Parallelizing Multigrid Solvers for Contact Problems on IBM's Cell Processor

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science

der Informatik

vorgelegt am

Fachbereich Mathematik und Informatik
Freie Universität Berlin

von

Christoph Thöns

am 1.7.2008

Betreuer: Prof. Dr. Ralf Kornhuber
Gutachter: Prof. Dr. Christof Schütte

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Berlin, 24.6.2008 _____

Unterschrift

Contents

Chapter 1

Introduction

In recent years microprocessor manufacturers have seen a number of conventional wisdoms reverted that had been around for decades. One of them states that transistors cost money, but power is free. Nowadays, as power dissipation increases with ever decreasing feature sizes on the chip, integrated circuit resources are no longer the limiting factor to performance. Thus power efficiency needs to be optimized at about the same rate as performance increases. This issue is often referred to as the "Power Wall".

Another one of those wisdoms concerns memory latency. For decades, processor clock frequencies have grown at a higher rate than memory latency has decreased. This has caused a gap in contemporary computer systems of about 1,000 clock cycles per I/O operation [20]. As a result, performance depends more and more on moving data than on the processor's computational power. This second issue is called the "Memory Wall".

The third and last wisdom that shall be mentioned here refers to clock frequencies. Increasing clock frequencies of single core processors used to be a key to achieving more performance. The idea of parallelism has been around for a long time, but the additional effort for parallelizing software used to be hardly justifiable since gains in sequential processing performance soon voided the performance advantage. Since physical limits to higher clock frequencies have been reached, performance gains are nowadays more and more achieved by putting several processing cores on a single chip. This change in paradigms has not been brought about by new developments in software architecture that decrease the effort of parallel programming, but is more of a retreat from the ever growing challenges that make optimizing the performance of single core processors difficult. This issue is also known as the "Frequency Wall".

The Cell Broadband Engine (CBE) is a representative of a new generation of microprocessors that push performance limits by massive parallelism. The novel architecture tries to mitigate all of the three issues presented above. The "Power Wall" issue is addressed by stripping the classical general purpose processing cores

of advanced logics, such as instruction reordering, that require a large number of transistors and are partly responsible for the growing need for energy and cooling of processors in recent years. This loss of computational power is compensated by augmenting the chip with eight special purpose processing elements that are designed to boost floating-point performance.

The decomposition of the chip into specialized but less complex parts also mitigates the "Frequency Wall" issue, since less power usage means less heat, which allows higher frequencies.

The CBEs solution to the "Memory Wall" issue is putting a small but fast memory called *local store* beside each of the special purpose processing elements, which they can access exclusively. The pathway between the processing element and its local store is short, allowing for short latency. The *Direct Memory Access* (DMA) controller of each processing element can transfer chunks of up to 16 KB of data ahead of time into local store while the execution unit is working. Each processing element can handle up to 16 parallel transfers. This allows the memory bandwidth of the CBE to surpasses the bandwidth of conventional processors by a factor of almost twenty [20].

One fundamental flaw of the first iteration of the CBE available to the market shall not be omitted: Its double-precision performance is about 10 times below its single-precision performance, which makes it less attractive for scientific computing. The chip designers have focused on optimizing the performance for computer graphics applications such as Sonys Playstation 3, for which single-precision generally suffices. The upside is that in form of the Playstation 3, the CBE is available inexpensively for evaluation. IBM has released the second iteration of the CBE platform in May 2008. They claim to have increased the double-precision performance by a factor of five [3]. However, we did not have it available for testing.

A number of algorithms from a wide range of fields, such as speech recognition [27], data mining [9], and drug design [30] have been implemented on the CBE. The performance of the *Stable Fluids* algorithm on a 'classical' CPU, the CBE and a *General Purpose Graphics Processing Unit* (GPGPU) are compared in [24]. Mixed-precision solvers for linear systems of equations based on the *Gaussian Elimination* algorithm are presented in [26, 10]. They bypass the double-precision performance weakness of the CBE by performing most of the computations in single-precision, but still deliver results in full double-precision accuracy. Frameworks that simplify software development for the CBE are presented in [16, 7]. IBM provides libraries that are optimized for the CBE architecture, such as the *Monte Carlo Library* [22] as well as the well established *Basic Linear Algebra Subprograms* (BLAS) [17] and *Linear Algebra Package* (LAPACK) [21] libraries.

This thesis evaluates the performance gains that are achievable by optimizing an algorithm for the solution of two-body contact problems for the CBE. We use a Finite Element Methods (FEM) based approach. FEM problems boil down to large systems of equations [25]. We use the *Truncated Nonsmooth Newton Multi-*

grid (TNNMG) algorithm [13] as an efficient solver for such a system in the context of contact problems. The parallelization is based on an existing sequential implementation of the algorithm [29]. We focused on parallelizing the Gauss-Seidel algorithm, which is an integral part of the solver. The matrices involved are generally sparse, meaning most elements are zero. Memory usage and performance are optimized by storing the non-zero elements only. This makes additional meta data necessary and access to the payload data irregular. Parallelizing the solver is challenging because the underlying algorithm has dependencies that make decomposing it into functionally independent parts that can be computed in parallel difficult. Computational intensity (the number of floating-point operations per data transfer from main memory) is low. This makes buffer management and load balancing more demanding and the impact of memory latency just as important for performance as computational power.

Chapter 2 gives an overview of the *two body contact* problem: The deformations that two elastic bodies undergo when forced upon each other are examined. We show how the problem can be discretized by using a Finite Element space and present solvers for the systems of equations that arise from the discretization.

In Chapter 3 we will discuss ways of parallelizing the solver. We focus on parallelizing the Gauss-Seidel algorithm, which is an integral part of it. Since the matrices that occur are sparse, they can be decomposed into parts that can be computed concurrently using an approach based on a graph coloring. The parallel Gauss-Seidel algorithm along with a heuristic for finding a suitable coloring is presented in this chapter.

Chapter 4 gives an introduction to the Cell Broadband Engine (CBE). We will have a look at the architecture of the hardware, illustrate the possibilities of its design and the facilities on the chip, explain the particular software development process for this platform and conclude by pointing out several pitfalls that should be avoided when designing software for the CBE.

In Chapter 5, we present the implementation of the parallel projected Gauss-Seidel algorithm for the CBE. We show the final overall architecture of the implementation, explain the communication and synchronizations between the processing elements and present low level optimizations that were done to adjust the source code to their special needs. These optimizations account for a considerable part of the performance gains.

In Chapter 6 the performance results that we measured are presented and interpreted. We list measurements of the Gauss-Seidel algorithm alone as well as the overall multigrid solver.

Chapter 2

Multigrid Methods for Two-Body Contact Problems

2.1 The Two-Body Contact Problem

This thesis deals with a static biomechanical problem: The behavior of tibia and femur of a human knee joint are examined. The two bones are placed in a configuration in which they are in contact with each other. We are interested in the deformations that the bodies undergo and stresses that occur. The abstract problem is called the *two-body contact problem*. Figure 2.1 shows a visualization of an instance of this problem along with the stress field along a cut through the contact boundary.

We only give a brief discussion of the key concepts of a solver for such a problem. This chapter is based on [25, 29], which provide more extensive introductions.

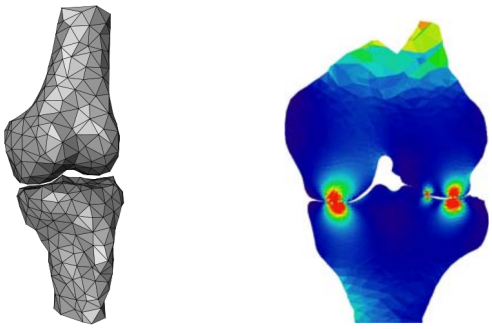

Figure 2.1: Left: Contact between Tibia and Femur of a human knee joint. Right: The stress field along a cut through the contact boundary.

2.1.1 Linear Elasticity

Let us first examine a single body in the absence of contact. Let the set $\Omega \subset \mathbb{R}^d$ be bounded, open, and connected. The boundary of Ω is decomposed into two disjoint subsets

$$\partial\Omega = \Gamma_D \cup \Gamma_N.$$

The body is clamped in place in Γ_D and surface force boundary conditions are described on Γ_N. Let $H_0^1(\Omega)$ be the first order Sobolev space of functions on Ω which are zero on Γ_D.

Lemma 2.1.1. *Let $u \in H_0^1(\Omega)$ describe the deformation of a body in the presence of a field of surface traction $t : \Gamma_N \to \mathbb{R}^d$ and a force density field $f : \Omega \to \mathbb{R}^d$. Then u is a solution of the minimization problem*

$$J(u) \leq J(v) \quad \forall v \in H_0^1(\Omega) \tag{2.1}$$

where the energy functional $J(v)$ is given by

$$J(v) = \frac{1}{2}a(v,v) - l(v), \tag{2.2}$$

with the bilinear form

$$a(v,w) = \int_\Omega \epsilon(v(x)) : C : \epsilon(w(x))\, dx \quad v, w \in H_0^1(\Omega)$$

and the linear form

$$l(v) = \int_\Omega fv\, dx + \int_{\Gamma_N} tv\, ds \quad v \in H_0^1(\Omega).$$

Proof. [29, Sec. 3.1]

The quadratic functional J is convex and coercive on H_0^1 [29, Lem. 3.1.1]. For small strains the fourth order Hooke tensor C can be derived from the linear relationship

$$\sigma = C : \epsilon, \tag{2.3}$$

between the material dependent Piola-Kirchhoff stress tensor σ and the material independent linearized strain tensor $\epsilon(u) = \frac{1}{2}(\nabla u + \nabla u^T)$ [29]. The : symbol denotes tensor multiplication. For isotropic materials, (2.3) can be simplified to

$$\sigma(u) = \frac{E}{1+\nu}(\epsilon + \frac{\nu}{1-2\nu}\mathrm{tr}\epsilon I).$$

where E and ν are material constants [29, Sec. 3.1].

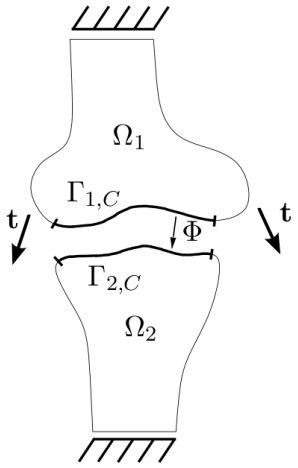

Figure 2.2: Two-body contact problem

2.1.2 Two-Body Contact

Let us now consider two bodies represented by the domains Ω_1 and Ω_2. Additional conditions that describe contact between the two bodies need to be stated. In particular, we need to make sure that the bodies do not penetrate each other. This only affects the set of admissible displacements from which a solution u can be chosen, the energy functional given in the previous section, however, remains unmodified.

The boundary of each domain is now decomposed into three disjoint subsets

$$\partial\Omega = \Gamma_{i,D} \cup \Gamma_{i,N} \cup \Gamma_{i,C},$$

where $\Gamma_{i,C}$ is the part of the boundary on which contact may occur. The *contact mapping* $\Phi : \Gamma_{1,C} \to \Gamma_{2,C}$ is a homeomorphism that identifies points on $\Gamma_{1,C}$ and $\Gamma_{2,C}$ that may come into contact with each other. With Φ we can define the *initial gap function*

$$\begin{aligned} g &: \Gamma_{1,C} \to \mathbb{R}; \\ g(x) &= \|\Phi(x) - x\|. \end{aligned}$$

The linear non-penetration condition states that the relative displacement in direction of the normal n of any two points on the boundary patches $\Gamma_{i,C}$ should not exceed their initial distance:

$$u|_{\Gamma_{1,C}} n_1 + (u|_{\Gamma_{2,C}} \circ \Phi) n_2 \leq g. \tag{2.4}$$

Let $\mathcal{K}$ be the set of functions u for which (2.4) holds. We call it the *set of admissible displacements*. The minimization problem (2.1) can now be modified to its constraint form: Find a $\mathbf{u} \in \mathcal{K}$ such that

$$J(u) \leq J(v) \quad \forall v \in \mathcal{K} \tag{2.5}$$

Lemma 2.1.2 ([29, Lem. 3.2.2])**.** *The minimization problem* (2.5) *has a unique solution.*

2.2 Discretization

2.2.1 Finite Elements

Let us go back to the linear elasticity problem in the absence of contact. The space $H_0^1(\Omega)$ is of infinite dimension, which makes finding a solution of the minimization problem (2.1) in it difficult. *Finite Elements Methods* (FEM) remedy this by using a linear Finite Element space S that is of finite dimension instead. The domain Ω is decomposed into a number of *elements*. The elements can be triangles in two dimensional problems or tetrahedrons in three dimensional problems. We assume the use of triangles for simplicity, but the considerations can easily be extended to other elements [8].

Definition 2.2.1. *The first order finite element space over a triangulation $\mathcal{T}$ of Ω is defined as*

$$\boxed{S_{\mathcal{T}} := \{v \in C(\overline{\Omega}) |\, v|_t \in \Pi \quad \forall t \in \mathcal{T}\}}$$

with Π being the set of first order polynomials [25].

The polynomials on the triangles are defined by interpolation between *nodes* $\mathcal{N}_{\mathcal{T}}$ that correspond to the vertices of the triangles. Let n be the number of nodes in $\mathcal{N}_{\mathcal{T}}$.

Definition 2.2.2. *The* nodal basis *of the space $S_{\mathcal{T}}$ is given by*

$$\Lambda_{\mathcal{T}} = \{\lambda_p |\, p \in \mathcal{N}_{\mathcal{T}}\}$$

where λ_p is defined by

$$\boxed{\lambda_p \in S_{\mathcal{T}} : \qquad \lambda_p(q) = \delta_{pq} \qquad \forall q \in \mathcal{N}_{\mathcal{T}} \qquad (\textit{Kronecker-}\delta)}$$

for all $p \in \mathcal{N}_{\mathcal{T}}$ [25].

