

# **The Fast Multipole Algorithm on the Cell Broadband Engine Architecture**

Christopher Fraser (u4395190)  
Supervisor: Dr. Eric McCreath  
Course: COMP3006 (6 units)

29th October 2009

## Abstract

Computing the interactions between  $N$  particles due to electrostatic or gravitational forces is a well known problem in scientific computing. Known as the  $N$ -body problem, it is a well known setback in scientific computing. Typical applications of the simulation, such as simulating star clusters, require large values of  $N$ , often in the range of many millions. The standard simulation technique of computing all pair-wise interactions requires  $O(N^2)$  calculations per time step, and does not parallelise well. This is quite inadequate most use cases, therefore there is a number of more efficient algorithms have been developed. These algorithms apply techniques to both reduce the order of required computations, and often introduce data parallelisation capability. One such algorithm is Greengard and Rohklin's Fast Multipole Algorithm (FMA), which computes potentials for groups of particles via multipole approximations. In this way, the number of computations required to compute the potentials for  $N$  particles is only  $O(N)$ , additionally allowing for parallelisation via data partitioning. An additional method of speeding up the simulation is by exploiting platform hardware features. The Cell Broadband Engine (CBE) is a specialised microprocessor architecture which has a number of properties and features which promise potential speedups to the FMA technique. This includes a number of SPE's for executing the simulation in parallel, SIMD operations, and a high memory bandwidth. This paper investigates the case for the use of the Fast Multipole Algorithm designed for the Cell architecture, specifically the Sony Playstation 3 hardware.

Though the original goal was to compare the performance off a Cell-optimised Fast Multipole algorithm to that of an unoptimised algorithm which could execute on any architecture, programming the FMA for Cell turned out to be more difficult than anticipated, and as of November 2009 the implementation cannot complete in order to gather performance results.

However, persisting to investigate the potential of the PS3/Cell as a platform for solving  $N$ -body problems, the direct pairwise algorithm for solving systems was timed on an Intel Core2Duo 2.6GHz compared to a PS3 executing the algorithm on its PPE, compared to a PS3 executing on the SPE. The PS3's PPE proved to be over 10 times slower than the Core2 in all tests, while the PS3 SPE was over 650 times slower for a test involving 25 particles.

Better performance may be demonstrated by the PS3/Cell with a completed, targeted and optimised algorithm, however it appears to be difficult even to achieve just the performance of a desktop machine without much work.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Algorithms for solving N-body systems</b>	<b>4</b>
2.1	The Basic Algorithm . . . . .	4
2.2	Hierarchical Tree Algorithms . . . . .	5
2.2.1	Advantages . . . . .	5
2.2.2	The Barnes-Hut Algorithm . . . . .	6
2.2.3	The Fast Multipole Algorithm . . . . .	6
2.2.3.1	Interaction lists . . . . .	7
2.2.3.2	The Algorithm . . . . .	7
<b>3</b>	<b>The Cell Broadband Engine Architecture and Playstation 3</b>	<b>10</b>
<b>4</b>	<b>Algorithm Implementation</b>	<b>12</b>
<b>5</b>	<b>Results</b>	<b>13</b>
<b>6</b>	<b>Reflection, Future Work</b>	<b>15</b>

# Chapter 1

## Introduction

The N-body problem is the problem of describing the interactions between a system of  $N$  particles given their initial state, in order to be able to determine their state at any given point in time. Mathematically, it may be described as an initial value problem involving ordinary differential equations. That is:

Given initial values for the positions  $q_j(0)$  and velocities  $\dot{q}_j(0)$  of  $n$  particles ( $j = 1 \dots n$ ) with  $q_j(0) \neq q_k(0)$  for all distinct  $j$  and  $k$ , find the solution of the second order system:

$$m_j \ddot{q}_j = \gamma \sum \frac{m_j m_k (q_k - q_j)}{|q_k - q_j|^3}, \quad j = 1 \dots n$$

