TECHNICAL UNIVERSITY OF CRETE
DEPARTMENT OF ELECTRONIC AND COMPUTER ENGINEERING

Master Thesis

Programming high-performance applications on the Cell BE processor

CHRISTOU PANAGIOTIS

Selection Committee

Ioannis Papaefstathiou, Assistant Professor of the Technical University of Crete (Supervisor)
Apostolos Dollas, Professor of the Technical University of Crete
Dionisios Pnevmatikatos, Associate Professor of the Technical University of Crete

CHANIA 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past decade, high-performance computing has ridden the wave of commodity computing, building clusterbased parallel computers that leverage the tremendous growth in processor performance fueled by the commercial world. As this pace slows, processor designers face complex problems in their efforts to increase gate density, reduce power consumption, and design efficient memory hierarchies. Processor developers are looking for solutions that can keep up with the scientific and industrial communities' insatiable demand for computing capability and that also have a sustainable market outside science and industry.

A major trend in computer architecture is integrating system components onto the processor chip. This trend is driving the development of processors that can perform functions typically associated with entire systems. Building modular processors with multiple cores is far more cost-effective than building monolithic processors, which are prohibitively expensive to develop, have high power consumption, and give limited return on investment. Multicore system-on-chip (SoC) processors integrate several identical, independent processing units on the same die, together with network interfaces, acceleration units, and other specialized units.

Researchers have explored several design avenues in both academia and industry. Examples include MIT's Raw multiprocessor, the University of Texas's Trips multiprocessor, AMD's Opteron, IBM's Power5, Sun's Niagara, and Intel's Montecito, among many others.

In all multicore processors, a major technological challenge is designing the internal, on-chip communication network. To realize the unprecedented computational power of the many available processing units, the network must provide very high performance in latency and in bandwidth. It must also resolve contention under heavy loads, provide fairness, and hide the processing units' physical distribution as completely as possible.

As the era of pure CMOS frequency scaling ends, architects must again respond to massive technological changes by more efficiently exploiting density scaling. The

Cell Broadband Engine (Cell BE) answers these challenges by providing the first implementation of a chip multiprocessor with a significant number of general-purpose programmable cores targeting a broad set of workloads, including intensive multimedia and scientific processing.

Jointly developed beginning in 2000 by IBM, Sony, and Toshiba (STI) for the PlayStation 3 as well as other data-processing-intensive environments, Cell's design goal was to improve performance an order of magnitude over that of desktop systems shipping in 2005 [et 05], [M. 06a], [M. 06b]. To meet that goal, designers had to optimize performance against area, power, volume, and cost in a manner not possible with legacy architectures. Thus, the design strategy was to exploit application parallelism through numerous cores that support established application models, thereby ensuring good programmability as well as programmer efficiency[A. 05].

The resulting Cell design is a heterogeneous, multicore chip capable of massive floating-point processing optimized for computation-intensive workloads and rich broadband media applications.

The rest of this thesis is organized as follows: **Chapter 2** introduces the Cell Broadband Engine processor and the Playstation 3 that was used in the implementation. **Chapter 3** outlines the gene identification problem, the Markov Chains and Interpolated Markov Models and the Glimmer algorithm. **Chapter 4** presents the Long-Range Noise Propagation and Helicopter Path Opti- mization for Noise Reduction algorithm. **Chapter 5 and Chapter 6** describe the development process that was followed for the implementations of the Glimmer and CNPE applications respectively. **Chapter 7 and Chapter 8** present the final results for the implementations of the Glimmer and CNPE application respectively. **Chapter 9** presents some conclusions from this work and proposes some ideas for future work.

Chapter 2

Platform

This chapter describes the platform that was used for the implementation. The Cell processor, its architecture as well as and the Playstation 3 game console are described in this chapter. The purpose of the chapter is to introduce the reader to the architecture of Cell and to the hardware that was used in the implementation.

2.1 Cell Broadband Engine

The Cell processor is the first implementation of the Cell Broadband Engine Architetcture (CBEA), which is a fully compatible extension of the 64-bit PowerPC Architecture. Its initial target is the PlayStation 3 game console, but its capabilities also make it well suited for other applications such as visualization, image and signal processing, and various scientific and technical workloads.

Figure 2.1 shows the Cell Broadband Engine, the first implementation of the CBEA . The processor is a heterogeneous, multicore chip capable of massive floating-point processing optimized for computation- intensive workloads and rich broadband media applications. It consists of one 64-bit power processor element (PPE), eight specialized coprocessors called synergistic processor elements (SPEs), a high-speed memory controller, and a high-bandwidth bus interface, all integrated on-chip. The PPE and SPEs communicate through an internal highspeed element interconnect bus (EIB).

With a clock speed of 3.2 GHz, the Cell processor has a theoretical peak performance of 204.8 Gflop/s (single precision) and 14.6 Gflop/s (double precision). The element interconnect bus supports a peak bandwidth of 204.8 Gbytes/s for intrachip data transfers, the memory interface controller provides a peak bandwidth of 25.6 Gbytes/s to main memory, and the I/O controller provides peak bandwidth of 25 Gbytes/s inbound and 35 Gbytes/s outbound[et 05].

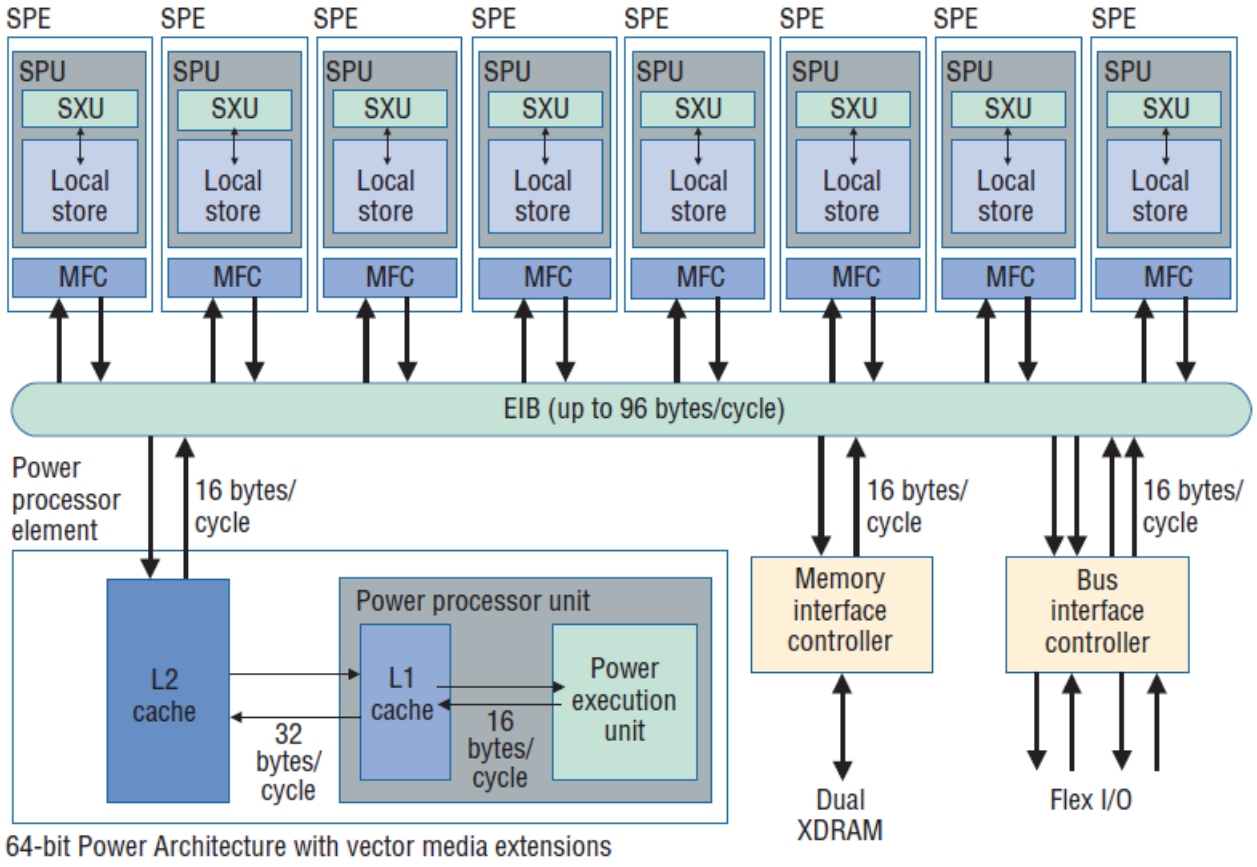

Figure 2.1: Cell Broadband Engine Architecture

2.1.1 Power Processor Element

The Power processor element (PPE)[et 05],[IBM08b], [Sca09] consists of a 64- bit, multithreaded Power Architecture processor with two concurrent hardware threads. The PPE supports the Power Architecture vector multimedia extensions to accelerate multimedia applications using SIMD execution units.

The PPE consists of two main units, the Power Processor Unit (PPU) and the Power Processor Storage Subsystem (PPSS), as shown in Figure 2.2. The PPE is responsible for overall control of the system. It runs the operating systems for all applications running on the Cell Broadband Engine.The PPU deals with instruction control and execution. It includes the full set of 64-bit PowerPC registers, 32 128-bit vector registers, a 32-KB level 1 (L1) instruction cache, a 32-KB level 1 (L1) data cache, an instruction-control unit, a load and store unit, a fixed-point integer unit, a floating point unit, a vector unit, a branch unit, and a virtual-memory management unit.

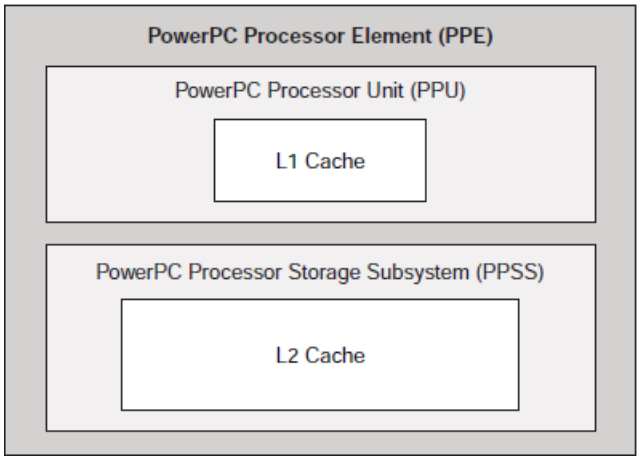

Figure 2.2: PPE Block Diagram

The PPU supports two simultaneous threads of execution and can be viewed as a 2-way multiprocessor with shared dataflow. This appears to software as two independent processing units. The state for each thread is duplicated, including all architected and special-purpose registers except those that deal with system-level resources, such as logical partitions, memory, and thread-control. Most nonarchitected resources, such as caches and queues, are shared by both threads, except in cases where the resource is small or offers a critical performance improvement to multithreaded applications.

The PPSS handles memory requests from the PPE and external requests to the PPE from other processors or I/O devices. It includes a unified 512-KB level 2 (L2) instruction and data cache, various queues, and a bus interface unit that handles bus arbitration and pacing on the EIB. Memory is seen as a linear array of bytes indexed from 0 to 264 - 1. Each byte is identified by its index, called an address, and each byte contains a value. One storage access occurs at a time, and all accesses appear to occur in program order.

The L2 cache and the address-translation caches use replacement-management tables that allow software to control use of the caches. This software control over cache resources is especially useful for real-time programming.

Although clocked at 3.2 GHz PPE looks like a quite potent processor, its main purpose is to serve as a controller and supervise the other cores on the chip. In a Cell based system the PPE will run the operating system (OS) and most of the applications but compute intensive parts of the OS and applications will be offloaded to the SPEs. Thanks to the PPE's compliance with the PowerPC architecture, existing applications can run on the Cell out of the box, and be gradually optimized for performance using the SPEs ,rather than written from scratch[et 05].

2.1.2 Synergistic Processor Element

Each of the eight Synergistic Processor Elements (SPEs)[et 05],[MS07],[IBM08b], [Sca09] is a 128-bit RISC processor specialized for data-rich, compute-intensive SIMD applications. It consists of two main units, the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC), as shown in Figure 2.3.

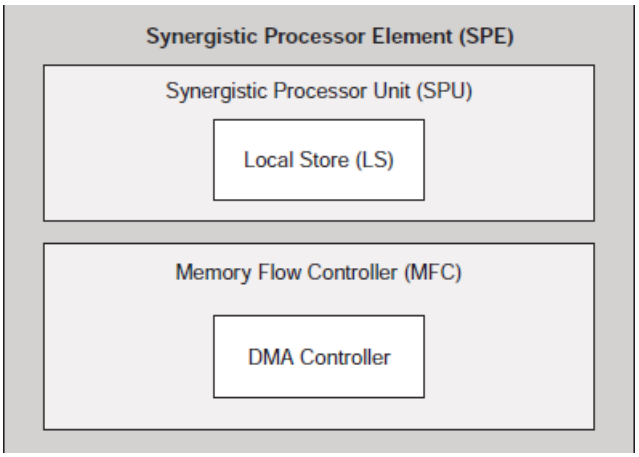

Figure 2.3: SPE Block Diagram

The SPU deals with instruction control and execution. It includes a single register file with 128 registers (each one 128 bits wide), a unified (instructions and data) 256-KB local store (LS), an instruction-control unit, a load and store unit, two fixed-point units, a floating-point unit, and a channel-and-DMA interface. The SPU implements a new SIMD instruction set, the SPU Instruction Set Architecture, that is specific to the Broadband Processor Architecture.

Each SPU is an independent processor with its own program counter and is optimized to run SPE threads spawned by the PPE. The SPU fetches instructions from its own LS, and it loads and stores data from and to its own LS. With respect to accesses by its SPU, the LS is unprotected and untranslated storage.

Most instructions operate in a SIMD fashion on 128 bits of data representing either two 64-bit doubleprecision floating-point numbers or longer integers, four 32-bit single-precision floating-point numbers or integers, eight 16-bit subwords, or sixteen 8-bit characters. The 128-bit operands are stored in a 128- entry unified register file. Instructions may take up to three operands and produce one result. The register file has a total of six read and two write ports.

The memory instructions also access 128 bits of data, with the additional constraint that the accessed data must reside at addresses that are multiples of 16 bytes. Thus, when addressing memory with vector load or store instructions, the lower four bits of the byte addresses are simply ignored. To facilitate the loading

and storing of individual values, such as a character or an integer, there is additional support to extract or merge an individual value from or into a 128-bit register.

An SPE can dispatch up to two instructions per cycle to seven execution units that are organized into even and odd instruction pipes, as shown in Figure 2.4. Instructions are issued in order and routed to their corresponding even or odd pipe by the issue logic, that is, a component which examines the instructions and determines how they are to be executed, based on a number of constraints. Independent instructions are detected by the issue logic and are dual-issued (i.e., dispatched two per cycle) provided they satisfy the following condition: the first instruction must come from an even word address and use the even pipe, and the second instruction must come from an odd word address and use the odd pipe. When this condition is not satisfied, the two instructions are executed sequentially. The instruction latencies and their pipe assignments are shown in Figure 2.5.

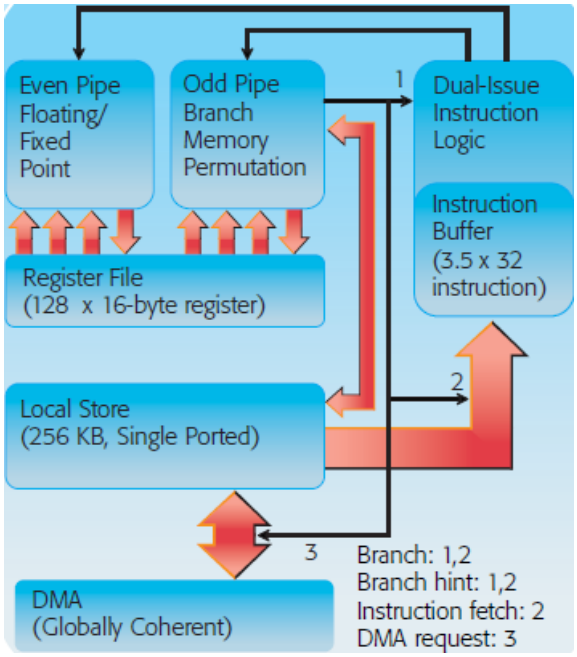

Figure 2.4: Synergistic Processor Element architecture

The SPE's 256-KB local memory supports fully pipelined 16-byte accesses (for memory instructions) and 128-byte accesses (for instruction fetches and DMA transfers). Because the memory has a single port, instruction fetches, DMA, and memory instructions compete for the same port. Instruction fetches occur during idle memory cycles, and up to 3.5 fetches may be buffered in the instruction fetch buffer to better tolerate bursty peak memory usage. The maximum capacity of the buffer is thus 112 32- bit instructions. An explicit instruction can be used to initiate an

inline instruction fetch.

Instruction	Pipe	Latency (cycles)
arithmetic, logical, compare, select	even	2
byte sum/diff/average	even	4
shift/rotate	even	4
float	even	6
integer multiply-accumulate	even	7
shift/rotate, shuffle, estimate	odd	4
load, store	odd	6
channel	odd	6
branch	odd	1–18

Figure 2.5: Latencies and pipe assignment for SPE

The SPE hardware assumes that branches are not taken, but the architecture allows for a "branch hint" instruction to override the default branch prediction policy. In addition, the branch hint instruction causes a prefetch of up to 32 instructions, starting from the branch target, so that a branch taken according to the correct branch hint incurs no penalty. One of the instruction fetch buffers is reserved for the branch-hint mechanism. In addition, there is extended support for eliminating short branches by using select instructions.

To access global data shared between threads executing on the PPE and other SPEs, each SPE includes an MFC, which performs data transfers between SPU-local storage and system memory. The MFC provides the SPEs with access to system memory by supporting high-performance direct memory access (DMA) data transfer between the system memory and the local store. Data transfers can range in size from a single byte to 16-Kbyte blocks.

The MFC transfers copy between local store and system memory. An MFC transfer request specifies the local store location as the physical address in the local store. It specifies the system memory address as a Power Architecture virtual address, which the MFC's memory management logic translates to a physical address based on system-wide page tables that the Power Architecture specification provides.

Using the same virtual addresses to specify system memory locations independent of processor element type enables seamless data sharing between threads executing on both the PPE and SPE. An application executing on Cell can pass a PPE-generated pointer to code executing on the SPE and use it to specify the source or target in an MFC transfer request. Using full memory translation also ensures data protection between processes, as a thread can only access the system memory

mapped into the associated process's virtual memory space.

Finally, using virtual addressing makes traditional operating system services such as demand paging available to SPE threads. When an SPE thread references paged-out memory via its associated MFC, the MFC's memory management unit generates a page-fault exception and delivers it to the PPE. The PPE then services the page fault on behalf of the SPE. When the page fault service has completed, the PPE restarts the MFC transfer that caused the page fault.

2.1.3 Element Interconnect Bus

The Element Interconnection Bus (EIB) [M. 06b], [et 05], [A. 05], [Sca09] is a communication bus internal to the Cell processor which connects the various on-chip system elements: the PPE processor, the memory controller (MIC), the eight SPE coprocessors, and two offchip I/O interfaces, for a total of 12 participants, as shown in Figure 2.6. The EIB has separate communication paths for commands (requests to transfer data to or from another element on the bus) and data. Each bus element is connected through a point-to-point link to the address concentrator, which receives and orders commands from bus elements, broadcasts the commands in order to all bus elements (for snooping), and then aggregates and broadcasts the command response. The command response is the signal to the appropriate bus elements to start the data transfer.