Lemma 2.2.1 ([29, Lem. 3.1.2])**.** *Let $u \in H_0^1(\Omega)$ be a solution to the minimization problem* (2.1)*. Then u is a solution of*

$$a(u, v) = l(v) \quad \forall v \in H_0^1(\Omega). \tag{2.6}$$

This is called the *variational formulation* of the minimization problem. In the Finite Element space $S_{\mathcal{T}}$ equation (2.6) can be expressed as

$$a(u_{\mathcal{T}}, v) = l(v) \quad \forall v \in S_{\mathcal{T}}. \tag{2.7}$$

The finite element function $u_{\mathcal{T}}$ can be represented through the basis functions $\{\lambda_p\}$ as

$$u_{\mathcal{T}} = \sum_{p \in N_{\mathcal{T}}} u_p \lambda_p.$$

Substituting $u_{\mathcal{T}}$ and $v = \lambda_q$, $q \in N_{\mathcal{T}}$ in equation (2.7) yields the linear system of equations

$$\sum_{p \in N_{\mathcal{T}}} a(\lambda_p, \lambda_q) u_p = l(\lambda_q) \quad \forall q \in N_{\mathcal{T}}$$

Thus, solving the variational problem (2.7) in the space $S_{\mathcal{T}}$ is equivalent to solving the algebraic problem

$$Au = b$$

with

$$\boxed{\begin{array}{lcllcl} A & = & (a_{p,q})_{p,q} \in N_{\mathcal{T}} & a_{pq} & = & a(\lambda_q, \lambda_p) \\ b & = & (b_p)_{p \in N_{\mathcal{T}}} & b_p & = & l(\lambda_p) \\ u & = & (u_p)_{p \in N_{\mathcal{T}}} & & & \end{array}} \tag{2.8}$$

The matrix A is commonly referred to as *stiffness matrix*. Note that A is symmetric [25].

Since we are looking for displacements in d directions, the components of the solution vector u are d-valued vectors. We define the d-valued nodal basis functions as $\lambda_{p,i} = \lambda_p e_i$ using the canonical basis vectors e_i, $i = 1, ..., d$. Thus, the elements of the stiffness matrix A are blocks of $d \times d$ matrices [29, Sec. 3.1], given by

$$(a_{pq})_{ij} = \int_{\Omega} \epsilon(\lambda_{p,i}(x)) : C : \epsilon(\lambda_{q,j}(x))\, dx.$$

Note that since most blocks are zero, the stiffness matrix is sparse. The right hand side b is a vector of d-vector blocks given by

$$(b_p)_i = \int_{\Omega} f \lambda_{p,i} ds + \int_{\Gamma_N} t \lambda_{p,i} ds.$$

The discretized energy functional can be written on $\mathbb{R}^{dn}$ as

$$J(v) = \frac{1}{2} v^T A v - bv \quad v \in \mathbb{R}^{dn}.$$

2.2.2 Discretization of the Nonpenetration Condition

For the discretization of the nonpenetration condition we use a weak form based on mortar elements

$$\int_{\Gamma_{1,C}} [u_1|_{\Gamma_{1,C}} \cdot n_1 + (u_2 \circ \Phi)|_{\Gamma_{2,C}} \cdot n_2]\theta ds \leq \int_{\Gamma_{1,C}} g\theta ds$$

for all θ from a suitable cone of mortar test functions defined on $\Gamma_{1,C}$ [29].

Let v_h be the vector of coefficients of a finite element function. The vector v_h is split into four parts, v_i and v_i^I, where v_i corresponds to the nodes of the grid that discretize the contact boundary $\Gamma_{i,C}$ and v_i^I represents the remaining nodes. The nonpentration condition only affects the elements of v_i. The set of admissible displacements can now be expressed in an abbreviated notation as

$$\mathcal{K}_{alg} = \{c \in \mathbb{R}^{dn} \mid Dv_1 - Mv_2 \leq g\}$$

with suitable matrices D and M. The concrete construction of D and M is given in [29].

2.3 Linear Solvers

In the remainder of this chapter we introduce the *Truncated Nonsmooth Newton Multigrid* (TNNMG) algorithm [13] as a fast an stable solver for algebraic constrained minimization problems. We start by introducing the *linear Gauss-Seidel* method as well as *Multigrid Methods* that are based on it. Then we turn to solvers for constrained problems by introducing the *projected Gauss-Seidel* algorithm [12]. This finally leads us to the TNNMG algorithm as an adaption of Multigrid Methods for constrained problems.

2.3.1 The Gauss-Seidel Iteration Scheme

Consider the algebraic minimization problem of finding $x \in \mathbb{R}^n$ such that the energy functional

$$\begin{aligned} J &: \mathbb{R}^n \to \mathbb{R} \\ J(x) &= \tfrac{1}{2}x^T Ax - bx \end{aligned}$$

is minimized on $\mathbb{R}^n$. To find a solution, let us consider the functional on a one dimensional subspace of $\mathbb{R}^n$ by assuming that all components of x except x_i are constant

$$\begin{aligned} J_i &: \mathbb{R} \to \mathbb{R} \\ J_i(x_i) &= \tfrac{1}{2}x^T Ax - bx. \end{aligned}$$

By deriving J_i, we can find that $\bar{x}_i \in \mathbb{R}$ minimizes J_i if

$$\bar{x}_i = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i+1}^{n} a_{ij}x_j).$$

By successively minimizing all components x_1 through x_n and immediately using the components $x_j^{\nu+1}$ that have already been minimized, we get the Gauss-Seidel iteration scheme:

$$x_i^{\nu+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{\nu+1} - \sum_{j=i+1}^{n} a_{ij}x_j^{\nu}). \tag{2.9}$$

Starting at an arbitrary x_0, an iteration scheme yields a sequence of *iterates* $x^0, x^1, ..., x^k$.

Definition 2.3.1. *An iteration sequence is said to be* convergent *if* $\lim_{k\to\infty} x_k$ *exists for all* $x^0 \in \mathbb{R}^n$*. Convergence is said to be* linear *if*

$$\exists\, \mu \in (0,1): \quad \|x - x^{k+1}\| \leq \mu \|x - x^k\|.$$

We call μ *the* rate of convergence*.*

Lemma 2.3.1. *If the Gauss-Seidel iteration scheme is convergent, its limit* x *is a solution of the system of equations* $Ax = b$*.*

Proof. [14].

For the algebraic linear elasticity problem the elements of A are blocks A_{ij} of $d \times d$-matrices. The *block Gauss-Seidel* method is the version of the Gauss-Seidel method for this kind of system. It can be derived by considering the energy functional on d-dimensional instead of one dimensional subspaces. This results in the same scheme as above, except that the division by the diagonal elements becomes a multiplication with the inverse of the diagonal element. This is equivalent to solving the system of equations

$$A_{ii}\, x_i^{\nu+1} = b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{\nu+1} - \sum_{j=i+1}^{n} A_{ij}x_j^{\nu}. \tag{2.10}$$

for $x_i^{\nu+1}$.

2.3.2 Multigrid Methods

With increasing fineness h of the grid the rate of convergence of the Gauss-Seidel algorithm drops exponentially [8]. The multigrid approach addresses this issue

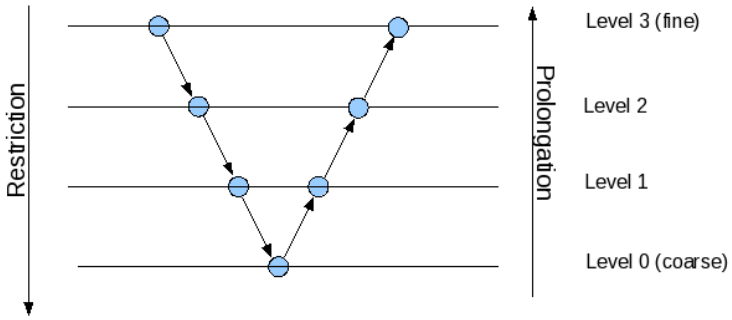

Figure 2.3: The multigrid V-cycle

by introducing a grid hierarchy of increasing fineness. The finer triangulations $\mathcal{T}_s$ on higher levels s of the hierarchy are derived by applying s refinement steps that decompose the triangles into smaller triangles to the original triangulation $\mathcal{T}_0$. Multigrid methods perform Gauss-Seidel iterations consecutively across all hierarchy levels by transferring the intermediate results between the levels. Thus, they benefit from the better rates of convergence on coarse hierarchy levels. The following listing shows the steps of an iteration of a multigrid method:

Algorithm 2.3.1 ([25, Alg. 6.2] Multigrid V-Cycle).

(1) Let there be a grid hierarchy of j levels;
(2)
(3) For each level k in (j,..,2)
(4) begin
(5) Perform ν_1 Gauss-Seidel presmoother iterations;
(6) Restrict the smoothed iterate to the level $k-1$;
(7) end
(8) Solve the coarse problem directly;
(9) For each level k in (2,..,j)
(10) begin
(11) Prolong the iterate from the coarse level $k-1$;
(12) Perform ν_2 Gauss-Seidel postsmoother iterations;
(13) end

Note that in the context of multigrid methods, the Gauss-Seidel algorithm is commonly referred to as *smoother*. The Multigrid V-cycle gets its name from the resemblance of the scheme shown in Figure 2.3 to the letter V. Iterations on the finer levels first presmooth the high frequency components of the error, then the lower frequency components are successively adjusted on the coarser levels until the bottom level is reached, and finally postsmoother iterations incorporate the coarse level adjustments into the overall result and eliminate high frequency error

components that might have been reintroduced by the transition from the lower levels. The *restriction* and *prolongation* operators handle the transitions of the operands from the fine to the coarse grid and back. See [8] for details. On the bottom level of the hierarchy, the *base solver* solves the remaining problem directly.

2.4 Solvers for Constrained Convex Problems

The restriction of the solution to the admissible set $\mathcal{K}_{alg}$ requires solvers for *constraint convex minimization problems*. In this section the projected Gauss-Seidel algorithm [12] is presented. It is the starting point for developing the *Truncated Nonsmooth Newton Multigrid*(TNNMG) algorithm [13], which was used in this thesis.

2.4.1 The Projected Gauss-Seidel Algorithm

In analogy to Section 2.3.1 the *projected Gauss-Seidel* algorithm [12] can be derived, which finds a solution to the minimization problem within a 'box' $[a_i, b_i]^n$ of admissible solutions instead of $\mathbb{R}^n$. Consider the minimization problem of finding an $x \in [a_i, b_i]^n$ such that the energy functional

$$\begin{aligned} J &: [a,b]^n \to \mathbb{R} \\ J(x) &= \tfrac{1}{2} x^T A x - bx. \end{aligned}$$

is minimized. Again we successively minimize all components x_1 through x_n, which gives us the Gauss-Seidel algorithm with the following extensions: If the minimum $\tilde{x}_i$ of a component is found within the admissible interval $[a_i, b_i]$, the algorithm remains unchanged. In case $\tilde{x}_i$ is outside of $[a_i, b_i]$, the actual minimum is either a_i or b_i depending on which side of the intervall $\tilde{x}_i$ is located. Formally, the iteration scheme becomes

$$\boxed{\begin{aligned} \tilde{x}_i^{\nu+1} &= \tfrac{1}{a_{ii}}\left(b_i - \textstyle\sum_{j=1}^{i-1} a_{ij} x_j^{\nu+1} - \sum_{j=i+1}^{n} a_{ij} x_j^{\nu}\right) \\ x_i^{\nu+1} &= \begin{cases} \tilde{x}_i^{\nu+1} & \tilde{x}_i^{m+1} \in [a_i, b_i] \\ a_i & \tilde{x}_i^{\nu+1} < a_i \\ b_i & \tilde{x}_i^{\nu+1} > b_i. \end{cases} \quad , \quad \text{i=1,...,n.} \end{aligned}}$$

Theorem 2.4.1. *For any initial iterate $x^0 \in K$ the projected Gauss-Seidel algorithm is globally convergent if the obstacles are of the box-constrained structure*

$$K = \prod_{0 \le i < n} [a_i, b_i] \qquad a_i \in \{-\infty\} \cup \mathbb{R}, \quad b_i \in \mathbb{R} \cup \{\infty\}.$$

Proof. [12].

In the case that the elements of A are blocks, we use the regular block Gauss-Seidel algorithm but solve the nested system of equations (2.10) with a projected Gauss-Seidel solver.

2.4.2 Multigrid Methods for Constrained Convex Problems

The *Truncated Nonsmooth Newton Multigrid* (TNNMG) algorithm [13] is a multigrid method for constrained problems. This section gives a brief overview of the key concepts behind the consecutive steps of the algorithm. A detailed explanation is given in [29].

- *Mortar transformation*

  Theorem 2.4.2. *There exists a transformation matrix B that transforms the nodal basis $\{\lambda_p\}$ to a new basis $\{\tilde{\lambda}_p\}$ in such a way that the transformed admissible set $\tilde{K}_{alg}$ has the box-constrained form*

  $$\tilde{K}_{alg} = \prod_{0 \le i < dn} [a_i, b_i] \qquad a_i \in \{-\infty\} \cup \mathbb{R}, \quad b_i \in \mathbb{R} \cup \{\infty\}$$

  The construction of B is explained in [29]. The transformed admissible set has the necessary box-constrained form to ensure convergence of the projected Gauss-Seidel algorithm. The energy functional $\tilde{J}$ in the new basis $\{\tilde{\lambda}_p\}$ is given by

  $$\tilde{J}(\tilde{v}) = \frac{1}{2}\tilde{v}^T \tilde{A}\tilde{v} - \tilde{b}\tilde{v}$$

  with

  $$\tilde{A} = BAB^T \quad \text{and} \quad \tilde{b} = Bb.$$

  Let $\tilde{u}^\nu$ be the current iterate in transformed coordinates. We perform one or more projected Gauss-Seidel steps which yields the *smoothed iterate* $\tilde{u}^{\nu+\frac{1}{2}}$.

- *Truncated defect problem*

  We now consider the algebraic defect problem with respect to the smoothed iterate $\tilde{u}^{\nu+\frac{1}{2}}$, which is to find a correction $\tilde{d} \in \mathbb{R}^{dn}$ such that

  $$\tilde{d}^T \tilde{A}(\tilde{v} - \tilde{d}) \ge \tilde{b} - (\tilde{u}^{\nu+\frac{1}{2}})^T \tilde{A}(\tilde{v} - \tilde{d}) \quad \text{for all } \tilde{v} \in \tilde{K}_{alg}^{\nu+\frac{1}{2}}$$

  with a suitable defect obstacle $\tilde{K}_{alg}^{\nu+\frac{1}{2}}$. Let $\mathcal{N}^\bullet(\tilde{u}^{\nu+\frac{1}{2}})$ be the set of nodes that have reached the obstacle. Further changes to the nodes in $\mathcal{N}^\bullet(\tilde{u}^{\nu+\frac{1}{2}})$ are unwanted. Thus, we would like to restrict the coarse grid correction $\tilde{d}$ such that

  $$\tilde{d}_{p,0} = 0 \quad \forall p \in \mathcal{N}^\bullet(\tilde{u}^{\nu+\frac{1}{2}})$$

To this end we define the truncation matrix as

$$T^{\nu}_{pq} = \begin{cases} \mathrm{Id} & p = q,\ p \notin \mathcal{N}^{\bullet}(\tilde{u}^{\nu+\frac{1}{2}}) \\ 0 & else. \end{cases}$$

With it we can define the truncated defect problem in canonical coordinates

$$d^T \hat{A}^{\nu}(v-d) \geq \hat{r}^{\nu}(v-d) \quad \text{for all } v \in K_{alg}^{\nu+\frac{1}{2}}$$

with

$$\hat{A}^{\nu} = (B^{-1}T^{\nu})\tilde{A}(B^{-1}T^{\nu})^T \quad \text{and} \quad \hat{r}^{\nu} = B^{-1}T^{\nu}(\tilde{b} - (\tilde{u}^{\nu+\frac{1}{2}})^T\tilde{A}). \quad (2.11)$$

- *Admissible coarse grid correction*

  Next we perform a linear multigrid step for the truncated linear defect problem

  $$\hat{A}^{\nu} d = \hat{r}^{\nu}$$

  disregarding the obstacles $\tilde{K}_{alg}^{\nu+\frac{1}{2}}$. Since the zero elements on the diagonal elements of $\hat{A}$ corresponding to truncated nodes would result in a division by zero, the linear Gauss-Seidel step for the coarse problem is adjusted to yield a correction of zero if the diagonal element is zero.

