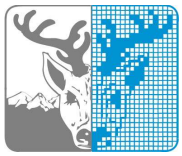
Smoothed Particle Hydrodynamics on the Cell Broadband Engine

Nils Hjelte

June 27, 2006

Master's Thesis in Computing Science, 20 credits

Supervisor at UmU-VRlab/HPC2N: Kenneth Bodin
Examiner at UmU-CS: Per Lindström



Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Smoothed Particle Hydrodynamics (SPH) is a method used mainly to simulate complex materials, such as water. It would be of great benefit for many application areas to be able to run large SPH systems at interactive speeds.

We have looked at ways to improve the performance of a SPH fluid simulation by designing algorithms for parallel execution. These have been used to develop an implementation for the target platform, the Cell Broadband Engine processor. Experimental results show great potential with linear performance scaling for increasing problem sizes, and excellent parallel efficiency.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A realistically behaving and good looking water simulation in a game will make the environment more immersive and lifelike. An interactive simulation of human organ tissue is of great use in medical simulators. Both these are examples of possible applications for a Smoothed Particle Hydrodynamics (SPH) simulation. SPH is a particle based simulation method, which is relatively inexpensive and can be run at interactive speeds, while at the same time giving relatively accurate simulations of different materials. We will focus on the usage of SPH to simulate incompressible fluids, such as water. See chapter 2 for some background and derivation of the equations used in SPH.

Interactive is a key word for this master thesis. It requires the simulation to run with real-time performance, meaning a high and consistent update frequency. The desire to run increasingly large simulations at interactive speeds creates a need for more sophisticated software methods and/or faster hardware.

Our motivation for implementing interactive particle simulations, in addition to an interest in purely algorithmic research, is application in vehicle simulators for modeling movement of earth, and as simulation of tissue and body fluids in medical training simulators.

A distinctive trend in hardware development is the rise of increasingly parallel architectures, with multiple computational units and great theoretical performance. But in order to take advantage of all those computational resources, software algorithms must be designed for parallel execution.

The main objective of this project is to explore ways to create a high performing SPH fluid simulation. There are many possible implementation targets and in chapter 3 we briefly analyze some available platforms. The chosen platform, Cell, is given a more in-depth look, and in chapter 4 the final implementation is described.

Performance analysis and profiling of the resulting implementation are presented in chapter 5.

Chapter 2

Theory

2.1 Methods

2.1.1 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is a meshless (no explicit connectivity between particles) particle based method, initially used for simulation of stellar gas in cosmology, when conceived in 1977. Monaghan [33] gives a in-depth compilation on SPH theory and application. It has over time been applied to numerous problems, such as, elasticity and fracture modeling [33], hair-hair interactions [21], and simulation of incompressible fluids [33].

The particles can be seen as interpolation points, approximating local field quantities. The basic SPH formula

$$A_s(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.1}$$

sums over nearby discritized points (particles) to calculate a scalar quantity A_s at point $\mathbf{r}$. The variable j loops over all neighbors, where m_j is the mass of j, $\mathbf{r}_j$ the position, ρ_j the density, and A_j the field quantity at $\mathbf{r}_j$. The weighting function W (also called smoothing kernel) makes the contribution from particle j dependent on its distance to $\mathbf{r}$, with h as a cut-off distance (points outside get $W = 0$). The function should be normalized, which means

$$\int_r W(\mathbf{r}, h) d\mathbf{r} = 1 \tag{2.2}$$

Each particle i represents a volume $V_i = m_i/\rho_i$, which is the reason why mass and density are part of equation (2.1). The mass is constant over the simulation, but the density has local variations and needs to be recalculated each time step. Inserting density as the field quantity in equation (2.1) gives a interpolation formula for the density at point $\mathbf{r}$,

$$\rho_s(\mathbf{r}) = \sum_j m_j \frac{\rho_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.3}$$

A nice property of SPH is that derivatives of field quantities only affect the kernel function. The gradient and Laplacian of A are shown in equations (2.4) and (2.5) respectivly.

$$\nabla A_s(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.4}$$

$$\nabla^2 A_s(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.5}$$

SPH fluid equations

For the task of simulating incompressible viscous fluids, the SPH formulations are derived from the Navier-Stokes equation. Müller *et al* [35] introduced SPH for interactive simulations in computer graphics, and our equations follow this paper. The acceleration $\mathbf{a}_i$ of a particle i is

$$\mathbf{a}_i = \frac{\mathbf{f}_i}{\rho_i} \tag{2.6}$$

where $\mathbf{f}_i$ and ρ_i are the force density field and the density field, locally at the position of particle i. The force density field $\mathbf{f}_i$ is the sum of force fields from pressure, viscosity, and external forces.

$$\mathbf{f}_i^{total} = \mathbf{f}_i^{pressure} + \mathbf{f}_i^{viscosity} + \mathbf{f}_i^{external} \tag{2.7}$$

Direct insertion of the pressure term from Navier-Stokes into the SPH formula yields asymmetric forces, which is not physically accurate. A simple modification is suggested, producing the following formula for calculating the pressure force density field:

$$\mathbf{f}_i^{pressure} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.8}$$

The variables p_i and p_j are the pressures at particles i and j respectively, and they are calculated using

$$p = c^2(\rho - \rho_0) \tag{2.9}$$

where c is the speed of sound (in the fluid) and ρ_0 is the reference density of the fluid.

As with the pressure force equation, when applying the SPH rule to give the viscosity force term, asymmetry appears. The fact that viscosity forces depends on relative velocities rather than absolute velocities allows for a simple modification, yielding the following (symmetric) equation.

$$\mathbf{f}_i^{viscosity} = \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.10}$$

Kernel functions

Three different smoothing kernels are presented by Müller *et al.*

$$W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9}(h^2 - r^2)^3 \tag{2.11}$$

The r is the length of the vector $\mathbf{r}$. Since only r^2 appears in equation (2.11) we can, using the W_{poly6} kernel, avoid expensive square root operations. However, because the gradient of the kernel approaches zero as r goes to zero, particles very close together will not generate enough pressure force to repel each other. A second kernel, for usage in pressure force calculations, was therefore devised:

$$W_{spiky}(\mathbf{r}, h) = \frac{15}{\pi h^6}(h - r)^3 \tag{2.12}$$

for which the gradient is

$$\nabla W_{spiky}(\mathbf{r}, h) = -\frac{45}{\pi h^6}(h - r)^2 \mathbf{r} \tag{2.13}$$

Viscosity turns kinetic energy into heat due to friction, thusly reducing the relative velocities of particles. But the Laplacian of equation (2.11) is negative for small values of r, which for particles close to each other will increase their relative velocities, and thereby violating the physical properties of the fluid. Kernel W_{spiky} exhibits the same problem, so for viscosity force calculations, a third kernel

$$W_{viscosity}(\mathbf{r}, h) = -\frac{15}{2\pi h^3}(-\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1) \tag{2.14}$$

with Laplacian

$$\nabla^2 W_{viscosity}(\mathbf{r}, h) = \frac{45}{\pi h^6}(h - r) \tag{2.15}$$

is used. Finally, remember that all kernel functions return 0 if the distance is greater than the cut-off value (i.e. if $r > h$)

2.1.2 Neighbor searching

In a basic SPH implementation we can calculate each field quantity by iterating over all particles in the system. On the downside, such a naive implementation will have $O(n^2)$ complexity, thus scaling poorly for large systems.

What we really need, since the contribution of particles outside the cut-off distance h is ignored, is a list for each particle i containing all neighboring particles j such that $|\mathbf{r}_i - \mathbf{r}_j| < h$.

There are many methods to decrease the computational cost of finding all neighbors in a system. The general idea is to perform the search in two passes. The first pass, called *broad phase*, generates groups of particles that possibly interact, which then are used during the second pass (*narrow phase*) to calculate exact distances.

This section describes a few available methods, Ericson [19] gives a much more thorough presentation on collision detection techniques.

Verlet neighbor lists

The Verlet neighbor list algorithm takes advantage of temporal locality of particles, and assumes they can only move within a certain radius each time step. Each particle has a list of possible neighbors to check distances against. If the distance between two particles exceeds a threshold value, they can be excluded from each others neighbor testing lists for several time steps. The computational cost for this algorithm is however still $O(n^2)$. Also, the large computational variations between time steps makes it not suitable for interactive simulations, where a consistent update frequency is desired.

Spatial partitioning divides space into cells, to which particle positions are mapped. The proximity of particles is (broadly) determined by the spatial relation of their cells. The spatial partitioning can either be uniform, using a regular grid, or hierarchical, where recursive subdivision of the space adapts the cell sizes to the variations in object density.

Uniform grids

By setting the length of the cells equal to the interaction radius h of the particles, the uniform partitioning guaranties that all neighbors to a particle are located in the same cell or one of the neighboring cells (figure 2.1).