The EIB data network consists of four 16- byte-wide data rings: two running clockwise, and the other two counterclockwise. Each ring potentially allows up to three concurrent data transfers, as long as their paths don't overlap. To initiate a data transfer, bus elements must request data bus access. The EIB data bus arbiter processes these requests and decides which ring should handle each request. The arbiter always selects one of the two rings that travel in the direction of the shortest transfer, thus ensuring that the data won't need to travel more than halfway around the ring to its destination. The arbiter also schedules the transfer to ensure that it won't interfere with other in-flight transactions. To minimize stalling on reads, the arbiter gives priority to requests coming from the memory controller. It treats all others equally in round-robin fashion. Thus, certain communication patterns will be more efficient than others.

The EIB operates at half the processor-clock speed. Each EIB unit can simultaneously send and receive 16 bytes of data every bus cycle. The EIB's maximum data bandwidth is limited by the rate at which addresses are snooped across all units in the system, which is one address per bus cycle. Each snooped address request can potentially transfer up to 128 bytes, so in a 3.2GHz Cell processor, the theoretical peak data bandwidth on the EIB is 128 bytes $\times$ 1.6 GHz = 204.8 Gbytes/s.

The on-chip I/O interfaces allow two Cell processors to be connected using a coherent protocol called the broadband interface (BIF), which effectively extends the

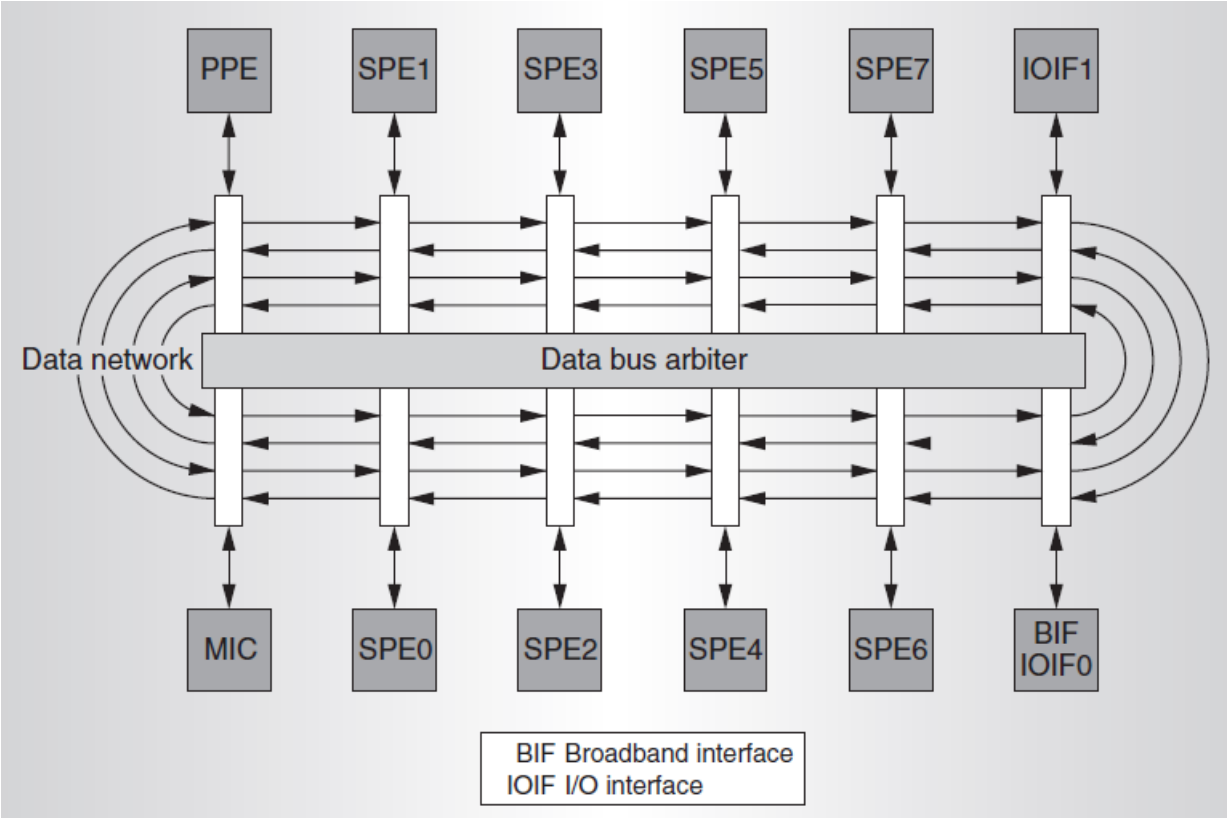

Figure 2.6: Element interconnect bus (EIB)

multiprocessor network to connect both PPEs and all 16 SPEs in a single coherent network.

2.2 Playstation3

Currently the easiest and the cheapest way to gain access to a Cell processor is the Sony PlayStation 3 (PS3) [Son]. As mentioned before, Cell processor was originally designed for PS3 and the vision was to achieve 1,000 times the performance of PlayStation 2. Due to the need of access to the Cell's computational power a Linux based operating system designed to run on PS3. The need for real Cell hardware mainly derives from the fact that IBM's Cell simulator is very slow. Today anybody can have access to the Cell processor by just installing an OS on PS3 and using it as a normal PC with high capabilities. Although PS3 is an easy solution it may not be the best, PS3 has some limitations on the performance of Cell. The main limitations are the small memory, only 256 MB and the availability of only six SPEs

out of eight. One of the eight SPEs is disabled at the hardware level due to yield reasons and another SPE is reserved for use by the PS3's operating system. Apart from these limitations PS3 remains a good choice for anybody who wants to have its own Cell processor.

2.2.1 Operating System

The PS3 is shipped with an operating system called Game OS but is capable of running Linux OS if installed on the console's hard drive. The Linux operating system runs on the PS3 on top of a virtualization layer, also called hypervisor, the Game OS. This means that all the hardware is accessible only through the hypervisor calls. The hardware signals the kernel through virtualized interrupts. The interrupts are used to implement callbacks for non-blocking system calls. The Game OS permanently occupies one of the SPEs and controls access to the hardware. A direct consequence of this is larger latency in accessing hardware such as the network card. Even worse, it makes some hardware inaccessible like the accelerated graphics card. At this point, there are numerous distributions that have official or unofficial support for PS3. The distributions that are currently known to work on PS3 (with varying levels of support and end-user experience) include:

- Fedora Core 7 [Red]
- YellowDog 6.0 [Ter]
- Gentoo PowerPC 64 edition [Gen]
- Debian [Deb]

All the distributions mentioned include Sony-contributed patches to the Linux kernel-2.6.16 to make it work on PS3 hardware and talk to the hypervisor. However, the Linux kernel version 2.6.20 has PS3 support already included in the source code without the need for external patches.

2.2.2 Memory System

The memory system is built of dual-channel Rambus Extreme Data Rate (XDR) memory. PS3 provides a modest amount of memory of 256 MB, out of which approximately 200 MB is accessible to Linux OS and applications. The memory is organized in 16 banks. Real addresses are interleaved across the 16 banks on a naturally aligned 128-byte (cache line) basis. Addresses 2 KB apart generate accesses to the same bank. For all practical purposes the memory can provide the bandwidth of 25.6 GB/s to the SPEs through the EIB, provided that accesses are distributed evenly across all the 16 banks.

2.2.3 Network Card

The PS3 has a built-in GigaBit Ethernet network card. However, unlike standard PC's Ethernet controllers, it is not attached to the PCI bus. It is directly connected to a companion chip. The network card has a dedicated DMA unit, which allows making data transfer without PPE's intervention. One of many advantages of GigaBit Ethernet is the possibility of increased frame size – so called Jumbo Frames. It can increase available bandwidth by 20% in some case and significantly decreases processor load when handling network traffic.

2.2.4 Graphics Card

PS3 features special edition from NVIDIA and 256 MB of video RAM. Unfortunately, the virtualization layer does not allow access to these resources. At issue is not as much accelerated graphics for gaming as is off-loading of some of the computations to GPU and scientific visualization.

Chapter 3

Glimmer Algorithm

This chapter presents the Gene Locator and Interpolated Markov Modeler (Glimmer) algorithm that has been ported on the Cell processor. Glimmer was the primary microbial gene finder used at The Institute for Genomic Research (TIGR), where it was first developed, and has been used to annotate the complete genomes of over 100 bacterial species from TIGR and other labs. The methodology, the basic steps of algorithm as well as matters of algorithm input and output are introduced in this chapter.

3.1 An Introductory Background On Biology

One of the fundamental principles of biology is that within each cell, DNA that comprises the genes encodes RNA which in turn produces the proteins that regulate all of the biological processes within an organism.

DNA is a double chain of simpler molecules called nucleotides, tied together in a double helix helical structure. The nucleotides are distinguished by a nitrogen base that can be of four kinds: adenine (A), cytosine (C), guanine (G) and thymine (T). Adenine (A) always bonds to thymine (T) whereas cytosine (C) always bonds to guanine (G), forming base pairs. DNA can be specified uniquely by listing its sequence of nucleotides, or base pairs. Proteins are molecules that accomplish most of the functions of a living cell, determining its shape and structure. A protein is a linear sequence of molecules called amino acids. Twenty different amino acids are commonly found in proteins. Similar to DNA, proteins are conveniently represented as a string of letters expressing their sequence of amino acids. A gene is a contiguous stretch of genetic code along the DNA that encodes a protein. Not all parts of a DNA molecule encode genes; some segments, called introns, have no influence on protein synthesis[Dav05].

In molecular genetics, an open reading frame (ORF) is a portion of an organism's

genome which contains a sequence of bases that could potentially encode a protein. In a gene, ORFs are located between the start-code sequence (initiation codon) and the stop-code sequence (termination codon). ORFs are usually encountered when sifting through pieces of DNA while trying to locate a gene. Since there exist variations in the start-code sequence of organisms with altered genetic code, the ORF will be identified differently[The].

3.2 Gene Identification Problem

Accurate microbial gene identification is becoming ever more important with the increasing rate of whole genome sequencing projects. In the past year alone, eight new bacterial and archaeal genomes have appeared, and the pace continues to accelerate. Each new genome contains thousands of new genes, all of which are deposited into public databases. These genes then become the basis for much further research into the biology of these organisms, and their sequences are used for further biological study. For work such as microarray analysis, in which specific sequences are arrayed onto a substrate and used as probes to measure expression levels, the accuracy of gene predictions is critical. The same point can be made about knockout experiments, which are an important tool to use in determining the function of the large numbers of genes whose function is unknown at the time of publication. Such hypothetical proteins typically comprise 30–40% of the genes in a newly sequenced genome.

The sizes of biological sequence databases are usually very large. Not all the sequences are coding, namely are a template for a protein. For example, in the human genome only 3%-5% of the sequences are coding. Due to the size of the database, manual searching of genes who do code for proteins is not practical. Gene-findings aim to provide computational methods to automatically identify genes that encode proteins[Ste99][Dav05].

3.3 Interpolated Markov Models and Markov Chains

Markov models are a well-known tool for analyzing biological sequence data, and the predominant model for microbial sequence analysis is a fixed-order Markov chain[BM93][BD95]. A fixed order Markov model predicts each base of a DNA sequence using a fixed number of preceding bases in the sequence. For example, a 5^{th}-order model uses the five previous bases to predict the next base. However, learning such models accurately can be difficult when there is insufficient training data to accurately estimate the probability of each base occurring after every possible combination of five preceding bases. In general, a k^{th}-order Markov model

for DNA sequences requires 4^{k+1} probabilities to be estimated from the training data (e.g., 4096 probabilities for a 5^{th}-order model). In order to estimate these probabilities, many occurrences of all possible kmers must be present in the data.

An IMM overcomes this problem by combining probabilities from contexts of varying lengths to make predictions, and by only using those contexts (oligomers) for which sufficient data are available. In a typical microbial genome some 5mers will occur too infrequently to give reliable estimates of the probability of the next base, while some 8mers may occur frequently enough to give very reliable estimates. In principle, using longer oligomers is always preferable to using shorter ones, but only if sufficient data is available to produce good probability estimates. An IMM uses a linear combination of probabilities obtained from several lengths of oligomers to make predictions, giving high weights to oligomers that occur frequently and low weights to those that do not. Thus an IMM uses a longer context to make a prediction whenever possible, taking advantage of the greater accuracy produced by higher-order Markov models. Where the statistics on longer oligomers are insufficient to produce good estimates, an IMM can fall back on shorter oligomers to make its predictions.

3.3.1 Markov Chains

A Markov chain[Ste98][Ste99][Twe05] is a sequence of random variables X_i, where the probability distribution for each X_i depends only on the preceding k variables X_{i-1}, ..., X_{i-k}, for some constant k. For DNA sequence analysis, a Markov chain models the probability of a given base b as depending only on the k bases immediately prior to b in the sequence. We refer to these preceding k bases as the context of base b in the sequence. A first order Markov chain is a sequence of random variables where the probability that X_i takes a particular value only depends on the preceding variable X_{i-1}. Note that for DNA sequences a first-order Markov chain is specified completely by a matrix of 16 probabilities: p(a|a), p(a|c), ..., p(t|t). The most common type of Markov chain is a fixed-order chain, in which the entire k-base context is used at every position. For example, a fixed 5^{th}-order Markov chain model of DNA sequences comprises $4^5 = 1024$ probability distributions, one for each possible 5mer context. Such fixed 5^{th}-order models have proven effective at gene prediction in bacterial genomes[BM93][BD95]. Ideally, larger values for k are always preferable.Unfortunately, because the training data available for building models is limited, we must limit k. In most collections of DNA coding sequences, however, there is substantial variability in the frequency of occurrence of different kmers.

The Glimmer algorithm uses seven submodels to find genes in microbial DNA. The algorithm builds six submodels one for each of the possible reading frames (three forward and three revers) and a seventh model for non-coding regions. Each

model makes different predictions for the bases in the three codon positions. Even with a 0^{th}-order model, the frequency of g in codon position 1 will be different from its frequency in another frame, so even this very weak model has some ability to identify the right reading frame for a gene.

Using the Markov models for each of the six possible frames plus a model of non-coding DNA, we can straightforwardly produce a simple algorithm for finding genes. Simply score every orf using all seven models, and choose the model with the highest score. The scores can be normalized so they represent the probability that a sequence is coding. If the model corresponding to the true coding region in the correct frame scores the highest, then the orf can be labeled as a gene.

3.3.2 Interpolated Markov Models (IMMs)

An Interpolated Markov Model (IMM)[Ste98][Ste99] uses a combination of all the probabilities based on 0, 1, 2, ..., k previous bases, where k is a parameter given to the algorithm. In GLIMMER, we use k = 8. Thus for oligomers that occur frequently, the IMM can use an 8^{th}-order model, while it might use a 5^{th} or even lower-order model for rare oligomers. In order to 'smooth' its predictions, an IMM uses predictions from the lower-order models, where much more data is available, to adjust the predictions made from higher-order models.

During training, GLIMMER computes the probability of each base a, c, g, t, following all kmers for $0 \leq k \leq 8$. Then, for each kmer it computes a weight to use in combining the predictions of different order models. Details of the algorithm for computing these weights are given in the Algorithm and system design section. Once the weights are computed, GLIMMER evaluates new sequences by computing the probability that the model M generated the sequence S, P (S|M). This probability is computed as

$$P(S|M) = \sum_{x=1}^{n} \mathbf{IMM_8}(S_x) \tag{3.1}$$

where S_x is the oligomer ending at position x, and n is the length of the sequence. $IMM_8(S_x)$, the 8^th-order interpolated Markov model score, is computed as

$$IMM_k(S_x) = \lambda_k(S_{x-1}) \cdot P_k(S_x) + [1-\lambda_k(S_{x-1})] \cdot IMM_{k-1}(S_x) \tag{3.2}$$

where $\lambda_k(S_{x-1})$ is the numeric weight associated with the kmer ending at position x – 1 in the sequence S and $P_k(S_x)$ is the estimate obtained from the training data of the probability of the base located at x in the k^{th}-order model. Thus, the 8^{th}-order IMM score of an oligomer is a linear combination of the predictions made by the 8^{th}, 7^{th} and lesser-order models all the way down to the 0^{th}-order model, which is just the simple prior probabilities of a, c, g, t.

3.3.3 Interpolated Context Models (ICMs)

Interpolated context models (ICMs)[Ste99] are a further extension of IMMs. For a given context C = $b_1 b_2 \ldots b_k$ of length k, the IMM in GLIMMER 1.0 computes a probability distribution for b_{k+1} using as many of the bases immediately preceding b_{k+1} as the training data set allows. The ICM is more flexible and can select any of the bases in C (not just those adjacent to b_{k+1}) to determine the probability of b_{k+1}. In general, from a given context, the ICM will choose approximately the same number of bases as the IMM. In GLIMMER 2.0, the motivation for choosing bases other than those at the end of the context is the fact that in coding regions the significance of a given base depends strongly on its position in a codon.

The criterion employed by the ICM to select which bases of a context C to use is mutual information. The mutual information between a given pair of discrete random variables X and Y is defined to be:

$$I(X;Y) = \sum_i \sum_j P(x_i, y_i) log(\frac{P(x_i)P(y_i)}{P(x_i, y_i)}) \tag{3.3}$$

where x_i and y_j are the values taken by random variables X and Y respectively, and $P(x_i, y_j)$ is the joint probability of x_i and y_j together.

To construct an ICM with context length k from a training set T of DNA sequences, we begin by considering all windows (i.e. oligomers) of length k+1 that occur in T. The algorithm lets random variable X_1 be the distribution of bases in the first position of those windows; X_2 be the distribution of bases in the second position; and so on through X_{k+1}.It then calculates the mutual information values $I(X_1; X_{k+1}), I(X_2; X_{k+1}), \ldots, I(X_k; X_{k+1})$, and choose the maximum. Suppose that maximum is $I(X_j; X_{k+1})$. It then partitions set of windows into four subsets based on the nucleotide that occurs in position j in the window.

The same procedure can now be performed again for each of the four sets of windows. Within each set, the position that has the highest mutual information with the base at position k+1 is chosen. The four nucleotide values at that position induce a further partitioning of the current set of windows into four subsets.

This process can be viewed as constructing a tree of positions within context strings. A sample portion of such a tree is shown in Figure 3.1. The construction is terminated when the tree depth reaches a predetermined limit, or when the size of a set of windows becomes too small to be useful to estimate the probability of the last base position.