  Let $\bar{d}$ be the resulting correction after one multigrid step in canonical coordinates and

  $$\tilde{d} = T^{\nu}B^{-1}\bar{d}$$

  the correction in transformed coordinates. Since $\tilde{d}$ may not be contained in the defect admissible set $\tilde{K}_{alg}^{\nu+\frac{1}{2}}$ we project it onto $\tilde{K}_{alg}^{\nu+\frac{1}{2}}$ in the Euclidean sense and obtain $\tilde{d}^P$.

- *Line search*

  The energy of the iterate after the coarse grid correction $\tilde{J}(\tilde{u}^{\nu+\frac{1}{2}} + \tilde{d}^P)$ may not be less or equal than the energy of the smoothed iterate $\tilde{J}(\tilde{u}^{\nu+\frac{1}{2}})$. To gain this *monotonocity*, a line search in the direction of the correction $\tilde{d}^P$ is done to find the minimal energy (see Figure 2.4):

  $$\tilde{u}^{\nu+1} = \tilde{u}^{\nu+\frac{1}{2}} + \alpha\tilde{d}^P$$

  How to find the optimal line search parameter α is explained [29].

In practice, the transformation (2.11) for the truncation is combined with the restriction operator R into a combined transition operator

$$\tilde{R} = R(B^{-1})^T T^{\nu}$$

and the linear multigrid step starts at the second-finest grid. Figure 2.5 gives an overview of the complete V-cycle of the TNNMG algorithm. After each TNNMG iteration the change in energy between the current and previous iterate is computed. The algorithm terminates when it drops below a given threshold.

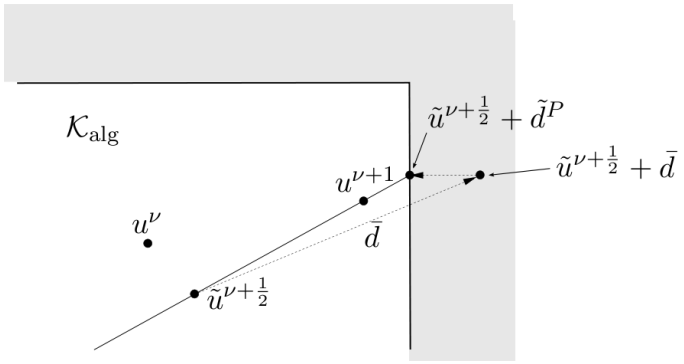

Figure 2.4: The coarse grid correction $\bar{d}$ of the smoothed iterate in transformed coordinates $\tilde{u}^{\nu+\frac{1}{2}}$ is projected onto the admissible set $\mathcal{K}_{alg}$ in $\tilde{u}^{\nu+\frac{1}{2}} + \tilde{d}^P$. A line search finds the minimum between $\tilde{u}^{\nu+\frac{1}{2}}$ and $\tilde{u}^{\nu+\frac{1}{2}} + \tilde{d}^P$, which yields the new iterate.

Theorem 2.4.3. *For any initial iterate $u^0 \in \mathcal{K}_{alg}$ the Truncated Nonsmooth Newton Multigrid algorithm converges to the unique minimum u of J in $\mathcal{K}_{alg}$.*

Proof. [29, Thm. 3.4.1].

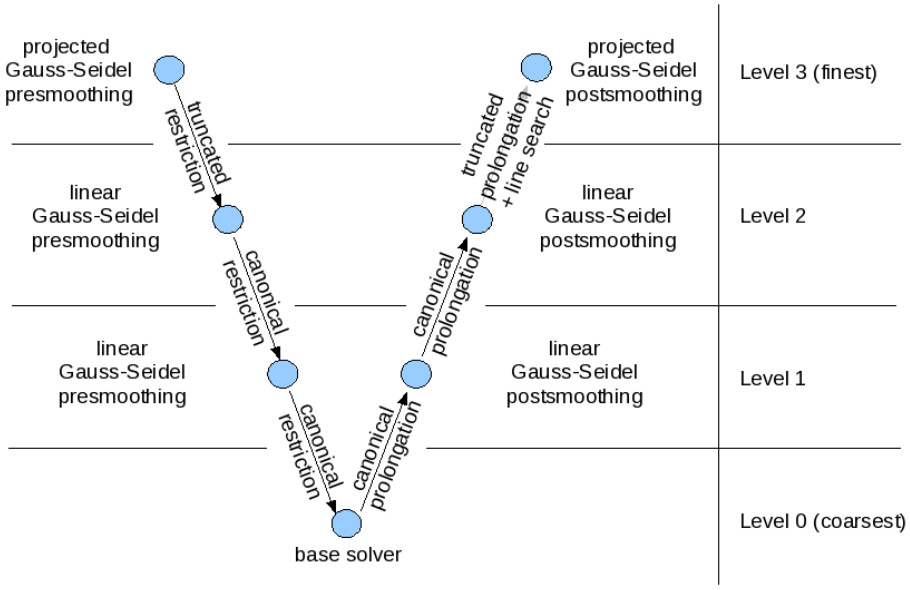

Figure 2.5: V-cycle of the TNNMG algorithm

Chapter 3

Concurrent Multigrid Solvers

Now that we have an algorithm for the solution of contact problems at hand, we can take care of finding ways of making the most of the Cell processor by parallelizing the solver. Since the time constraint of this master thesis does not allow to parallelize the entire algorithm, we had to pick a part of the algorithm to focus on. Here is a list of parts of the TNNMG algorithm that are eligible for parallelization:

- Linear / Projected Gauss-Seidel Smoothers
- Truncated Restriction / Prolongation
- Regular Restriction / Prolongation
- Base Solver
- Line Search
- Computation of the change in energy

We picked the Gauss-Seidel smoothers, since they are the foundation of the solver, are the only part of the solver that is non trivial to parallelize by feeding different rows of the matrix to different processors, and are of use in other algorithms as well. It needs to be expected, however, that the speed-up effect of parallelizing a single part of an algorithm has limited effect on the overall performance. This is where *Amdahls argument* [35] comes in. It states that when a part $p \in [0, 1]$ of an algorithm is parallelized, leaving the remainder $(1 - p)$ sequential, the maximum speed-up that can be expected on n processors is

$$\text{speed-up(n)} = \frac{1}{(1-p) + \frac{p}{n}}.$$

This means for example that if 50% of an algorithm are parallelized the maximum speed-up on 2^{16} processors is about 1.99.

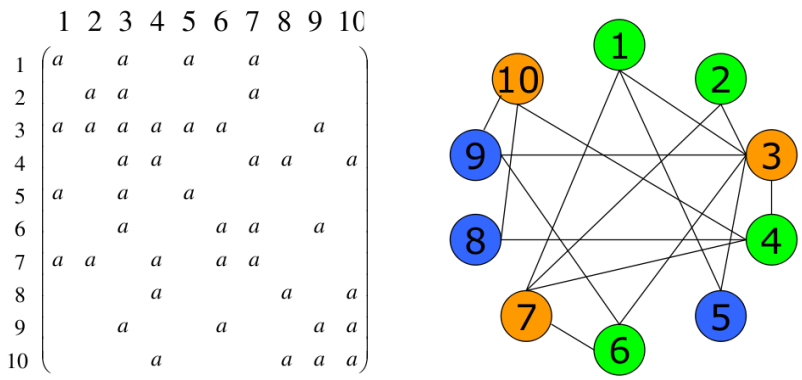

Figure 3.1: Deriving the dependency graph for a sparse matrix. The blanks in the matrix represent zero elements. Each non-zero element corresponds to one edge in the graph.

3.1 Parallelizing the Gauss-Seidel Algorithm

The Gauss-Seidel algorithm cannot be parallelized by putting multiple processors to work on different components x_i^ν of the current iterate x^ν at the same time, because component i depends on all $i-1$ preceding components. This is why the Gauss-Seidel algorithm is considered to be not parallelizable in general. There is, however, a way of parallelizing the algorithm for the special case of sparse matrices. If the matrix is sparse, the elements of the rows of the iteration matrix A that are zero do not need to be multiplied with the respective elements of the current iterate, since the product is always zero, thus eliminating some of the dependencies. This can go as far as to make sets of rows of A completely independent from one another. The key to parallelizing the Gauss-Seidel algorithm lies in disjointly partitioning A into functionally independent sets of rows that can be processed concurrently [15].

The functional dependencies of the algorithm for a matrix A are represented in the *functional dependency graph* $G(V, E)$, which is defined as:

$$V: \quad \text{rows of } A$$

$$E: \quad \{(i,j) : i,j \in V \,|\, i \neq j,\ a_{ij} \neq 0 \text{ or } a_{ji} \neq 0\}$$

Figures 3.1 shows an example of how such a graph can be derived from a concrete matrix. The problem of finding concurrently processable sets of rows of A is equivalent to finding a coloring of the nodes of G, such that none of the nodes of each pair of neighbors is assigned the same color. Rows of the matrix that correspond to nodes of the same color are functionally independent. With a coloring available, we get the following algorithm for updating the components x_i^ν:

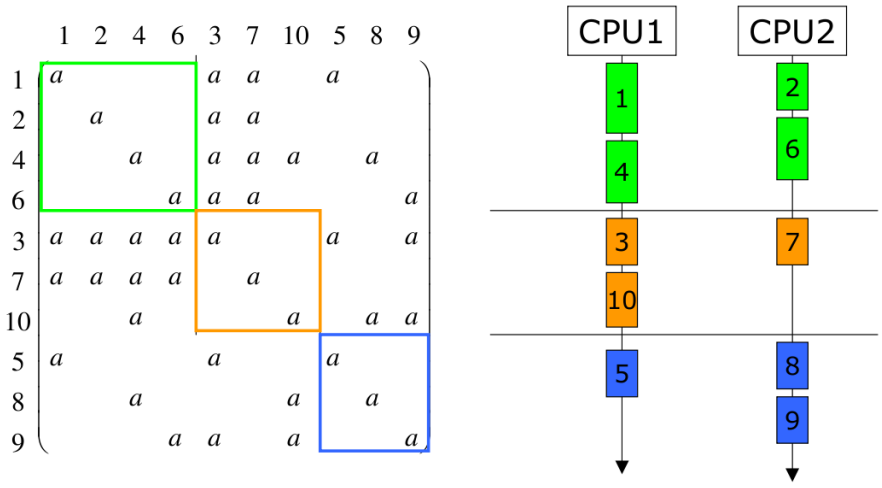

Figure 3.2: Left: Rearranging the rows and columns illustrates that rows of the same color are now independent from one another. Right: A possible way of spreading the workload over two processors.

Algorithm 3.1.1 (Concurrent Gauss-Seidel).

(1) For k=1 to #colors
(2) begin
(3) For each node n of color c_k
(4) begin
(5) Update component x_i^ν that corresponds to n;
(6) end
(7) end

The *For each* loop in the algorithm indicates an arbitrary update order that allows for parallelization. Note that the algorithm requires synchronization of all processing elements involved after all nodes of a color have been processed. This makes load balancing crucial, since the first processor to finish will idle until the last one catches up.

Figure 3.2 shows a rearrangement of the rows and columns of the matrix in such a way that rows of the same color are next to each other. The empty boxes around the diagonal of the matrix indicate that the respective rows do not depend on any of the rows in their immediate neighborhood and can be processed in parallel.

3.2 Heuristics for Graph Coloring

The perfect graph coloring algorithm returns a coloring that uses the fewest possible number of colors $\chi(G)$, which is called the *chromatic number* of G. The problem of finding such a coloring is proven to be NP-hard [11, Sec. 10.3]. There are, however, a number of heuristics that produce non optimal colorings in acceptable time. The choice of a suitable heuristic is governed by two factors: runtime versus the ratio of $\chi(G)$ to the number of colors produced by the heuristic. We have chosen a simple algorithm known as *sequential coloring* [15]:

Algorithm 3.2.1 (Sequential coloring).

(1) Choose an order v_1,v_2,...,v_n of the vertices of G;
(2) For i = 1 to n
(3) begin
(4) Assign v_i the smallest color yet unused in
neighbors(v_i) $\cap$ $\{v_1,v_2,...,v_{i-1}\}$;
(5) end

Let $deg(G)$ denote the *degree* of a graph $G(V, E)$, defined as the maximum number of neighbors of a node in V. Under the assumption made in [15, Sec. 6] that an upper bound for $deg(G)$ in the context of matrices arising from FE Methods is $\alpha \log |V|$, sequential coloring takes $O(n \log n)$ time. An upper bound for the number of colors produced by this algorithm is deg(G)+1, because the worst case for assigning a new color occurs when all other colors have been used up by the neighbors of a node.

Coloring the graph is a preprocessing step not necessary in the sequential version of the Gauss-Seidel algorithm. That is why we chose this algorithm for its favorable time complexity rather then an algorithm that might need fewer colors. Since the ratio of non-zero elements of the matrices we are dealing with increases with growing problem sizes, leading to fewer edges in the dependency graph, the number of additional colors needed grows very slowly with the problem size, as shown in the next section. Note that with the TNNMG algorithm the occupation patterns of the matrices on all levels of the hierarchy are stable across iterations, which allows to use the same coloring in all iterations.

The following listing shows the adaption of the heuristic for coloring the rows of a symmetric matrix $M^{n\times n}$.

Algorithm 3.2.2 (Coloring the rows of a symmetric matrix $M^{n\times n}$).