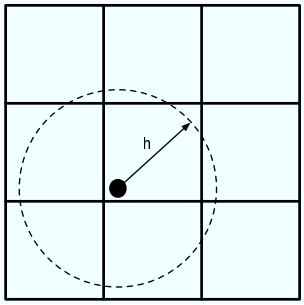


Figure 2.1: The set of cells containing all possible neighbors to a particle. For this specific example, the upper right cell can not contain a neighbor to the shown particle, but this is not true in general. In three dimensions, the region containing all possible neighbors to a particle is a `3x3x3` cube.

When dealing with large systems, storing the entire grid in memory can be overwhelming, in particular since most grid cells will be empty when simulating a heterogeneous particle system. A common technique to reduce the memory footprint is to store the grid cells in a *hash table* (see e.g. [45]). Through a hash function, such as described in [42], it is possible to map the cells in a potentially infinite grid on to a small number of hash buckets. The particle system simulation in this paper uses spatial hashing for neighbor searching (see chapter 4 for more implementation details).

Hierarchical partitioning

Hierarchical partitioning is done by recursively dividing cells into sub-cells. The method can adapt very well to non-uniform distribution of objects in space. A single large cell can cover a large region of empty space, while a region containing a cluster of objects will get a much more fine grained partitioning. This recursive behavior maps naturally to a tree data structure.

A common method using hierarchical partitioning is the *octree* (*quadtree* in 2d), which uses axis-aligned partitioning. The recursive step is done by splitting a cell in half along every dimension, as depicted in figure 2.2.

Other methods for hierarchical subdivision include the k-d tree [19], the Barnes-Hut tree [43], and the balanced binary tree [43].

See [19] for more information on the subject of spatial partitioning.

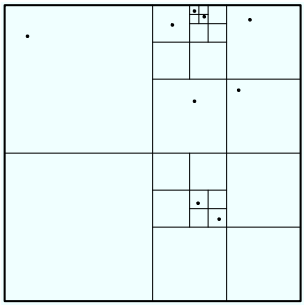


Figure 2.2: A two-dimensional space, recursively partitioned by a quadtree until each cell contain at most one particle.

2.2 Performance Aspects

When it comes to physical simulations there is always a desire to simulate even larger and more complex systems. Performance is therefore a very important factor, especially for real-time simulations. This section will discuss different aspects of the performance of computer programs, and SPH simulations in particular. More implementation specific optimization techniques, such as loop unrolling, are described in [19].

2.2.1 Asymptotic Complexity Of Algorithms

An implementation independent comparison of algorithms can be done using asymptotic complexity analysis. It does not measure performance in absolute terms, but can be used to describe how the time and space requirements scale with the size of the problem[45].

For SPH, the local local field quantities are calculated using only neighboring particles. Assuming the neighboring particles are known, the cost of computing the SPH forces is of order $O(kn)$, where n is the number of particles in the system and k is the average number of neighbors/particle. By keeping the average number of neighbors/particle constant when simulating larger systems the computational requirements should scale linearly. The drawback of keeping k constant when n grows is that additional time steps are required to propagate forces through the system.

The cost of finding neighbors can be overwhelming if a bad algorithm is chosen, such as the exhaustive all-to-all $O(n^2)$ method, which will quickly become the bottleneck. Adding a broad phase searching algorithm (described in section 2.1.2) is essential when simulating large systems.

2.2.2 Parallel Execution

Although it does not lower the computational cost, parallel algorithms allow more hardware to be used to decrease the runtime of a program. Instruction level parallelism can be exploited using SIMD (Single Instruction Multiple Data) units, which are commonly available on modern CPUs. A machine with multiple CPUs and/or multiple cores can be used to run multiple threads in parallel, called MIMD (Multiple Instruction Multiple Data) programming. The processors can have shared memory (for example a conventional multi-CPU workstation) or distributed memory (normally a multi-node PC cluster). Instruction level parallelism and thread level parallelism can preferably be used together since they are generally not mutually exclusive.

The SPH method is inherently parallel due to the local independent force calculations. A number of different parallel SPH implementations on different platforms have been made. Some of these are described in chapter 3.

2.2.3 The Memory Bottleneck

The performance of microprocessors has grown at a much faster rate than the performance of memory subsystems. Many programs are unable to fully utilize the CPU due to expensive memory transfers. To help reduce the cost of accessing memory, modern computers use a hierarchical layering with small but fast cache memories close to the processor. However, to get good cache utilization, algorithms and data structures must be designed for high data locality, in both time and space. Temporal locality is achieved by keeping accesses to the same data structure grouped together in time. Spatial locality means high probability that a data element is accessed if a nearby element was recently accessed. By having good spatial and temporal reuse of data, memory latency will drop and less bandwidth will be consumed.

Unfortunately, interacting particle simulations such as SPH have irregular (not fixed) memory access patterns, which gives high cache miss rates on large simulations (small simulations may fit entirely in cache, thus avoiding the problem). Due to the irregularities, it is impossible to perform compile-time data reordering. It is however possible to restructure data at run-time to improve cache utilization. Papers by Ding and Kennedy[16], and Badawy *et al*[13] show that dynamic data transformations can give substantial gains in run-time performance.

Chapter 3

Platforms

3.1 Platform survey

This section gives a brief overview of different implementation targets for a real-time Smoothed Particle Hydrodynamics (SPH) fluid simulation. Each platform is discussed with focus on how well suited it is for this specific problem type. Also rather important, since this is a master thesis with a large practical component, is how well explored the platform is, and how interesting an implementation will be, and if the results can provide any new knowledge.

3.1.1 Conventional desktop CPU

Conventional desktop Central Processing Units (CPUs) - such as AMD's Athlon series [1], IBM's 970/970FX processors [9], and Intel's Pentium 4 family [7] - can be considered the reference platform, against which other implementations are compared. They are extremely common and relatively cheap, with many different languages and programming models available when developing software.

There are however many factors to consider when trying to minimize application runtime. Memory bandwidth and latency has become an increasingly large bottleneck on modern CPUs, where efficient cache utilization is vital for computationally intensive software. Large speedups can be gained using data prefetching and exploiting temporal and spatial data locality [13]. It may also be important to consider aspects such as the latency for different hardware instructions, where e.g the cost of floating point division typically is much higher than that of floating point multiplication [26]. Some general programming techniques for software optimization are shown by Ericson [19].

Most modern desktop CPUs contain a Single Instruction Multiple Data (SIMD) unit, which process multiple data elements stuffed in (typically) 128-bit vectors. And at a more abstract level, a growing number of processors provide multiple execution cores,

running separate instruction streams. In order to take advantage of these features, software must must be designed for parallel execution. Many steps in the SPH neighbor search and force calculations are naturally vectorizable, and multithreading is to some degree possible. These possibilities are explored further by the implementation presented in chapter 4.

3.1.2 Graphics Processing Unit

The programmable shaders on modern Graphics Processing Units (GPUs) allows general-purpose calculations to be performed on the GPU, using the stream programming model [37]. Uberflow [28] and "Million Particle System" [32] are two projects, using ATI and NVIDIA graphics hardware respectively, to simulate large uncoupled particle systems i real-time. The simulations run entirely on the GPU and provides a large speedup compared to CPU-based implementations.

Takashi *et al* presents a method for solving the SPH equations on a GPU [12], but the neighbor searching is still done using the host CPU. This induces not only a computational bottleneck, but also a bandwidth bottleneck since all particles and neighbor search data must be transferred between GPU and CPU each time step.

An SPH implementation on GPU using techniques to remove the need for global sorting or explicit n-nearest neighbor searching is presented by Kolb and Cuntz [30]. The paper is however rather short on details, and the performance is lower than the serial CPU implementation in this paper (see section 5.1 for performance results).

A set of different methods for locating nearest photons in a photon mapping algorithm is described in [38], but they are limited by not having random write access on GPUs. And while it is possible to construct such algorithms for the GPU, the problem of neighbor searching seems to map poorly on the stream processing model.

Also, the problem with all these GPU implementations is that they are standalone simulations. Integrating them into e.g. a complex game environment may induce new limitations. The simulation will have to share the resources on the GPU with other parts of the application. And, if for example a SPH fluid simulated solely on the GPU is to interact with other entities simulated on the CPU, it may be necessary to transfer all particles back to the CPU each time step, which would nullify an otherwise major advantage for pure GPU-based simulations.

3.1.3 Distributed memory PC cluster

A popular and inexpensive way of building a super computer is to connect a large number of PC workstations or blade servers using high speed interconnects. Commonly used network topologies include the hypercube and k-d mesh[20].

Within the fields of molecular dynamics and astrophysics there are many parallel N-body implementations for distributed memory machines. Warren and Salmon [44] per-

form spatial subdivision using an oct-tree, which is stored in a hash table for efficient distributed access.

A large long range N-body simulation may take a year to complete on a single computer, but using a cluster may reduce this to a day or two. This extremely high computational cost would explain why so much research has been done on parallelizing long range force calculations.