Each node in the ICM decomposition tree represents a set of windows that provide a probability distribution for the final base position. The root node, which includes all possible windows, represents a 0^th-order Markov model. All other nodes give a probability distribution for the final base position, conditional on a specific

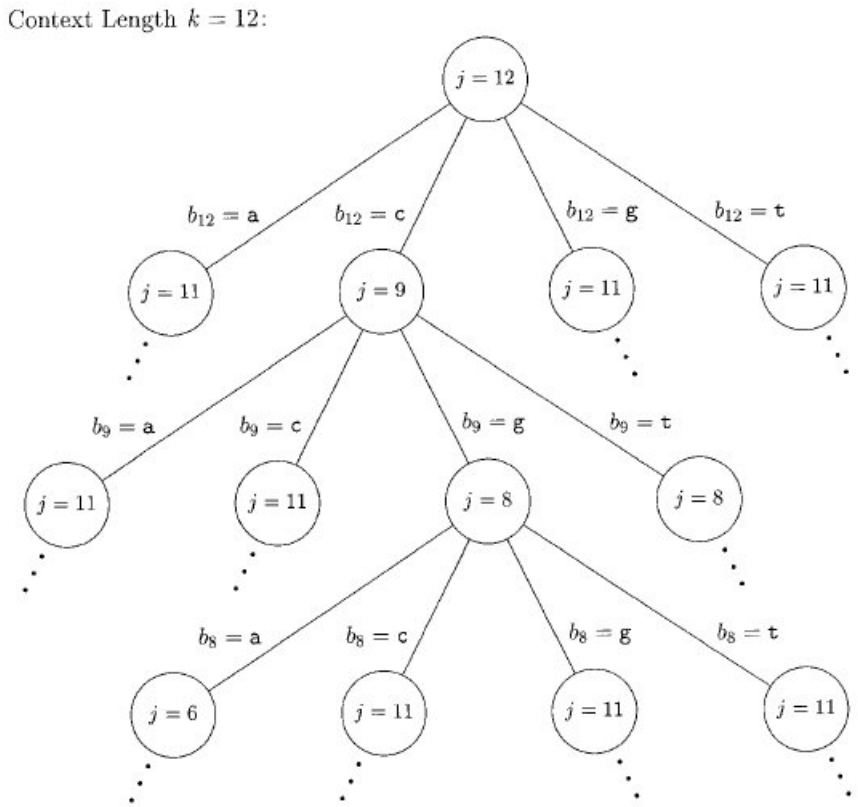

Figure 3.1: Sample ICM decomposition tree

set of bases occurring at the positions indicated on the path to the root from that node.

3.4 Glimmer Algorithm

Glimmer (Gene Locator and Interpolated Markov Modeler) [Ste98][Dav05] finds genes in microbial DNA. Its uses interpolated Markov models (IMMs) to identify coding and noncoding regions in the DNA. The program consists of essentially two steps: the first step trains the IMM from an input set of sequences, the second step uses this trained IMM for finding putative genes in the input genome.

3.4.1 Algorithm Input

Glimmer takes a sequence file in FASTA format. FASTA format is a text-based format for representing either nucleotide sequences or peptide sequences, in which

base pairs or amino acids are represented using single-letter codes. The format also allows for sequence names and comments to precede the sequences. The simplicity of FASTA format makes it easy to manipulate and parse sequences using text-processing tools and scripting languages like Python and Perl.

3.4.2 Algorithm Output

The algorithm outputs a list of all open reading frames (orfs) together with scores for each as a gene. An open reading frame (ORF) is a portion of an organism's genome which contains a sequence of bases that could potentially encode a protein.

3.4.3 The Glimmer System

The GLIMMER system consists of four programs.The first program long-orfs takes a sequence file in FASTA format and outputs a list of all long "potential genes" in it that do not overlap by too much. The extract program takes a FASTA format sequence file and a file with a list of start/stop positions in that file as produced by the long-orfs program and extracts and outputs the specified sequences. Program build-icm creates and outputs an interpolated Markov model (IMM) as describe in the section 3.3.2. Final, the glimmer2 takes a sequence file in FASTA format and a collection of Markov models for genes as produced by the program build-icm . It outputs a list of all open reading frames (orfs) together with scores for each as a gene.

Glimmer does not use sliding windows to score regions. Instead, it first identifies all orfs longer than some specified threshold value, and scores each one in all six reading frames. Those that score higher than a designated threshold in the correct reading frame are then selected for further processing. These selected orfs are then examined for overlaps. If two orfs in different reading frames overlap (by more than some designated minimum length), the overlapping region alone is scored separately. The overlap region's six reading frame scores are then compared with those of the two overlapping orfs to see which frame scores highest. In general, when a longer orf overlaps a shorter orf and the overlap region scores highest in the reading frame of the longer orf, then the shorter orf is eliminated as a gene candidate. The final output of the program is a list of putative gene coordinates in the genome, together with notations for each one that may have had a suspicious overlap with another gene candidate. These 'suspect' gene candidates (usually a very small percentage of the total) can then be examined manually to determine if they are in fact genes.

Chapter 4

Long-Range Noise Propagation and Helicopter Path Optimization for Noise Reduction

This chapter presents the Long-Range Noise Propagation and Helicopter Path Optimization for Noise Reduction algorithm that has been ported on the Cell processor. The algorithm developed by FORTH Research at Heraklion of the Crete [FOR].

This application is an accurate and computationally efficient procedure for the prediction of long-range propagation over realistic, complex terrain of the low frequency noise components from helicopter rotors. The sound propagation method is based on the computationally efficient parabolic equation (PE) approach that describes accurately the time-harmonic, far-field sound propagation. Accurate and efficient methods developed in the past are used for the numerical solution. Atmospheric, terrain impedance, Geographic Information System (GIS) models or measurements are used for realistic PE predictions. Standard optimization techniques are combined with the computationally efficient long-range noise propagation method to prescribe the helicopter flight path that minimizes noise level at the reception point. The methodology, the basic steps of algorithm as well as matters of algorithm input and output are introduced in this chapter.

4.1 Introduction and Background

The aerodynamic noise generated by helicopter rotors has been thoroughly studied in the past. Despite the efforts for improved aerodynamic designs that reduce noise emission from helicopter rotors, helicopter noise remains a problem for both civilian and military missions. Low frequency noise components from helicopter rotors propagate with small attenuation over long distances in the atmosphere. As

a result, noise received at remote locations gives warning for the approaching vehicle long before it is visible at the reception point. Prediction of long-range propagation of noise over complex terrain and selection of the optimal helicopter flight path that yields the minimum noise level at the reception point is therefore important for mission success.

Long-range propagation of noise over complex terrain is described by the Euler linear equations [J. 09]. However, the numerical solution of these equations is far from trivial and computationally intensive even for moderate-size domains, because it requires application of high-order accurate in space and time, low-diffusion numerical methods, which are needed in order to propagate the sound waves with minimal distortion. On the other hand, application of the efficient but approximate ray methods for noise propagation does not yield accurate predictions for sound waves propagating over complex terrain, in inhomogeneous environment, and strong wind gradients. Understanding noise propagation mechanisms in the atmosphere has attracted significant attention due to the great socio-economic impact of controlling environmental noise levels. The atmosphere is an inhomogeneous moving medium where winds, atmospheric turbulence, and temperature variations result in a height- and range-varying sound speed in the air. The significant variation of the sound speed in the atmosphere combined with the complexity of the terrain adds more difficulties to long-range noise prediction.

For near-horizontal sound propagation in an inhomogeneous atmosphere, "paraxial approximations" of the Helmholtz equation[S. 08] in cylindrical coordinates that yield a parabolic equation in range have been effectively used in the past to model long-range propagation. In particular, the standard narrow-angle parabolic equation approximation introduced by Tappert [N. 08] and the third-order wide-angle parabolic approximation introduced by Claerbout [N. 08] were used to model sound propagation in axially symmetric atmosphere. In addition, these methods were coupled with atmospheric turbulence models to study the effects of atmospheric turbulence on sound propagation. Numerical simulations of acoustic propagation in an atmosphere with height-dependent sound speed and surfaces with range varying impedance were also carried out. Recently, numerical models based on the parabolic approximation for sound propagation in the atmosphere were interfaced with a geographic information system (GIS) for the prediction of sound propagation over irregular terrain.

An additional simplification may be introduced with the use of Green functions [G. 89] approach for the Parabolic Equation. The Green function approach is expected to significantly reduce the computing cost of three-dimensional solutions. However, the Green function approach introduces additional modeling error and evaluation of this approach through comparisons with the full parabolic equation solutions is needed especially for the three-dimensional cases

4.2 Problem Statement

The objective of this research is to identify the flight path of a flying object that yields minimum sound levels at a given reception point. This approach is demonstrated with the simple two-dimensional schematic of Fig. 4.1 where the optimization parameter is the flight height, H, of the noise source (helicopter) from an irregular terrain. The optimal height is determined for several stations, $O_1, O_2, \ldots, O_N$, by repeating the optimization problem N times for progressively reduced domains, and the optimal flight path is determined as shown in Fig. 4.1.

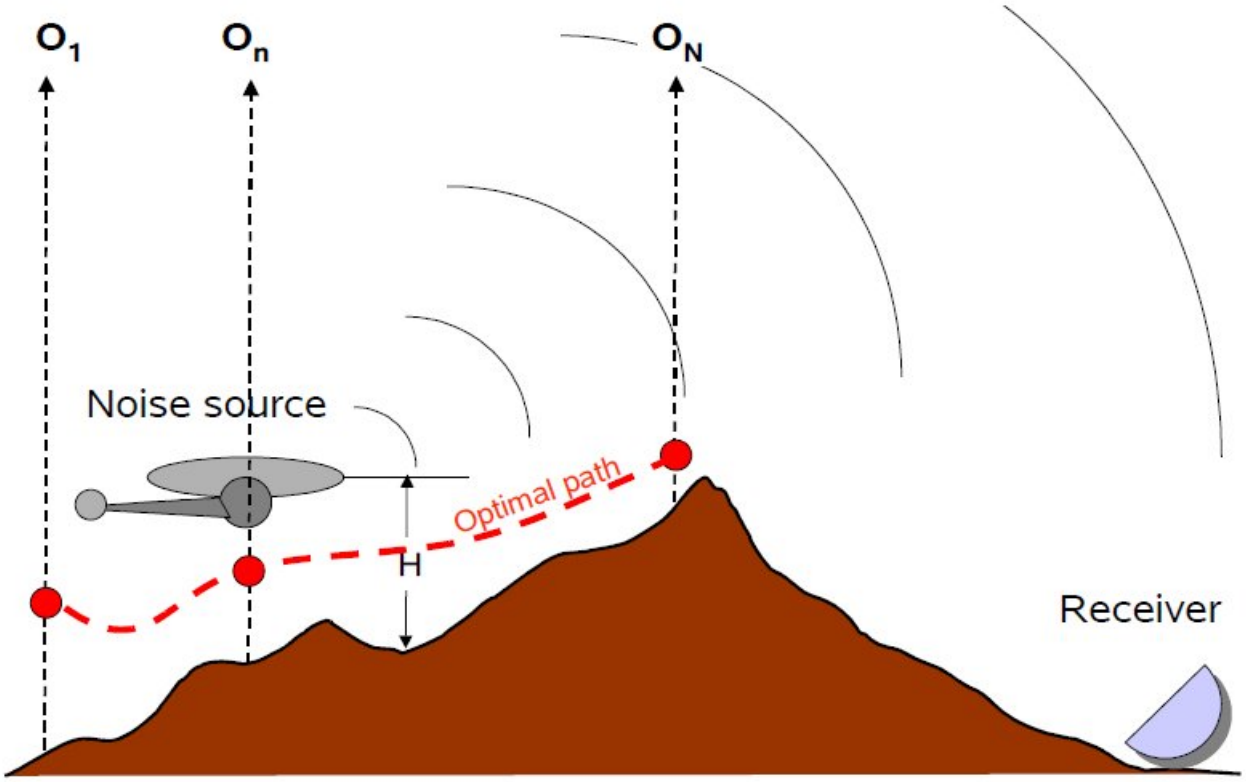

Figure 4.1: Schematic of long-range noise propagation of a helicopter flying at height H.

In three dimensions, the optimization problem is more complex but still possible to be solved (even the three-dimensional parabolic equation can be solved very efficiently).

4.3 The PE-method in sound propagation

The standard parabolic equation (PE) was introduced by Tappert [F.] as a model to describe time-harmonic, narrow-angle, far-field sound propagation in underwater acoustics. For the range r and depth z, using the approximation

$$\phi(r,z) = u(r,z)H(r), \tag{4.1}$$

where H(r) is the Hankel function of the first kind and zero order, the PE can be obtained by decomposing the elliptic wave equation

$$[\partial_{rr} + (\frac{1}{r})\partial_r + \partial_{zz} + k^2(r,z)]\phi = 0 \tag{4.2}$$

Furthermore, by ignoring the term u_{rr} the following parabolic approximation of the elliptic wave equation is obtained

$$u_r = \frac{ik_0}{2}[n^2(r,z) - 1]u + \frac{i}{2k_0}u_{zz}, \tag{4.3}$$

where $n(r,z) = \frac{c_0}{c(r,z)}$ is the index of refraction, *c(r,z)* is the local sound speed, c_0 is a reference sound speed, $k_0 = \frac{2\pi f}{c_0}$ the associated reference wavenumber, and f is the source frequency. Substituting $\alpha(r,z) = \frac{ik_0}{2}[n^2(r,z) - 1]$ and $\beta = \frac{i}{2k_0}$,Eq.(4.3) is written as

$$u_r = \alpha(r,z)u + \beta u_{zz}. \tag{4.4}$$

For the solution of the Eq.(4.4) are used the following simple boundary conditions

$$u(r,0) = u(r,z_{max}) = 0. \tag{4.5}$$

4.4 PDE Solution with Crank-Nicholson Algorithm

Crank-Nicholson Algorithm is a classical model, which has been prove to be an accurate and reliable approach in both 2D and 3D propagation [Cra]. CN-PE can incorporate complex environments, while a lot of research has been carried out and various techniques have been proposed, part of which was due to its application in underwater acoustics, also. The main disadvantage of the method is the computational cost, resulting from the requirement for small discretization step in range.

For the solution of the Eq.(4.4) over a domain, it is necessairy the boundary conditions. The Eq.(4.5) are the boundary conditions just over the terrain. We want to find a numerical solution u that satisfies Eq.(4.4) over an orthogonal domain. We assume that the domain is divided normally in the smaller orthogonals ($\Delta r \times \Delta z$) by using a mesh. The solution u is calculated over the nodes of the mesh by using the 2nd degree Crank-Nikolson.

A second-order accurate algorithm for implicit marching of Eq.(4.4) in the time-like direction is obtained with the second-order accurate Crank-Nicholson (CN-2) method

$$\left[U - \frac{\Delta r}{2}\mathbf{R}\right]_k^{n+1} = \left[U + \frac{\Delta r}{2}\mathbf{R}\right]_k^n \tag{4.6}$$

where U_k^n is the approximal solution u of the Eq.(4.4) over node $(n\Delta r, k\Delta z)$ and $\mathbf{R}(u) = u_r$.

Second-order accurate in space numerical solutions with the CN-2 method of Eq.(4.6) imply the following tridiagonal matrix inversion

$$a_l U_{k-1}^{n+1} + b_l U_k^{n+1} + a_l U_{k+1}^{n+1} = a_r U_{k-1}^n + b_r U_k^n + a_r U_{k+1}^n, \tag{4.7}$$

where the coefficients for the CN-2 scheme of Eq.(4.7) are

$$a_l = -\beta h_r h_z, a_r = \beta h_r h_z,$$

$$b_l = 1 - \alpha h_r + 2\beta h_r h_z, b_r = 1 + \alpha h_r - 2\beta h_r h_z$$

with $h_r = \frac{\Delta r}{2}$ and $h_z = \frac{1}{(\Delta z)^2}$.

The Eq. (4.7) produces a tridiagonal system with unknown values the solution of the PE of Eq. (4.4) on the discretization nodes. A tridiagonal algorithm [C. 00] resolves this system (a variance of LU method is used when the system table is zero-valued apart from the central diagonal and sub-diagonals above and below).

4.5 Curvilinear Transformations of Coordinates

Numerical solutions of sound propagation in complex domains are obtained using curvilinear transformations of coordinates $\xi = \xi(r, z), \eta = \eta(r, z)$. These transformations map the physical domain (r,z), which is unequally spaced and irregular, to an equally spaced rectangular domain, referred to as the (ξ, η) computational domain. The numerical solution is performed in the computational domain equally spaced grid using unweighted finite-difference formulas. A typical body-fitted grid in which the η = constant lines follow the shape of the bottom and the ξ = constant lines are approximately orthogonal to the bottom surface to facilitate application of Neumann-type boundary conditions, is shown in Figure 4.2.

In most applications it is sufficient to use an equally spaced mesh with straight lines along time-like, or ξ, direction and stretched grid lines fitted to the bottom surface with irregular shape. Then,the simpler transformation of coordinates $\xi = \xi(r), \eta = \eta(r, z)$, is sufficient. A typical body-fitted grid, in which the η = constant lines follow the shape of the bottom and the ξ =constant lines for each time-like step are parallel lines not orthogonal to the bottom surface, is shown in Figure 4.3.

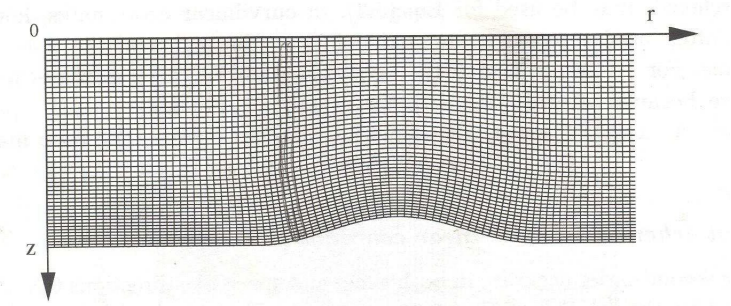

Figure 4.2: Orthogonal mesh over an irregular bottom.

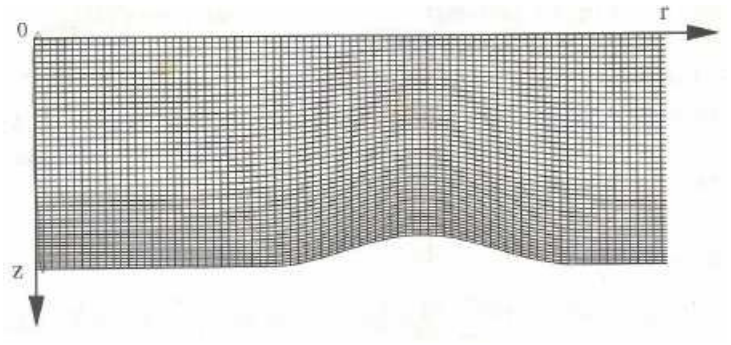

Figure 4.3: Non-Orthogonal mesh over an irregular bottom.

4.6 CNPE Algorithm

This section describes the input, the output and the basic steps of the CNPE application. The code of the algorithm has been developed by Dr. C. Arvanitis in programming language C.

4.6.1 Algorithm Input

The CNPE application accepts as entry the position of the receiver, the height of the transmitter and the topographic elements of a region.

4.6.2 Algorithm Output

The CNPE application returns the biggest received sound pressure by the receiver.

4.6.3 Basic Steps Of The Application

Step 1: The territorial region is separated in orthogonal departments by using curvilinear transformations of coordinates as shown in Figure 4.4.