Many physics problems, including those in the field of electro-magnetics and astronomy, require such a system to be described. However, resolving an N-body system is complicated by the fact that for  $N \geq 3$ , no analytical method is known to provide an exact solution. Instead, it must be solved numerically, by simulating the interactions of the particles over discrete time steps. This leads to the unwanted effects of a loss of accuracy over the course of the simulation, and requiring a large number of intermediate calculations. The simplest, most direct method for simulating an N-body system is to treat the system as a set of pairwise particle-particle interactions per time step. This therefore requires  $O(N^2)$  time in order to simulate, which is generally considered less than ideal. Many alternate simulation methods have thus been developed in an effort to achieve more efficient use of available computing resources.

The most successful algorithms have used divide-and-conquer hierarchical techniques, such as the Barnes-Hut Algorithm from [4]. An overview of these algorithms is presented in (2). The Fast Multipole Algorithm (FMA), described in [3] is one of the more optimal of these techniques. The premise of this algorithm is to separate the system clusters of nearby particles. Particles which are considered as close clusters are simply computed directly in the pairwise fashion. Other particles are considered distant enough to approximate, and are therefore calculated via multipole expansions. The advantage to using multipole expansions is that they may be truncated, with little loss of accuracy. Therefore the Fast Multipole Algorithm promises a running time of  $O(N)$  in most cases, at worst only  $O(N \log N)$ . This algorithm thus appears to provide a worthy basis for a fast particle simulator, thus I have chosen it for my solver implementation. This algorithm is detailed in (2.2.3).

Crucial to the performance of any algorithm is the platform on which it is designed for. The Cell Broadband Engine (CellBE) is a microprocessor architecture designed jointly by Sony, Toshiba and IBM. It has been designed with an emphasis on parallel execution, high bandwidth, and computational throughput. It is therefore well matched for scientific computing applications such as the FMA. The basic configuration of a Cell platform is that of a

multi-core chip consisting of a single Power Processing Element (PPE) and multiple Synergistic Processing Elements (SPE). The Sony Playstation 3 (PS3) is the first and most successful commercial implementation of the Cell architecture, due to its positioning as an affordable gaming machine. The PS3 version of the CellBE contains a dual threaded PowerPC processor clocked at 3.2GHz as the PPE coupled with 8 SPEs, 6 of which are available to the developer.

I have set out to implement the Fast Multiple Algorithm optimised for the Cell Broadband Engine, the PS3 implementation in particular. I then went on to explore the performance potential of this algorithm executing on the PS3, comparing its performance to that of an unoptimised basic pairwise algorithm, both when executed on an x86 machine and on the PS3's PPU.

## Chapter 2

# Algorithms for solving N-body systems

### 2.1 The Basic Algorithm

The intuitive algorithm 2.1, which solves an N-body system by computing all interactions directly, while simple to implement, is less than ideal for a number of reasons:

- The nested looping over  $N$  particles causes the algorithm to require  $N^2$  computations.
- It is possible to parallelise the algorithm over  $p$  processors in two ways, however neither is overly efficient:
  - Data partitioning is possible in that each processor is responsible for  $N/p$  particles. However each must then broadcast all of its  $N/p$  particles to the  $p - 1$  other processors after each timestep.
  - Space partitioning is also possible, in that each processor is responsible for a volume within the simulation. However, particles may converge into a smaller number of spaces, leaving processors idle.

Due to these inadequacies, there have been several efforts to implement N-body algorithms which circumvent the aforementioned limitations. Following this, the algorithms which use hierarchical tree-based methods have proven quite successful.