But for short ranged interaction systems (such as SPH), it is hard to find implementations for distributed memory machines. Probably mainly because most short ranged simulations do not have real-time requirements, and then there is little reason to use a cluster when a single workstation can produce results within reasonable time.

However, the locality of the force calculations in SPH should enable relatively clean separation of data dependencies and low communication costs, and it would indeed be interesting to test the scalability of a real-time SPH implementation on a distributed memory machine.

3.1.4 Ageia PhysX

Ageia hopes that the concept of a Physics Processing Unit (PPU) will do for physics what the Graphics Processing Unit has done for graphics. The Ageia PhysX [11] is the first commercial PPU hardware implementation. Unfortunately, very little in-depth information on the PhysX card is publicly available. According to [39] Ageia has shown some impressive demonstrations using real-time fluid and rigid body simulations, and a number of upcoming game titles will support the PhysX card [8].

Programming for the PhysX PPU is done using Ageia's physics software API, NovodeX. It is unclear whether developers will be have fine grained control of the hardware resources, using a more low level programming model, which is necessary for PhysX to become an interesting implementation target for computing scientists.

3.1.5 MDGRAPE-3

MDGRAPE-3 (Molecular Dynamics GRAvity PipE) is a special-purpose computing chip for molecular dynamics (MD) simulations [41]. It is specialized in calculating particle interaction forces - such as gravitational, Coulomb, and van der Waals forces - and must be connected to a host computer. The "Protein Explorer" is a project to build a petaflop computer using a cluster of 256 PCs, each with 2 CPUs and a PCI-X board with 24 MDGRAPE-3 chips.

The advantage of using MDGRAPE-3 is that it provides very low cost/Gflop and power/Gflop [40]. Being a special-purpose processor also means simulations will achieve close to peak performance, and the computational accuracy can be optimized for the specific problem domain (e.g. forces are accumulated in 80-bit fixed-point format).

The idea behind MDGRAPE is that the $O(N^2)$ cost to calculate long range interactions will clearly dominate the $O(N)$ communication cost between host and the special-purpose engine. The SPH method used in this paper however use a cutoff radius with a relatively low number of neighbors, and a force calculation cost of $O(NN_c)$, Nc being the average number of neighbors. In such a setting, the communication cost can have a large impact. And as seen in section 5.1, the force calculation cost is nowhere as dominant as for the MDGRAPE targeted applications.

Also, the programming model for the MDGRAPE allows for very little flexibility. There is pretty much only one way of doing an implementation, using the supplied subroutine packages. This may simplify things and can be positive when using such a system in practice, but it makes MDGRAPE a not so exciting implementation target for this master thesis.

3.1.6 Cell Broadband Engine

The Cell Broadband Engine Processor is part of a new design philosophy for desktop-grade CPUs. It is developed by IBM, Sony and Toshiba and has a very parallel design with lots of computational resources. It is mainly targeted at multimedia and game applications and is the chosen CPU for the upcoming Playstation 3 game console [10]. The processor consist of one "conventional" processing core, mainly responsible for running the OS, and eight autonomous SIMD processing units, to be used for highly computationally intensive tasks [27]. It has very impressive peak performance, both computationally and bandwidth-wise, and is designed for efficient performance/area and performance/power ratios. This comes at the expense of lowered single thread performance, and applications must specifically target the Cell processor to efficiently utilize its resources. Furthermore, the eight SIMD processing cores (SPEs) have poor double-precision floating point performance [15], and their single-precision floating point units are not IEEE compliant [25].

The Cell Broadband Engine is, however, the chosen platform for the implementation in this paper. The reasoning behind this is the potential performance available, and the fact that it is a very new and unexplored platform with no, to our knowledge, prior SPH implementation specifically written for it. Other important factors are that IBM provides extensive, well written, documentation, and a publicly available Cell system simulator [3].

3.1.7 Closing comments

What can be said in general for all the above platforms is that in order to utilize them optimally and achieve a high performing simulation it is necessary to exploit parallelism. From 2-way or 4-way multicore desktop CPUs to extremely parallel platforms such as a MDGRAPE-3 system or a modern GPU it is clear that parallel algorithms are the way of the future. The SPH implementation presented in this paper (chapter 4) is heavily focused on utilizing the many levels of parallelism available on the Cell Broadband Engine processor.

3.2 The Cell Broadband Engine Processor

Developed by IBM, Sony and Toshiba, the Cell Broadband Engine Architecture (CBEA) [22] represents a radical change in processor design. Designed to increase performance/area and performance/power ratios, and to overcome some of the bottlenecks seen in conventional desktop CPUs [27].

The Cell is a parallel design with multiple computational cores connected using the Element Interconnect Bus (EIB). Two types of computational units are specifies by the CBEA:

- The PowerPC Processing Element (PPE), a 64-bit general purpose processor implementing the PowerPC instruction set, including the Vector Multimedia eXtensions (VMX) for SIMD processing.
- The Synergistic Processor Element (SPE), which is a high performing SIMD processing unit designed for streaming workloads.

Modular by design, a CBEA compliant system can contain multiple PPEs and SPEs. The first CBEA implementation is the Cell Broadband Engine processor, hereafter referred to as the Cell. It consists of one PPE and eight SPEs, organized around the high-speed EIB ring bus, as shown in figure 3.1.

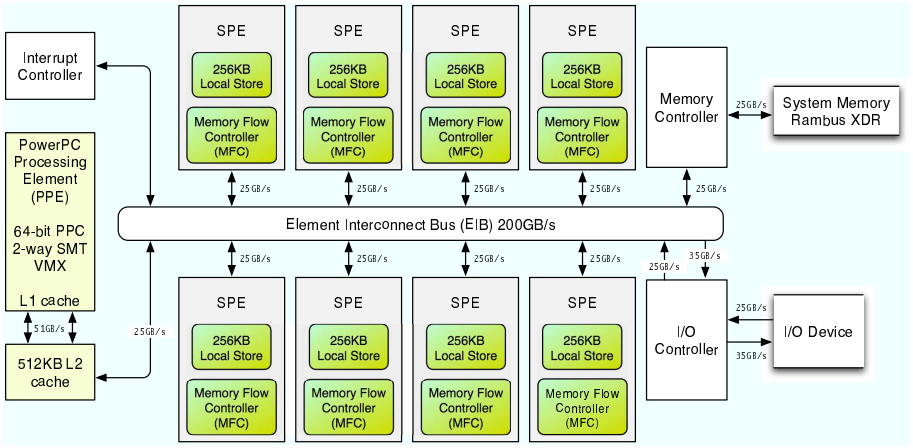


Figure 3.1: Abstract block diagram of the Cell BE Processor.

The initial implementation has a targeted clock frequency of 3.2GHz, and uses Rambus XDR DRAM memory to achieve high main-memory bandwidth.

The Broadband Engine Interface (BIF) I/O interface allows two Cell processors to communicate in a symmetric multiprocessor (SMP) system, and using a switched bus allows even larger systems [31].

3.2.1 The PowerPC Processing Element

The PowerPC Processing Element (PPE) is a general purpose, 64-bit PowerPC processor. It is responsible for running the operating system and managing the SPEs. Two instructions can be issued each clock cycle, which are executed in-order (with limited out-of-order execution of load instructions). Instructions from two threads are interleaved using two-way simultaneous multithreading (SMT). The cache hierarchy consist of 32KB L1 data and instruction caches, and a 512KB L2 cache.

In addition to the PowerPC instruction set architecture, the PPE core also supports SIMD instructions by implementing the Vector Multimedia eXtensions (VMX) instruction set.

3.2.2 The Synergistic Processing Element

The Synergestic Processing Element (SPE) is an autonomous processor designed for game and multimedia applications and other computationally intensive tasks. It consists of a execution core, called Synergestic Processing Unit (SPU), with an associated 256KB local storage (LS) memory, and a Memory Flow Controller (MFC) for transferring data between the LS and system memory. The SPU is a RISC processor implementing a new, multimedia oriented, 128-bit SIMD instruction set architecture. The operands are typically composed of 16 x 8-bit values, 8 x 16-bit values, 4 x 32-bit values or 2 x 64-bit values, and are stored in a large 128-entry unified register file (unified meaning all data types use the same register file).

The SPU has two non-identical execution pipelines; one (called "even pipe") for fixed and floating point arithmetic instructions, the other (called "odd pipe") for loads/stores, permutations, and branching. Instructions are executed in-order, and independent instructions can be dual-issued, provided that the first instruction comes from an even word address and uses the even pipe, and the second instruction comes from an odd word address and uses the odd pipe. All instructions are fully pipelined, except for double-precision floating point operations, which have a maximum issue rate of one SIMD instruction per seven cycles.

The SPE hardware does not include any complex branch speculation logic, all branches are assumed not taken. There is however a "branch hint" instruction, overriding the default behavior, which allows software to use compile-time branch prediction. In addition, short branches can be eliminated by executing all paths and choosing the correct result using the bit-select instruction.

