required from main memory two buffers are allocated in the local store. While one is filled, the data in the second one can already be manipulated and vice versa. The same holds also for writing operations: While the content of one buffer is copied to the memory, data can be written to the second one by the user. However, scheduling has to be done carefully to ensure that transfers are finished when the data is needed and no dead locks are caused. When accessing the first line, loading will delay the execution since the required data is not loaded in advance, whereas for all following lines it is loaded in advance. Analog, storing of the last data line will cause a delay.

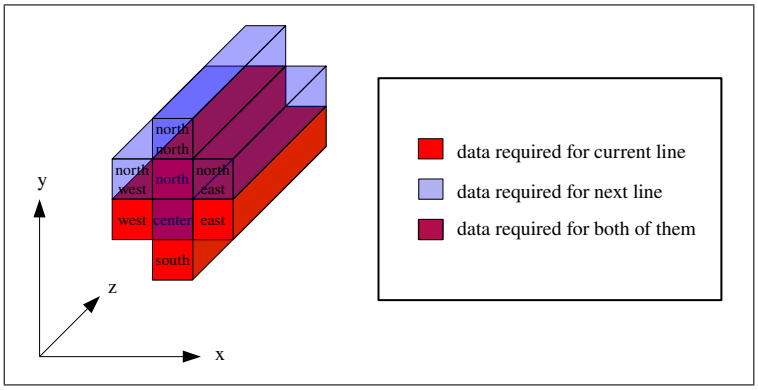


Figure 3: Lines of Domain required for Jacobi Stencil

Local Operators The multigrid method can be parallelized easily by domain decomposition because the operators that are applied onto the data are local ones. 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 in this context that those operators have only read dependencies on a small area within the whole domain and the results are only written to a small area. Assuming line–wise processing, also neighboring lines have to be loaded for every operation on the domain. This is shown in figure (3) 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 three ones (marked blue: north–north, north–west and north–east) must be loaded from memory.

Multi–Threading of the Program The Cell Software Development Kit (cellsdk) offers the possibility of multi–threaded programs. The framework for each application on the Cell/BE is a PPE program, that is used for initialization, thread creation and cleanup afterward. 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 corresponding PPE thread is sent to sleep until the SPE program is finished.
Additionally, synchronization of the different threads is required to insure the data consistency. Hence, synchronization points are introduced after each Jacobi iteration, coarsening and interpolation step. The cellsdk offers libsync, a library that provides variables for inter–process communication. Since the synchronization is a cheap function call and the load is quite good balanced, this locking communication is preferred over dealing with overlapping domains in the Jacobi solver.

SIMDization of the Kernels Every SPE provides 128 SIMD registers of 128 bit each. Using these registers is crucial for efficient execution of the program. The compiler does not create vectorized programs from standard code, but special vector data types have to be chosen for the operands. For floating point data this is the `vector float` type which contains four 32 bit floats. For these data types adapted operations (e. g. logical, arithmetic, shifting, shuffling) must be used. All the numerical kernels (Jacobi, Laplace operator, restriction and prolongation) were completely SIMDized within the program code. In case that array dimensions are not multiples of 4, 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`. The results are shuffled within the regarding vectors afterward.
The SPEs have a bad performance in predicting branches and each misprediction will cause a break in the pipeline. Therefore, branches in the computation kernels are eliminated. This is possible in all kernels of the multigrid method, but quite tricky for the treatment of the interfaces between coarse and fine grid.