---

**Algorithm 2.1** The direct algorithm for solving an N-body system.

---

```
for dt in timesteps:
    potentials = [0.0]
    for p1 in particles:
        potentials[p1] = 0
        for p2 in particles, p2 != p1:
            potentials[p1] += potential(p1, p2)
    for p in particles:
        position[p] += move_from(potentials[p])
```

---

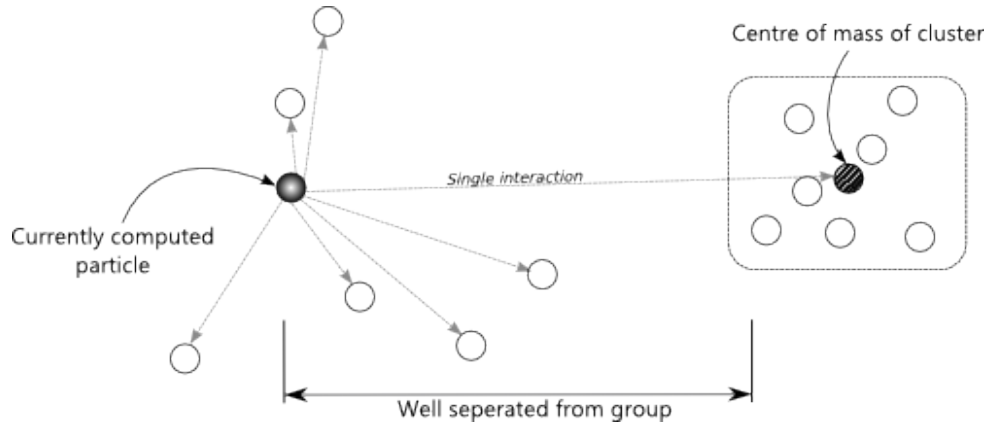


Figure 2.1: Graphical representation of clustering a group of particles which are well separated from the current particle. The clustered particles are no longer considered individually, reducing pairwise computations.

## 2.2 Hierarchical Tree Algorithms

Hierarchical Tree Algorithms are based around the technique of recursively dividing the particle space into sub-trees. Algorithms which use hierarchical tree structures include the Barnes-Hut algorithm of [4] and the more complex Fast Multipole Algorithm of [3], which I have chosen to implement.

### 2.2.1 Advantages

There are two primary speedups which can be obtained from the structure of these algorithms. The first is that they are highly parallelisable, due to their divide-and-conquer nature. The second is that due to their design of partitioning the space into cells, they are intuitively able to detect when a particle is *well separated* from cells of other particles, and thus compute an accurate approximation of potential exerted on the particle in much fewer operations.

**Definition 1.** A cell/group of particles  $c$  is well separated from a particle  $p$  if  $c$  exerts a decidedly minute potential on  $p$ .

The potential that  $c$  will exert on  $p$  is most influenced by two parameters - the size  $r$  of  $c$ , which bounds the proximity of particles within, as well as the distance  $z$  of the  $c$ 's centre of mass from  $p$ . While the function which determines if an instance of  $c$  is well separated from a particle depends on the algorithm and implementation, it will be dependant on  $r$  and  $z$ . If an instance of  $g$  satisfies 1, it is ensured that the particles in  $g$  will be sufficiently close, and the aggregate sum of their potentials  $\phi_c$  will be sufficiently small such that  $\phi_c$  may be calculated via approximation with a minimal degree of error. If a particle is known to be well separated from  $x$  cells ( $cell = 1 \dots c$ ) then the potential  $\phi$  exerted on each particle can then be broken down into two components:

$$\phi = \phi_{near} + \phi_{far}$$

- $\phi_{near}$  is a rapidly decaying potential due to close-range forces:

$$\phi_{near} = \sum \phi_y, y = 1 \dots n, y \text{ not in } \phi_{far}$$

Computing  $\phi_{near}$  is an expensive operation and is to be avoided.