Program instructions and data are stored in the 256KB local memory, which is directly addressable by software. Data transfers between LS and system memory are initiated by the SPU by sending Direct Memory Access (DMA) commands to the Memory Flow Controller via the SPU channel interface, or externally (from the PPE or other SPEs) using Memory Mapped IO (MMIO) resources. The DMA commands are executed asynchronously by the MFC, allowing software to hide memory latency using double buffering, where one data buffer is processed by the SPU while the MFC simultaneously

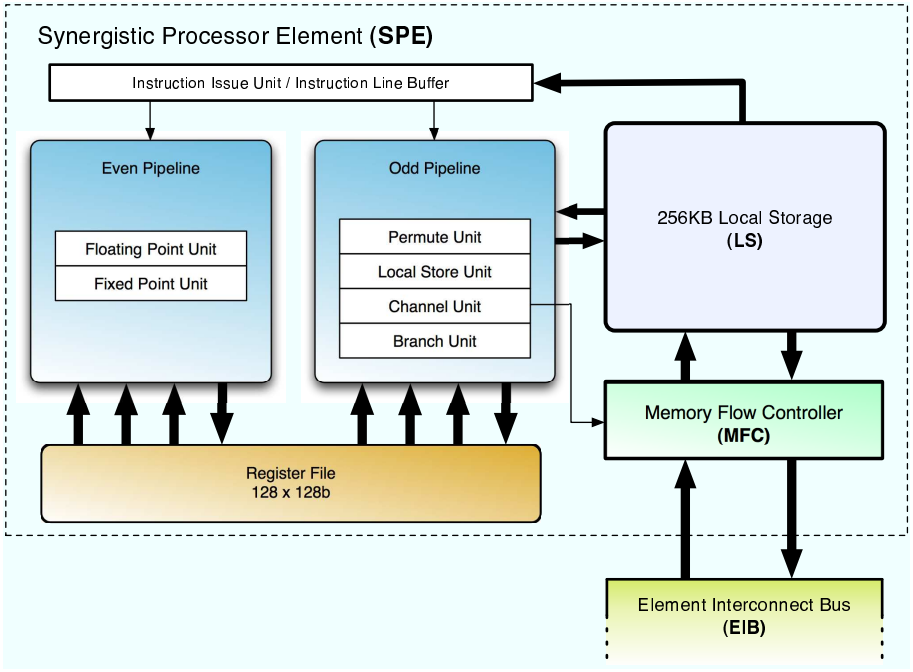


Figure 3.2: SPE block diagram.

transfers the next buffer. The MFC can handle 16 simultaneous transfers, which are completed out-of-order. Each transfer must be naturally aligned with a size of 1B, 2B, 4B, 8B, or multiples of 16B, with a maximum transfer size of 16KB. For optimal performance the transfer size must be a multiple of 128B, with source and target addresses 128B aligned. Discontinuous memory transfers, like gather/scatter operations [20], can be achieved using DMA lists, which allows up to 2048 DMA transfers to be issued using a single DMA command.

3.2.3 Communication between the PPE and SPEs

There are two primary facilities for sending 32-bit messages between the PPE and SPEs; mailboxes and signal notification registers. Each SPE has two outbound, single-entry, mailboxes; the "SPU Write Outbound Mailbox" and the "SPU Write Outbound Interrupt Mailbox". Incoming messages are queued in the "SPU Read Inbound Mailbox", which has capacity to hold four messages.

The signal notification registers are inbound (to the SPEs) 32-bit registers. They can be configured for one-to-one signaling or many-to-one signaling. A comparison of mailboxes and signal notification registers is available in section 3.1.3 of [23].

Chapter 4

Implementation

The primary focus of this implementation is the Cell platform. However, the methods used can easily be modified for multicore/SMP machines using more conventional CPUs. All development was done using the IBM Full-System Simulator for the Cell Broadband Engine Processor [6].

"Cell's greatest strength is that there's a lot of hardware on that chip. And Cell's greatest weakness is that there's a lot of hardware on that chip"
- John "Hannibal" Stokes, ArsTechnica.com

The Cell architecture provides great theoretical performance, but to take full advantage of its many resources requires much more than a recompile of existing serial programs. A set of different programming models for the Cell are described in [27] and [36]. And by offloading work to the compiler, Eichenberger *et al.* looks at how to help developers exploit the Cell's resources [17].

For those interested in seeing what Cell specific code looks like, a hands on tutorial complete with source code examples is available in section 3.6.3 of the CBE Tutorial PDF document, which is included when downloading the IBM Cell Broadband Engine Software Sample and Library Source Code [5].

In order to maximize performance on Cell it is important to utilize many levels of parallelism. How these are used in the implementation is described below.

Single Instruction Multiple Data
At the lowest level there is instruction level parallelism, the SPEs perform operations on 128 bit vectors, using SIMD instructions. A large portion of the algorithms used for this implementation are well suited for vectorizing. When a set of operations are to be performed on all particles, multiple particles can be processed simultaneously. Or when performing operations on 3-dimensional attributes, SIMD instructions are naturally applied.

The data structures are designed to allow for vector processing of the elements (e.g. 3-

dimensional attributes such as position and velocity include a dummy value to maintain 16 byte alignment). Also, similar to the vector permute instruction of the VMX/Altivec SIMD instruction set, the SPU shuffle instruction [25] allows data to be restructured dynamically to exploit instruction level parallelism.

Superscalar pipelining
The superscalar design of the SPEs allows them to execute two instructions per clock cycle, but due to pipeline restrictions the compiler may not always be able to schedule instructions for dual issuing. The type of instructions which can be dispatched into which pipeline is described in section 3.1.1.4 of [23]. The use of a static analyzer like `spu-gcc_timing` (section 3.7.1 in [23]) can give information on dual-issue rates and also helps finding pipeline stalls caused by instruction/data dependencies.

Our implementation does however not include any explicit effort to remove pipeline stalls.

Task level parallelism
The 8 SPEs on the Cell run independent instruction streams, providing MIMD parallelism. Additionally, multiple Cell processors can be used in a SMP fashion [4], which further increases the degree of thread parallelism.

The implementation is designed to be scalable, where the number of SPE threads is user selectable. To achieve good scalability the parallel work overhead is kept to a minimum, although there are some redundant calculations to avoid memory transfers. The threads are synchronized between each subtask, but no inter-thread communication is needed during the subtasks. To balance the workload evenly over the SPEs, a simple workpool is used.

Data transfer parallelism
The DMA engines (MFCs) allows transfers data between local storage of a SPE and main memory in a non-blocking manner. Each MFC has a queue with capacity for 16 MFC commands, which means 128 simultaneous data transfers when utilizing all 8 SPEs. The implementation makes extensive use of double buffering to allow the SPU to process of one buffer while the MFC is concurrently fetching the next one. Also, the data structures are designed to fit the characteristics of the MFCs with, which is necessary to get good memory transfer throughput.

4.1 Simulation Overview

The simulation loop (algorithm 1) consists of a set of subtasks, each described in more detail in the following sections.

Algorithm 1 Simulation loop

while simulation is running **do**
  Neighbor search
  SPH force calculations
  Collision handling - first pass (velocity reflection)
  Time integration
  Collision handling - second pass (position projection)
end while

The data structures used in the simulation are designed for efficient DMA transfers. The particle system is stored as a structure of arrays (SOA).

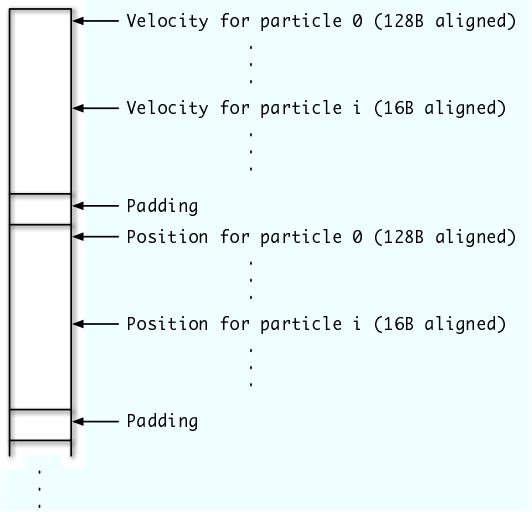


Figure 4.1: Particle system memory layout.

Every subtask running on the SPEs have the same basic structure, as described in algorithm 2, using double buffering to hide data transfer latencies as much as possible.

Algorithm 2 Basic SPE subtask loop

```
Prefetch first workblock
while workpool has more workblocks do
    Prefetch the next workblock
    Wait for current transfer to complete
    Process data in workblock
    Pre-send back results
end while
Wait for last block to arrive
Process last block
Send back last results
```

4.2 Neighbor Search

The neighbor search is done using spatial hashing. The length of the grid cells is equal to the particle interaction radius, which means that all possible neighbors for a particle will be placed in the same cell or one of the neighboring cells. Since interaction pairs are symmetrical it is enough to check 13 of the 26 neighbors from each cell.