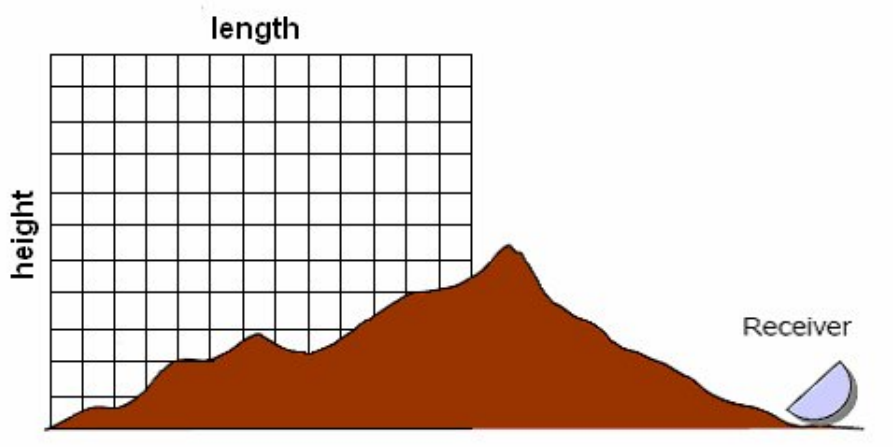

Figure 4.4: Schematic of the seperated territorial region.

Step 2: The CNPE algorithm is applied for various heights and lengths as shown in Figure 4.5.

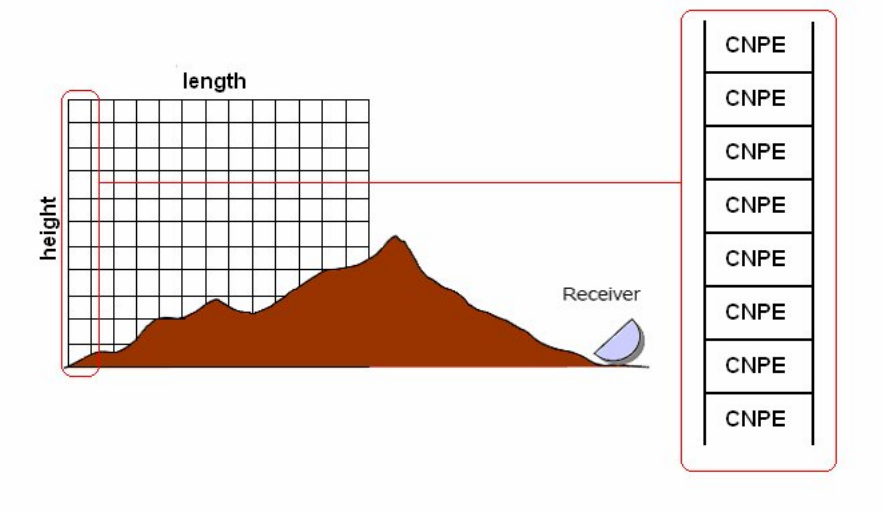

Figure 4.5: The CNPE is applied for each height.

Step 3: For every height and length the CNPE method resolves the tridiagonal system and calculates Max Receiving Pressure to the receiver as shown in Figure 4.6.

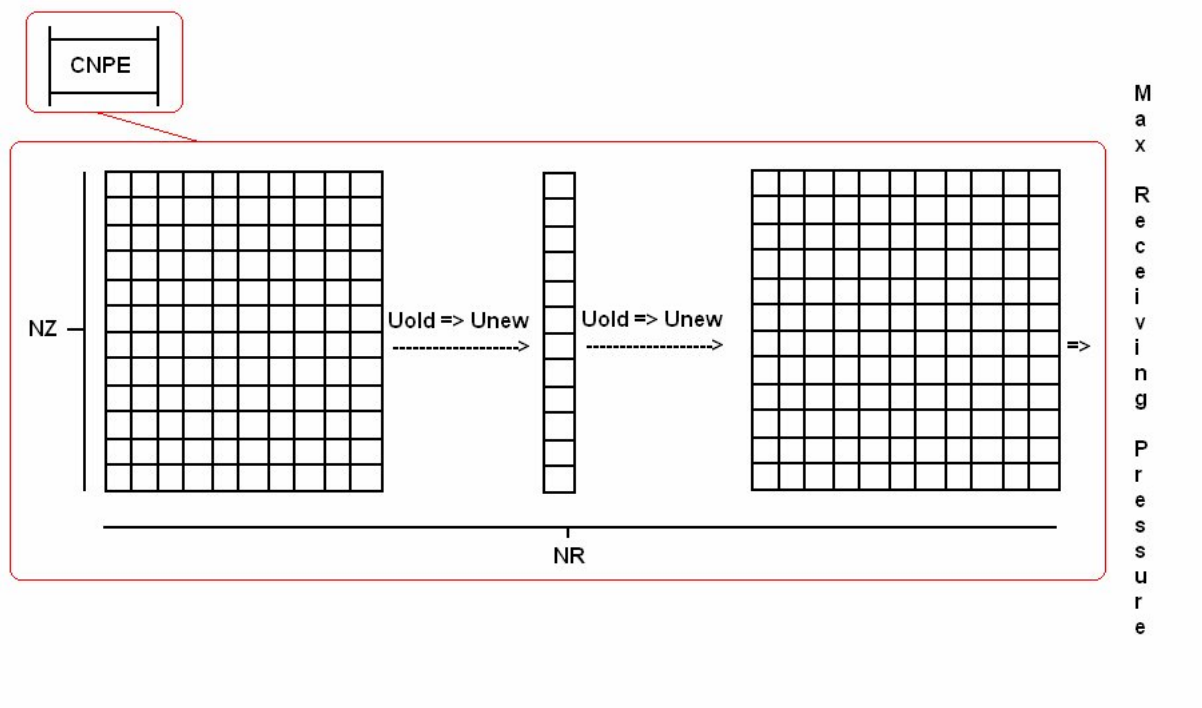

Figure 4.6: The calculation of the Max Receiving Pressure of the receiver.

Step 4: The pressure vector is created for every height as shown in Figure 4.7.

Step 5: The height that causes the minimum sound pressure to the receiver is calculated. Repeating the same process for each length the optimal path is designed as shown in Figure 4.8.

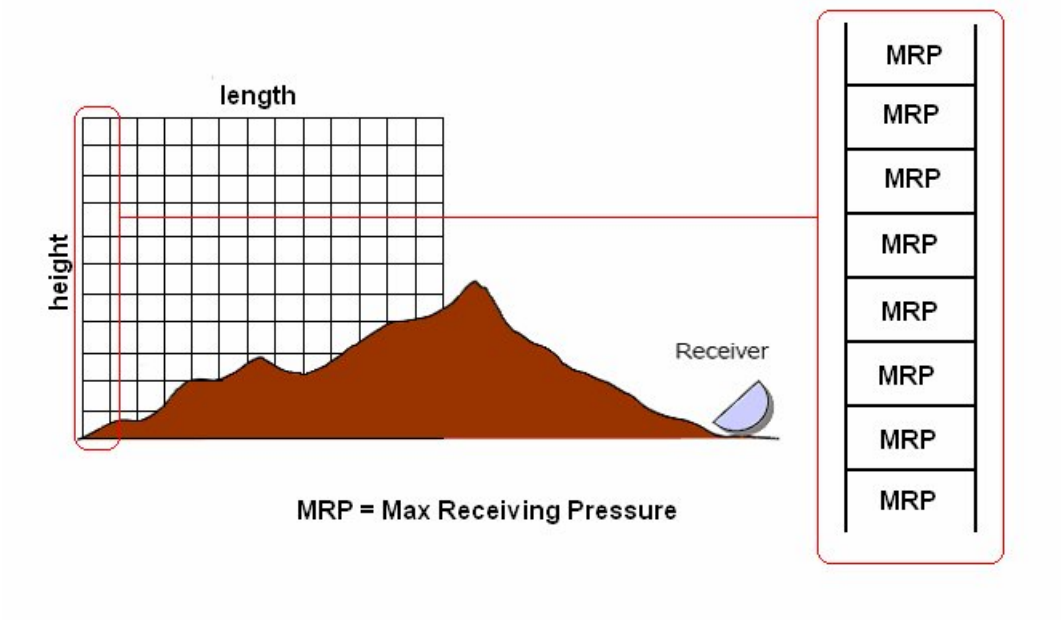

Figure 4.7: The Max Receiving Pressure of the receiver for each height.

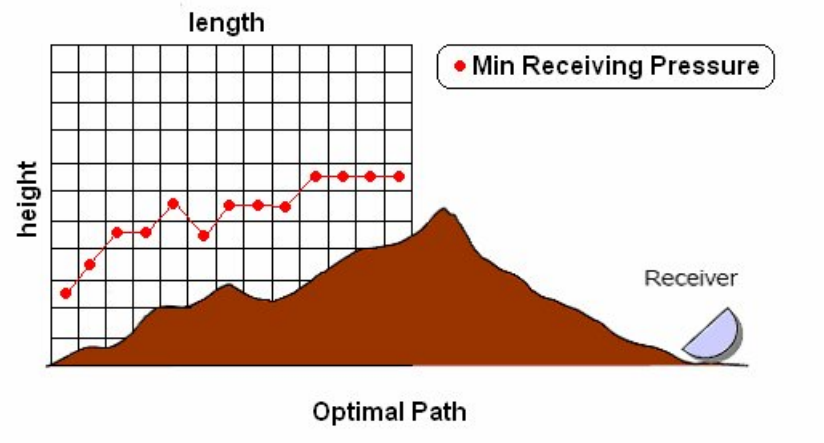

Figure 4.8: The optimal flight path.

Chapter 5

Implementation Of The Glimmer

This chapter describes the process of enabling the Glimmer algorithm to the Cell processor on the Playstation 3. Programming for the CBE processor requires an understanding of parallel programming. Traditional computing platforms contain a single processor, which computes a single thread of control. High-performance computing platforms contain many processors, with potentially many threads of control.

The first, vital step in parallelizing any program is to consider where there might be exploitable concurrency. Time spent analyzing the program and its algorithms and data structures will be repaid many-fold in the implementation and coding phase. In other words, by no means immediately start to code a program to take advantage of this or that parallel programming model. Spend time understanding the data flow, data dependencies, and functional dependencies.

The most important question is: Will the anticipated speedup from parallelizing a program be greater than the effort to parallelize a program, which includes any overhead for synchronizing different tasks or access to shared data? The second question is: Which parts of the program are the most computationally intensive? It is worthwhile to do initial performance analysis on typical data sets, to be sure the hot spots in the program are being targeted.

When you know which parts of the program can benefit from parallelization, you can consider different patterns for breaking down the problem. Ideally, you can identify ways to parallelize the computationally-intensive parts:

- Break down the program into tasks that can execute in parallel.
- Identify data that is local to each subtask.
- Analyze dependencies among tasks.

In this context, a task is a unit of execution that is enough to keep one processor

element busy for a significant amount of time, and that performs enough work to justify any overhead for managing data dependencies. Key elements to examine are:

- Function calls.
- Loops.
- Large data structures that could be operated on in chunks.

This chapter refers with details to explain the overall development flow that was followed in our implementation as the data partitioning procedure, the levels of parallelism, the data transfers and the code optimizations. It also describes some of the problems that were encounter during the development and the solutions that were provided.

5.1 The Programming Model

The programming model that was chosen for our implementation was the function offload model [IBM08b],[A. 07],[et 05]. The function offload model is the quickest way to effectively use the Cell processor with an existing application. In this model, the main application runs on the PPE and calls selected procedures to run on one or more SPEs. In this programming model, the SPEs are used as accelerators for certain types of performance-critical functions, hotspots. This model replaces complex or performance-critical functions invoked by the main application with functions offloaded into one or more SPEs, without changing the main application logic at all. The original performance-critical function is optimized and recompiled for the SPE environment and an SPE-executable program is being created.

Currently, the programmer statically identifies which functions should execute on the PPE and which should be offloaded to SPEs by utilizing separate source and compilation for the PPE and SPE components. It is also programmer's responsibility to manually partition and schedule the work to one or more SPEs. This model was selected because we already had an implementation of the Glimmer algorithm and we just wanted to improve the performance of the algorithm with parallelism.

5.2 The Application Enablement Process

The process of enabling an application on Cell BE can be incremental and iterative [A. 07]. It is incremental in the sense that the hotspots of the application should be moved progressively off the PPE to the SPE. It is iterative as for each hotspot, the optimization can be refined at the SIMD, synchronization and data movement levels until satisfactory levels of performance are obtained.

As for the starting point, a thorough profiling of the application on a general purpose system (PPE is just fine for this) will give all the hotspots that need to be looked at. Then, for each hotspot, we can write a multi-SPE implementation with all the data transfer and synchronization between the PPE and the SPE. Once this first implementation is working, we then turn to the SIMDization and tuning of the SPE code. The last two steps can be repeated in a tight loop until we get a good performance. We can repeat the same process for all the major hotspots. This is shown in Figure 5.1.

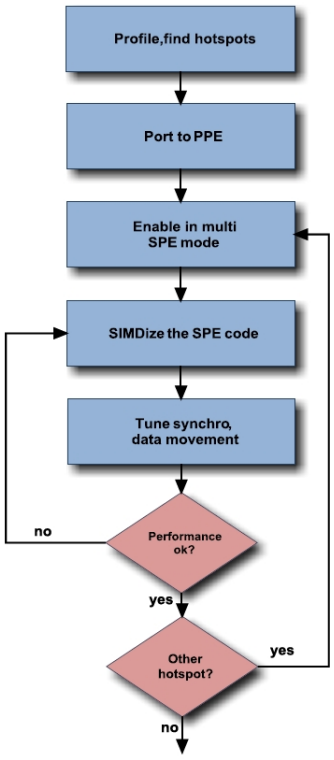

Figure 5.1: Application enablement process.

5.3 Profiling

The first step of the application enablement process is the profiling, as shown in Figure 5.1. The system that we used was a Intel P4 at 2,66Mhz with 1GB memory, operating system Ubuntu 8.04 and Vtune Performance Analyzer 9.0 for Linux. For profiling we have used input datasets of various sizes from Bioperf suite. Below, we present the results of profiling for various datasets.

- Table 5.1 and Table 5.2 show the results of the profiling for dataset CLASS A NC004061.fna.

Program	**Clockticks (%)**	**Clockticks Events**
Build-icm	48.02	5894526000
Glimmer2	16.96	2082146000
Long-orfs	2.72	333250000
Extract	1.02	125302000
Get-Putative	0.09	10664000

Table 5.1: Results of the profiling GLIMMER for Dataset CLASS A NC004061.fna

Build-icm		
Function	**Clockticks (%)**	**Clockticks Events**
count_window	40.78	1940848000
count_base_pairs_with	35.24	1676914000
get_mut_info	18.21	866450000

Table 5.2: Function-wise breakout of Build-icm program

- Table 5.3 and Table 5.4 show the results of the profiling for dataset CLASS B NC003062.fna.

Program	**Clockticks (%)**	**Clockticks Events**
Glimmer2	73.47	83909684000
Build-icm	7.84	8955094000
Long-orfs	1.25	1428976000
Extract	0.36	413230000
Get-Putative	0.14	162626000

Table 5.3: Results of the profiling GLIMMER for Dataset CLASS B NC003062.fna

- Table 5.5 and Table 5.6 show the results of the profiling for dataset CLASS C NC004463.fna.

Glimmer2		
Function	**Clockticks (%)**	**Clockticks Events**
get_prob_of_window1	63.78	35948344000
filter	17.26	9728234000

Table 5.4: Function-wise breakout of Glimmer2 program

Program	**Clockticks (%)**	**Clockticks Events**
Glimmer2	82.20	399329476000
Build-icm	3.99	19389818000
Long-orfs	0.94	4572190000
Extract	0.26	1271682000
Get-Putative	0.11	551862000

Table 5.5: Results of the profiling GLIMMER for Dataset CLASS C NC004463.fna

Glimmer2		
Function	**Clockticks (%)**	**Clockticks Events**
get_prob_of_window1	61.82	159272172000
filter	19.78	50976586000

Table 5.6: Function-wise breakout of Glimmer2 program

Figure 5.2 shows the most time-consuming part of the Glimmer application for various datasets. After observating the results, we decided to offload the function *get_prob_of_window1* to the SPEs. This is a useful fact for an implementation on the Cell processor as significant speed up might be obtained for the Glimmer application by only offloading function *get_prob_of_window1* to the SPUs.

5.4 The Hotspot of the Glimmer

An important thing that makes the function *get_prob_of_window1* the most time-consuming part of the Glimmer application is the fact that it is executed many times, as shown in Table 5.7. From Figure 5.3, we observe that the computation-intensive part of the *get_prob_of_window1* is the *for-loop* statement. For each call of the function, the processor executes seven iretations of the *for-loop*, seven multiplications, seven additions, seventeen loads and many branches. The SPEs are dual-issue processors, and can perform a load, store, shuffle, channel or branch op-

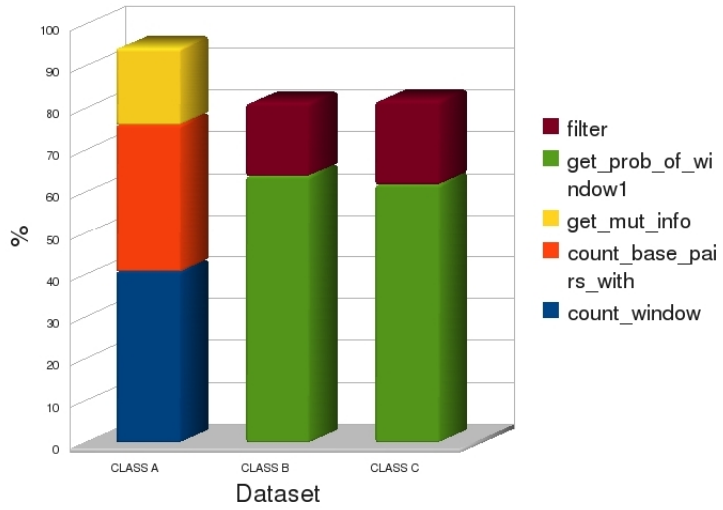

Figure 5.2: Function-Wise Breakout for various datasets.

eration in parallel with a computation. With a 6 cycle load latency to the 256kB local store and software controlled branch prediction, the SPE is highly effective at computation (basically anything with a loop that can be unrolled and interleaved), but not optimally efficient at "gcc/TPCC" (load-compare-add- branch) type codes. From the above it is concluded that the code of the function *get-prob-of-window1* might not to be suitable to offload to the SPUs.

	CLASS A	**CLASS B**	**CLASS C**
TOTAL CALLS	1109432	54549884	259799328

Table 5.7: Total calls of the function get_prob_of_window1

5.5 Data Flow Analysis

After having decided which function would run on the SPEs, a data flow analysis was required to determine the amount of data that had to be transferred to the SPEs LS. The function has three input arguments, a value type of *integer*, a data structure of type *tModel*, an array of type *char* and returns a value type of *double*. The data structure contains an *integer* and a *floating point* array with four elements. The size of the *Delta* array remains the same except for the array *orf*. The number of the *orf* array elements varies from 4281 to 17067 depending on the dataset. Table

```
double get_prob_of_window1 (int start, tModel * Delta, char *orf)
{
  int i, num_node = 0;
  double prob;
  int pos;
  for (i = 0; i < MAX_DEPTH; i++)
  { pos = Delta [num_node] . mut_info_pos;
    if (pos == -1)
    {
      num_node = PARENT (num_node);
      pos = Delta [num_node] . mut_info_pos;
      break;
    }
    switch (Filter (orf [pos+start]))
    {  case 'a': case 'A':
         num_node = (num_node * 4) + 1;
         break;
       case 'c': case 'C':
         num_node = (num_node * 4) + 2;
         break;
       case 'g': case 'G':
         num_node = (num_node * 4) + 3;
         break;
       case 't': case 'T':
         num_node = (num_node * 4) + 4;
         break;
       default:
         fprintf (stderr, "switch error .%c. in
         get_prob_of_window1 start=%d\n", orf [pos+start], start);
         exit (1);}
  }
  pos = Delta[num_node].mut_info_pos;
  if (pos == -1)
  { num_node = PARENT (num_node);
    pos = Delta[num_node] . mut_info_pos;  }
  switch ( Filter (orf [start+INTERVAL-1]))
  {  case 'a': case 'A':
       prob = Delta[num_node].prob[0];
       break;
     case 'c': case 'C':
       prob =  Delta[num_node].prob[1];
       break;
     case 'g': case 'G':
       prob =  Delta[num_node].prob[2];
       break;
     case 't': case 'T':
       prob =  Delta[num_node].prob[3];
       break;
     default:
       fprintf (stderr, "switch error .%c. in 2nd switch of
       get_prob_of_window1 start=%d\n", orf [start+INTERVAL-1], start);
       exit (1);
  }
  return prob;}
```

Figure 5.3: The code of the function get_prob_of_window1.