- $\phi_{far}$  is the potential exerted by well separated cells:

$$\phi_{far} = \sum \phi_{cell}, \text{ cell} = 1 \dots c$$

These algorithms may approximate  $\phi_{far}$  by computing the potential due to well separated groups via *clustering*. Clustering is an approximation technique in which a group of particles are treated as a single virtual particle  $v$ , with mass equal to the total of the clustered particles, centered at the centre-point of the cluster. If a cell  $c$  consists of  $x$  particles ( $d = 1 \dots x$ ), then rather than fully calculating  $\phi_{far}$  for a particle as

$$\phi_{far} = \sum \phi_d \text{ } d = 1 \dots x$$

it may be approximated down to

$$\phi_{far} = \phi_v$$

As Figure 2.1 on page 5 demonstrates, many computations are saved in this way, abolishing the  $O(N^2)$  performance upper bound. While this comes at the expense of some amount of approximation error which is dependant on the degree of separation. As clustering is only applied to groups of particles which are considered well separated from the particle in question, this error is kept to a minimal.

Thus we see that via the ability to detect well separated groups of particles, tree-based N-body solving algorithms can vastly improve upon performance, at minimal expense of accuracy.

## 2.2.2 The Barnes-Hut Algorithm

This algorithm is based on an octree representation of three-dimensional space. It initially defines a cell structure, where each cell stores the total mass and centre of mass of all particles contained within. It then defines a root cell containing all particles in the system. It is then split into two phases. The first recursively divides cells into eight cubical subcells of equal size, until each cell has at most a single particle. The second phase traverses the tree once per particle to compute the net force acting upon it. Traversal begins at the root, and for each cell traversed, if it is considered well separated from the particle, the cell is approximated via the aforementioned clustering technique. This algorithm defines a cell well separated from a particle if it satisfies 2.

**Proposition 2.**  $\frac{r}{z} < \theta$  | *implementation defined*

This algorithm thus requires  $\log_8 N$  computations to traverse the octree and compute interactions for a particle, and is traversed all of  $N$  times. Therefore this algorithm requires only a very respectable  $N \log N$  computations for a simulation step of an N-body system.

Many variants of this algorithm have been developed, such as an implementation that allows for greater code vectorization, though at the expense of more floating point operations[6].

## 2.2.3 The Fast Multipole Algorithm

The Fast Multipole Algorithm of Green was developed 'for the rapid evaluation of the potential and force fields in systems involving large numbers of particles whose interactions are Coulombic or gravitational in nature'[3]. The algorithm requires a maximum level of refinement and a precision bound as parameters. The premise of acceleration in this algorithm is to always approximate particles' potentials via the largest well separated cells as possible, and in it's use of multipoles in potential calculations. These multipoles can be translated and truncated efficiently in order to approximate potentials to within the defined error bounds.

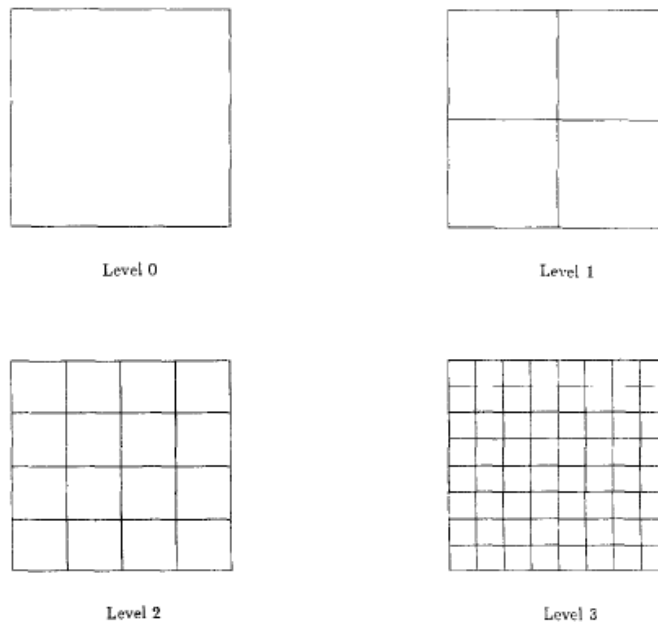


Figure 2.2: The first four levels of refinement within the FMA[3]

### 2.2.3.1 Interaction lists

The algorithm uses a structure referred to as an interaction list in order to compute the minimal number of interactions between cells as necessary. [3]page 336 defines an interaction list as

**Interaction list** for box  $i$  at level  $l$ , it is the set of boxes which are children of the nearest neighbors of  $i$ 's parent and which are well separated from box  $i$ . (box: cell)

### 2.2.3.2 The Algorithm

The FMA, like the Barnes-Hut algorithm, begins with a root cell containing all particles and recursively subdivides cells, using a quadtree rather than an octree however. The FMA considers well separation between pairs of cells on equivalent subdivision levels, being defined as well separated if they simply are non-adjacent.