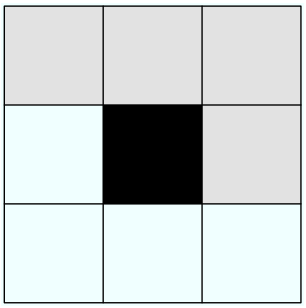


Figure 4.2: A two-dimensional grid where, since interactions are symmetrical, only the gray neighbor cells needs to be examined from the black center cell.

This does however induce data dependencies in the SPH force calculations. So in order to get a clean separation during the parallel SPH calculations, the implementation does in fact look in all 26 neighboring cells for interactions.

4.2.1 Particle hashing

First the cell of the particle is calculated.

$cell$ = floor(particle position / cell length)

The coordinates of the cell are then used to calculate a hash value for the particle. The hash function used is described in [42]. Using three large primes; $P1 = 73856093$, $P2 = 19349663$, $P3 = 83492791$ the hash value is

$hash = (P1 * cell.x \textbf{ xor } P2 * cell.y \textbf{ xor } P3 * cell.z) \textbf{ mod } N$

where N is the size of the hash table.

4.2.2 Hash table

The hash table is a general data structure that can be implemented in many different ways. An interesting property for the hash table in the neighbor search is that it all objects (particles) will be inserted during hash table initialization, after which no more alterations (insertions or deletions) are done. This static property enables the use of a hash table described in [19], section 7.1.4.

The creation of the hash table is done in passes. The first step applies the hash function to every particle, which gives an array of hash values (depicted in figure 4.3).

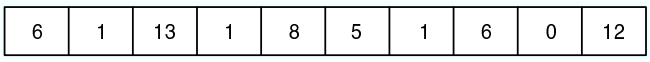


Figure 4.3: Hash value buffer, where the hash value for particle i is located at hashvalues[i].

A histogram (a table where there for each hash value is a count for how many particles that hashed to that value) is then constructed by iterating over the hash values.

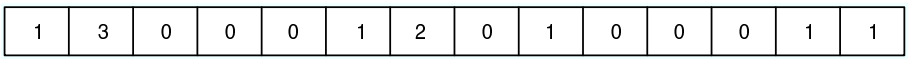


Figure 4.4: Histogram for the hash values.

By performing a prefix sum calculation on the histogram, the offset of every bucket in the hash table is found. The histogram can be now be discarded since the size of hash bucket i can be calculated easily by subtracting offsettable[i] from offsettable[$i + 1$].

With the size and offset of each hash bucket known, the particles can be inserted into the hash table.

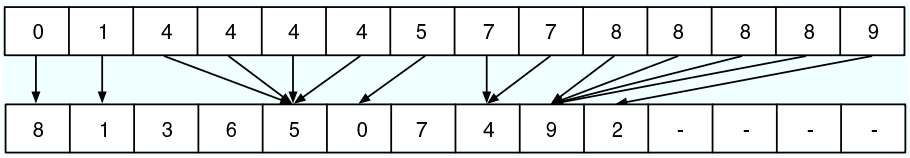


Figure 4.5: Offset table (above), which stores indices to the hash buckets in the hash table (below)

The hash buckets are stored in contiguous memory, reducing memory overhead and providing better cache utilization, compared to a linked list implementation.

The first pass, where the hash value of every particle is calculated, is an inherently parallel task. The SPE threads process particles in batches from the workpool (see algorithm 2).

The building of the hash table is done solely on the PPE, in a serial manner. This algorithm is very similar to parts of the Radix sort algorithm, and there could be some parallelism to exploit, as shown by [46].

4.2.3 Calculating the interaction lists

Even though the building of the hash table is done serially on the PPE, the SPEs are not left idle during this task. When finding interacting particles the neighboring cells for each particle must be searched, which means the hash values for the neighboring cells must be calculated (so the corresponding hash buckets can be retrieved from the hash table).

These hash values are precalculated while the PPE builds the hash table using a slight modification of algorithm 2.

The interaction list is created by iterating over the hash buckets, calculating distances between particles.

Algorithm 3 Serially calculating the interaction lists

```
for all hash buckets do
  Check every particle in bucket against every other particle in bucket
  for every particle in bucket do
    for all 26 neighboring cells do
      Check particle against every particle in neighboring cells bucket
    end for
  end for
end for
```

Because of hash value collisions, it is possible that duplicate particle interactions are found. Therefore, in addition to checking distances between two particles, their grid cell relation must be tested.

The implementation uses workblocks of N buckets, which are processed in parallel on the SPEs, using the basic structure from algorithm 2. The algorithm uses additional layers of prefetching due to multi-layered data dependencies (figure 4.6).

And the algorithm is further extended by the fact that the SPH algorithm also contains a multi-layered dependency. The force summations for each particle depends on the pressure and density of all its neighbors, but the calculation of neighbor density/pressure values depends subsequently on *their* neighbors. This forces us to sum over the interaction lists in two passes.

However, right after the interaction list for a particle is calculated, all data required for density/pressure calculations for that particle is locally available. This allows us to

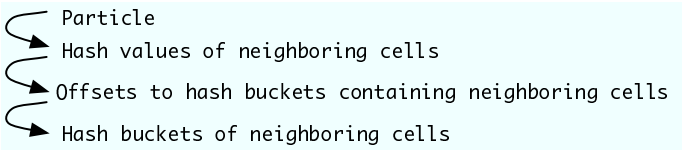


Figure 4.6: Data dependencies when searching for neighbors to a particle. The arrows show the sequence in which data must be fetched from main memory.

perform the first SPH calculation pass inside the neighbor search subtask.

The final algorithm for calculating interaction lists is presented below in algorithm 4.

Algorithm 4 Calculating interaction lists for one workblock

```
for all non-empty hash buckets in workblock do
  Prefetch neighbor cell hash values for first particle
  for all particles in hash bucket do
    Prefetch neighbor cell hash values for next particle
    Wait for current transfer to complete
    Use neighbor cell hash values to prefetch offset to neighbor hash buckets
    Check particle against all particles in same hash bucket
    Wait for offset transfer to complete

    Prefetch first neighbor hash bucket
    for first 25 neighboring cells do
      Prefetch next neighbor hash bucket
      Wait for current transfer to complete
      Check particle against all particles in neighboring hash bucket
    end for
    Wait for last hash bucket
    Check particle against all particles in neighboring hash bucket

    Calculate density and pressure for the particle
  end for
end for
```

4.3 SPH force calculations

The inter-particle forces are symmetrical, which can be exploited to decrease the number of arithmetic operations. It does however force more data to be transfered and creates dependancies between parallel threads. The implementation ignores the symmetrical property in order to maximize data locality and to allow asynchronous execution of the SPE threads.

The SPH densities and pressures are already calculated from the neighbor search task. Additionally, the squared distances between interacting particles are stored in the interaction lists. The algorithm used for parallel calculations of SPH forces in this implementation is pretty straight forward, and is described below.

Algorithm 5 Calculating SPH forces for one workblock

```
Prefetch particles for the first list
Calculate the distances for all interactions in the workblock
for all interaction lists in workblock do
    Prefetch particles for the next list
    Wait for the current transfer to complete
    Calculate SPH forces acting on particle
end for
Wait for the last transfer to complete
Calculate SPH forces acting on particle
```

4.4 Time Integration and Collision Handling

Time integration is done using the velocity Verlet method [18],

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{\mathbf{a}(t)+\mathbf{a}(t+\Delta t)}{2}\Delta t \tag{4.1}$$

$$\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + \frac{\mathbf{a}(t)}{2}\Delta t^2 \tag{4.2}$$

which, in addition to the normal state vectors (position, velocity, and acceleration), also requires the acceleration from the previous time step to calculate the new state.

The collision handling in the implementation only supports a bounding box which will contain the simulated fluid. Two passes of collision resolving are performed in the simulation loop. The first pass resolves collisions using a simple impulse method, reflecting the velocities of particles outside the box. After time integration, a second collision resolving pass projects the positions of particles still outside the box (only those who were also outside the box before the time integration).

Both time integration and collision handling are trivially parallelized over the SPEs, and have very little impact on the performance of the simulation.

4.5 Bandwidth Requirements

The memory transfer performance of a program is depending on many factors. For the Cell it is in practice possible to achieve about 15-20GB/s throughput [29] when taking full advantage of the MFCs. One important aspect to get good perfomance is to align source and target addresses on 128B boundaries, and use transfer sizes of multiples of 128B [23]. A rough theoretical analysis of the bandwidth requirements for the implementation is here presented.

N is the number of particles.
M is the number of buckets in the hash table.
G is the average number of neighbors per particle.

Hashing

$N * 4 * 4$	Read particle positions.
$N * 4$	Write resulting hash values.