(1) for i=1 to n *//iterate through the rows*
(2) begin
(3) current_color=0;
(4) for j=1 to i *//iterate through the columns*

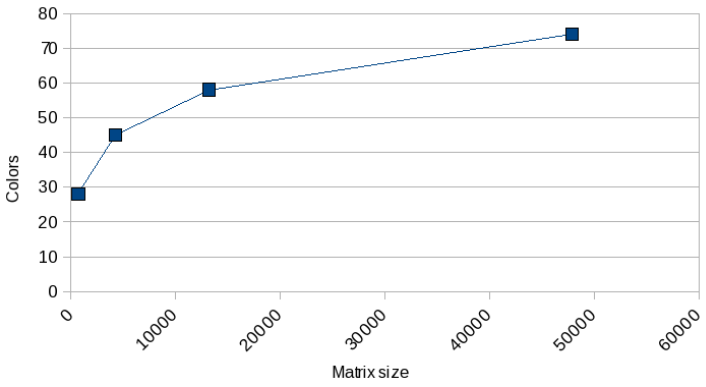

Figure 3.3: The number of colors needed by the coloring heuristics with growing problem sizes

```
(5)     begin
(6)       if (a_ij ≠ 0) and (row_color[j]==current_color)
(7)       begin
(8)         current_color=current_color+1;
(9)         j=1;
(10)      end
(11)    end
(12)    color[i]=current_color;
(13) end
```

Note that $a_{ij} = a_{ji}$ since the matrices we are dealing with are symmetric. Thus, it suffices to check for $a_{ij} \neq 0$ in line (6) to get all nodes that are connected to node i. The produced number of rows per color is unevenly distributed: The colors that are introduced for the first couple of rows generally comprise more rows than the colors introduced later, since the number of neighbors that are already colored is larger for higher row numbers. Thus, it is more likely that their neighbors have already used up all colors and a new color has to be introduced.

3.3 Results

The following table lists results we got from Algorithm 3.2.2. The matrices used resulted from several refinements of the discretization of the knee problem presented in Chapter 2:

Matrix size	Non zeros	Fill rate	deg(G)	Colors
684×684	10024	2.14 %	55	28
4226×4226	69778	0.39 %	121	45
13198×13198	249128	0.14 %	192	58
47893×47893	971009	0.04 %	194	74

The deg(G) columns lists the degree of the functional dependency graph, which is equal to the maximum number of non-zero elements of a row of the respective matrix. Note that the fill rate drops as the matrices grow larger. The number of additional colors needed by the heuristic behaves in the same way. Figure 3.3 shows the results graphically.

Chapter 4

The Cell Broadband Engine (CBE)

This chapter gives an introduction to the CBE platform. First the elements on the chip and the overall architecture will be explained. Then we will have a closer look at the facilities that allow communication between the elements, and conclude with a discussion of compiler support and issues to keep in mind for achieving optimal performance. Only concepts that are relevant for understanding this document will be discussed. An in-depth insight to the CBE architecture and API is provided in [20].

4.1 Architecture

Figure 4.1 provides an overview of the CBEs architecture. The Cell processor is a heterogeneous multicore processor: A single Cell chip features one classical PowerPC core (PPU), and 8 *Synergistic Processing Elements* (SPE). The PPU is a general purpose processor closely related to the PowerPC processors that were used in Mac computers. Therefore a large software base is available that can run on the PPU without changes. The PPU's design has, however, been simplified to make way on the chip for the SPEs. Branch predication logics and instruction reordering, for instance, have been cut back or even completely omitted. Those simplifications make the PPU very slow compared to modern general purpose processors and puts it on about the same level as an AMD Athlon XP at 1,3 GHz [33].

The poor computational performance of the PPU is compensated by the SPEs, which provide the bulk of the CBE's computational power. SPEs are special purpose processors designed to boost floating-point performance on vectors of four single-precision or two double-precision components. This specialization makes an SPE unsuitable for control intensive tasks like executing an operating system. The responsibilities of the two types of elements on the chip are clearly defined: The

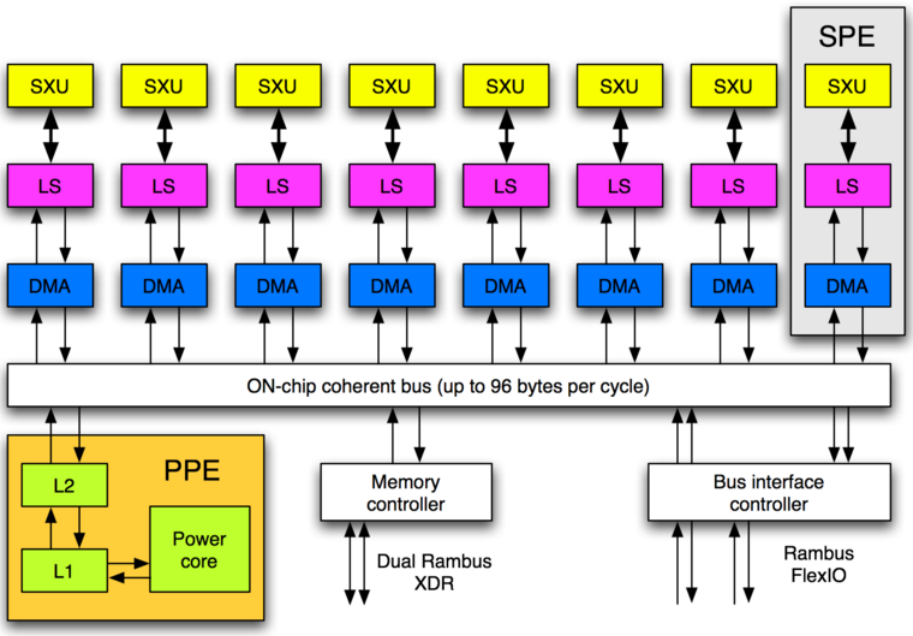

Figure 4.1: CBE architecture [36]

PPU provides a well established platform for the operating system and puts the SPEs to work should the need for high computational performance arise. Additionally, the PPU is responsible for relaying services of the operating system to the SPEs.

The three most important parts that make up an SPE are its *Synergistic Execution Unit* (SXU), the *Local Store* (LS), and the *Memory Flow Controller* (MFC). The SXU, which is responsible for executing the SPE's program, can only access its local store directly. To access the system's main memory, it needs to have the DMA controller, which is a part of the MFC, transfer parts of it to the local store and back. This makes each SXU completely autonomous and almost unaware of the rest of the world. It also eliminates the need for time consuming cache coherence checks when SXUs access their memory, as opposed to classical shared memory architectures [31, Sec. 6.2.1]. The MFC provides the only interface of the SXU to the outside world. To this end, its DMA controller can autonomously transfer chunks of data between the host systems main memory and an SPE's local store. Next to DMAs, the MFC provides *Mailboxes* and *Signal channels* for passing messages between the elements of the chip.

Note that only the PPU uses caches to speed up memory access, the SPEs, however, do not. The pathway between an SPE and its local store is very short. Thus it can be accessed at low cost, making caches superfluous.

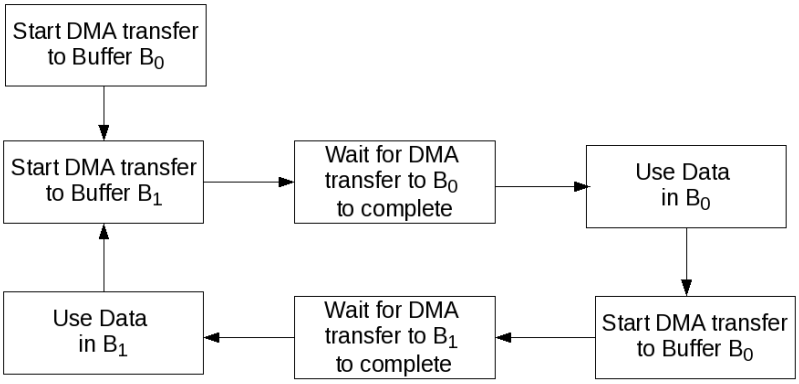

Figure 4.2: Double buffering [19]

4.2 Communication Facilities of the MFC

This section provides an overview of the facilities that the MFC provides to connect its SXU to the outside world.

4.2.1 Direct Memory Access Controller (DMA)

The DMA controller transfers chunks of data between the host systems main memory and an SPE's local store. An SXU can issue commands that the DMA controller executes independently. On a single command the DMA controller can transfer up to 16 KB of data (32 MB when DMA lists are used as explained in the next section). Since the SXU has no further part in transferring the data after issuing the command, it can process a chunk of data while the DMA controller brings in the next chunk. The SXU can continue seamlessly with that next chunk of data when it catches up. This concept is called double buffering and is essential for achieving high performance with the CBE architecture. It is illustrated in Figure 4.2. As long as the time needed to bring in the data does not exceed the time needed to process it, the SXU can work without interruptions, thus effectively voiding the cost of memory transfers, since they are completely concurrent. Considering this, it becomes clear why the limited size of the local store of 256 KB is not a major restriction: The SPE can process arbitrary amounts of data without wasting any time by handling the transfer of data that exceeds its local store.

4.2.2 DMA Lists

Next to transferring continuous chunks of data, the DMA controller supports so-called DMA lists. They are lists of commands placed in local store, which the DMA controller processes consecutively. This allows to transfer data that lies

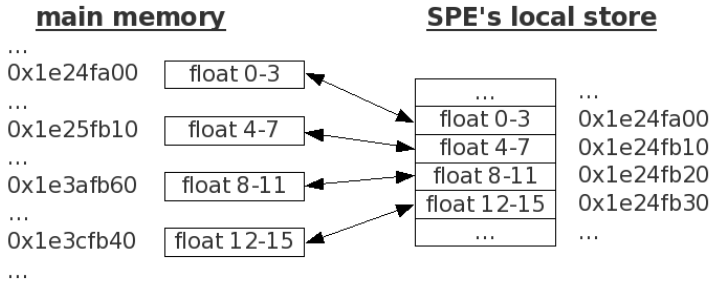

Figure 4.3: Scatter-gather example

scatter across main memory and place it continuously into an SPE's local store or the other way around. This concept is also referred to as *scatter-gather operation* by the CBE documentation. Figure 4.3 shows an example.

A DMA list can specify up to 2048 transfers. Each element of the list consists of a pointer into main memory and a transfer size. Thus, a DMA list command can transfer up to 32 MB of data, which is 128 times the size of the local store.

4.2.3 Limitations to DMAs

Transfer sizes and memory locations that can be handled by the DMA controller underlie some restrictions that can have major consequences for the overall architecture of applications:

1. Transfer sizes are restricted to 1, 2, 4, 8, and multiples of 16 bytes.
2. The starting address of the data to be transferred has to be 16 byte aligned (the four least significant bits of the address must be zero).
3. Transfers of size 1, 2, 4 and 8 bytes are also possible if the four least significant bits of the local store and main memory address are equal.

Figure 4.4 shows an array of 3-dimensional single-precision vectors. Suppose vector 1 starting at address 0x1e24fb18 needs to be transferred to an SPE's local store. The alignment constraint, however, prevents a DMA transfer from starting at such an address. A workaround is to start at the previous 16 byte aligned address (0x1e24fb10), thus bringing in the y and z-components of vector 0 as well. Obviously the SPE code needs to ignore the additional data. This adjustment results in a chunk of 20 bytes of data that needs to be brought in. The DMA controller, however, does not support transfers of 20 bytes. Again, the workaround is to transfer more data then needed by adjusting the transfer size to the next multiple of 16, which is 32, thus the entire vector 2 is transferred as well. In total, 32 bytes of data need to be transferred in order to bring in an unaligned 12 byte vector of data.

...	...
0x1e24fa0c	vec[0].x
0x1e24fb1**0**	vec[0].y
0x1e24fb14	vec[0].z
0x1e24fb18	vec[1].x
0x1e24fb1c	vec[1].y
0x1e24fb2**0**	vec[1].z
0x1e24fb24	vec[2].x
...	...

Figure 4.4: The alignment issue. The DMA controller can only access data that starts at one of the underlined addresses

The root of the problem lies within the overall design of the data structure. The CBE architecture is designed to work with vectors of four single-precision components. Using float vectors with three components goes against this principle. A better solution to the problems discussed above is to pad each vector with a fourth component and make sure the array starts at a 16 byte aligned address. This way all elements automatically start at a 16 byte aligned address and have a valid transfer size. This simplifies the code and thus improves the performance, but still leads to a waste of 25 percent of the memory bandwidth, which is, however, unavoidable.

4.2.4 Mailboxes

As opposed to the DMA controller that handles transfers of large amounts of data, mailboxes and signals only handle 32 bit messages for control and synchronization purposes. Both the SXU and PPU can access mailbox channels, whereas DMAs are usually initiated by the SXU. The MFC provides the following kinds of mailboxes:

Name	Queue depth	Direction
Inbound Mailbox	4	PPU to SPE
Outbound Mailbox	1	SPE to PPU
Interrupting Outbound Mailbox	1	SPE to PPU

The queue depth of a mailbox it the number of messages it can hold. Mailboxes cannot be used to exchange messages between SPEs. When an SPE reads an incoming mailbox that has no messages pending, it stalls until a message is available. The same happens when an SPE tries to write to a full mailbox.

Since the PPU is share by several processes, it cannot stall like an SPE. Thus, an attempt to write another message to a full mailbox causes the oldest message in it to be lost. Likewise, reading an empty outbound mailbox returns immediately even if the mailbox is empty, so the outbound mailbox needs to be polled.

Alternatively, the *Interrupting Outbound Mailbox* can be used. Writing a message to this mailbox causes an interrupt to be thrown on the PPU that can be routed by the operating system to notify the application. SPEs can throw interrupts in the following events:

1. The Interrupting Outbound Mailbox has been written
2. The Inbound Mailbox has been read and now has slots available
3. An SPE has terminated

Latency times of PPU interrupts are considerable, which renders the interrupt mechanism too slow for our purposes as will be further explained in Section 5.1.1.

4.2.5 Signals

Signals are similar to mailboxes but have some additional traits. Other then mailboxes, signals can be used to pass messages between SPEs. There are two signal channels on each SPE, which can be configured independently to work in either overwrite or *OR*-mode. A signal channel in overwrite mode behaves exactly like a mailbox. In *OR*-mode, the current content of the signal channel is bitwise *OR*ed with incoming messages. When the channel is read, the current value is reset to zero. Section 5.4 shows an application of signal channels in *OR*-mode.

4.3 The SXU

The SXU is the part of an SPE that is responsible for executing the program. An SPE is a special purpose processor designed to boost floating-point performance. This specialization makes programming it different from programming other processors in several aspects that will be explained in this section.