Similarly to the Barnes Hut algorithm, the FMA consists of two phases: The Upwards Pass and the Downwards Pass. The Upwards Pass works from the finest level up to create the initial multipole expansions which are required for continuous evaluation in the downwards pass. The purpose of the Downwards Pass is to compute interactions at the most coarse level possible. That is, it begins at the coarsest level of refinement (level 0, consisting of only the root cell) and for each cell in the level, the pass will only compute interactions for cells which are well separated, and have not already been accounted for in the parent level. The levels are computed from top to bottom in this fashion.

As is evident from Figure 2.4 on page 9, all steps within the algorithm operate in linear time with  $N$ . This makes the Fast Multipole algorithm one of the most optimal  $N$ -body solvers. Therefore it was chosen as the algorithm for this project.

		x	x	x	x	x	x
		x				x	x
		x		i		x	x
		x				x	x
		x	x	x	x	x	x
		x	x	x	x	x	x

Figure 2.3: The interaction list for cell  $i$ , consisting of all cells which are children of cells adjacent to  $i$ 's parent cell, but not adjacent to  $i$ . [3]

---

**Algorithm 2.2** The Fast Multipole algorithm

---

**Upwards Pass**

1. The quadtree is built, and multipole expansions are calculated for the particles in each cell at the highest subdivision level.
2. Multipole expansions are formed for each higher mesh level by shifting the multipoles up from the child cells.

**Downwards Pass**

3. For each level from top to bottom, form a local expansion about the center of each cell in the level from all cells non-adjacent to the cell.
  4. For each cell in the finest mesh level, form a local expansion by summing together the expansions of each cell in it's interaction list.
  5. Evaluate the multipole for each particle in every cell in the finest mesh level to receive  $\phi_{far}$ .
  6. For each particle in each cell in the finest mesh level, compute direct interactions with other particles in this cell and adjacent cells to receive  $\phi_{near}$ .
  7. Add together  $\phi_{near}$  and  $\phi_{far}$ .
-

Step	Operation count	Explanation
1	order $Np$	Each particle contributes to one expansion at the finest level.
2	order $Np^2$	At the $l$ th level, $4^l$ shifts involving order $p^2$ work per shift must be performed.
3	order $\leq 28Np^2$	There are at most 27 entries in the interaction list for each box at each level. An extra order $Np^2$ work is required for the second loop.
4	order $\leq 27Np^2$	Again, there are at most 27 entries in the interaction list for each box and $\approx N$ boxes.
5	order $\leq 27Np^2$	One $p$ -term expansion is evaluated for each particle.
6	order $\frac{9}{2}Nk_n$	Let $k_n$ be a bound on the number of particles per box at the finest mesh level. Interactions must be computed within the box and its eight nearest neighbors, but using Newton's third law, we need only compute half of the pairwise interactions.
7	order $N$	Adding two terms for each particle.