4 Tests and Results

The pivotal question is what the outcome of the applied effort in terms of computational performance is. In order to find out performance tests were executed on both the Playstation™ 3 and the IBM 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™ 3, but 8 on the QS20, and the overall memory size of the Playstation™ 3 is 256 MB, while the QS20 has got 1 GB of main memory. This restricts the maximal problem size on the PS3, i.e. bigger grid sizes are only tested on the QS20. It turned out that the speed on both the QS20 and the PS3 is fairly the same, so within this section the results on the QS20 are shown only. Numerical results were already presented in [1], so the focus in this paper is on the computational speed.
The first test showed that only half of the theoretical bandwidth of the Cell/BE could be reached. Given a closer look the reason was identified to be the improper alignment of data. Hence, in a second step the tests were re–run with adapted code that operates on data that is 128 byte aligned both in main memory and local store of the SPE. The results show that much higher bandwidth can be achieved. Finally, tests were performed using both of the Cell/BEs on the QS20 and both available memory buses, forcing interleaved memory scheduling. This a technique that spreads subsequent logical addresses to different 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 this scheduling is forced on the QS20, it enables both memory buses for the executed program and doubles the theoretical peak to 50 GB/s in theory. For the interleaved case, SPE threads are assigned to both Cell processors on the blade, while in the other cases only one of them is used.

4.1 Calculation of the Performance Measures

Runtime analysis was done exemplary for the Jacobi Solver and conclusions are drawn for the limiting factor 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 Cell/BE. These measures are computed directly from the runtime, which is calculated via `gettimeofday()` C function calls that are injected at the synchronization points of the program. The measurements are accurate down to microseconds and are averaged over 10 test runs each. Timing for one Jacobi iteration was considered to be typical for all computational kernels because all of them were implemented using the same optimization techniques.
The floating-point performance of the Jacobi solver is computed directly from runtime: For every inner grid point a fixed number of floating point operations has to be done: 10 operations per point. So, the overall number 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 elapsed time t as $P_{\text{flop}} = \frac{\#\text{op}}{t}$.
The memory bandwidth of the Jacobi solver is calculated in a similar way. Consider figure 3 again: Per line that is processed, 3 lines had to be loaded while processing the previous one, i. e. north, east and west w.r.t. the processed line. Additionally, one line of the right hand side has to be loaded and one result line has to be stored: Altogether $\#mem = 5 \cdot 4\text{byte} = 20\text{byte}$ of data are transferred per inner grid point. So, the memory performance calculates as $P_{\text{mem}} = \frac{\#\text{mem}}{t}$ for elapsed runtime t.

4.2 Results for Unaligned Arrays

The floating point performance that has been computed from runtimes for different problem sizes depending on the number of SPE threads is visualized in figure 4 regarding the left y–axis. The maximal achieved performance

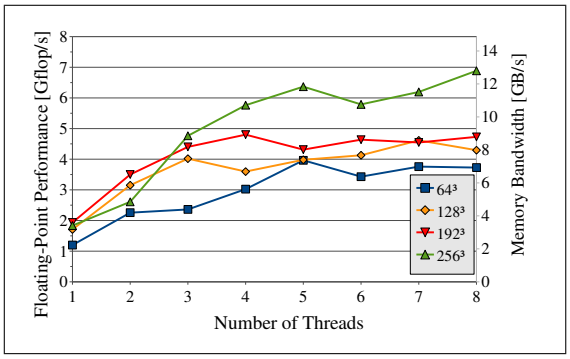


Figure 4: Floating Point and Memory Performance of Jacobi Smoother for Unaligned Arrays

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 be excluded as limiting factor.
The graph corresponding to the utilized memory bandwidth is contained in figure (4), 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 GB/s, whereas the achieved one is maximal 12.8 GB/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 effective for more than 4 threads. At least, performance is not dropping for these cases. The memory bandwidth seems to be the limiting factor of the speed for the Jacobi solver. Because no consideration was given to proper alignment each DMA transfer is split up to two ines, which gives a 50% drop in bandwidth performance.