4.3.1 The SPE Complier

The PPU and SPEs of a Cell chip are two fundamentally different processors, which is why they need separate compilers. The complier that produces code that is to be executed by an SPE will be referred to as the *SPE Compiler*. It is always a cross-platform compiler, since the SPE cannot run a compiler itself due to its limited resources. The currently available version 3.0 of IBM's CBE SDK ships with a GNU GCC C++ compiler in version 4.1.1, including versions for both the PowerPC and I386 architecture. The GCC developers have currently released version 4.3 of the compiler suite, but since it is not supported by any of the distributions we use on the Playstion 3 (Fedora 7 and Yellow Dog Linux), we did not test it. The developers claim, to have addressed some of the issues explained below in this release, extended auto-vectorization capabilities in particular [1]. An

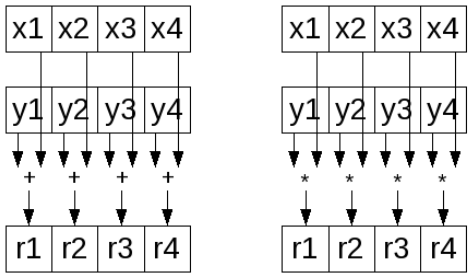

Figure 4.5: SIMD instructions act across an entire vector

alternative to GCC compilers is IBM own C++ compiler called *XL*. It is, however, not publicly available, which is why we could not test it.

The SPE compiler can handle a wide range of existing source codes, including API-calls like `printf` or `fopen`. The CBE architecture includes means to arbitrated them by the PPU. This makes a great deal of legacy code directly compilable for execution on an SPE without any changes. An exception is the *Standard Input / Output Streams Library* [5] introduced with C++. It includes facilities provided by the <iostream> header file such as `cout`, which are currently not implemented by the libraries shipping with the SDK. However, the CBE API offers, standardized ways of implementing custom handlers for C++ API calls [20].

Even though the compiler can handle a wide range of legacy source code, the resulting program will perform poorly if it is not adjusted to the SXU's special needs. We will point out some of those needs in the following sections.

4.3.2 SIMD Instruction Set

The SXU is designed entirely after the *Single Instruction Multiple Data* (SIMD) principle. SIMD is the concept of applying a single operation to multiple operands at the same time (see Figure 4.5) [19]. Consequently, the instruction set of the SXU consists entirely of SIMD instructions. The data paths and registers of an SPE are 128 bit wide, which allows for vectors of four single-precision or two double-precision components. All instructions act across the entire register. Since classical source code usually addresses one element of a vector at a time, the compiler needs to find ways of joining those *scalar* instructions to a SIMD instruction, which is called *vectorization* [20, Sec. 22.2]. This is not a simple task at all, and might not be possible in some cases. An alternative is to vectorize algorithms manually. This requires more work but can lead to more efficient solutions than the compiler could find on its own.

The SPE compiler provided with the CBE SDK has limited auto-vectorization capabilities. The SDK documentation recommends vectorization by hand [19, p. 108ff]. To this end, the SPE compiler has extensions to make SIMD instructions available through so-called C-language *intrinsics*. Intrinsics are commands in the

form of function calls that are substitutes for one or more inline assembly-language instructions. They include instructions like `spu_add` or `spu_mul` for vector addition and multiplication [18]. Most intrinsics expect operands of the type `vector float` or `vector double` that designate vectors of four single-precision or two double-precision components.

Operations that act on only a single component are sometimes unavoidable. Since they are not directly supported by the instruction set, they need to be emulated, which can cause considerable overhead. Emulating scalar instructions can entail the need to shift operands to matching position within registers and back. It can also be necessary to load an entire vector from memory, modify the element that has changed and write the modified vector back. This increases the impact of missed vectorizations by the compiler.

4.3.3 Floating-Point Unit Latency

The *Floating-Point Unit* (FPU) is the part of an SXU that is responsible for performing computations that involve floating-point numbers. The FPU needs 6 clock cycles to finish a single-precision floating-point operation [20, Sec. 24.2]. The registers that hold the operands are not accessible during this time. This means that when there are two consecutive floating-point instructions that use the same registers, the second instruction will stall until the first one finishes. This problem can be solved by having the instructions use different registers. The register file of an SXU provides 128 registers, which is potentially enough to make stalls completely avoidable. In practice, an instruction might depend on the result of the previous instruction, which makes stalls unavoidable in some situations.

The high number of latency cycles can particularly become a problem in loops. To prevent stalls in loops, we use a technique called *software pipelining* [20, Sec. 24.4]. The loop is *unwound*, which means that the instructions in it are written out several times for different sets of registers. Since single-precision floating-point instructions have a latency of 6 cycles, the loop might have to be written out up to 5 times to remove all stall. Thus, proper unwinding of loops is essential for optimizing performance.

The following example illustrates software pipelining:

Listing 4.1: Classical code example

```
1 vector float in_buffer_1[100], in_buffer_2[100],
2     in_buffer_3[100];
3 vector float result[100];
4
5 for (int i=0; i<100; i++)
6 {
7     vector float tmp = in_buffer_1[i] + in_buffer_2[i];
8     result[i] = tmp * in_buffer_3[i];
9 }
```

The code example adds the arrays of vectors `in_buffer_1` and `in_buffer_2` and multiplies the result with `in_buffer_3`. Note that the multiplication in line 8 depends on the result of line 7. The compiler has to insert five NOP (No Operation) instructions after the instruction in line 7 to give the FPU enough time to complete it before line 8 is executed.

Here is how this can be improved:

Listing 4.2: Unwound example

```
1 vector float in_buffer_1[100], in_buffer_2[100],
2    in_buffer_3[100];
3 vector float result[100];
4
5 for (int i=0; i<50; i++)
6 {
7    vector float tmp1 = Win_buffer_1[i*2+0] +
8       in_buffer_2[i*2+0];
9    vector float tmp2 = in_buffer_1[i*2+1] +
10      in_buffer_2[i*2+1];
11
12   result[i*2+0] = tmp1 * in_buffer_3[i*2+0];
13   result[i*2+1] = tmp2 * in_buffer_3[i*2+1];
14 }
```

In the second listing an iteration of the loop handles two elements at a time, cutting the number of iterations needed in half. Note that line 7 and 9 are now independent from each other, thus the computation of line 9 can start before line 7 is completed, significantly reducing the number of waiting cycles. The same applies to line 12 and 13.

To remove all stalls, the loop needs to be further unwound to handle five elements of the arrays at a time. With the fully unwound version of the code, the SXU can complete one floating-point instruction per cycle, leading to a considerable speed-up compared to the first version of the code. As with vectorization, the support for automatic software pipelinig of the GNU C++ 4.1.1 compiler is very limited, which makes it necessary to do this optimization by hand.

Note that the latency for double-precision floating-point instructions is 13 cycles [20, Sec. 24.2], which is the reason for the poor double-precision performance of the first implementation of the CBE.

4.4 PPU Compiler Issues

Like the SPE compiler, the PPU compiler shipped with IBM's CBE SDK 3.0 is a GCC compiler in version 4.1.1, which is again included in versions for PowerPC and as a cross-platform compiler for the I386 architecture. An alternative is IBM's

XL compiler, which is, however, not publicly available. In this section we will point out some issues of the GCC compiler.

4.4.1 Alignment

As explained in Section 4.2.3, the DMA controller can only access data that resides at 16 byte aligned addresses. The GCC compiler we used provides the *__attribute__((aligned(16)))* modifier to influence the alignment of variables and types of any kind. It does, however, not always work as expect. Arrays and objects that are dynamically instantiated using the `new` operator or `malloc` generally cause problems, because neither of them respects this modifier since the underlying `malloc` function implemented in *glibc* only guarantees 4 byte aligned pointers. Member variables of classes are only aligned as requested with respect to the beginning of the classes data. This entails the need to make sure that a dynamically instantiated class is placed at a 16 byte aligned location in memory for its member variables to be accessible by the DMA controller, a requirement that the standard `new` operator does not provide.

This issue has been discussed extensively in the GCC mailing list and currently has the status of WONTFIX, meaning that the cost of fixing it is considered to be too high [34]. The following workarounds can be used to address the problem:

1. Allocate 16 bytes more then needed and discard the first bytes up to the first correctly aligned address. The original pointer, however, needs to be stored as well, since `free` and `delete` will not accept the aligned version.

2. Use the `malloc_aligned` and `malloc_free` functions that are provided by the CBE SDK. The `malloc_aligned` function also requests more space than needed, but places a header in front of the payload data that contains the original pointer. The `malloc_free` function evaluates this header when freeing the memory. This is more convenient then storing the original pointer.

3. For classes, *placement new* can be used, a technique in which a special version of the `new` operator is used that does not allocate the memory itself, but is handed over a pointer to a buffer into which the newly created instance of the class is to be placed.

4. Also for classes, the `new` operator can be overloaded and replaced by a version that handles the allocation of memory on its own.

The same problem arises when it comes to *Standard Template Library* (STL) containers like `std::vector`. The STL uses the concept of *allocators* [4] to give programmers a way of influencing how memory is allocated. All classes that need to allocate memory dynamically provide an additional template parameter that is by default set to the STL's standard allocator. It is easily possible to build your own version of an allocator by using one of the existing implementations as a template.

One issue, however, remains: For the compiler the vector `std::vector<int>` is of a different type then `std::vector<int, aligned_allocator<int> >`, so they cannot be assigned to each other. This is particularly a problem for legacy code, which generally expects the first version. The only way to deal with this problem is to adjust the legacy code by means of an additional template parameter to work with arbitrary allocators.

Chapter 5

Implementation on the CBE

This chapter will describe the adaption of the graph coloring approach to parallelizing the Gauss-Seidel algorithm for the Cell processor. First we will have a look at the implementation from the global point of view of the PPU and discuss the overall architecture and data structures, followed by a detailed look at the SPE side implementation.

5.1 The PPU Side

5.1.1 The Overall Architecture

Figure 5.1 shows the overall architecture of the implementation. One of the SPEs is designated as the *supervisor* whose job it is to dispatch tasks and put the other SPEs to work. The reason for an SPE doing this job rather than the PPU is the PPU's insufficient response time. Suppose the SPEs notify the PPU about having finished processing a block of data by writing a message to their *Interrupting Outbound Mailbox*. However, before the interrupt notification reaches the PPU side code that reads the mailbox, the SPE has often already processed the next block, which leads to a stall, since the mailbox can only hold a single message.

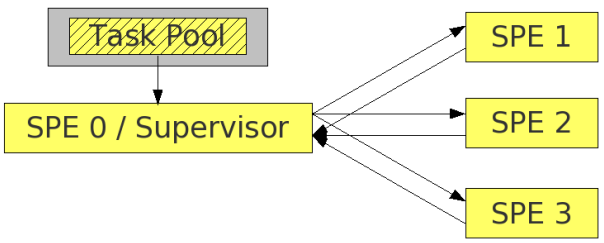

Figure 5.1: Overview of the architecture used in the implementation. The task pool is located in main memory.

Other than the PPU, an SPE can block when awaiting a message since it is not shared by multiple processes. Also, it can react on incoming messages much faster, since the notification of a new message does not need to be propagated through several layers of the operating system. The only use of the PPU in this architecture is to do the graph coloring, assemble the tasks, and fire up the SPEs. It does not have any part in the actual computations and just waits for all SPEs to return.

5.1.2 Vector and Matrix Types

As explained in Chapter 4, the DMA controller can only access data a 16 byte aligned addresses. Thus, we use 16 byte aligned vector types in the implementation. The compiler automatically extends the size of aligned types to the next multiple of 16 to make sure that all components of an array start at a properly aligned address. Since we are using vectors of three components that occupy 12 bytes in single-precision and 24 bytes in double-precision, those vectors are padded to occupy 16 and 32 bytes respectively. Matrix types are made up of an array of three padded vectors, resulting in 48 bytes for single-precision and 96 bytes for double-precision matrices. This waste of memory and overhead to DMA transfers is unavoidable since workarounds would affect the performance negatively.

5.1.3 Block Compressed Row Storage (BCRS)

Block Compressed Row Storage (BCRS) [28, Sec. 2.7.5] is a data structure for efficiently storing sparse matrices. It allows to only provide memory for the non-zero elements of a matrix. The drawback is, that additional meta data needs to be stored and time complexity for random access to a specific column increases compared to storing dense matrices. The data structure manages the following buffers to allow reconstruction of the original matrix:

- The *elements* buffer holds the non-zero elements of the matrix densely in the order in which they appear in the columns.
- The *column numbers* buffer is an array of integers that holds the column number of the respective element in the *elements* buffer. The column numbers for each row are stored in ascending order.
- The *row windows* buffer is an array of a structure that describes the window of the *elements* and *columns* buffers that is occupied by a row. The structure consists of an *offset* element and a *count* element that describe the offset of the row's first element within the *elements* and *columns* buffer and the number of the row's non-zero elements.

Figure 5.2 shows an example. Access to rows can be done in constant time using the *row windows* buffer. For random access to a column, a binary search

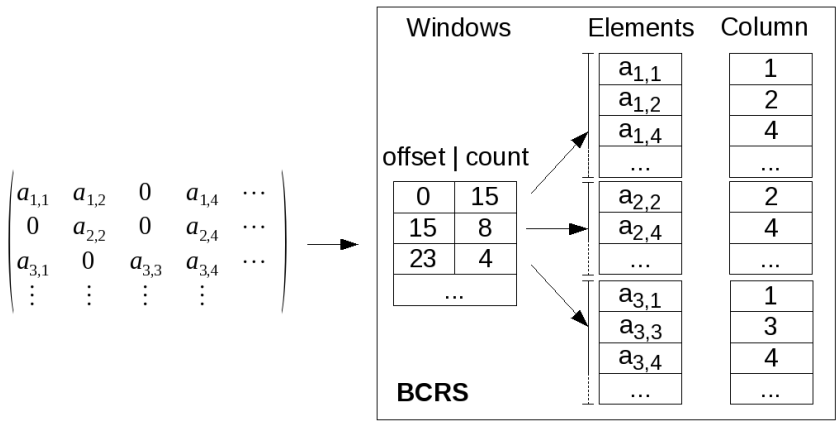

Figure 5.2: Block Compressed Row Store (BCRS) example

in the *column numbers* buffer needs to be done, which leads to a time complexity of $O(log\, c)$, where c is the number of non-zero elements of the row. The time complexity of iterating through the non-zero elements of the row is $O(c)$.