Figure 2.4: Computational complexity of each step in the Fast Multipole Algorithm[3]

## Chapter 3

# The Cell Broadband Engine Architecture and Playstation 3

The Cell Broadband Engine, developed by the STI alliance consisting of Sony CE, Toshiba and IBM, is a multimedia-targeted microprocessor architecture. As described in Chapter 1, it offers a number of unique architectural features which can assist in developing a high-performance parallel floating point algorithm. It has been designed to break down the three walls which have limited processor development recently[2]:

- The "Frequency Wall" - Increasing clock frequency typically requires a deeper pipelining, making it increasingly more difficult to maintain an efficient pipeline while keeping stalls to a minimum.
- The "Power Wall". This is a problem which has reared its ugly head in the field of computing lately. This is the fact that power consumption of a processor increases at a faster rate than the frequency increases. As CPUs reach high clock frequencies of 3GHz, CPU manufacturers have found it increasingly difficult to deal with the greatly increased power drain, heat production, as well as Passiver Power (leakage).
- The "Memory Wall". This is the problem which has arisen lately that memory technology has not developed at the same rate as CPU technology, creating a large bottleneck as modern CPU's are limited by memory performance. The move to multi-core chips further exacerbated this problem due to the use of shared channels into the memory between cores.

The Cell tackles all three issues with a heterogenous multi-core architecture, consisting of processing cores which are specialised for their designated tasks, allowing for simpler chips which make more efficient use of their available transistors. The processors themselves have been designed to be simple in hardware, with typical CPU features such as hardware branch prediction replaced by compiler assistance, in favour of more efficient chips.

The Cell is a highly complex architecture, but has been thoroughly explored and described [1, 5, ?]. The Playstation 3, despite being marketed as a gaming console, contains an implementation of the CellBE, and coupled with the ability to boot into a Linux platform, made for a powerful and unique scientific computing platform. Unfortunately the recent refresh of the PS3 lineup removed the ability to run a Linux OS, leaving the machine a pure entertainment system.

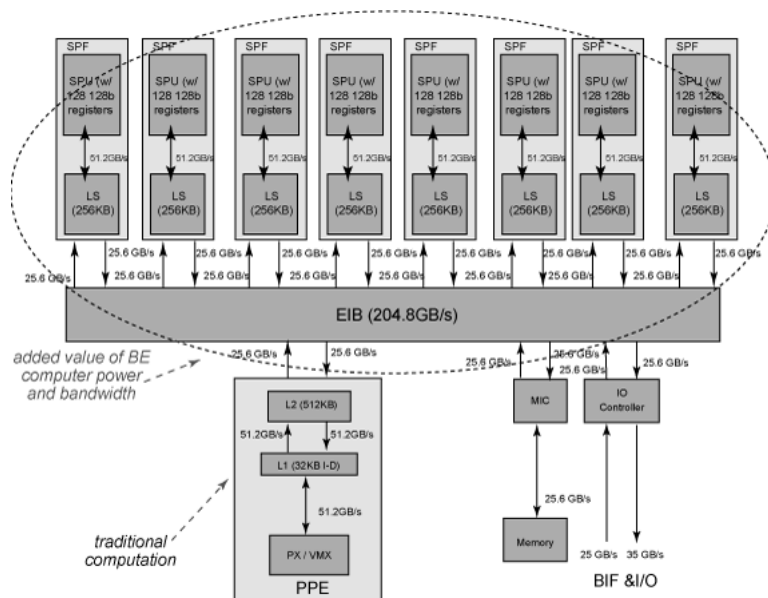


Figure 3.1: Block diagram of the PS3 architecture with theoretical memory bandwidth limits [7]

The PS3 is equipped with a 3.2GHz Cell platform, including a dual-threaded PowerPC PPE with 512Kb of L2 cache, 256MB of Rambus XDR memory, and 8 SPEs, 6 of which are user-accessible. The SPEs are 128bit SIMD capable, allowing them to perform vector operations on four floating point numbers from a single pipelined instruction.

While extremely powerful, a large hurdle in developing for the Cell Architecture lies in the fact that the SPEs do not have access to shared memory, main or otherwise. The individual SPEs are only able to access their 256K of local store, which is used to hold both instructions and data. To compensate for this, they are instead equipped with the capability to transfer data between main memory via DMA. An SPE channel has 25.6GB/s bandwidth to main memory, and DMA operations can execute independantly from executing code. Therefore optimal performance may be achieved by double or even triple buffering DMA transfers, aiming to saturate the memory pipeline and hide latency while waiting on memory.