4.3 Results for Aligned Arrays

The tests were repeated with properly aligned arrays and the performance measures are shown in figure 5. Analog

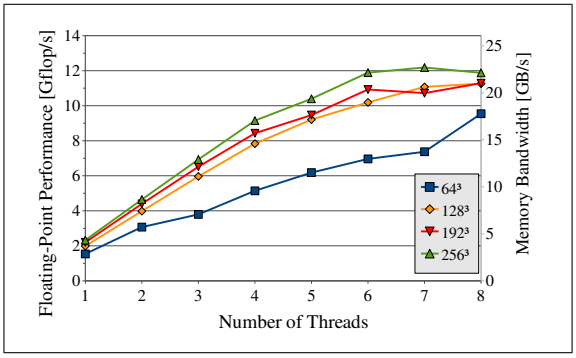


Figure 5: Floating Point and Memory Performance of Jacobi Smoother with 128 byte Alignment

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 memory bandwidth within this test case was 22.7 GB/s, which is 88% of the theoretical peak. Other measurements, e.g. in [11] affirm, that this is the maximum reachable bandwidth for scientific computation codes on the Cell/BE.
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 medium 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 up to 6 threads, until the peak in memory bandwidth is reached. For the small test case it is even better for 7 and 8 threads also, since this peak is not reached.
Altogether, the alignment can be identified as crucial factor for performance 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. So, higher floating point performance could be gained.

4.4 Results for Interleaved Memory

Finally, measurements were made with properly aligned arrays (compare the previous subsection) and an interleaved memory strategy as explained. To get an impression of the achieved performance, in figure 6 the floating point and memory performance are displayed. Like in the previous test cases the floating point operations do not

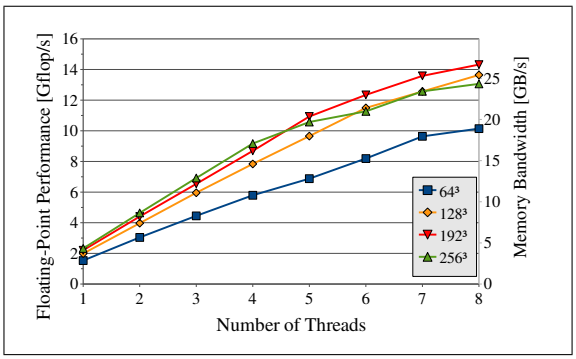


Figure 6: Floating Point and Memory Performance of Jacobi Smoother with Interleaving

limit the performance. The maximal one is 14.3 GFlop/s, i.e. 7% of the theoretical peak performance. Regarding 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 up to eight ones, which is better than before. 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.

The maximal achieved memory bandwidth is 26.7 GB/s, being better than in the previous subsection, but relatively low compared to around 50 GB/s that are possible with interleaved memory in theory. However, interleaved memory gives additional performance on the QS20 at the price of occupying more resources.

5 Conclusion

A parallel FAC method could be implemented successfully on the Cell/BE. The main effort within this implementation was the proper design of data sets and structures as well as scheduling memory transfer operations. These are tasks which are done to a major part by either the compiler or the processor itself (e. g. cache hierarchies, out–of–order execution) on other modern architectures. It was shown that the hardware limits of the Cell/BE can be reached if advanced programming techniques are applied, however, a deeper understanding of the underlying hardware properties is required to achieve that. A higher floating point performance could be gained if the overall parallelization model is revisited and data locality techniques like advanced blocking developed in part III of [9] are applied. Another future goal is to extend the discussed simulation model to higher error orders. For this purpose the discretization at grid interfaces is a key criterion. From algorithmic view it is also still an open question how these interfaces should be handled for more complex stencils. At the moment methods for hierarchical grid coarsening are developed to make the implementation easier without affecting the accuracy, respectively feasible for higher order accuracy.

References

[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. *Cell Broadband Engine Architecture*, October 2007.

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

[7] IBM. *Cell BE Programming Handbook Including PowerXCell 8i*, May 2008.

[8] H. Köstler. An accurate multigrid solver for computing singular solutions of elliptic problems. In *Abstracts Of the 12th Copper Mountain Conference on Multigrid Methods*, pages 1–11. SIAM, SIAM, Apr 2005.

[9] M. Kowarschik. *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures*. PhD thesis, Jun 2004. Advances in Simulation 13.

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

[11] M. Stürmer, G. Wellein, G. Hager, H. Köstler and U. 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.

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

[13] 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.