5.8 shows the total size in Bytes for each input and output of the function and the total size needed by the procedure *get_prob_of_window1*.

From the data flow analysis, we observed that it was unable to execute the function *get-prob_of_window1* at the SPEs due to the limited size of their Local Store (LS). This observation led us to the conclusion that in order to execute the function *get_prob_of_window1* at the SPEs it was necessary to partition the data in order to fit in the Local Store. The data partitioning on the six SPEs of the PS3 has not been possible due to the arbitrary values of the variable *num_node* in the function *get_prob_of_window1*. Thus, the total size of data needed by each SPE was over 441 KBytes, see Table 5.8.

From the code of the function, see Figure 5.3, we observed that the computation-intensive part of the *get_prob_of_window1* is the *for-loop* statement, as described in section 5.4. Thus, we decided to implement this part of the function at the SPEs

and the rest of the code at the PPE. From this assumption, the amount of data at the Local Store is reduced due to the second member of the data structure *tModel*, which is not needed for the execution of the *for-loop* statement. Furthermore, we observed that this part of the code uses 8000 elements from the total 21845 of the *Delta* array and returns a value type of *integer*. Thus, the new data flow analysis is shown in Table 5.9.

Dataset	**Start** (Bytes)	**Delta** (Bytes)	**orf** (Bytes)	**ret_value** (Bytes)	**Total** (Bytes)
CLASS A	4	436900	4281	8	441193
CLASS B	4	436900	8547	8	445459
CLASS C	4	436900	17067	8	453979

Table 5.8: The amount of data is needed by procedure for various datasets.

Dataset	**Start** (Bytes)	**Delta** (Bytes)	**orf** (Bytes)	**ret_value** (Bytes)	**Total** (Bytes)
CLASS A	4	32000	4281	4	36289
CLASS B	4	32000	8547	4	40555
CLASS C	4	32000	17067	4	49075

Table 5.9: The new amount of data is needed by procedure for various datasets.

Figure 5.4 shows us the part of the code where the function *get_prob_of_window1* is called. From this Figure, we have observed that the procedure is called with different arguments *Delta* and namely *Delta0*, *Delta1* and *Delta2*. Thus, we need to transfer two more arrays type of *integer* to the Local Store. Table 5.10 shows the total amount of data that we must transfer to the Local Store.

Dataset	**Start** (Bytes)	**Delta0,1,2** (Bytes)	**orf** (Bytes)	**ret_value** (Bytes)	**Total** (Bytes)
CLASS A	4	3x32000	4281	4	100289
CLASS B	4	3x32000	8547	4	104555
CLASS C	4	3x32000	17067	4	113075

Table 5.10: The size of data is needed to store to Local Store.

```
for (start = 0; stop < length; start++, stop++)
  {
    switch (start % 3)
    {
       case 0:
         score += log (get_prob_of_window1 (start, Delta0, orf));
         break;
       case 1:
         score += log (get_prob_of_window1 (start, Delta1, orf));
         break;
       case 2:
         score += log (get_prob_of_window1 (start, Delta2, orf));
         break;
       default:
         fprintf (stderr, "ERROR in switch in Fast Eval\n");
         exit (1);
    }
  }
```

Figure 5.4: The code where the function *get_prob_of_window1* is called.

5.6 Development Stages

This section refers with details the overall development flow that was followed in our implementation as the levels of parallelism, the data transfers and the code optimizations. The development stages were chosen prior to the beginning of the code development and later revised. The purpose of introducing these stages was to provide a way to monitor progress of the project and seemed like a practical way to develop the solution. A detail development flow chart is shown on Figure 5.5.

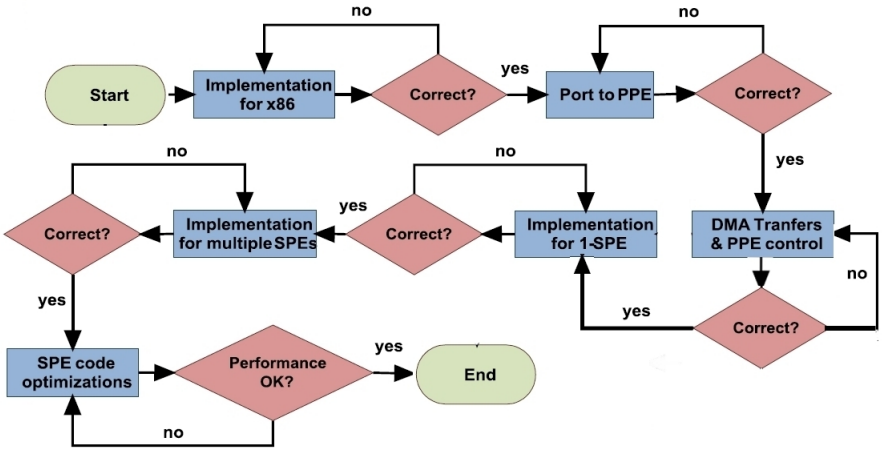

Figure 5.5: The development flow chart.

For some or all of the development stages, we used an interative development

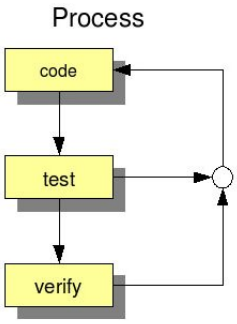

Figure 5.6: An interative development process.

process that includes the stages *code*,*test* and *verify*, as shown in Figure 5.6. The *code* stage represents the actual physical writing of code. This may include the implementation of a formal program specification or coding done without any documentation other than code directly from the programmers brain. The *test* stage represents the testing of code that has been compiled. For Cell BE applications, testing is still a necessary and critical step in producing well performing applications. The test process for Cell BE application will undoubtably require testing for code performance. Testing is normally focused on code that runs on the Cell BE SPU and the focus is more on making sure all code paths are processed and the accuracy of the output. The *verify* stage represents an important aspect of many Cell BE applications. The need for verification of code that runs on the SPU is of special importance. Code being ported from other processor platforms presents a unique oppurtunity. The verification of Cell BE programs is done by comparing output from the original application for accuracy.

All the code has been developed with the use of IBM's Cell SDK 3.1 and Full-System Simulator. For the final stage of the development a PS3 was used with Yellow Dog Linux 6.0 OS.

5.6.1 Implementation on x86 Architecture

At this stage we used the code of the GLIMMER algorithm from Bioperf suite. We split the code of the *get_prob_of_window1* in two parts, as described in section 5.5. Furthermore, we used 8000 elements for the array *Delta* instead of 21845 and a maximum of 8000 elements for the array orf. The goal from this stage was to prove that the basic algorithm produced correct results and would meet our requirements for running on an SPU.

The implementation was run on P4 machine at 3.0 GHz with memory 512 MB. The operating system was *fedora 9*, the code was developed in the Linux based editor *Geany 0.16* and the debugging of the compiled code was made with *GNU*

gdb debugger.

5.6.2 Port to PPE

At this stage, the x86 GLIMMER implementation was ported to PowerPC hardware (PPE) by recompiling the x86 source code with *ppu-gcc* compiler and the necessairy makefiles for the Cell processor [IBM08b], [IBM08a], [Sca09]. This porting to the PPE was made to confirm the correct execution of x86 implementation on the PPE. The PowerPC version of the code performed much slower than the x86 version of the code. This is due to the difference in the relative power of the PPU portion of the Cell BE compared to a fairly high end x86 CPU on which the x86 code was developed. Even though this code was simple and single threaded, the x86 processor core used for development is a much more power processor core than the PPU. After this step the application was running on PPU and the next step was to begin offloading the functions to the SPEs.

5.6.3 PPE control

The PPU's most important role is managing the Synergistic Processor Elements (SPEs) [IBM08b], [A. 07], [IBM08a], [Sca09]. Below, we describe two implementation approaches which we followed in our design.

1st Implementation

In our first implementation, the function offload model implemented using stubs as proxies. A method stub, or simply stub, is a small piece of code used to stand in for some other code. The stub or proxy acts as a local surrogate for the remote procedure, hiding the details of server communication. The main code on the PPE contains a stub for each remote procedure on the SPEs. Each procedure on an SPE has a stub that takes care of running the procedure and communicating with the PPE.

The stub code, together with the runtime code, controls the execution, data transfer, and program coordination between the PPE and SPE during program execution. A procedure is loaded onto an SPE only once, and the program on the PPE can then make multiple calls to that procedure without having to reload it.

When the program on the PPE calls a remote procedure, it actually calls that procedure's stub located on the PPE. The stub code initializes the SPE with the necessary data and code, packs the procedure's parameters, and sends a mailbox message to the SPE to start its stub procedure. Figure 5.7 shows an example of a program using this method. Converting a PPE program to use RPCs requires the following steps:

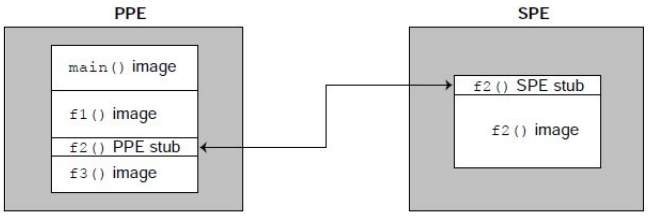

Figure 5.7: Function-Offload (or RPC) Model with stubs.

- Produce an Interface Definition Language file (IDL file) [Cen05]. The IDL file defines the interface between the main program on the PPE and the remote procedures on the SPEs. This specification of the program's remote procedures is defined using the Cell Broadband Engine's IDL.
- Process the IDL file using the IDL compiler. The IDL compiler produces three files to be used in the program-compilation phase. One file is a C header file and the other two are C source files—one to be compiled with the PPE program and the other to be compiled with the SPE procedures. The generated header file contains the declarations and data structures required by both stubs for data transfer between the PPE and the SPE.
- Compile the PPE and SPE code into separate programs. The PPE code must be compiled with the PPE stub code produced by the IDL compiler, and the SPE code must be compiled with its stub code, thus producing two program files.

Figure 5.8 shows the production flow for producing an application. Boxes with bold borders represent source-code files.

The *RPC* model is implemented with the use of specific library calls provided by the RPC runtime management library. PPE requests for SPE executions are represented in the RPC runtime code as task structures. As each remote procedure is invoked, a new task is created and placed in a task queue. Each SPE has its own task queue, so having the procedure loaded on multiple SPEs does not increase the size of the queue, it only enables the procedure to execute on multiple SPEs at the same time. The number of slots in a queue is fixed. If the PPE requests a remote procedure call and the queue is full, the application must wait for a free slot in the queue. When a remote procedure call returns, a slot in the queue becomes available.

On invocation of a remote procedure call, the PPE program can either wait for the procedure to return (synchronous execution), or continue processing and synchronize with the procedure later (asynchronous execution). Whether a remote procedure is synchronous or asynchronous is specified by the procedure's definition in the IDL file.

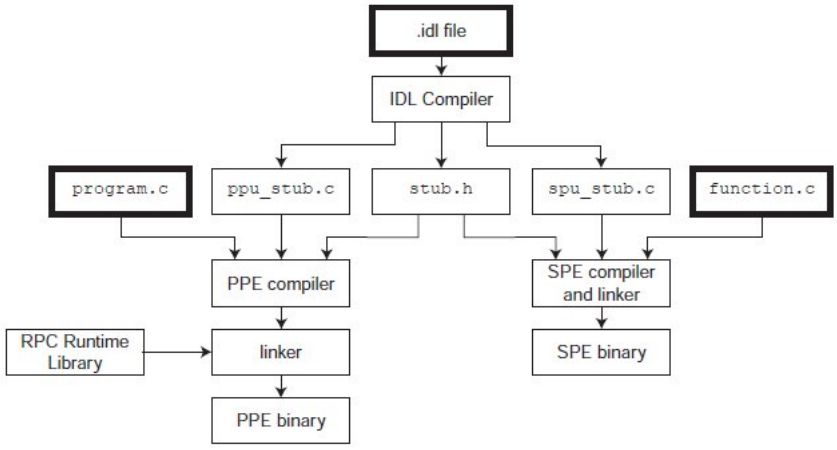

Figure 5.8: Production Flow for Function Offload (or RPC) Model.

All remote procedure calls are RPC functions that return a value of type idl_id_t. The value returned is unique, and identifies that instance of the procedure call. This value is used by the PPE program to synchronize with asynchronous procedure calls. There are three synchronization functions used for this purpose:

- *int idl_join_foo(idl_id_t id)*. This function blocks until the remote procedure with the idl_id_t value of id completes execution on the SPE. When the SPE function finishes, it sends a signal to the PPE.

- *int idl_poll_foo(idl_id_t id)*. This function polls to see if the SPE remote procedure with the idl_id_t value of id has finished.

- *int idl_join_all_foo( )*. This function blocks and waits for all instances of remote procedure foo to complete.

In our implementation, before the first function-call of the *get_prob_of_the_window1*, we introduce on the PPE a stub for remote procedure on the SPEs which transfers the data from main memory to the Local Store of the SPEs. When the program on the PPE calls a remote procedure, it actually calls that procedure's stub located on the PPE. The stub code initializes the SPE with the necessary data and code, packs the procedure's parameters, and sends a mailbox message to the SPE to start its stub procedure. At the first time of the SPE's program execution, the arrays *Delta0*, *Delta1* and *Delta2* are transferred from the main memory to the Local Store of each SPE. This happens only once as the data of these arrays do not change during the execution of the program. When the SPE code has been executed the program returns to the PPE side.

Furthermore, for each function-call of the *get_prob_of_the_window1* in the loop, we introduce on the PPE a stub for remote procedure on the SPEs which transfers the proper data to Local Store and executes the function *get_prob_of_the_window1*. When the SPE code has been executed, the program returns to the PPE side and creates the result array until the next function-call, as shown in Figure 5.9.

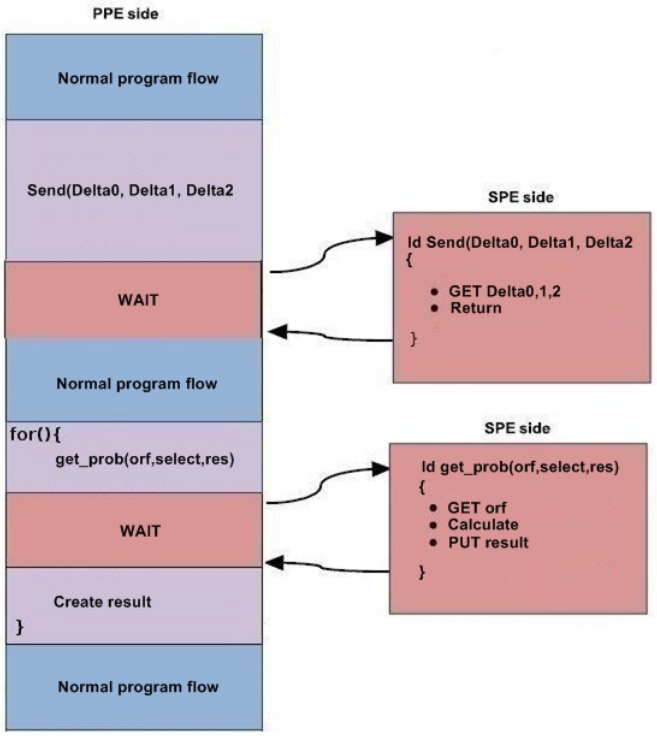

Figure 5.9: Overall scheduling process for 1st implementation of GLIMMER.

It has been proved that this implementation did not work well when we applied the optimization PPU_unroll to reduce the overhead from the scheduling process. For more details see in **Section 5.7**.

2nd Implementation

In this implementation, the scheduling and control of the threads are based on a *fork-join* model, as shown in Figure 5.10. *Fork-join* is a model where a master execution thread (PPE) calls (fork) multiple parallel execution SPEs threads and waits for their completion (join). While the threads are running the PPE can either continue execution (*asynchronous* execution) or can wait the SPE threads to finish (*synchronous* execution).

The *fork-join* model is implemented with the use of specific library calls provided

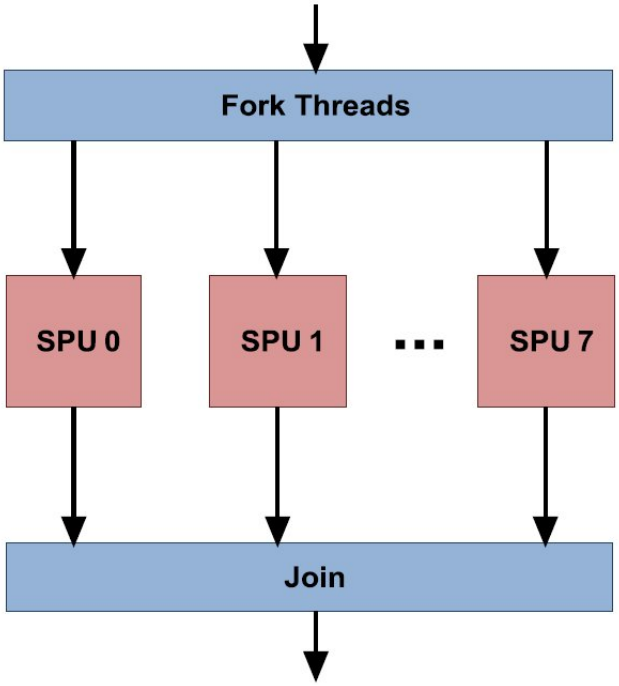

Figure 5.10: Fork-join model.

by the SPE runtime management library (*libspe2*) [IBM08c], [IBM08d]. A PPE module starts an SPE module running by creating a thread on the SPE, using the *spe_context_create*, *spe_program_load*, and *spe_context_run* library calls, as shown in Figure 5.11.

The *libspe2* functions don't access SPE resources directly but they operate on data structure that represent aspects of an SPE's operation. PPU code accesses SPEs through data structures called *contexts*, and each *context* represents a single SPE. This data structure contains fields that access the SPE's processing unit, memory, and communication resources.

The *spe_context_create* call creates a *context* for the SPE thread which contains the persistent information about a logical SPE. Before being able to run a SPE *context*, a SPE program has to be loaded into the *context* using the *spe_program_load* subroutine. A SPE *context* is executed on a physical SPE by calling the *spe_context_run* function.

The total scheduling procedure consumes a significant amount of time compared with the total execution time. Some overhead from the scheduling process is unavoidable but it was managed to be reduced as much as possible. In order to reduce the overhead the context was created only once and the program was loaded to the SPEs only once, when it was possible.