Due to the Fast Multipole algorithm's tree-based divide and conquer nature, it has a high level of memory spatial locality, that is memory accesses are often adjacent or close by. This helpfully offsets the memory limitation of Cell's SPE's, and improves the case for parallelisation of the algorithm. This is well matched to the platform's 6 available SIMD processors and high memory performance.

## Chapter 4

# Algorithm Implementation

The first algorithm implemented was the standard, single-threaded pairwise algorithm, with no architecture-specific optimisations, as described in 2.1. It was designed to be used as a benchmark and results bench test. An implementation was also produced which could be deployed on a single SPE of the PS3.

As such a simple algorithm, there were few design decisions involved. One significant, yet simple optimisation was made within the loop which calculated Force and Potential energy. This involves updating force for both particles involved in a pairwise calculation, rather than the single outer-loop particle. This eliminates the redundant second pairwise calculations, therefore requiring only half the computations for this phase. It does not, however, improve upon the property of the algorithm's  $O(N^2)$  performance.

The next algorithm attempted was the main goal of the Fast Multipole Algorithm designed for the Cell Broadband Architecture. A number of factors needed to be considered during the design of this algorithm on such a unique architecture.

One must note that the double precision floating point performance of the Cell is orders of magnitude lower than single precision performance, therefore this algorithm was implemented using single floating point precision.

The next important decision made was the layout of required data structures. There are multiple methods for storing 2D position, velocity, force etc. arrays, the two most common being:

**Array of Structures (AOS)** Each index within an array points to a structure representing the (x,y) point. This makes for very readable code, however it eliminates the ability for the algorithm to take advantage of the SPE's SIMD features, as memory storing consecutive co-ordinates is not coherent.

**Structure of Arrays:** Multiple arrays are maintained, one for each component of the co-ordinates. This ends in more verbose, harder to track code, but allows for SIMD operations, in which the SPE could operate for example on a component of 4 different co-ordinates simultaneously. This would lead to a large performance advantage. Therefore this method was implemented.

# Chapter 5

## Results

Unfortunately due to coding failure, a successful implementation of the FMA on the PS3 was not achieved in time. Although most of the algorithm had been implemented, it was completely unable to complete a simulation, seemingly stalling mid-execution. This points to a segfault-causing bug in the SPE code, causing the SPE threads to die and leaving the PPU waiting.

Therefore the only results currently available are that of the basic, single threaded pairwise algorithm.

As a benchmark, the code was tested on an Intel Core2Duo Q6600 2.6GHz machine with 2GB DDR2 RAM, running on the 64 bit Linux kernel, version 2.6.31. This is a common mid-range setup, and can be used to compare how efficiently the Cell platform can execute general non-optimised code. As well on the reference machine, the code was executed on both types of processor equipped with the PS3 - the PPE and an SPE. The PS3 runs atop the 2.6.20 64 bit Linux kernel.

The results of these executions, presented in Table 5.1 on page 13 and Figure 5.1 on page 14, were quite strong.

The timing results for the PS3 running the code on the SPU are unusual. as the OS reports that although much time was taken to execute the code, only a very small amount was spent in the context of the process. This may be attributed to a lack of kernel support for the SPU hardware, causing faulty timing reports. Ignoring this anomaly, it becomes apparent that the standard Core2 machine is much more efficient at executing the unoptimised algorithm than the PS3 using either the PPU or SPU. The PPU follows by a long time, followed even further behind by the SPU. For example, in the first test, the PS3 PPU took *13x longer to execute* than the Core2, while the SPE took *over 650x the time* of the Core2. These results are somewhat explainable, as the PS3 chips (and the SPU in particular) sacrifice auto-optimisation

<i>N</i>	<i>Core2</i>	<i>PS3 - PPU</i>	<i>PS3 - SPU</i>
25	0m0.267s / 0m0.260s	0m1.277s / 0m1.270s	3m1.334s / 0m0.001s
100	0m3.521s / 0m3.510s	0m19.442s / 0m19.423s	4m12.688s / 0m0.002
196	0m13.175s / 0m13.160s	1m14.426s / 1m14.365s	DNF
400	0m54.261s / 0m54.250s	5m8.674s / 5m8.444s	DNF

Table 5.1: Performance comparison of the pairwise algorithm on different execution platforms. Timing format real,user. Each timing the result of an average of 5 runs. All runs executed with time-step 0.01s, over 10000 steps.