$= 20N$ bytes

Building hash table

$N * 4$	Read hash values.
$M * 2$	Clear count table.
$N * 2$	Calculate count table.
$M * 2$	Read count table.
$M * 4$	Write offset table.
$N * 4$	Read hash values.
$N * 4$	Read offset for each hash value.
$N * 4 * 4$	Read particle positions.
$N * 4 * 4$	Write particle positions.

$= 46N + 8M$ bytes

Neighbor hashing

$N * 4 * 4$	Read particle positions.
$N * 26 * 4$	Write resulting neighbor hash values.

$= 120N$ bytes

Neighbor searching

$N(N/M * 4 + 26 * N/M * 4)$	Read position of every particle in same and neighboring hash bucket.
$N * G * 4 * 2$	Write interaction lists.
$N(4 + 26 * 4)$	Read offsets to same and neighboring hash buckets.
$N(4 + 26 * 4)$	Read particle and neighbor hash values.

$= 108N^2/M + 8NG + 216N$ bytes

SPH calculations

$N * 4 * 2$	Write pressures and densities.
$N * 4 * 2$	Read pressures and densities.
$8 * N * G$	Read interaction lists.
$N(G(4*4+4*4)+(4*4+4*4))$	Read velocities and positions of particle and neighbors.
$N * 4 * 4$	Write resulting forces.

$= 64N + 40NG$ bytes

Collision handling - pass one

$N * 4 * 4$	Read particle positions.
$N * 4 * 4$	Read particle velocities.
$N * 4 * 4$	Write particle positions.
$N * 4 * 4$	Write particle velocities.

$= 64N$ bytes

Time integration

$N * 4 * 4$	Read particle positions.
$N * 4 * 4$	Read particle velocities.
$N * 4 * 4$	Read particle accelerations.
$N * 4 * 4$	Read previous particle accelerations.
$N * 4 * 4$	Write particle positions.
$N * 4 * 4$	Write particle velocities.
$N * 4 * 4$	Write particle accelerations.

$= 112N$ bytes

Collision handling - pass two

$N * 4 * 4$	Read particle positions.
$N * 4 * 4$	Write particle positions.

$= 32N$ bytes

TOTAL: $674N + 8M + 108N^2/M + 48NG$ bytes/timestep

Of these are $452N$ bytes transfered with 128B alignment and 128B or larger transfer sizes.

As an example, we assume a particle system with 50000 particles and a hash table with

75000 buckets, and each particle has 30 neighbors on average, and we want to run the simulation at 50Hz. The required bandwidth should be roughly 5GB/s (of which 20% are 128B sized/aligned).

4.6 Performance vs. Stability

Both performance and stability of the simulation are depending on the accuracy of the calculations. Typically, numerical precision is crucial when simulating interacting particles, since the force sums involves small additions to a large number. However, the compact support of the SPH kernel limits the number of neighbors to 15-60 per particle and therefore numerical precision is not as important as in the general N^2 case. The implementation uses single-precision (32bit) arithmetic for floating point calculations, which on the Cell gives significantly (roughly 10x) better performance [15], compared to double-precision (64bit) calculations.

The SPEs does not provide hardware support for floating point division or square root operations. The software library functions availible [24] provides two different versions. One is fully IEEE compliant, but rather expensive. The other is a fast, but less accurate implementation.

Also, as mentioned in the neighbor search section, the grid cell relation of neighboring particles must be verified when calculating interaction lists in order to prevent duplicate interactions. The implementation allows this test to be disabled, which can have a positive impact on performance, which is shown in section 5.1.

Chapter 5

Results and Discussion

This chapter analyses the performance of the implementation presented in chapter 4. The simulations are contained inside a static box, with particles given random initial positions. System sizes between 1000 and 35000 particles are used in the tests. To keep the average number of neighbors (roughly) equal for the different system sizes, the size of the box is adjusted accordingly. The simulations run 3000 time steps (using $\Delta t = 0.003s$), which allows the system to stabilize, as shown in figure 5.1. Two different SPH smoothing lengths are used; one yielding approximately 15 average neighbors per particle, and the other giving roughly 25 average neighbors per particle.

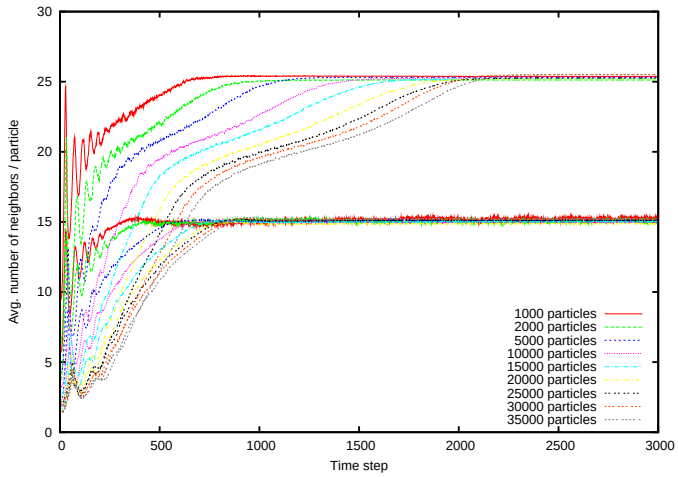


Figure 5.1: The different configurations used for testing and how they stabilize from an initial randomized state.

5.1 Scalar implementation

A scalar implementation is used as a reference, against which the performance of the Cell implementation is compared. Both implementations use the same basic data structures and algorithms. The scalar tests are performed on an Apple PowerMac with dual 2.0GHz IBM PPC 970 (*aka* G5) processors, but only one is used by the simulation. The program is compiled using gcc 4.0.0 using the `-fast` flag, which optimizes for the G5 architecture. The scalar implementation has no manual optimizations and using `-fast` gives a 3x to 4.5x speedup compared to unoptimized (`-O0`) code. Although capable of it, gcc does not indicate any automatic vectorization being done. So the speedup must come from other optimization techniques such as loop unrolling, automatic inlining, and processor model specific instruction scheduling.

The performance for different system sizes is presented in table 5.1. Using 10000 particles with 15 avg. neighbors / particle the simulation runs at 24Hz. Given a large enough time step can be used, this is somewhere near the limit for what could be called a real-time simulation.

System size	**Update frequency [Hz]** 15 neighbors / particle	**Update frequency [Hz]** 25 neighbors /particle
1000	312.5	242.4
2000	146.2	118.0
5000	54.2	43.7
10000	24.1	18.9
15000	14.0	11.5
20000	9.6	7.8
25000	7.0	5.9
30000	5.5	4.7
35000	4.5	3.9

Table 5.1: Performance of the scalar implementation on a 2.0GHz PPC 970 processor.

The cost of the different subtasks is show in figure 5.2. Time integration and collision detection against the box come at practically zero cost and are excluded from the figure. Perhaps somewhat surprising is that the SPH force calculation subtask use less than 40% of the clock cycles, and it is actually the neighbor searching that is most expensive part. The Cell implementation parallelizes all tasks except the building of the hash table, which fortunately is quite inexpensive. However, as shown by Amdahl's law [20], even a small serial dependency can heavily limit the scalability of a parallel program. See section 5.2 for further analysis of the Cell implementation.

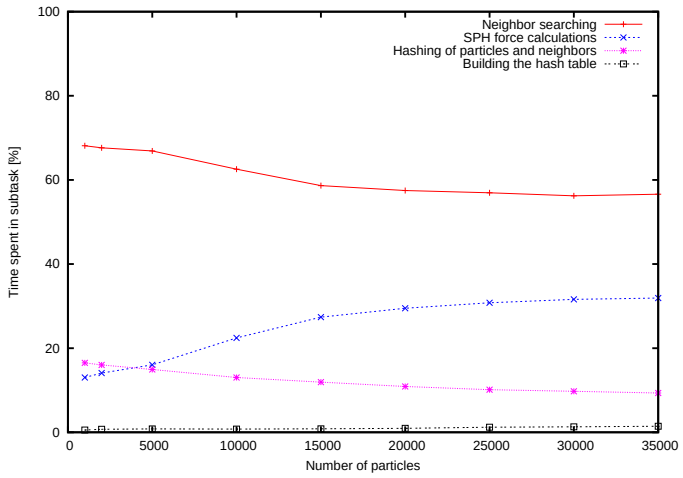


Figure 5.2: Profiling of subtasks using different system sizes, with 25 avg. neighbors / particle

Figure 5.3 shows the average time spent on each particle each time step. For the performance to scale linearly with the number of particles, the curves should have a constant y-value, which they don't. The reason for this could be increasingly large cache miss rates for larger problems, or simply that the algorithm is of higher order complexity, such as $O(n \log n)$ or $O(n^2)$. See section 5.2 for further analysis of asymptotical complexity.