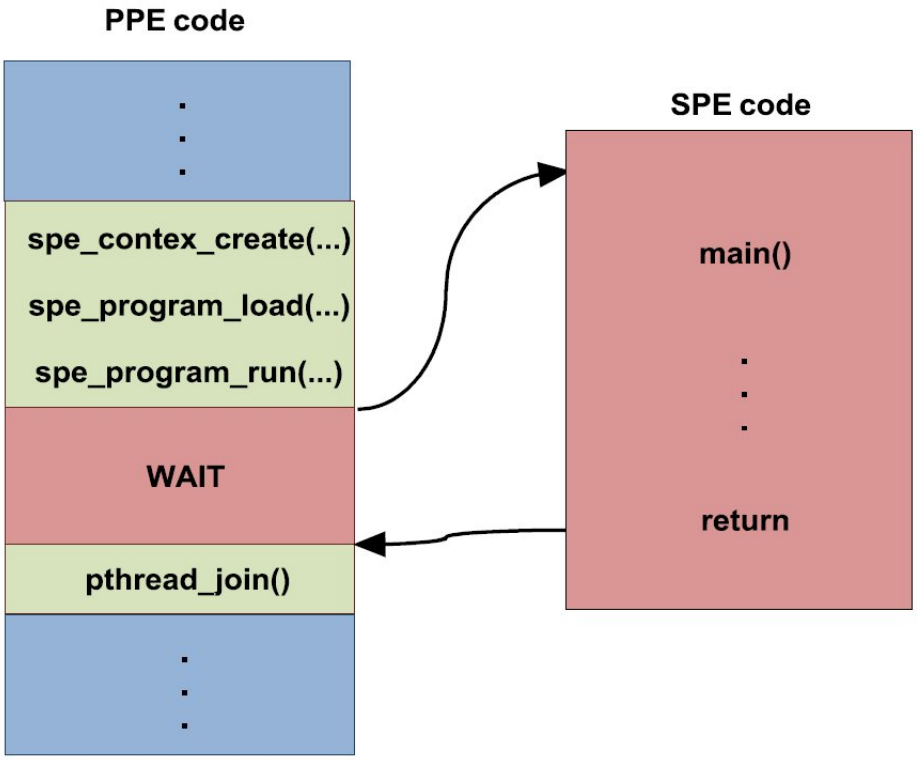

Figure 5.11: PPE control.

Before the first function-call of the *get_prob_of_the_window1*, we introduce a series of scheduling operations that initiates the execution of the SPE code, as shown in Figure 5.12. More specifically, the program creates the control block, creates context, loads program to the SPEs, creates SPE threads and runs SPE threads. At the first time of the SPE's program execution, the data of the arrays *Delta0*, *Delta1* and *Delta2* is transferred from the main memory to the Local Store of each SPE. This happens only once as the data of these arrays do not change during the execution of the program. When the SPE code has been executed the program returns to the PPE side.

Since the SPEs were running the same program it wasn't necessary to create new context and load the program each time. Thus, for each function-call of the *get_prob_of_the_window1* in the loop the program updates the *control block*, which is different for every function call and executes the SPE thread, as shown in Figure 5.12. When the SPE code has been executed, the program returns to the PPE side and creates the result array until the next function-call. With this approach, the creation of the context and the program loading was done outside the loop to avoid their unnecessary repeat.

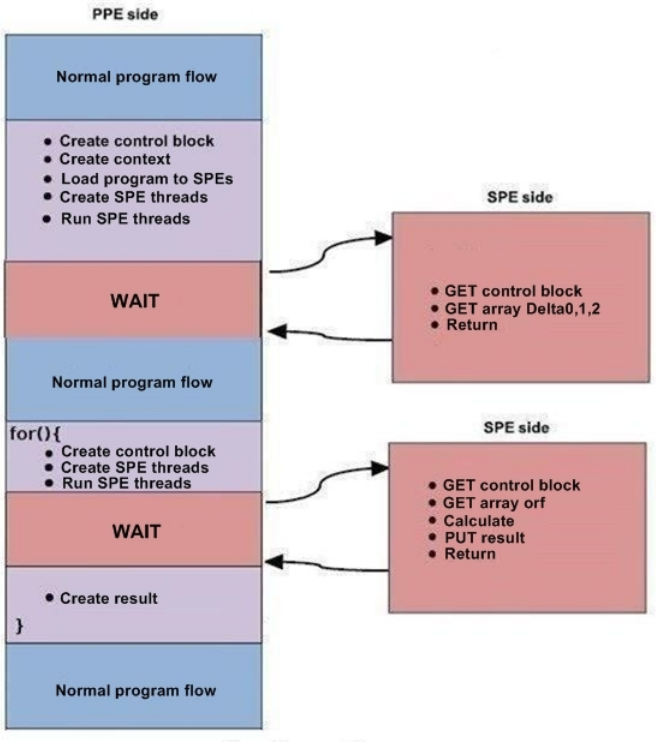

Figure 5.12: Overall scheduling process for GLIMMER.

5.6.4 DMA Transfer

Being a multiprocessor system on a chip, the CBE processor has many attributes of a shared-memory system. The PPE and all SPEs have coherent access to main storage. But the CBE processor is not a traditional shared-memory multiprocessor. For example, an SPE can execute programs and directly load and store data only from and to its private local store (LS). In a traditional shared-memory multiprocessor, data communication and synchronization among processors happen at least partially as a side-effect of the fact that all processors use the same shared memory.

Since SPEs lack shared memory, they must communicate explicitly with other entities in the system using DMA (Direct Memory Access) communication mechanism [IBM08b], [IBM08a], [A. 07], [Sca09]. This mechanism is implemented and controlled by the SPE's MFC (Memory Flow Controller). The MFC is a coprocessor specifically designed to send and receive data on the EIB. The advantage of performing data transfer outside the SPU is that the MFC can perform its job without interfering with SPU's regular operation.

An MFC supports naturally aligned DMA transfer sizes of 1, 2, 4, 8, and 16 bytes and multiples of 16 bytes up to 16 KB of data between an LS of and main storage. There are two categories of DMA commands the **mfc_put** and the **mfc_get**; the **mfc_put** commands move data from LS to main storage and the **mfc_get** commands move data from main storage to LS.

Cell processor supports two kinds of DMA transfers, PPE initiated and SPE initiated, in our implementation only SPE initiated DMA transfers were used. This choice was made because there are eight times more SPEs than PPEs and the number of cycles to initiate a transfer from the SPEs is smaller than the number of cycles to initiate the same transfer from the PPE.

The data transferring process can be described from the following steps:

- SPU needs data.

1. SPU initiates DMA request for data.
2. DMA requests data from the memory.
3. Data is copied to the LS.
4. SPU can access data from the local store.

- SPU operates on data then copies data from local store back to memory in a similar process

Figure 5.13 describes the total procedure for data transfer to and from the LS. The circled numbers shown in the figure correspond to the steps of the data

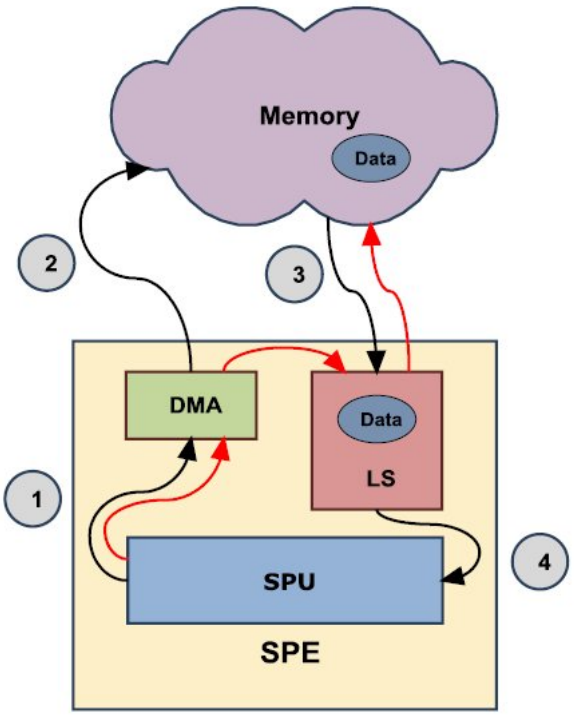

Figure 5.13: Data Transfer from and to LS.

transferring process as it was defined above. The black arrows are for data transfer from the main memory to the LS and the red arrows are for the opposite process, data transfers from the LS to the main memory.

In our implementation, we need to transfer from main memory to the Local Store three arrays type of *integer* and an array type of *char*. Furthermore, we need to transfer the result fron Local Store to the main memory. Each array type of *integer* has 8000 elements, the array type of *char* has maximum 8000 elements and the result is a variable type of *integer*. Thus, we used 2 *mfc_get* for each array type of *integer*, 1 *mfc_get* for *orf* array and 1 *mfc_put* for the result. Table 5.11 shows us a summary of the required DMA transfers.

Data	**Size (KBytes)**	**DMA-get (KBytes)**	**DMA-put (KBytes)**
Delta0	32	2x16	-
Delta1	32	2x16	-
Delta2	32	2x16	-
orf	8	1x8	-
num_node	0.004	-	1x0.004

Table 5.11: DMA Transfers.

5.6.5 Implementation with One and Multiple SPEs

At this stage, we take the PPU code of the function *get_prob_of_window1* and more specifically the *for-loop* statement and run it on a single SPU. This involved restructuring the code by adding calls to the SDK to load an SPU module. More code was added to implement a simple DMA model for streaming data into the SPU and streaming it back to main store memory. At this stage there is no parallelism since only one SPE it was used.

The design was extended on two and finally on six SPEs, the procedure remains the same as in the case of one SPE. The advantage now is that the total number of iterations that are required, as shown in Figure 5.4, is distributed on multiple SPEs. In each iteration, the SPE should known informations about which data must fetch, where should store the result and which array *Delta* should use. This information is passed once at each SPE through a structure called *control block*. The *control block* contains information such as, arrays addresses, result address and information related to the selection of the suitable *Delta* that should be used.

5.6.6 Code Optimizations

Unlike conventional processors, near theoretical-maximum performance can be achieved for real applications on the Cell Broadband Engine processor. However, we must be aware of the architectural characteristics of the processor to achieve optimal performance [D. 06], [A. 07], [IBM08b], [IBM08a]. These characteristics include multiple heterogeneous execution units, Single Instruction Multiple Data (SIMD) processing engines, limited local store, software managed cache, memory access latencies, dual instruction issue rules, both large and wide register files, quad-word memory accesses, branch prediction, and synchronization facilities. Below, we describe the code optimizations that we applied to improve the performance.

Reduction overhead from the scheduling process

As described in section 5.6.3, a SPE thread executes only once the function *get_prob_of_window1*. The PPU code, at this point, was gradually unrolled until the improvement was not important. Finally, the code was unrolled 624 times that means a SPE thread executes 624 times the function *get_prob_of_window1*. This approach has resulted in the reduction of the overhead from the scheduling process compared with the total execution time.

Branch Elimination

The SPU hardware assumes linear instruction flow and produces no stall penalties from sequential instruction execution. A branch instruction has the potential of disrupting the assumed sequential flow. Correctly predicted branches execute in one cycle, but a mispredicted branch (conditional or unconditional) incurs a penalty of approximately 18-19 cycles. Considering the typical SPU instruction latency of two-to-seven cycles, mispredicted branches can seriously degrade program performance. Branches also create scheduling barriers, reducing the opportunity of for dual issue and covering up dependency stalls.

The most effective means of reducing the impact of branches is to eliminate them using two primary methods:

- Loop Unrolling
- Function Inline

Loop-unrolling technique can be used to increase the size of basic blocks (sequences of consecutive instructions without branches), which increases scheduling opportunities. It eliminates branches by decreasing the number of loop iterations. The compiler SPU by default has the optimization level set to three, *-O3*, which does some loop-unrolling. Furthermore, we made extra loop-unrolling that did not

increase important the performance. The SPE code was gradually unrolled until the code fit in the Local Store. Finally, the code was unrolled seventy-five times (75x).

Function-inlining is another technique that can be used to increase the size of basic blocks (sequences of consecutive instructions without branches). This technique eliminates the two branches associated with function-call linkage - the branch for function-call entry and the branch indirect for function-call return.

The first step of the optimizations was the use of the compiler in such way to produce optimized code. A specific flag was used, the *-Winline*, in the makefiles to force the compiler produce code with function inlining. The compiler by default has the optimization level set to three, *-O3*, which does some function inlining so the extra inlining that was applied didn't had significant increase in performance.

5.7 Software Tools problems

During the development process, apart from the design problems that were described in the previous paragraph, many other problems came up, mostly related with the software tools. In this section a list of these problems is presented, as well as how each problem solved or avoided.

Initially, we used the IBM SDK for Multicore Acceleration Version 2.1 (Development Tools + Full-System Simulator) on server for compilation. The source code compilation and execution with SDK 2.1 were working fine untill the PPU_unroll optimization was applied in the first implementation of the PPE control. The problem was that the execution of Glimmer was very slow as the processor remained idle unduly nevertheless the application returned correct results after considerable time. The solution of the problem was to setup SDK 3.1 on our P4 host machine with OS fedora 9. In the SDK 3.1 the IDL tools was not included, so we implemented the PPE control with Fork-Join model.

The Full-System Simulator application that is included in the SDK is a very demanding application and especially the cycle-mode was extremely slow on the host machine. This was delaying the development process and it was necessary to avoid it, the solution was to execute the applications directly on hardware and measure the performance in different way. The only available hardware was a PS3 and to be able executing applications on the PS3 the installation of an OS was required.

The Yellow Dog Linux 6.0 (YDL) was installed on the PS3, the installation procedure was done by following the detailed guide for YDL installation. The only conflict during the installation was the monitor configuration, because PS3 normally is connected on HDMI monitor, the installation was not working until the proper settings for the monitor were chosen.

Chapter 6

Implementation Of the CNPE algorithm

This chapter describes the process of enabling the CNPE algorithm to the Cell processor on the Playstation 3. It refers with details to explain the overall development flow that was followed in our implementation as the data partitioning procedure, the levels of parallelism, the data transfers and the code optimizations.

6.1 The Programming Model

The programming model that was chosen for our implementation was the function offload model [IBM08b],[A. 07],[et 05]. See more details **Chapter 5, section 5.1**.

6.2 The Application Enablement Process

See more details **Chapter 5, section 5.2**.

6.3 Profiling

The first step of the application enablement process is the profiling, as shown in Figure 5.1. The system that we used was a Intel P4 at 2,66Mhz with 1GB memory, operating system Ubuntu 8.04 and Vtune Performance Analyzer 9.0 for Linux. Below, we present the results of profiling for one and twelve calls of the *CNPE* function.

- Table 6.1 and Table 6.2 show the results of the profiling for one call of the *CNPE* function.

Program	Clockticks (%)	Clockticks Events
CNPE	66.03	3777722000
Others	33.97	1888861000

Table 6.1: Results of the profiling CNPE for one call of the *CNPE* function

CNPE		
Function	**Clockticks (%)**	**Clockticks Events**
Znorm	23.47	306590000
ZdivZ	17.96	234608000
rhs	15.10	197284000
trid	12.04	157294000
amat	4.69	53320000
ZsubZmulZ	3.88	7998000

Table 6.2: Function-wise breakout of CNPE program

- Table 6.3 and Table 6.4 show the results of the profiling for twelve calls of the *CNPE* function.

Program	Clockticks (%)	Clockticks Events
CNPE	88.93	41706904000
Others	11.07	5154785888

Table 6.3: Results of the profiling CNPE for twelve call of the *CNPE* function

Figure 6.1 shows the most time-consuming part of the CNPE application. After observating the results, we decided to offload the function *CNPE* to the SPEs. This is a useful fact for an implementation on the Cell processor as significant speed up might be obtained for the CNPE application by only offloading function *CNPE* to the SPUs.

6.4 The Hotspot of the CNPE

The most time-consuming part of the CNPE application is the *CNPE* function, as described in previous section. The following analysis explains the reasons that make computation-intensive the *CNPE* function for the processors. Figure 6.2 shows the

CNPE		
Function	**Clockticks (%)**	**Clockticks Events**
Znorm	28.14	3617762000
rhs	19.18	2466050000
ZdivZ	18.77	2412730000
trid	10.99	1412980000
ZsubZmulZ	5.08	767808000
amat	4.79	653170000

Table 6.4: Function-wise breakout of CNPE program

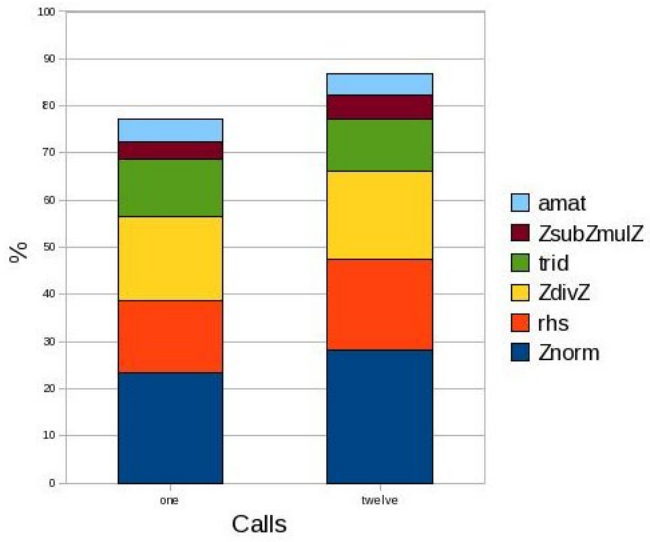

Figure 6.1: Function-Wise Breakout for CNPE.

code of the *CNPE* function. From Figure 6.2, we observe that the computation-intensive part of the *CNPE* function is the *for-loop* statement. This *for-loop* executes (NR-1) iterations, in each iretation it calls three time-consuming functions *rhs*, *amat* and *trid*.

Amat function contains a *for-loop* statement, as shown in Figure 6.3. This *for-loop* is executed (NZ-1) times, in each iteration a number of the complex operations (multiplications, additions, substractions) is executed.

Rhs function contains two *for-loop* statement, as shown in Figure 6.4. The first *for-loop* is the most time-consuming part of the *rhs*. This *for-loop* is executed (NZ-2) times, in each iteration a number of the complex operations (multiplications, additions, substractions) is executed.

Trid contains two *for-loop* statement, as shown in Figure 6.5. The first *for-loop*

```
floatType CNPE( floatType zs )
{ int i, j;
  floatType *OutSol=contour;
  floatType cc1=pi/(2.*zs);
  complex *Unew, *Uold;
  Uold=U1, Unew=U2;
  InsertInitialCond( zs, Uold);
  for( i=1; i<NR; i++)
  { bc( Uold);
    rhs( i, Uold);
    bc( Uold);
    amat( i, al, ad, au);
    al[NZ-1].re=al[NZ-1].im=0., au[0].re=au[0].im=0.;
    trid( al, ad, au, Uold, Unew);
    bc( Unew);
    if( Uold==U1) Uold=U2, Unew=U1;
    else          Uold=U1, Unew=U2;
    for( j=0; j<NZ; OutSol[j]=Znorm( &Uold[j]), j++);
    OutSol+=NZ;
  }
  PrintPressureField( );
  return FindMaxReceivingPressure();
}
```

Figure 6.2: The code of the function CNPE.

```
void amat( int i, complex *al, complex *ad, complex *au)
{
  int j;
  complex cca, ccb, ccc, ccb2;
  floatType hr =.5, hr2 = hr*hr;
  for( j=0; j<NZ; j++)
  {
       cca.re= hr  * ca[i][j].re,  cca.im= hr  * ca[i][j].im;
    ccb.re= hr2 * cb[i][j].re,  ccb.im= hr2 * cb[i][j].im;
    ccc.re= hr  * cc[i][j].re,  ccc.im= hr  * cc[i][j].im;
    al[j].re=     ccb.re -      ccc.re,  al[j].im=   ccb.im -      ccc.im;
    ad[j].re= 1.- cca.re + 2. * ccc.re,  ad[j].im= - cca.im + 2. * ccc.im;
    au[j].re=   - ccb.re -      ccc.re,  au[j].im= - ccb.im -      ccc.im;
    }
}
```

Figure 6.3: The code of the function Amat.

is the most time-consuming part of the *trid*. This *for-loop* is executed (NZ-1) times, in each iteration a number of the complex operations (divisions, multiplications, additions, substractions) is executed.