5.1.4 Task Creation

This section gives an overview of how the workload is decomposed into tasks that can be dispatched to the SPEs. The data necessary for computing the next iterate of a single row is generally not large enough to fill the available buffer space of an SPE, so a task can consist of multiple rows. These rows are, as a consequence of the graph coloring approach, not necessarily consecutive within the matrix. A row should be processed completely by a single task to keep communication to a minimum. All rows of a task must be processable independently from each other (all rows must be of the same color) to keep the computational kernel simple. Thus a task also has a unique color, which is the color of the rows that it contains. Furthermore, the data of a task should fill the buffer space in the SPEs local store as much as possible to keep the overhead of dispatching tasks to a minimum. The implementation manages three buffers in main memory to describe the tasks:

- The *row windows* buffer, which is a permutation of the *row windows* buffer of the BCRS in such a way that rows of the same color are consecutive.
- The *diagonal index* buffer. The diagonal indices need to be determined during the graph coloring and can be reused in the computational kernel of the worker SPEs.
- The *row numbers* buffer that contains the original row numbers within the BCRS.

Part of the data in the buffers is redundantly stored in the BCRS. They are necessary, however, since the rows are processed in a different order than they are stored. A task has to be dispatchable by a single 32 bit message that fits into a mailbox or signal channel. This allows to dispatch as many tasks as possible ahead of time but makes the use of pointers impossible. We use offsets to the base address of the buffers instead and pass the base addresses to the SPEs on initialization. The first 24 bit of the message that dispatches a task serve as offset to the first row of the task and the remaining 8 bit hold the number of rows.

When there are only a limited number of rows of the same color available it needs to be avoided that all of those rows are placed into a single task, since only one worker would process this task. Thus, when the tasks are being assembled, the number of worker SPEs has to be known. When there are n workers available, the implementation fills n tasks at the same time. When adding a new row it is added to the task that has the most buffer space left. This helps to keep the load of the tasks balanced.

Note that the architecture can do without a dependency graph for the tasks as it is used in frameworks that simplify software development for the CBE like [16], which is why we did not use them. The colors of the tasks suffice to represent the dependencies and offer a much more lightweight way of synchronizing dependent tasks.

5.2 The Worker SPEs

The worker SPE implementation consists of a part that sets up DMA transfers to bring the data into local store and the computational kernel that computes the new iterate. In this section we will first cover the computational kernel to point out design decisions and then turn to explaining how data transfers are handled.

5.2.1 The Computational Kernel

The following listing shows the computational kernel of the SPE side code. It computes the new iterate for each row of a task t. Thus, it is basically an adaption of the projected block Gauss-Seidel algorithm. Note that the operands are 3×3-matrices and 3-vectors as explained in Chapter 2.

Algorithm 5.2.1 (Compute the new iterates of task t).

(1) for each row r in t
(2) begin
(3) residual = rhs[r];
(4) for i=0 to row_window[r].count do
(5) residual -= row[r][i] * x^{ν}[r][i];
(6) diagonal = row[r][diagonal[r]];
(7) find a correction vector d that minimizes $J(x) = \frac{1}{2}x^T Ax - bx$

with A=diagonal and b=residual, such that $d \in$ obstacle[r];
(8) $x^{\nu+1}[r] = x^{\nu}[r]$[diagonal[$r$]] + d;
(9) write $x^{\nu+1}[r]$ back to main memory;
(10) end

The following table explains the meaning of the buffers. We call them the *payload buffers* since they hold the data that is required by the computational kernel:

Payload Buffers		
Buffer	Elements	Description
rhs	vectors	The elements of the right hand side vector corresponding to each row
row_window	structures (see 5.1.3)	The window within the BCRS buffers that is occupied by each row
row	matrices	The non-zero elements of each row
obstacle	boxes (see 2.4.1)	The *obstacle* boxes describing the set of admissible solutions for each row
x^{ν} buffer	vectors	The elements of the current iterate corresponding to the elements in the *row* buffer
$x^{\nu+1}$ buffer	vectors	The next iterate for each row
diagonal	integers	An index to the diagonal element of each row within the *row* buffer

The algorithm computes the residual, finds the correction vector d by solving the minimization problem in line (7), and computes the new iterate. The implementation of the linear and projected Gauss-Seidel smoother only differ in the way that the correction d is computed in line (7). For the projected Gauss-Seidel smoother, we use an iterative projected Gauss-Seidel solver that performs 20 iterations. In the linear Gauss-Seidel smoother, d is computed by solving *diagonal*d=residual* (see Section 2.3.1), which can be done in a single step. This accounts for the difference in runtime for the two variants of the algorithm shown in Table 6.2.

Note that not all rows have an obstacle, since the non-penetration condition only affects the nodes on the boundary patch $\Gamma_{i,C}$. Thus, line (7) can be solved directly in the projected Gauss-Seidel smoother for rows that do not have an obstacle. To this end the implementation uses one bit of each element of the *diagonal* buffer to indicate weather a row has an obstacle or not.

Let x^{ν}_{ppu} denote the buffer holding the current iterate in main memory. Note that the vector x^{ν}_{ppu} is dense as opposed to the rows of the matrix. Thus, one could expect the x^{ν} buffer to be accessed in the computation of the residual in line (5) like this:

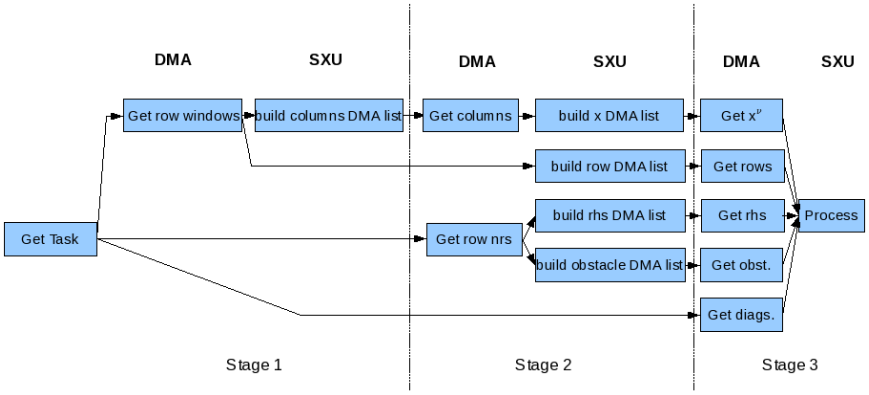

Figure 5.3: Dependencies in the steps necessary to bring in all data for a task. The steps in the first row represent the time critical path.

(5) residual -= row$[r][i]$ * x^{ν}[column_number$[r][i]$];

This is not the case in our implementation since we put only those elements of the current iterate into the x^{ν} buffer that correspond to non-zero elements of the rows of the task. Thus, the *column numbers* buffer of the BCRS has to be resolved during the transfer of the data as explained in the next section. This can lead to redundant data in the x^{ν} buffer when multiple rows of a task have a non-zero element in the same column. The complexity added by removing duplicates would, however, outweigh the benefits. The total number of elements transferred from the x^{ν}_{ppu} buffer in a complete Gauss-Seidel iteration equals the number of non-zero elements of the matrix.

The alternative would be to bring in the entire current iterate x^{ν}_{ppu} from main memory. Since it is generally too large for the local store, it would have to be broken down into pieces that need to be transferred into local store again and again for each task causing high load to the system bus. Furthermore, most of the elements of x^{ν}_{ppu} are unneeded, since the rate of non-zero elements of each row is very low and decreasing with growing problem sizes.

Note that the current iterate for each row is always contained in the x^{ν} buffer, since it is the element that corresponds to the diagonal element of the row. Thus, it can be extracted from the x^{ν} buffer as is done in line (8).

5.2.2 Data Transfer

Figure 5.3 shows the dependencies in the steps necessary for bringing in all data needed for a single task. The steps that resolve the *column numbers* buffer of the BCRS and fill the x^{ν} buffer represent the time critical path. They partition the process into three stages, each of which consists of a DMA operation and a

corresponding SXU operation. Stage one brings in the windows of the rows. Stage two brings in the column numbers of the non-zero elements and processes them to pointers into x^{ν}_{ppu}. Stage three brings in the required elements of x^{ν}_{ppu} along with the other payload data.

There are a number of possibilities for placing the steps that are not on the critical path. We chose not to bring in any data sooner than necessary to avoid unnecessary load on the system bus as well as tieing up buffer space prematurely.

Since the data for the rows of the tasks is scattered across main memory, the implementation uses several DMA lists that require buffer space in local store. Table 5.1 shows the auxiliary buffers for the transfer and how they are accessed during each step. Note that the row numbers, which are necessary for bringing in the respective elements of the right hand side vector and obstacle, correspond to the column numbers of the diagonal elements and could be extracted from the *column numbers* buffer. However, tests have shown that having the DMA controller transfer this redundant data from main memory is faster than having the SXU extract it from the *column numbers* buffer.

Operation	reads	writes
Stream in the row windows	[main memory]	row windows
Set up the column DMA list	row windows	columns DMA list
Stream in columns and row numbers	columns DMA list [main memory]	columns buffer row numbers
Set up payload DMA lists	columns buffer row windows row numbers row numbers	x DMA list row DMA list rhs DMA list obstacles DMA list
Stream in the payload data	x DMA list row DMA list rhs DMA list obstacle DMA list [main memory]	x^{ν} buffer row buffer rhs buffer obstacles buffer diagonal indices
Compute the new iterates	payload buffers	[main memory]

Table 5.1: Access to the buffers in each stage

5.2.3 Pipelining Data Transfers

In the previous section we dealt with the data transfer as a sequence of steps. However, since DMA and SXU operations take turns and those two parts of the chip can work concurrently, the process can be parallelized. In this section we describe how pipelining can be used to achieve this. The idea is to have the DMA

controller and SXU do each stage in parallel on data of two different tasks. To allow them to work on the same buffer concurrently, all buffers need to be present twice. The two versions of the same buffer will be denoted as the *even* and *odd* buffer. The following table shows for the critical path how DMA and SXU operations can be done in parallel in each stage:

DMA			**SXU**		
Operation	Buffer	Task	Operation	Buffer	Task
Get windows	even	$i+3$	build columns DMA list	odd	$i+2$
Get columns	odd	$i+2$	build payload DMA lists	even	$i+1$
Get payload	even	$i+1$	Process payload data	odd	i

Table 5.2: Pipelining of DMA transfers for the data on the critical path. The data of task i is fully available in local store in the current iteration of the SPE-side main loop.

In each iteration of the SPE-side main loop, the data of the tasks is propagated further through the pipeline and one block of payload data is completed and placed into the *even* buffer. Meanwhile, the computational kernel computes the new iterates for the payload data in the *odd* buffer that has been brought in during the previous iteration. After each iteration the buffers are swapped.

At any given time the buffers are filled with the data of four different tasks. The SXU operation in each stage operates on the data that was brought in by the DMA operation in the previous iteration of the main loop. Note that the amount of data handled by the DMAs equals the amount of data handled by the parallel SXU operation to balance their load as much as possible.

Two consecutive steps on the same set of data in Table 5.1 are actually an entire iteration of the main loop apart. Note that most of the buffers that are brought in during an iteration are used in the next iteration. Data that is needed again in later iterations will be overwritten by the intermediate transfers. Since the *row windows* and *row numbers* buffers are used in the computational kernel, these buffers actually have to be present four times to prevent premature overwriting.

5.3 Optimizations

5.3.1 Vectorized Computation of the Residual

Computing the residuals is the most time consuming part of the Gauss-Seidel algorithm. Therefore, optimizing it yields the most significant effects. The com-

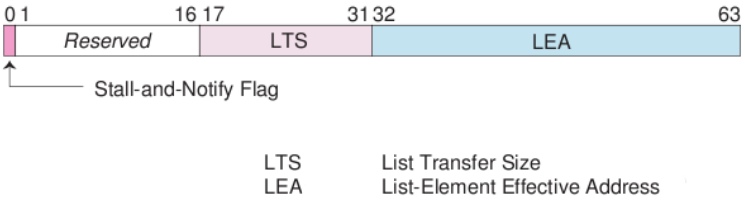

Figure 5.4: The format of a DMA list element [20]

putation consists of a number of multiplications of 3×3-matrices and 3-vectors and the summation of the result vectors.

Let the rows of matrix M be represented by three floating-point vectors M_1 through M_3. The classical approach to computing the matrix-vector product $y = Mv$ is to determine the dot product of each row of M_i and v and merge the resulting scalars into the result vector. Let y_i be the components of the floating-point vector y and *dot* denote the dot product instruction. The multiplication can be written as:

$$\begin{aligned} y_1 &= \mathrm{dot}(M_1, v) \\ y_2 &= \mathrm{dot}(M_2, v) \\ y_3 &= \mathrm{dot}(M_3, v) \end{aligned}$$

However, the SIMD instruction set does not provide a dot product instruction, since the result is a single scalar and not a vector. This means that the dot product needs to be computed using a number of other instructions, which is time consuming.

The key to an approach that is better suited for vectorization is the realization that all elements of a column of M are multiplied with the same element v_i [23]. So we splat each element v_i of v over all three components of an auxiliary vector $s^{(i)}$ such that $s^{(i)} = \{v_i, v_i, v_i\}$. The auxiliary vectors are then multiplied with the columns of M. In order to get all elements of a column of M into the same floating-point vector, M needs to be transposed:

$$y = \mathrm{mul}(M_1^T, s^{(1)}) + \mathrm{mul}(M_2^T, s^{(2)}) + \mathrm{mul}(M_3^T, s^{(3)})$$

The *mul* instruction multiplies two vectors element-wise and returns the result in a new vector. Note that all arithmetic instructions affect an entire floating-point vector as opposed to the first approach.

In Table 5.3 the results of the final and fully optimized SPE code and a version without the manually vectorized computation of the residual are compared.

Matrix size	Optimized	Unoptimized	Speed-up
linear Gauss-Seidel			
684 × 684	2.82	16.01	5.68
4226 × 4226	13.84	106.86	7.72
13198 × 13198	42.3	374.85	8.86
47893 × 47893	151.34	1450.69	9.59
projected Gauss-Seidel			
684 × 684	3.53	16.72	4.73
4226 × 4226	17.05	110.08	6.46
13198 × 13198	54.76	387.06	7.07
47893 × 47893	199.72	1499.34	7.51

Table 5.3: Runtimes in milliseconds for three Gauss-Seidel iterations with a single worker SPEs.

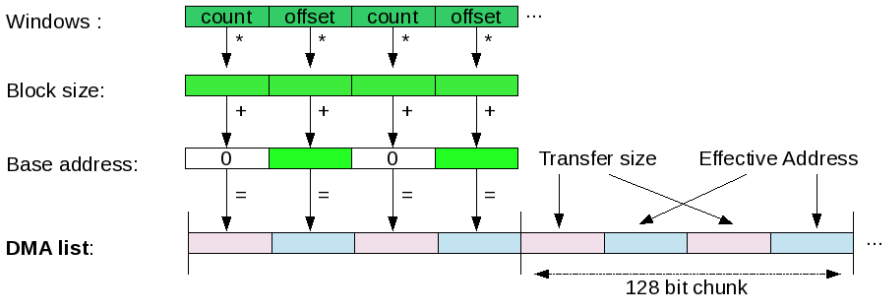

Figure 5.5: Vectorized filling of the *rows* and *columns* DMA lists

5.3.2 Vectorized Filling of DMA Lists

The implementation uses five DMA lists (listed in Table 5.1) that need to be filled. This section shows ways of vectorizing this process. Figure 5.4 shows the format of a DMA list element. To simplify matters when filling the list, the transfer size word can be accessed as a single 32 bit chunk, ignoring the lower 16 bit that comprise the *Stall-and-Notify* flag and a number of reserved bits. They will be implicitly set to zero as long as the transfer size does not exceed 64 KB.