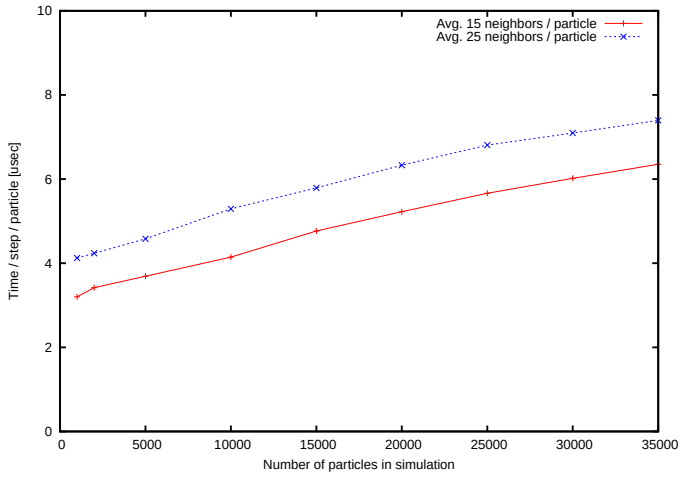


Figure 5.3: Cost per particle for different system sizes.

As mentioned in section 4.2.3 the possibility of hash value collisions require an extra check during neighbor search to guarantee that no duplicate interactions are found. The number of duplicate interactions is however relatively low (typically below 1%), as seen in table 5.2, and may be acceptable in some simulations. Removing this check speeds up the neighbor search, and since that is the most expensive subtask it will also give a noticeable difference in overall performance. Depending on system size and neighbors/particle a 20%-50% increase in overall performance can typically be seen.

Hash table size:	24001 (1.6x number of particles)
Number of empty hash buckets:	89% of all buckets
Hash buckets with collisions:	5% of non-empty buckets
Avg. number of particles in each non-empty hash bucket:	5.94
Redundant neighbor tests due to hash collisions:	13% of all tests
Duplicate interactions due to hash collisions:	0.2% of all interactions

Table 5.2: Hash table statistics for a simulation with 15000 particles and 25 avg. neighbors / particle.

5.2 Cell performance

As we were unable to get access to Cell hardware, the Cell performance analysis is done using test results from the Cell simulator [6]. The simulator does have an accurate performance model for SPE computations but does not make any attempt to accurately model DMA memory transfers. This is very important and means that, even though the implementation extensively uses double buffering to hide memory transfer latencies, there could be additional costs that does not show up in the simulator.

Table 5.3 shows performance results using different system sizes. Even though the Cell performance numbers are not entirely reliable, a single SPE seems to provide roughly the same performance as the PPC 970.

System size	**Update freq. [Hz]** PPC 970	**Update freq. [Hz]** Cell (1 SPE)	**Update freq. [Hz]** Cell (8 SPEs)
1000	242.4	168.5	545.9
2000	118.0	85.2	519.2
5000	43.7	34.6	252.1
10000	18.9	17.2	132.7
15000	11.5	11.4	87.2
20000	7.8	8.5	66.3
25000	5.9	6.9	53.2
30000	4.7	5.7	44.5
35000	3.9	4.9	38.4

Table 5.3: Performance using avg. 25 neighbors / particle.

Figure 5.4 shows the speedup of one SPE compared to the PPC 970. The PPC 970 is faster using smaller systems, probably due to good cache efficiency, but as the system grows the cache efficiency decreases, giving the SPE the advantage.

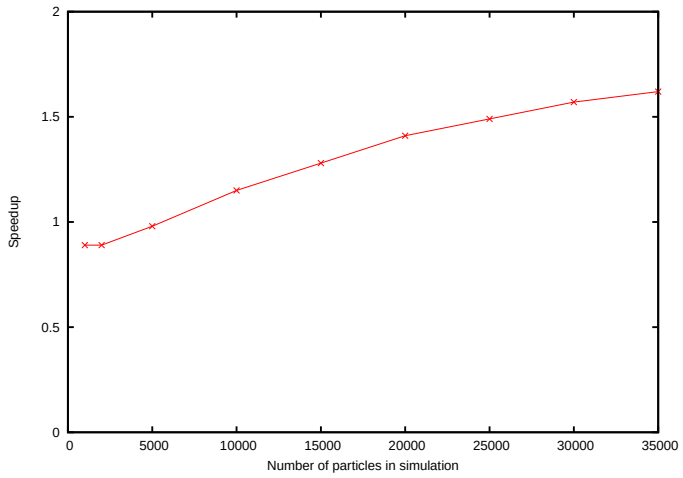


Figure 5.4: Speedup of one SPE compared to to the PPC 970.

The fact that the Cell implementation does not include a software cache for the SPEs enables us to determine the theoretical complexity of the simulation. In figure 5.5 the cost per particle is plotted for both the SPE and the PPC 970. We can see that, for the SPE, the cost per particle is constant with increasing system sizes, which would indicate $O(n)$ complexity for the simulation (when keeping the number of neighbors per particle constant). Note that the analysis is valid even though the simulator does not model memory transfer performance, since the asymptotical complexity is based purely on computational cost, which the simulator *does* accurately measure.

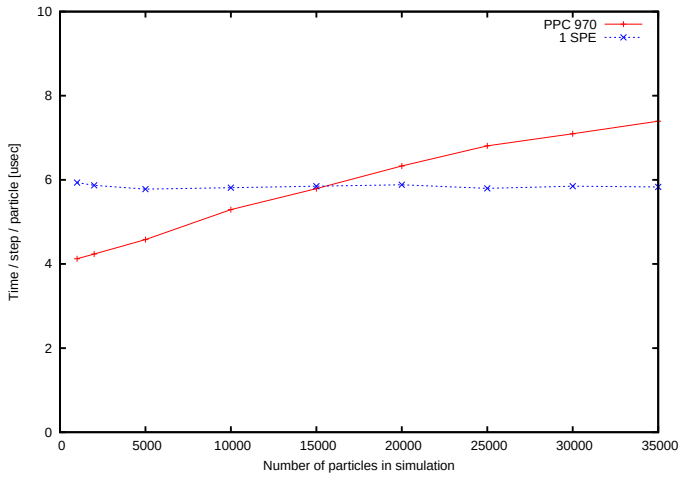


Figure 5.5: Cost per particle for different system sizes.

Profiling of the different subtasks when running on one SPE is shown in figure 5.6. We can see that the cost of the SPH force calculations is even less than for the scalar version.

And since the hashing subtasks are actually part of the neighbor searching algorithm (together 80% of total cost), it is clear that any further optimization attempts should focus mainly on improving the neighbor searching method.

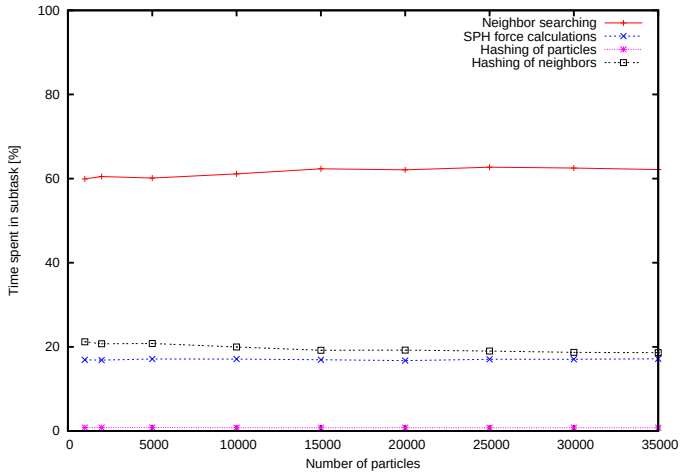


Figure 5.6: Profiling of subtasks using one SPE, with 25 avg. neighbors / particle.

There are two potential bottlenecks hindering the parallel scalability of the implementation; the cost of the serial part of the program, and saturation of system bandwidth. These are hard to determine using the simulator because of the non-complete performance modeling. Additionally, while the simulator can only simulate a single Cell processor, a real Cell blade server with dual Cell processors would allow us to test scalability using up to 16 SPEs. The scalability results from the simulator are still presented here for the sake of completeness. The parallel efficiency is defined as

$$E(p) = \frac{S(p)}{p} \tag{5.1}$$

where S is the speedup and p is the number of processing elements. An efficiency graph using the maximum number of SPEs available in the simulator is shown in figure 5.7. We get near perfect scalability for all medium and large sized systems.

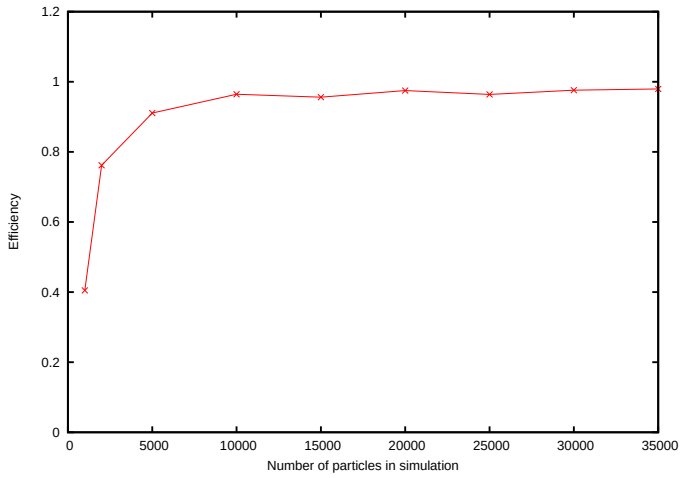


Figure 5.7: Parallel efficiency for different system sizes when using 8 SPEs.