The functions *ZmulZ* and *ZdivZ* calculate the multiplication and the division of the complex numbers. The *Znorm* calculates the magnitude of the complex number and the *ZsubZmulZ* calculates the multiplications and substractions of the complex numbers . Another important thing that makes even harder the *CNPE* function is that the complex numbers have single-precision floating-point real and imaginary part.

```
void rhs( int i, complex *u)
{ int im1=i-1, NZm1=NZ-1, j;
  floatType  hr=.5,  hr2=hr*hr;
  complex bl, bm, bu, c1, c2,  cca, ccb, ccc, ccb2, cmp1, cmp2, cmp3;

  bm.re=1.+ ca[im1][0].re*hr - 2.*cc[im1][0].re*hr,  bm.im= ca[im1][0].im*hr  - 2.*cc[im1][0].im*hr;
  bu.re=    cb[im1][0].re*hr2 +   cc[im1][0].re*hr,  bu.im= cb[im1][0].im*hr2 +    cc[im1][0].im*hr;

  ZmulZ( bm, u[0], cmp1); ZmulZ( bu, u[1], cmp2);
  uu[0].re=cmp1.re+cmp2.re,  uu[0].im=cmp1.im+cmp2.im;

  for( j=1; j<NZm1; j++)
  {
    cca.re = ca[im1][j].re*hr,  cca.im = ca[im1][j].im*hr;
    ccb.re = cb[im1][j].re*hr2, ccb.im = cb[im1][j].im*hr2;
    ccc.re = cc[im1][j].re*hr,  ccc.im = cc[im1][j].im*hr;
    bl.re  =   - ccb.re +    ccc.re,  bl.im= - ccb.im +    ccc.im;
    bm.re  = 1.+ cca.re - 2.*ccc.re,  bm.im=   cca.im - 2.*ccc.im;
    bu.re  =     ccb.re +    ccc.re,  bu.im=   ccb.im +    ccc.im;
    ZmulZ( bl, u[j-1], cmp1), ZmulZ( bm, u[j], cmp2), ZmulZ( bu, u[j+1], cmp3);
    uu[j].re=cmp1.re+cmp2.re+cmp3.re,   uu[j].im=cmp1.im+cmp2.im+cmp3.im;
  }
  bl.re =    - cb[im1][NZ-1].re*hr2 +    cc[im1][NZ-1].re*hr, bl.im= - cb[im1][NZ-1].im*hr2 +    cc[im1][NZ-1].im*hr;
  bm.re = 1. + ca[im1][NZ-1].re*hr  - 2.*cc[im1][NZ-1].re*hr, bm.im=   ca[im1][NZ-1].im*hr  - 2.*cc[im1][NZ-1].im*hr;
  ZmulZ( bl, u[NZ-2], cmp1), ZmulZ( bm, u[NZ-1], cmp2);
  uu[NZ-1].re=cmp1.re+cmp2.re,  uu[j].im=cmp1.im+cmp2.im;
  for( j=0; j<NZ; u[j].re = uu[j].re, u[j].im = uu[j].im, j++);
}
```

Figure 6.4: The code of the function Rhs.

```
void trid( complex *al, complex *am, complex *au, complex *b, complex *u)
{ static int k;
  complex bb, cmp1;
  bb.re = am[0].re, bb.im = am[0].im;
  if( Znorm( &bb)< 1.e-5) printf("BAD Matrix cond 0\n");
  ZdivZ( &b[0], &bb, &u[0]);
  for( k=1; k<NZ; k++)
  { ZdivZ( &au[k-1], &bb, &c[k]);
    ZsubZmulZ( &am[k], &al[k], &c[k], &bb);
    if( Znorm( &bb)< 1.e-5) printf("BAD Matrix cond %d\n", k);
    ZsubZmulZ( &b[k], &al[k], &u[k-1], &cmp1); ZdivZ( &cmp1, &bb, &u[k]);
  }
  for( k--; k>0; ZmulZ( c[k], u[k], cmp1), k--, u[k].re -= cmp1.re, u[k].im -= cmp1.im);
}
```

Figure 6.5: The code of the function Trid.

6.5 Dataflow Analysis

After the function that would run on SPEs was chosen, a data flow analysis was required to determine the amount of data that had to be transferred to the SPEs LS. The function accepts a *float* as input and returns a value type of *float*. Furthermore, the function *CNPE* needs the following data:

- The matrices *ca[NR][NZ]*, *cb[NR][NZ]* and *cc[NR][NZ]* of complex type.
- The arrays *al[NZ]*, *ad[NZ]*, *au[NZ]*, *uu[NZ]* and *c[NZ]* of complex type.
- The arrays *U1[NZ]*, *U2[NZ]*, *contour[NZ]*, *zz[NZ]* of *float* type.

where NR=1005 and NZ=501.

The matrices *ca[NR][NZ]*, *cb[NR][NZ]*, *cc[NR][NZ]* and the array *zz[NZ]* must be transferred from the main memory to the Local Store each SPE. The array *contour[NZ]* must be transferred from the Local Store to main memory. The remaining data is created during the execution of the*CNPE* function. Table 6.5 shows the amount of Local Store space needed.

Data	**Size (Bytes)**
ca,cb,cc	12084120
al,ad,au,uu,c	12024
U1,U2,contour,zz	8016
Total Size	**12104160**

Table 6.5: The amount of Local Store space needed.

From Table 6.5, we observed that was unable to execute the function *CNPE* at the SPEs due to the limited size of their Local Store (LS). This observation led us to the conclusion that in order to execute the function *CNPE* at the SPEs it was necessary to partition the data in order to fit in the Local Store of the SPEs. More specifically, it was necessary to partition the data *ca*, *cb* and *cc*. For more details, see **section 6.6.4** .

6.6 Development Stages

This section refers with details the overall development flow that was followed in our implementation as the levels of parallelism, the data transfers and the code optimizations. The development stages were arrived at prior to the beginning of the code development and later revised. The purpose of introducing these stages

was to provide a way to monitor progress of the project and seemed like a practical way to develop the solution. A detail development flow chart is shown on Figure 6.6. For some or all of the development stages, we used an interative development process that includes the stages code,test and verify, as shown in Figure 6.7. For more details, see in **section 5.6**.

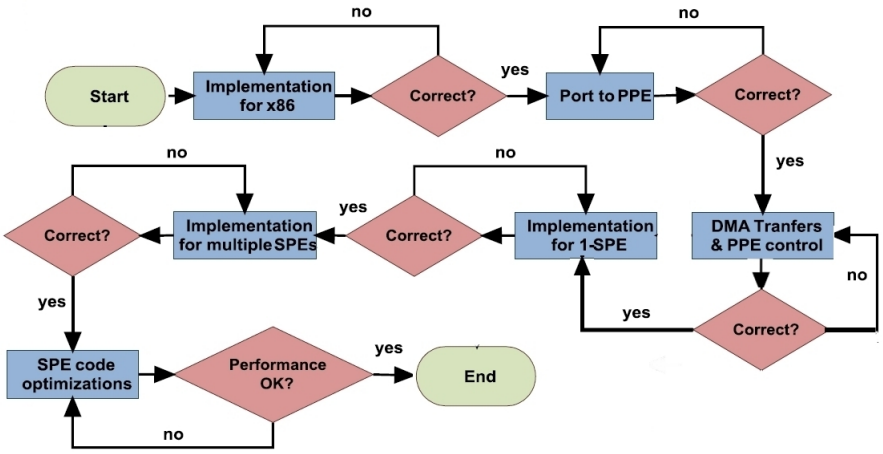

Figure 6.6: The development flow chart.

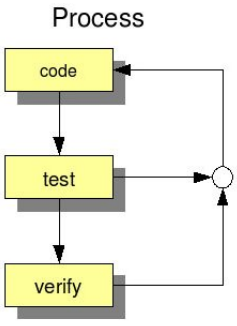

Figure 6.7: An interative development process.

All the code was developed with the use of IBM's Cell SDK 3.1 and Full-System Simulator. For the final stage of the development a PS3 was used with Yellow Dog Linux 6.0 OS.

6.6.1 Implementation on x86 Architecture

At this stage we used the code of the CNPE algorithm that was developed by FORTH Research at Heraklion of the Crete. The only change in the code that we made was the splitting of the matrix type of complex in two matrices type of *float*. The first matrix stores the real part of the complex number and the second the imaginary part. For example, the matrix *ca[NR][NZ]* is split in the matrices *ca_re[NR][NZ]* and *ca_im[NR][NZ]*. The goal from this stage was to prove that the basic algorithm produced correct results and would meet our requirements for running in an SPU.

The implementation was run on P4 machine at 3.0 GHz with memory 512 MB. The operating system was *fedora 9* and the code was developed in the Linux based editor *Geany 0.16* and the debugging of the compiled code was made with *GNU gdb* debugger.

6.6.2 Port to PPE

At this stage, the x86 CNPE implementation was ported to PowerPC hardware (PPE) by recompiling the x86 source code with *ppu-gcc* compiler and the necessairy makefiles for the Cell processor [IBM08b], [IBM08a], [Sca09]. This porting to the PPE was made to confirm the correct execution of x86 implementation on the PPE. The PowerPC version of the code performed much slower than the x86 version of the code. This is due to the difference in the relative power of the PPU portion of the Cell BE compared to a fairly high end x86 CPU on which the x86 code was developed. Even though this code was simple and single threaded, the x86 processor core used for development is a much more power processor core than the PPU. After this step the application was running on PPU and the next step was to begin offloading the functions to the SPEs.

6.6.3 PPE control

The PPU's most important role is managing the Synergistic Processor Elements (SPEs) [IBM08b], [A. 07], [IBM08a], [Sca09]. In our implementation, the scheduling and control of the threads are based on a *fork-join* model, as shown in Figure 5.7. For more details, see in **section 5.6.3**.

The total scheduling procedure consumes a significant amount of time compared with the total execution time. Some overhead from the scheduling process is unavoidable but it was managed to be reduced as much as possible. In order to reduce the overhead the context was created only once and the program was loaded to the SPEs only once, when it was possible. Before the first function-call of the function *CNPE*, we introduce a series of scheduling operations that creates the control block,

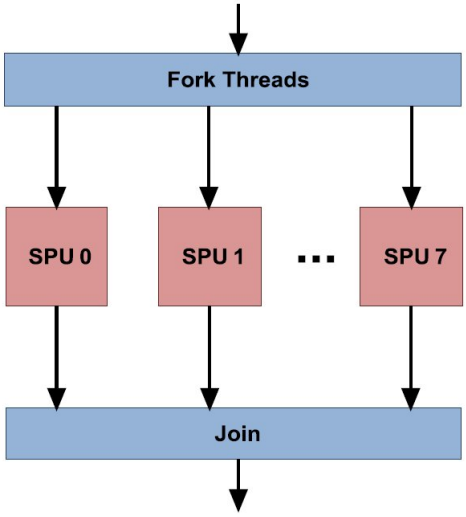

Figure 6.8: Fork-join model.

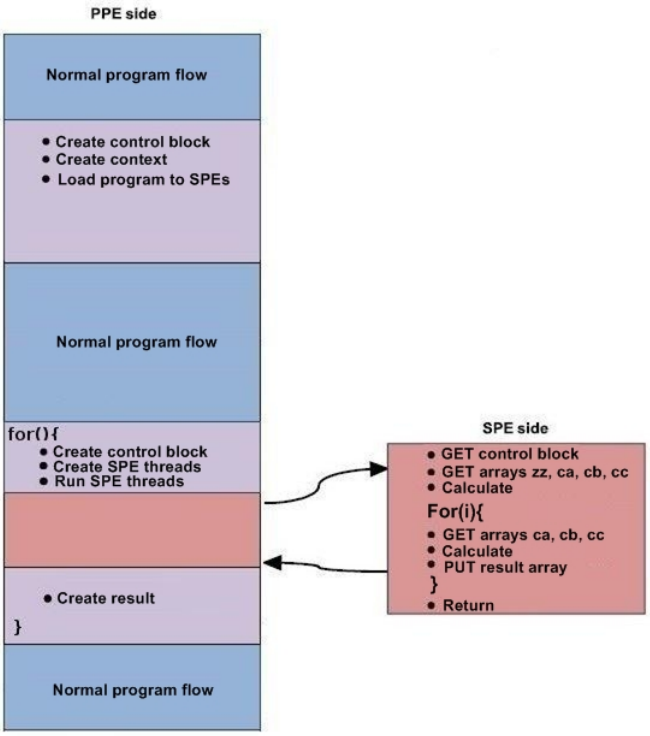

Figure 6.9: Overall scheduling process for CNPE.

creates context and loads program to the SPEs, as shown in Figure 6.9. Since the SPEs were running the same program it wasn't necessary to create new context and load the program each time. Thus, for each function-call of the *CNPE* in the loop the program updates the *control block*, which is different for every function call and executes the SPE thread, as shown in Figure 6.9. When the SPE code has been executed, the program returns to the PPE side and creates the result. With this approach, the creation of the context and the program loading was done outside the loop to avoid their unnecessary repeat.

6.6.4 DMA Transfer

The DMA mechanism is described in **Section 5.6.4**. In our implementation, we need to transfer from main memory to the Local Store the matrices *ca_re[NR][NZ]*, *cb_re[NR][NZ]*, *cc_re[NR][NZ]*, *ca_im[NR][NZ]*, *cb_im[NR][NZ]*, *cc_im[NR][NZ]* and an array *zz[NZ]* type of *float*. Initially, we used a *mfc_get* command for the array *zz*. The matrices do not fit in the Local Store.Thus, we segment them in NR arrays of each NZ elements because each iteration of the *for-loop* the program needs a row from each matrix , as shown in Figure 6.2, 6.3, 6.4. With this approach, in each iteration, we store a row from each matrix in the LS. Furthermore, in each iteration we used a *mfc_put* command for the array *contour[NZ]*. Table 6.6 shows us a summary of the required DMA transfers.

Data	Size (Bytes)	DMA-get (Bytes)	DMA-put (Bytes)
zz	2004	1x2004	-
ca_re	2014020	1005x2004	-
ca_im	2014020	1005x2004	-
cb_re	2014020	1005x2004	-
cb_im	2014020	1005x2004	-
cc_re	2014020	1005x2004	-
cc_im	2014020	1005x2004	-
contour	2014020	-	1005x2004

Table 6.6: DMA Transfers.

6.6.5 Implementation with One and Multiple SPEs

At this stage, we take the PPU code of the function *CNPE* and run it on a single SPU. This involved restructuring the code by adding calls to the SDK to load an SPU module. More code was added to implement a simple DMA model for

streaming data into the SPU and streaming it back to main store memory. At this stage there is no parallelism since only one SPE it was used.

The design was extended on two and finally on six SPEs, the procedure remains the same as in the case of one SPE. The advantage now is that the total number of functions calls *CNPE* that are required is distributed on multiple SPEs. In each function call, the SPE should known informations about which data must fetch and where should store the result. This information is passed once at each SPE through a structure called *control block*. The *control block* contains information such as, arrays addresses, result address.

6.6.6 Code Optimizations

In this section, we describe the code optimizations that we applied to improve the performance.

Overlap data with computations

One of the unique features of the Cell BE architecture is the DMA engines in each of the SPEs which enables asynchronous data transfer. Thus, we used this feature to achieve overlapping between data transfers and computations. With this technique, we achieved the SPUs to execute computations untill of completion of the DMA transfers

Trid optimization

From the code of the function *trid*, we observed that the division of the complex numbers was made by using the polar form. This form helps the programmer to write code easily but it requires a great number of computations. Thus, we implemented the division by using the normal form.

Furthermore, the product of two complex numbers is given by the following equation:

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc) \tag{6.1}$$

From mathematics, we have the following equation:

$$(a + b)(c + d)–ac–bd = ac + ad + bc + bd–ac–bd = ad + bc \tag{6.2}$$

From equations 6.1 and 6.2, we observed that we can implement the multiplication of two complex numbers by using optimized version $(ad + bc)$ to get rid of multiply. The improvement was important by applying these optimizations, as is shown in **Chapter 8 section 8.2**.

SIMD Programming

Both the PPE and the SPEs support parallel processing of Single Instruction Multiple Data (SIMD) vector elements. A vector is an instruction operand containing a set of data elements packed into a one-dimensional array, as shown in Figure 6.10. In the CBE processor, the vector elements can be fixed-point (integer) or floating-point values. Almost all Vector/SIMD Multimedia Extension and SPU instructions operate on vector operands. Vectors are also called SIMD operands or packed operands.

Figure 6.10: A vector with four elements.

As the SIMD name implies, this style of programming allows one instruction to be applied to the multiple data elements of a vector in parallel. In this way, SIMD processing exploits data-level parallelism. SIMD programming is prevalent in multimedia, graphics-intensive stream processing (such as gaming), and high performance computing—basically, in any compute-intensive application. The CBE processor is designed to operate efficiently on SIMD code, so programmers benefit from learning how to efficiently exploit data parallelism in their programs and how to take advantage of compiler optimizations for SIMD code.

In both the PPE and SPEs, the vector registers hold multiple data elements as a single vector. The data paths and registers supporting SIMD operations are 128 bits wide, corresponding to four full 32-bit words. This means that four 32-bit words can be loaded into a single register, and, for example, added to four other words in a different register in a single operation. Similar operations can be performed on vector operands containing 16 bytes, 8 halfwords, or 2 doublewords. Both the vector unit of PPE and SPE instruction set have extensions that support C-language intrinsics [IBM08b], [IBM08a], [IBM08d]. Intrinsics are C-language commands, in the form of function-calls that are convenient substitutes for one or more inline assembly-language instructions. In our design all the data were *float*, so the vector registers could only hold four 32-bit values and the data parallelism offered by the SIMD vectorization is reduced to four simultaneous operations. This means that once the data were promoted from *float* to vector type it was able to execute four loads, four multiplications, four additions, etc, simultaneously and so the iterations were reduced by a factor of four.

In our implementation, we vectorized the functions *amat* and *rhs*. Below, we describe how the code of the functions was vectorized. Let us suppose we have the following code part of the *amat*, as shown in Figure 6.11.