The Rows and Columns DMA Lists

The *rows* and *columns* DMA lists both transfer data from the buffers of the BCRS, which is why they are filled similarly. The input in both cases is the *row windows* buffer. The format of the elements of this buffer has been chosen deliberately to resemble a DMA list element: The *offset* field corresponds to the *effective address* field and the *count* field corresponds to the *transfer size* field.

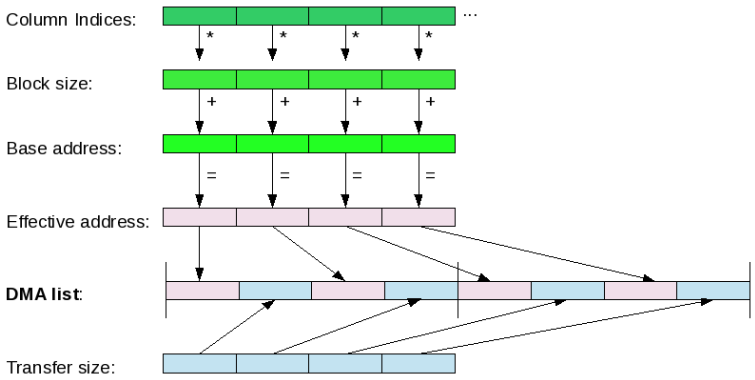

Figure 5.6: Vectorized filling of the *rhs* and *obstacle* DMA lists

Figure 5.5 illustrates how the row windows are transformed into DMA list elements. Both the *offset* and *count* fields are multiplied with the block size, which is the size of an element of the respective buffer. Then the base address of the array in main memory is added to the offset field to create the actual pointer. To this end, two auxiliary floating-point vectors are created. The first one has all its components set to the block size and the second one has the elements that correspond to the *offset* field set to the base address and the others to zero. By multiplying the row windows with the block size vector and adding the base address vector two elements of the list can be processed at a time.

The columns DMA list needs additional treatment since the transfer from the columns buffer is unaligned: Each column index occupies only 4 bytes, thus the starting addresses of the column numbers of a row may not be 16 byte aligned and the transfer size may not be a multiple of 16 bytes as is required by the DMA controller. This issue is solved by transferring additional data before and after the payload data as discussed in Section 4.2.3. To move the pointer of a DMA list element to the previous 16 byte aligned address, the four least significant bit are set to zero. To compensate for this, the transfer size has to be increased by the value of the four bit that were truncated from the pointer. Then the transfer size is increased to the next multiple of 16 by applying

```
transfer size = (transfer size + 0xF) & 0xFFFFFFF0
```

where `&` denotes the bitwise *and* operator.

The RHS and Obstacle DMA lists

The pointers for the *rhs* and *obstacle* DMA lists are computed based on the *row numbers* buffer. The transfer size is constant for each element. The row numbers are processed into pointers by multiplying them with the block size and adding the base address of the respective buffer in main memory. The effective addresses

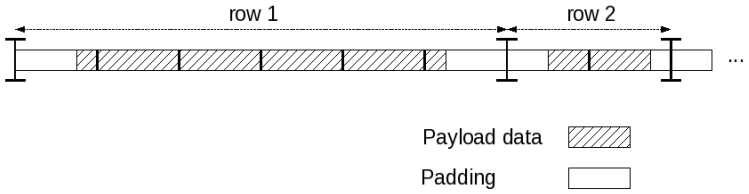

Figure 5.7: Possible arrangement of the *column numbers* buffer

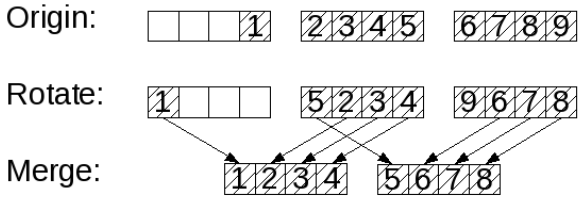

Figure 5.8: Removing excess indices

have to be intertwined with the transfer sizes to form the DMA list elements as shown in Figure 5.6. This is done using a *splat* instruction, which takes two vectors and merges those elements selected by a third parameter into the result vector. A slightly simplified version of the splat instructions used for merging the effective addresses and transfer sizes looks like this:

```
vector int splat_word_0 = { 0x0000, 0x0100, 0x0001, 0x0101 };
vector int splat_word_1 = { 0x0002, 0x0102, 0x0003, 0x0103 };

vec_dma_list[i]   = spe_splat(sizes, addresses, splat_word_0);
vec_dma_list[i+1] = spe_splat(sizes, addresses, splat_word_1);
```

A value of 0x0b0a in the splat word says to take component a of input vector b and write it into the result vector. The value of a can be between 0 and 3 and b can be 0 or 1. This allows to process four DMA list elements at a time.

The x DMA list

The *x DMA list* is the largest one in the set, since it contains one element per block instead of one per row. As explained above, the column numbers are unaligned in main memory, since a single column index occupies only 4 bytes. Figure 5.7 shows an example of how the column numbers could be arranged in local store.

In this section we present a way of filtering out the excess elements when building the *x DMA list*. Each element of the input floating-point vector is rotated to the left by the number of excess elements in front of it. Then each pair of

consecutive vectors is merged into a single vector that contains only the payload bytes in the correct order. This vector can then be used as input for the procedure described in the previous section. Figure 5.8 illustrates this approach.

In the following table, the results of the final and fully optimized SPE code and a version without the manually vectorized filling of the DMA lists are compared:

Matrix size	Optimized	Unoptimized	Speed-up
linear Gauss-Seidel			
684×684	2.82	3.28	1.16
4226×4226	13.84	18.49	1.34
13198×13198	42.3	57.68	1.36
47893×47893	151.34	213.38	1.41
projected Gauss-Seidel			
684×684	3.53	3.99	1.13
4226×4226	17.05	21.15	1.24
13198×13198	54.76	70.56	1.29
47893×47893	199.72	263.08	1.32

Table 5.4: Runtimes in milliseconds for three Gauss-Seidel iterations with a single worker SPEs.

5.4 The Supervisor SPE

The supervisor SPE acts as a central hub for the communication between the workers. Its main function is to wait for the workers to signal that they have finished tasks and dispatch new tasks to them. Additionally, when all tasks of the current color have been processed, the supervisor signals to all workers that tasks of the next color are cleared for processing. This section explains how the communication between the supervisor and workers is done.

5.4.1 Dispatching Tasks to the Workers

Communication between the supervisor and worker SPEs is done in both directions by means of signal channels (see Section 4.2.5). To dispatch a task, the supervisor writes a message that describes the window of the tasks data to a signal channel of the worker in overwrite mode. One task is dispatched to each worker in advance and held by its signal channel until the message is read. Before a worker signals that it has finished a task, it reads the next task from its signal channel. This

ensures that the supervisor has processed the previous message and confirmed it by dispatching the next task. Otherwise consecutive messages of the same worker could overwrite each other.

To signal the finishing of a task, the workers use a signal channel of the supervisor in *OR*-mode. Each worker uses a different bit of the channel. Since the signal channel works in *OR*-mode, the messages of several workers can accumulate without interfering with each other.

5.4.2 Synchronization of Task Colors

The second function of the supervisor is to clear the tasks of the next color for processing when all tasks of the current color are done. When a worker notices that the color of a new task is different from the current color, it delays the transfer of the current iterate for this task and stalls until the color is cleared. As soon as the task is cleared, the transfer of the current iterate is resumed.

On the other side, the supervisor counts the number of tasks of the current color that have been processed. When all tasks are done it clears the next color by writing a message that contains the number of the color into the signal channels of the workers.

Chapter 6

Results

In this chapter the performance gains that we achieved with our implementation on the CBE platform compared to two 'classical' processors are presented. First the performance of the Gauss-Seidel algorithm alone will be examined, since it is the part of the TNNMG algorithm that has undergone the most optimizations. Then we will see how the implementation scales when the number of worker SPEs increases and conclude with measurements of the performance of the complete TNNMG solver.

6.1 The Test Environment

We used the same example geometry as in [29, Sec. 3.8]. The left distal femur and proximal tibia from the Visible Human data set [6] is used. The data was segmented and a high-resolution boundary surface was extracted. The femur surface consisted of 7236 vertices and 14468 triangles, and the tibia surface of 7453 vertices and 14902 triangles. They were simplified as described in [29, Sec. 3.6] to yield coarse surfaces with 268 vertices and 532 triangles for the femur and 224 vertices and 444 triangles for the tibia. The Amira [32] grid generator produced two tetrahedral grids with 378 and 306 vertices, and 1328 and 1044 elements, respectively (see Figure 6.1, left).

The bone was modeled with an isotropic, homogeneous, linear elastic material with $E = 17$ GPa and $\nu = 0.3$. The bottom section of the proximal tibia was clamped and a downward displacement of 6 mm was prescribed on the upper section of the femur (see Figure 6.1, right).

We used the hierarchy of matrices shown in Table 6.1 in our tests. It was obtained using the implementation of the two body contact problem described in [29]. It makes heavy use of the *Distributed and Unified Numerics Environment* (Dune) [2]. The matrices were obtained by adaptively refining the 20 percent of the triangles for which the a posteriory error estimator described in [29, Sec. 3.7] yielded the largest error.

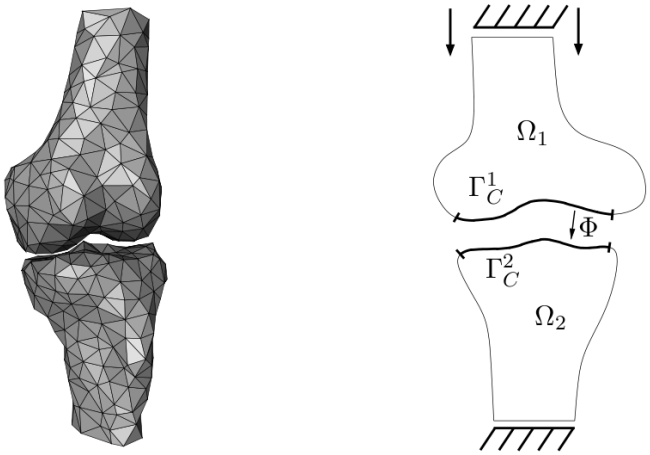

Figure 6.1: Two-body contact problem

Matrix size	Non zeros	Fill rate	deg(G)	Colors	Obstacles
684×684	10024	2.14 %	55	28	17
4226×4226	69778	0.39 %	121	45	77
13198×13198	249128	0.14 %	192	58	293
47893×47893	971009	0.04 %	194	74	1167

Table 6.1: The hierarchy of matrices that we used in our tests. The matrix sizes refer to blocks of 3×3 matrices. The deg(G) columns lists the degree of the corresponding functional dependency graph, which is equal to the maximum number of non-zero elements of a row of the respective matrix.

We used a Sony Playstation 3 (PS3) for our tests on the Cell processor. This entails two restrictions: Only six of the eight SPEs available on a Cell processor can be accessed on the PS3. Since in our architecture one SPE acts exclusively as supervisor, only 5 SPEs remain to do the actual work. The second restriction concerns memory: A PS3 only features 256 MB of RAM that cannot be extended. This allows only relatively small problems to be computed on the PS3. For this reason the hierarchy of grids we used for testing the complete TNNMG algorithm comprises three levels only.

For comparing the results to a 'classical' scalar and sequential architecture, we ran the test suite on two machines with Intel processors. The first one is a desktop computer that uses an Intel Pentium 4 processor (P4) at 3.00 GHz and has a system memory that runs at 400 MHz. The second machine is a laptop computer that uses a weaker Intel T2250 processor at 1.73 GHz but has a faster system memory that runs at 533 MHz. As table 6.2 shows, the Gauss-Seidel algorithm is only slightly faster on the Pentium 4 than on the T2250. This goes to show that

the performance of the algorithm depends not only on the processor but just as much on the performance of the system memory.

The source code was compiled with the GNU C++ Compiler in version 4.1.1 for all three platforms (PPU, SPE and Intel x86). The IBM Cell SDK 3.0 was used. See Chapter 4 for more information about the compilers and SDK. All times have been taken using the *gettimeofday()* function.

Note that the Cell processor used in the PS3 is optimized for single-precision floating-point computations and therefore performs poorly when it comes to double-precision computations. IBM has released the next iteration of the CBE platform in which the problem was addressed in May 2008 [3]. We did, however, not have it available for testing. This is why double-precision performance is only briefly discussed at the end of this chapter. All other results listed refer to computations in single-precision.

6.2 Single-Precision

6.2.1 The Gauss-Seidel Algorithm

In this section we present the results for the Gauss-Seidel algorithm alone. This algorithm has undergone the most optimizations for the Cell platform and thus exhibits the most performance gains. The following table shows the runtimes in milliseconds of three iterations for both the linear as well as the projected form of the Gauss-Seidel algorithm. The runtimes on the Cell processor were measured using all five available worker SPEs:

Problem Size	IBM Cell 3.20 Ghz	Intel T2250 1.73 Ghz	Intel P4: 3.00 Ghz	Speed-up (P4)
linear Gauss-Seidel				
684×684	2.39	1.29	1.42	0.59
4226×4226	5.81	9.01	9.38	1.62
13198×13198	13.53	32.64	31.45	2.32
47893×47893	45.90	136.00	120.82	2.63
projected Gauss-Seidel				
684×684	2.55	2.64	2.49	0.97
4226×4226	7.19	14.48	15.8	2.20
13198×13198	15.91	54.22	53.72	3.38
47893×47893	52.12	196.25	205.69	3.95

Table 6.2: Runtimes for three iteration of the Gauss-Seidel algorithm in milliseconds. The speed-up column shows to the reduction of the runtime relative to the P4 processor.