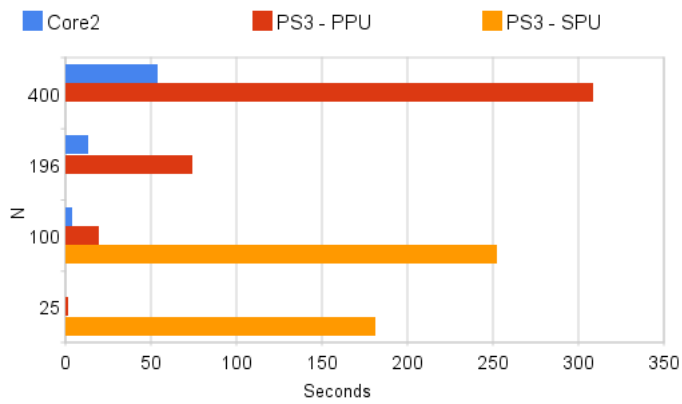


Figure 5.1: Performance of the Core 2 compared to the PS3 Cell, PPU and SPU

features such as branch prediction and out-of-order execution in favor of simpler, more specific silicon. It is expected that the developer codes with the Cell hardware in mind, using its advanced features and supplementing the lack of advanced hardware, for example dropping branch prediction hints to the compiler via special code macros.

## Chapter 6

# Reflection, Future Work

It was quite unfortunate that the advanced algorithm was not completed in time to report on. It is expected that it would have been able to extract much greater performance out of the specialised Cell hardware. However, programming for the Cell architecture, and learning its quirks, proved much more difficult than anticipated. Considering the machine's somewhat dismal performance in executing standard code, it is clear that the Cell architecture is indeed a highly specialised platform, requiring a greater deal of knowledge and skill to work with. It would seem that even working on the basic pairwise algorithm, it would take a great deal of re-factoring and tweaking to match or exceed the performance of a standard desktop machine. It would seem to be a greater payoff to use that effort to simply optimise the general purpose code on a standard x86 machine, given it's much greater initial performance.

The Fast Multipole Algorithm however, seems to be a very effective method of optimising particle simulations. It is not overly complex, yet has been shown to achieve performance gains in orders of magnitude over the standard algorithm.

This project may be continued in order to complete the Cell implementation of the FMA, however, it is questionable whether there will be any considerable gain, if any, from implementing the algorithm on this platform.

# Bibliography

- [1] Jakub Kurzak Jack Dongarra Alfredo Buttari, Piotr Luszczek and George Bosilca. Scop3: A rough guide to scientific computing on the playstation 3. Technical report, Innovative Computing Laboratory, University of Tennessee Knoxville, April 2007.
- [2] David Erb. Creating high performance radar applications with the cell broadband engine. Presentation, April 2007.
- [3] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. volume 73, pages 315 -- 348. 1987.
- [4] J.Barnes & P. Hut. A hierarchical  $O(n \log n)$  force calculation algorithm. volume 324. 1986.
- [5] H.P. Hofstee C.R. Johns T.R. Maerur J.A. Kahleand, M.N. Day and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Research and Development*, 49:589--604, 2005.
- [6] A modified tree code: don't laugh; it runs. J.e. barnes. volume 87, pages 161--170. 1990.
- [7] J. Dale T. Chen, R. Raghavan and E. Iwata. Cell broadband engine architecture and its first implementation - a performance view, 2005.