Finally, a somewhat surprising result is that disabling the "duplicate interactions check", which for the serial implementation gave a substantial performance gain, only increase performance by approximately 7% on the Cell.

Chapter 6

Conclusions

The Cell is a brand new platform, and due to the difficulty of getting real hardware, it was fortunate that IBM released their Full System Simulator early in the project. However, the slowness of the simulator made development frustrating at times, especially since the Cell is a rather complex platform. The asynchronous data transfers and the parallel nature of the Cell makes programming non-trivial, and yielded some rather challenging bugs.

6.1 Limitations

Since we were unable to get hold of Cell hardware in time for experimental performance testing, the results in section 5.2 must not be taken as absolute performance numbers.

6.2 Future work

We will hopefully get access to real Cell hardware soon, on which a thorough analysis of the implementation will be made. Mainly to get more reliable performance data, but it will also allow us to further test the scalability since the Cell blade servers have dual Cell processors (the simulator can only simulate a single Cell). The scalable design of the implementation means we can spawn as many threads as there are SPEs available on the system, but with two possible bottlenecks. The serial portion (filling the hash table) will at some point become too expensive to maintain good parallel efficiency. The other bottleneck is memory bandwidth, which given enough SPE threads, will be saturated. Finding those thresholds would certainly give valuable information.

It would also be interesting to see what kind of gains IBM's new compiler technology [17] could give. Currently we have only tried gcc as a compiler.

For further development of the implementation, it would be interesting add another level of parallelization to enable the system to run as a distributed simulation (e.g. on a rack of interconnected Cell blade servers).

Another possible large gain in performance is by implementing some sort of dynamic data reordering together with a software cache for the SPEs. Eichenberger *et al* [17] has shown that a 4-way associative software cache can be implemented with a mere 12 instruction cache hit latency. For data reordering it would be possible to use the PPE which is mostly idle in the current implementation. There are a number of different data clustering methods, such as the Hilbert space-filling curve [34], and Morton ordering (*aka* z-ordering) [19]. Initial tests on a 2GHz G5 show that 100000 keys can be sorted in less than 5 msec using radixsort, which would indicate potential for dynamic data reordering.

High cache hit rate would not only mean higher performance because of lower latency, but would also lower the bandwidth requirements which could enable the system to scale even further.

It would also be interesting to use Cell to implement a new, not yet finalized, method for simulating incompressible fluids as well as viscoelastoplastic materials [14]. It uses regularized constraint methods in combination with an iterative solver, that together ensure both the physical conservation laws and stability.

6.3 Acknowledgements

I would like to thank my supervisor, Kenneth Bodin at VRLab, for help and feedback throughout the project.

Additionally, I am grateful for the quick and informative responses received from the IBM Cell architecture forum [2] during development of the implementation.

References

[1] AMD Athlon XP Processor Tech Docs. http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_3748,00.html (visited 2006-04-14).

[2] Cell architecture forum. http://www-128.ibm.com/developerworks/forums/dw_forum.jsp?forum=739&cat=46 (visited 2006-06-13).

[3] Cell Broadband Engine resource center. http://www-128.ibm.com/developerworks/power/cell/ (visited 2006-04-12).

[4] Dual Cell-Based Blade. http://www.mc.com/literature/literature_files/Cell_blade-ds.pdf (visited 2006-03-17).

[5] IBM Cell Broadband Engine Software Sample and Library Source Code. http://www.alphaworks.ibm.com/tech/cellsw?open&S_TACT=105AGX16&S_CMP=DWPA (visited 2006-06-14).

[6] IBM Full-System Simulator for the Cell Broadband Engine Processor. http://www.alphaworks.ibm.com/tech/cellsystemsim (visited 2006-03-17).

[7] Intel Pentium 4 Processor Family. http://www.intel.com/design/Pentium4/documentation.htm (visited 2006-04-14).

[8] PhysX-Optimized Titles. http://physx.ageia.com/titles.html (visited 2006-04-12).

[9] PowerPC 970 and 970FX Microprocessors. http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_970_and_970FX_Microprocessors (visited 2006-04-14).

[10] Sony Computer Entertainment Inc. to launch its next generation computer entertainment system, Playstation 3 in spring 2006. http://www.scei.co.jp/corporate/release/pdf/050517e.pdf.

[11] Ageia. *A White Paper: Physics, Gameplay and the Physics Processing Unit*, 2005.

[12] T. Amada, M. Imura, Y. Yasumuro, Y. Manabe, and K. Chihara. Particle-Based Fluid Simulation on GPU. In *ACM Workshop on General-Purpose Computing on Graphics Processors*, 2004.

[13] A-H A. Badawy, A. Aggarwal, D. Yeung, and C-W. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimization. In *Proceedings of the 15th international conference on Supercomputing*, 2001.

[14] K. Bodin, C. Lacoursiére, and K. Wiklund. Personal communication.

[15] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation. http://www-128.ibm.com/developerworks/power/library/pa-cellperf/ (visited 2006-03-17).

[16] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, 1999.

[17] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, , and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine (TM) architecture. *IBM System Journal*, 45, 2006.

[18] F. Ercolessi. The Verlet algorithm. http://www.fisica.uniud.it/∼ercolessi/md/md/node21.html (visited 2006-03-17).

[19] C. Ericson. *Real Time Collision Detection*. Morgan Kaufmann, 2005.

[20] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing, Second Edition*. Addison Wesley Pearson, 2003.

[21] S. Hadap and N. Magnenat-Thalmann. Modeling dynamic hair as continuum. In *Eurographics Proceedings, Computer Graphics Forum*, 2001.

[22] IBM. *Cell Broadband Engine Architecture*. http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA.

[23] IBM. *Cell Broadband Engine Programming Tutorial V1.0*. http://www.alphaworks.ibm.com/tech/cellsw (visited 2006-03-17).

[24] IBM. *Cell Broadband Engine SDK Libraries: Overview and Users Guide V1.0*. http://www.alphaworks.ibm.com/tech/cellsw (visited 2006-03-17).

[25] IBM. *SPU C/C++ Language Extensions V2.1*. http://www-128.ibm.com/developerworks/power/cell/downloads_doc.html (visited 2006-03-17).

[26] Intel. *IA-32 Intel Architecture Optimization Reference Manual*, 2005.

[27] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49, 2005.

[28] P. Kipfer, M. Segal, and R. Westermann. UberFlow: a GPU-based particle engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004.

[29] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 2006.

[30] A. Kolb and N. Cuntz. Dynamic Particle Coupling for GPU-based Fluid Simulation. *TODO*, 2005.

[31] D. Krolak. Unleashing the Cell Broadband Engine Processor. In *Papers from the Fall Processor Forum 2005*, 2005.

[32] L. Latta. Building a Million Particle System. In *Game Developers Conference, 2004*, 2004.

[33] J.J. Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68, 2005.

[34] B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of Hilbert space-filling curve. *IEEE Transactions on knowledge and data engineering*, 13, 2001.

[35] M. Müller, D. Charypar, and M. Gross. Particle-Based Fluid Simulation for Interactive Applications.

[36] M. Ohara, H. Inoue, Y. Sohda abd H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 45, 2006.

[37] M. Pharr and R. Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[38] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2003.

[39] R. Shrout. AGEIA PhysX Physics Processing Unit Preview. http://www.pcper.com/article.php?aid=140 (visited 2006-04-12).

[40] M. Taiji. MDGRAPE-3 chip: A 165-Gflops application-specific LSI for Molecular Dynamics Simulations. In *Proceedings of Hot Chips 16*, 2004.

[41] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, and A. Konagaya. Protein Explorer: A Petaflops Special-Purpose Computer System for Molecular Dynamics Simulations. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[42] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, and M. Gross. Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *Proceedings of Vision, Modeling, Visualization VMV'03*, 2003.

[43] J. Waltz, G.L. Page, S.D. Milder, J. Wallin, and A. Antunes. A Performance Comparison of Tree Data Structures for N-Body Simulation. *Journal of Computational Physics*, 178, 2002.

[44] M. S. Warren and J. K. Salon. A parallel hashed Oct-Tree N-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, 1993.

[45] M.A. Weiss. *Data Structures & Algorithm Analysis in Java*. Addison Wesley Longman, 1999.

[46] M. Zagha and G.E. Blelloch. Radix Sort For Vector Multiprocessors.