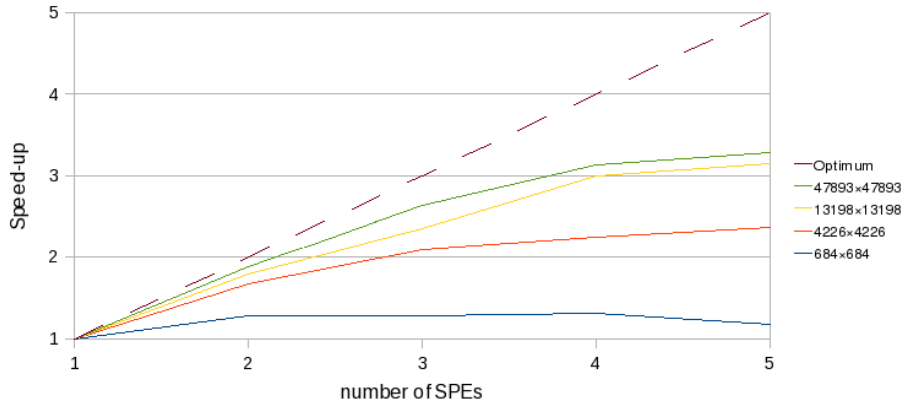

Figure 6.2: Speed-ups for the linear Gauss-Seidel algorithm on up to five SPEs with varying problem sizes.

The results exhibit two trends: Firstly, the Cell processor performs better the bigger the problem gets. This can be explained by the fact that the number of non-zero elements of the rows increases and thus the tasks comprise a smaller number of rows. This allows the SXU to spend more time in the inner loop of the highly optimized computational kernel rather than the control intensive part that switches between rows. Also, as shown in Chapter 3.2, the number of colors needed by the graph coloring heuristic decreases relative to the size of the matrix with bigger problems, which causes a decline of the overhead for synchronization.

Secondly, note that the Cell processor needs only slightly more time for the projected Gauss-Seidel algorithm than for the linear version, whereas the differences are larger on the Intel processors. This indicates that the Cell processor has more floating-point performance reserves, since the increased computational complexity introduced by the projection has little affect on it. The DMA transfers profit from the increased computational complexity as well, since the DMA controller has more time to bring in the data, which reduces idle times of the SXUs while waiting for transfers to finish.

6.2.2 Scaling of the Gauss-Seidel Algorithm

In this section we examine how the implementation scales when additional SPEs are added. The speed-up when using n SPEs is defined as

$$\text{speed-up}(n) = \frac{\text{runtime using 1 worker SPE}}{\text{runtime using n worker SPEs}}.$$

The optimum speed-up when using n SPEs is n. This is, however, unachievable in practice, since adding processors increases the communication overhead.

Figure 6.2 and Figure 6.3 show speed-ups for the two versions of the Gauss-Seidel algorithm. Consistently with the results from the previous section, the

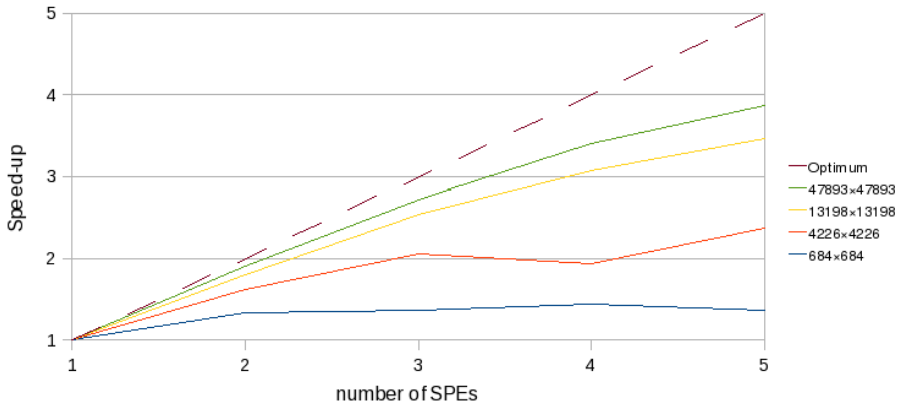

Figure 6.3: Speed-ups for the projected Gauss-Seidel algorithm on up to five SPEs with varying problem sizes.

implementation scales best with the projected Gauss-Seidel algorithm and large problem sizes. These constellations can be expected to profit most when increasing the number of worker SPEs beyond five. The two smallest problem sizes exhibit irregular behavior: For the smallest problem the implementation becomes slower when more than four SPEs are used. This is because the added communication overhead outweighs the contribution of the fifth SPEs to the computation. The bump in the curve for the second smallest problem in Figure 6.3 may be explained by delays due to unfortunate synchronization circumstances that we could, however, not track down further.

6.2.3 The Complete Multigrid Solver

In this section we will examine how the complete TNNMG solver performs on the Cell processor. The hierarchy we used for testing comprises only the three smallest matrices listed in table 6.1 due to the limited memory resources. We used an iterative Gauss-Seidel solver as base solver. As explained in Chapter 3, a significant part of the TNNMG solver has not been optimized for the Cell platform since this would have exceeded the time constraint of this master thesis. The unoptimized part is executed by the PowerPC core and therefore performs poorly as expected.

Table 6.3 lists the average runtime for each part of the algorithm. The effect of the unoptimized code on the overall performance is devastating. Note that the base solver uses an optimized Gauss-Seidel step, but since the difference in energy computed after each iteration is computed by the PPU, the complete base solver is slower on the PS3 than on the other machines.

Operation	level	IBM Cell 3.20 Ghz	Intel T2250 1.73 Ghz	Intel P4: 3.00 Ghz	Speed-up
projected smoother	2	14.77	50.10	48.97	↑3.32
truncation	2	23.14	0.46	0.34	↓67.77
matrix restriction	2	33943.00	1331.50	1055.06	↓32.17
vector restriction	2	163.60	11.53	10.53	↓15.54
linear smoother	1	4.26	8.76	7.40	↑1.74
matrix restriction	1	2498.17	98.81	77.65	↓32.17
vector restriction	1	43.32	3.07	2.69	↓16.10
base solver	0	752.69	108.32	100.29	↓7.50
vector prolongation	0	21.86	0.97	0.77	↓28.57
linear smoother	1	4.24	8.74	7.51	↑1.77
vector prolongation	1	82.59	3.74	2.77	↓29.83
projected smoother	2	14.82	50.02	48.95	↑3.30
line search	2	101.44	8.74	8.80	↓11.52
error computation		184.34	16.31	16.39	↓11.25
miscellaneous		17.47	0.59	0.52	↓33.56
total		37685.38	1685.35	1372.24	↓27.46

Table 6.3: Average runtime for each part of the TNNMG algorithm. Green arrows (↑) indicate speed-ups and red arrows (↓) indicate slow-downs in comparison to the P4 processor.

6.3 Double-Precision

For reasons given in Section 6.1 the results for the double-precision version of the Gauss-Seidel algorithm are only discussed briefly in this section. They are given in Table 6.4. Despite the weakness in double-precision performance, we still observe a speed-up in comparison to the other two processors. The decreased computational power is at least partly mitigated by the dependency of the Gauss-Seidel algorithm on the speed of the system memory.

Problem Size	IBM Cell 3.20 Ghz	Intel T2250 1.73 Ghz	Intel P4: 3.00 Ghz	Speed-up (P4)
linear Gauss-Seidel				
684×684	2.90	2.01	1.89	0.65
4226×4226	8.26	16.33	12.24	1.48
13198×13198	32.47	51.19	46.17	1.42
47893×47893	111.54	194.79	170.16	1.53
projected Gauss-Seidel				
684×684	3.44	3.09	3.06	0.89
4226×4226	10.94	21.69	18.76	1.71
13198×13198	44.78	67.02	69.00	1.54
47893×47893	151.79	287.05	255.01	1.68

Table 6.4: Runtimes for three iteration of the Gauss-Seidel algorithm in milliseconds using double-precision. The speed-up column shows to the reduction of the runtime relative to the P4 processor.

Chapter 7

Conclusion

The results for the complete TNNMG algorithm show that there is still a lot of work to be done until the performance of the implementation on the Cell processor can compete with the performance on the other tested platforms. It has also become clear that the CBE is, at least in its tested incarnation, not at all suited to speed-up a time critical part of the application and leave the rest unmodified. The dramatically reduced performance of the parts of the implementation that run on the PPU leaves no doubt that these parts of the solver need to be ported to the SPEs as well, before the implementation is suited for practical use. This will, however, most likely require additional months of work.

The performance of the Gauss-Seidel algorithm that has been optimized for the SPEs is, however, promising. The restricted resources of the PS3 have limited the test we could make to a minimum. The problem sizes we could use for testing are relatively small and the number of available SPEs is limited. A full grown IBM *Cell BladeCenter* like the QS22 [3], however, provides much larger memory and a total of 16 SPEs. The trends exhibited by our test results show that the implementation could benefit from these additional resources. The performance gains increase with growing problem sizes and the scaling behavior observed when adding SPEs indicates that the implementation could improve its performance if additional SPEs were available.

Bibliography

[1] Auto-vectorization in GCC. `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`.

[2] DUNE Distributed and Unified Numerics Environment. `http://dune-project.org/`. [Online; accessed 26-June-2008].

[3] IBM BladeCenter QS22. `http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs22/index.html`. [Online; accessed 26-June-2008].

[4] STL Allocators. `http://www.sgi.com/tech/stl/Allocators.html`. [Online; accessed 26-June-2008].

[5] The Standard Input / Output Streams Library. `http://www.cplusplus.com/reference/iostream/`. [Online; accessed 26-June-2008].

[6] The Visible Human Project. `http://www.nlm.nih.gov/research/visible/visible_human.html`. [Online; accessed 26-June-2008].

[7] Markus Alind, Mattias V. Eriksson, and Christoph W. Kessler. Blocklib: a skeleton library for cell broadband engine. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14, New York, NY, USA, 2008. ACM.

[8] Dietrich Braess. *Finite Elements.* Cambridge University Press, 2001.

[9] Gregory Buehrer, Srinivasan Parthasarathy, and Matthew Goyder. Data mining on the cell broadband engine. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 26–35, New York, NY, USA, 2008. ACM.

[10] Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perform. Comput. Appl.*, 21(4):457–466, 2007.

[11] Shimon Even. *Graph algorithms.* Computer software engineering series. Pitman, 1979.

[12] R. Glowinski. *Numerical Methods for Nonlinear Variational Problems*. Series in Computational Physics. Springer Verlag, 1984. cited in [29].

[13] C. Gräser, U. Sack, and O. Sander. Truncated nonsmooth Newton multigrid methaods for convex minimization problems. In *Proc. of DD18*, submitted. cited in [29].

[14] Wolfgang Hackbusch. *Iterative solution of large sparse systems of equations*. Applied mathematical sciences; 95. Springer Verlag, 1994.

[15] Hans-Christian Hege and Hinnerk Stüben. Vectorization and parallelization of irregular problems via graph coloring. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 47–56, New York, NY, USA, 1991. ACM.

[16] IBM Corporation. *Accelerated Library Framework for Cell Broadband Engine Programmers Guide and API Reference*, 2007. `http://www.ibm.com/developerworks/power/cell/documents.html?S_TACT=105AGX16&S_CMP=LP`.

[17] IBM Corporation. *Basic Linear Algebra Subprograms Programmer's Guide and API Reference*, 2007. `http://www.ibm.com/developerworks/power/cell/documents.html?S_TACT=105AGX16&S_CMP=LP`.

[18] IBM Corporation. *C/C++ Language Extensions for Cell Broadband Engine Architecture Version 2.5*, 2007. `http://www.ibm.com/developerworks/power/cell/documents.html?S_TACT=105AGX16&S_CMP=LP`.

[19] IBM Corporation. *Cell BE Programming Tutorial Version 3.0*, 2007. `http://www.ibm.com/developerworks/power/cell/documents.html?S_TACT=105AGX16&S_CMP=LP`.

[20] IBM Corporation. *Cell Broadband Engine Programming Handbook Version 1.1*, 2007. `http://www.ibm.com/developerworks/power/cell/documents.html?S_TACT=105AGX16&S_CMP=LP`.

[21] IBM Corporation. *LAPACK Programmer's Guide and API Reference*, 2007. `http://www.ibm.com/developerworks/power/cell/documents.html?S_TACT=105AGX16&S_CMP=LP`.

[22] IBM Corporation. *Monte Carlo Library API Reference Manual*, 2007. `http://www.ibm.com/developerworks/power/cell/documents.html?S_TACT=105AGX16&S_CMP=LP`.

[23] Jeff Derby. Cell Programming Tutorial. `http://www.cs.unc.edu/~geom/EDGE/SLIDES/derby.ppt`. [Online; accessed 26-June-2008].

[24] Theodore Kim. Hardware-aware analysis and optimization of stable fluids. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 99–106, New York, NY, USA, 2008. ACM.

[25] Ralf Kornhuber and Christian Schütte. Numerik von partiellen differentialgleichungen. Lecture notes SS 2007, Freie Universität Berlin.

[26] Jakub Kurzak and Jack Dongarra. Implementation of mixed precision in solving systems of linear equations on the cell processor. *Concurr. Comput. : Pract. Exper.*, 19(10):1371–1385, 2007.

[27] Y. Liu, H. Jones, S. Vaidya, M. Perrone, B. Tydlitát, and A. K. Nanda. Speech recognition systems on the cell broadband engine processor. *IBM J. Res. Dev.*, 51(5):583–591, 2007.

[28] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007.

[29] Oliver Sander. *Multidimensional Coupling in a Human Knee Problem*. PhD thesis, Freie Universität Berlin, 2008.

[30] Harald Servat, Cecilia Gonzalez, Xavier Aguilar, Daniel Cabrera, and Daniel Jimenez. Drug design on the cell broadband engine. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, page 425, Washington, DC, USA, 2007. IEEE Computer Society.

[31] Sajjan G. Shiva. *Advanced Computer Architectures*. Taylor and Francis, 2005.

[32] Detlev Stalling, Malte Westerhoff, and Hans-Christian Hege. Amira: A highly interactive system for visual data analysis. In Charles D. Hansen and Christopher R. Johnson, editors, *The Visualization Handbook*, chapter 38, pages 749–767. Elsevier, 2005. cited in [29].

[33] Andreas Stiller. Cell-Kultur - Innenleben und Programmierung des Cell-Prozessors. *c't Playstation 3 special*, 2007.

[34] trevor_smigiel@playstation.sony.com. aligned attribute and the new operator (pr/15795). archived at *http://gcc.gnu.org/ml/gcc/2006-10/msg00166.html*. GCC mailing list, October 2006.

[35] Wikipedia. Amdahl's law — Wikipedia, The Free Encyclopedia, 2008. [Online; accessed 26-June-2008].

[36] Wikipedia. Cell (Prozessor) — Wikipedia, Die freie Enzyklopdie. `http://de.wikipedia.org/w/index.php?title=Cell_%28Prozessor%29&oldid=45212818`, 2008. [Online; accessed 27-May-2008].