Instead of loading one element into one 128-bit register we use the data type

```
for( j=0; j<NZ; j++)
{
ad[j].re= 1.- cca.re + 2. * ccc.re
}
```

Figure 6.11: A part of the amat code.

vector and load four elements into one register. Furthermore, we use the function *spu_splats* to convert scalar data to vectors whose elements equal the scalar. Thus, for the constants '1' and '2' we have the vectors *vone* and *vtwo* respectively. The function *spu_sub(vone,vcca_re)* executes four operations of type $(1. - cca.re)$ simultaneously, the function *spu_madd(vtwo,vccc_re,spu_sub(vone,vcca_re))* executes four operations of type $(1. - cca.re + 2. * ccc.re)$ simultaneously and so the iretations were reduced by a factor of four. Figure 6.12 shows the vector implementation of the scalar code in Figure 6.11.

```
vector floatType  vcca_re,vccc_re;
vector floatType vone=spu_splats((floatType)1.0);
vector floatType vtwo=spu_splats((floatType)2.0);
for( j=0; j<NZ>>2; j++)
{
      vad_re[j]=spu_madd(vtwo,vccc_re,spu_sub(vone,vcca_re));
}
```

Figure 6.12: A part of the amat's vector code.

We followed the same process to vectorize the function *rhs*. One important difference was that the data must be reorganized in registers to produce a correct result because the vector register needs data from the compination of the two vectors in each iretation of the loop. The reorganization of the data in registers was made by using the instruction *spu_shuffle*. The *spu_shuffle* accepts two input vectors of similar type and a third control vector. Each byte in the result vector is determined by the corresponding byte in the control vector, as shown in Figure 6.13.

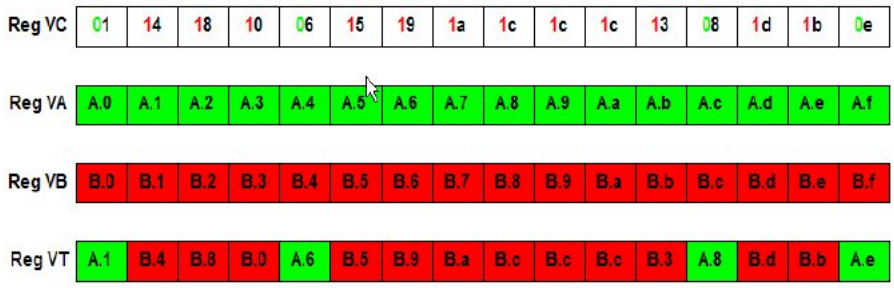

Figure 6.13: Shuffle example: spu_shuffle VT,VA,VB,VC instruction.

Pipeline loops

Most loops generally have the same basic structure. Per iteration, they load input data, perform computation, and finally store the results. Since loads, stores, quadword rotates, and shuffles execute on pipeline 1, and most computation instructions execute on pipeline 0, we applied software pipelined loops technique to improve dual-issue rates by computing at the same time as loading and storing data, as shown in Figure 6.14.

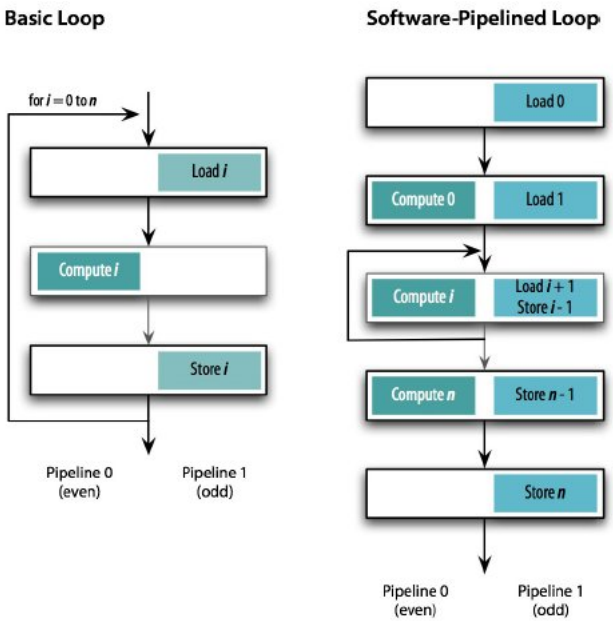

Figure 6.14: Pipelining and dual-issue.

Branch Elimination

We applied some extra loop-unrolling and function-inlining for branch elimination but we did not had significant increase in performance.

Chapter 7

Evaluation and Verification Of the Glimmer

This chapter presents the performance of the design and compares it with the performance of other processors for the specific algorithm. The measuring procedure is also described here as well as the verification of the implementation.

7.1 Measuring Performance

Measuring the performance of an application is a very important step and provides the programmer with critical information about its design. For the Cell processor there are currently three ways to measure the performance of an application running on Cell. The first two methods are using two software tools that are available in the SDK to assist in measuring the performance, the spu-timing analyzer and the IBM Full System Simulator for Cell B.E [IBM07b], [IBM07c]. The last method for measuring and the one that was followed in the design is the dynamic profiling using the hardware counters.

The processor includes two software-visible 64-bit time-base registers in the PPE one for configuration and one for counting and eleven software-visible 32-bit decrementers (down-counters), three in the PPE and one in each of the eight SPEs [IBM07a]. The time-base registers and the decrementers are not clocked at the 3.2 GHz as the core clock, they have their one frequency called time-base frequency. This frequency is different on the PS3 than on the Cell Blades [IBM], the PS3 time-base frequency is 79.8 MHz and this value was used for our measurements.

During the measuring procedure the one 64-bit time-base register in the PPE was used to measure the total execution time and execution time of code segments at the PPE. The SPEs performance was measured with the use of the decrementers of each SPE. Both types of time-base registers were providing us with a number

of clockticks which was converted to execution time by dividing with the time-base frequency. In the case of the SPEs, when multiple SPEs were used the greater time was considered as the SPEs execution time.

In order to have a fair comparison the total execution time for P4 was measured in a similar way. The time.h library was used for the P4 to measure the real execution time through the OS. The main purpose was to compare the processors and not the systems, so for both measurements the amount of time for loading data to the main memory and for storing data to the hard disk was taken out. Furthermore all the printf system-calls were removed from the programs to avoid as much as possible the OS since the two processors are running different OS. Due to the OS measurements of the same code had a small variation, so ten measurements were taken for each case and the average is being presented as the final result.

7.2 Performance

This section presents all the performance measurements that were done to evaluate the implementation. First the performance of the total code running on SPEs was measured and compared with the various optimizations. Finally the total execution time of Glimmer algorithm was evaluated and compared with the P4 at 2.66 GHz with 1 GB memory.

7.2.1 Performance of SPEs

This section presents the total execution time of the Glimmer algorithm with various optimizations for input datasets CLASS B NC003062.fna and CLASS C NC004463.fna from Bioperf suite. The summary of the results is shown in the next tables and figures.

Execution Time (sec)	**Original**	**Unroll_ppu 48x**	**Unroll_ppu 288x**	**Unroll_ppu 576x**	**Unroll_ppu 624x**
1-SPU	4,714.448	2,212.978	1,301.550	656.508	625.109
2-SPU	2,357.224	1,106.354	650.675	328.104	312.545
4-SPU	1,178.612	553.397	324.725	163.987	156.172
6-SPU	810.547	368.931	216.283	109.245	107.705

Table 7.1: Execution time of Glimmer for NC003062.fna.

As shown in Figures 7.1 and 7.2, the performance of the code had a significant improvement with the use of multiple SPEs and the optimizations. The use of the technique PPU_unroll reduces the overhead from the scheduling process, as it was

Execution Time (sec)	Original	Unroll_ppu 48x	Unroll_ppu 288x	Unroll_ppu 576x	Unroll_ppu 624x
1-SPU	20,748.067	10,145.846	3,062.252	2,675.632	2,673.604
2-SPU	10,474.034	5,022.923	1,539.626	1,339.816	1,335.252
4-SPU	5,187.017	2,521.462	764.813	669.408	668.626
6-SPU	3,577.253	1,725.146	527.457	460.971	459.742

Table 7.2: Execution time of Glimmer for NC004463.fna.

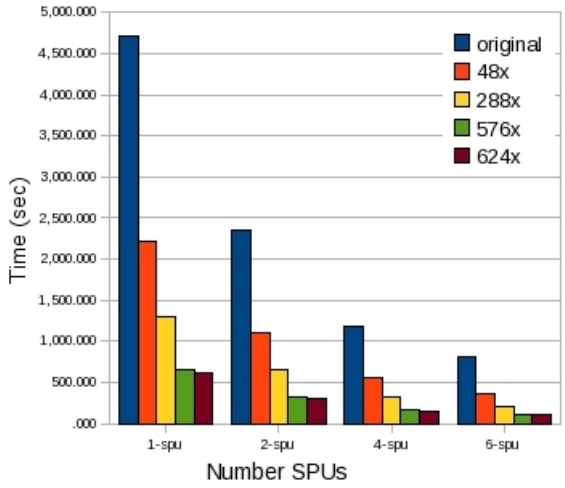

Figure 7.1: Performance impact of various optimizations for NC003062.fna.

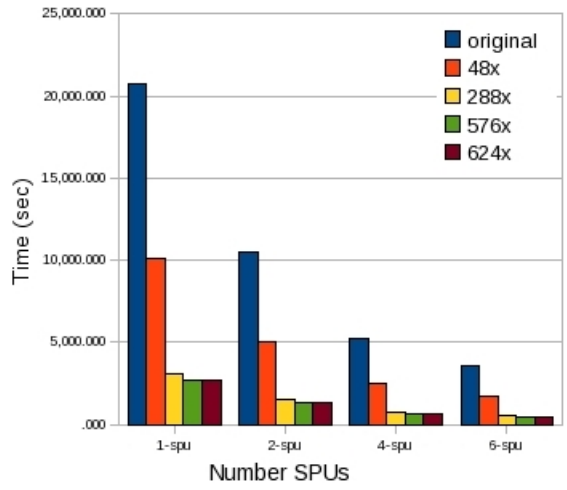

Figure 7.2: Performance impact of various optimizations for NC004463.fna.

mentioned in **Section 5.6.6**. From the 576x to 624x PPU_unroll, we observe that the reduction of the total execution time is not significant. Thus, we could have further reduction of the total execution time by applying more optimizations in the SPU code.

The next step was to run the Glimmer on the simulator to understand where the SPUs stall. From Figure 7.3 we observe that the SPUs stall due to branch misses and data dependencies. As it was mentioned in **Section 5.6.6**, we tried to eliminate branches by applying loop unrolling and function inlining but the improvement of the performance was not significant. Furthermore, the structure of the SPU code prevented us from using SIMD technique. For the above reasons we have these undesirable results.

```
SPU DD3.0
***
Total cycle count              11673719578
Total Instruction count        643
Total CPI                      18155085.24
***
Performance cycle count        11673719578
Performance Instruction count  3760402619 (3267373052)
Performance CPI                3.10 (3.57)

Branch instructions            559417868
Branch taken                   270020413
Branch not taken               289397455

Hint instructions              19463686
Hint hit                       32342530

Contention at LS between Load/Store and Prefetch 30616548

Single cycle                                  2256886618 ( 19.3%)
Dual cycle                                     505243217 (  4.3%)
Nop cycle                                      294838594 (  2.5%)
Stall due to branch miss                      4279080464 ( 36.7%)
Stall due to prefetch miss                             0 (  0.0%)
Stall due to dependency                       3139872572 ( 26.9%)
Stall due to fp resource conflict                      0 (  0.0%)
Stall due to waiting for hint target            19000261 (  0.2%)
Issue stalls due to pipe hazards                  216522 (  0.0%)
Channel stall cycle                           1178581321 ( 10.1%)
SPU Initialization cycle                               9 (  0.0%)
-------------------------------------------------------------------
Total cycle                                  11673719578 (100.0%)
```

Figure 7.3: SPE statistics for the Glimmer.

7.2.2 Final Comparison

This section presents the performance of Glimmer algorithm compared with the reference P4 machine and with the execution of the application on the PPE only. For the P4 machine, the Glimmer algorithm was measured with -O3 option which had an important improvement in performance. The following figures show the performance comparisons of the Glimmer for datasets NC003062.fna and NC004463.fna.

From figures 7.4 and 7.5, we observe that the cell processor is slower than PPU in all cases. As described in previous section, the structure of the SPU code prevented us to achieve desirable results.

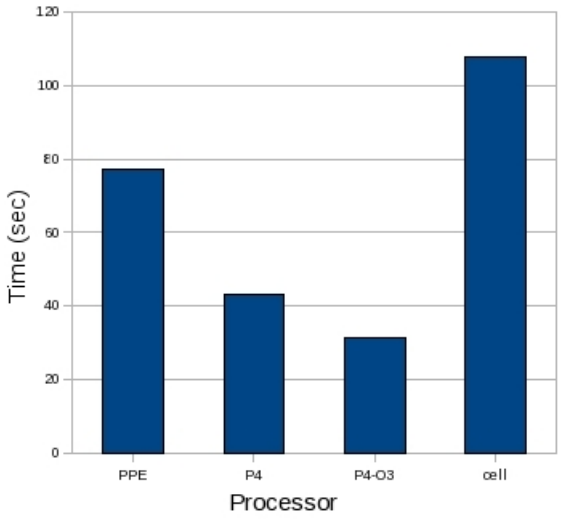

Figure 7.4: Performance comparisons of the Glimmer for dataset NC003062.fna.

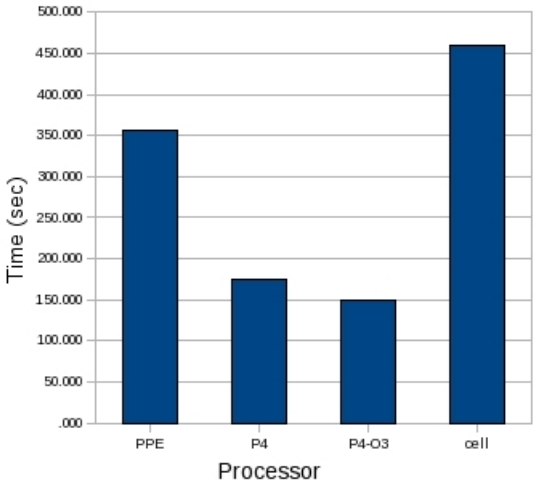

Figure 7.5: Performance comparisons of the Glimmer for dataset NC004463.fna.

7.3 Verification

The last but not least step of the design was the overall verification of the application. As it was mentioned in section 4.6.2 the output of the application is a list of all open reading frames (orfs) together with scores for each as a gene, so a comparison of this list and these scores were made to verify the results. The results produced by the execution of the original code were compared with the results produced from the execution on PS3. The verification process was successful and all the results were the same.

Chapter 8

Evaluation and Verification Of the CNPE

This chapter presents the performance of the design and compares it with the performance of other processors for the specific algorithm. The measuring procedure is also described here as well as the verification of the implementation.

8.1 Measuring Performance

Measuring the performance of an application is a very important step and provides the programmer with critical information about its design. The measuring procedure that was followed is same with the Glimmer algorithm, as described in **Chapter 7 section 7.1**.

8.2 Performance

This section presents all the performance measurements that were done to evaluate the implementation. First the performance of the total code running on SPEs was measured and compared with the various optimizations. Finally the total execution time of CNPE algorithm was evaluated and compared with the P4 at 3.0 GHz with 512 MB memory and Xeon at 2.66 GHz with 9 GB memory.

8.2.1 Performance of SPEs

This section presents the total execution time of the CNPE algorithm for 36 and 360 calls of the function CNPE. Furthermore, in the two cases it presents the total execution time of the CNPE algorithm with various optimizations for different NZ and NR. The summary of the results is shown in the following tables and figures.

- Table 8.1 shows the total execution time of the CNPE algorithm for 36 calls of the function CNPE with NZ=128 and NR=1005. Figure 8.1 shows the gradual improvement of the performance for this case.

Execution Time (sec)	**Original**	**+Overlap data comput**	**+Trid_opt**	**+SIMD**	**+Unroll&Pipeline Loops**
1-SPU	4.363	4.342	1.359	0.724	0.589
2-SPU	2.218	2.198	0.717	0.401	0.335
4-SPU	1.152	1.131	0.396	0.243	0.211
6-SPU	0.797	0.777	0.295	0.193	0.175

Table 8.1: Execution time of 36 CNPE for NZ=128 and NR=1005.

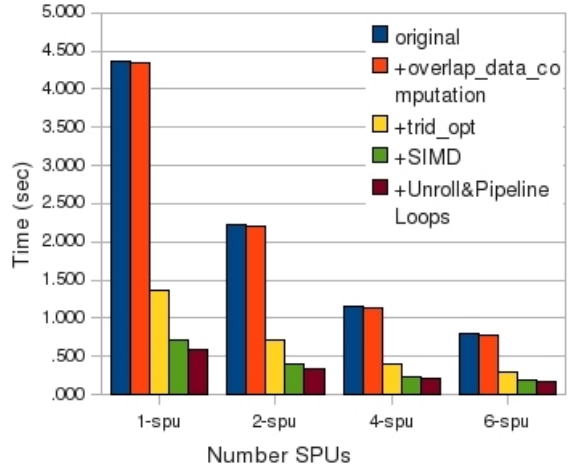

Figure 8.1: Performance impact of various optimizations.

- Table 8.2 shows the total execution time of the CNPE algorithm for 36 calls of the function CNPE with NZ=256 and NR=500. Figure 8.2 shows the gradual improvement of the performance for this case.

- Table 8.3 shows the total execution time of the CNPE algorithm for 36 calls of the function CNPE with NZ=512 and NR=1005. Figure 8.3 shows the gradual improvement of the performance for this case.

Execution Time (sec)	Original	+Overlap data comput	+Trid_opt	+SIMD	+Unroll&Pipeline Loops
1-SPU	4.415	4.394	1.342	0.703	0.578
2-SPU	2.246	2.226	0.707	0.388	0.324
4-SPU	1.163	1.143	0.389	0.233	0.204
6-SPU	0.806	0.789	0.286	0.197	0.176

Table 8.2: Execution time of 36 CNPE for NZ=256 and NR=500.

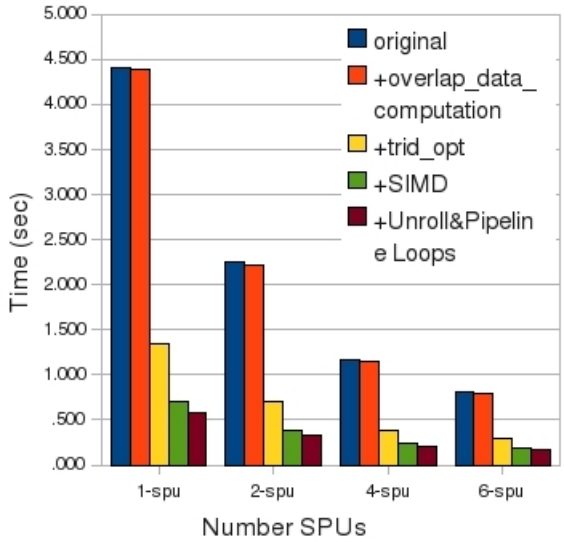

Figure 8.2: Performance impact of various optimizations.

Execution Time (sec)	Original	+Overlap data comput	+Trid_opt	+SIMD	+Unroll&Pipeline Loops
1-SPU	17.378	17.351	5.335	2.767	2.256
2-SPU	8.818	8.799	2.793	1.517	1.262
4-SPU	4.544	4.526	1.529	0.899	0.773
6-SPU	3.125	3.107	1.117	0.715	0.634

Table 8.3: Execution time of 36 CNPE for NZ=512 and NR=1005.

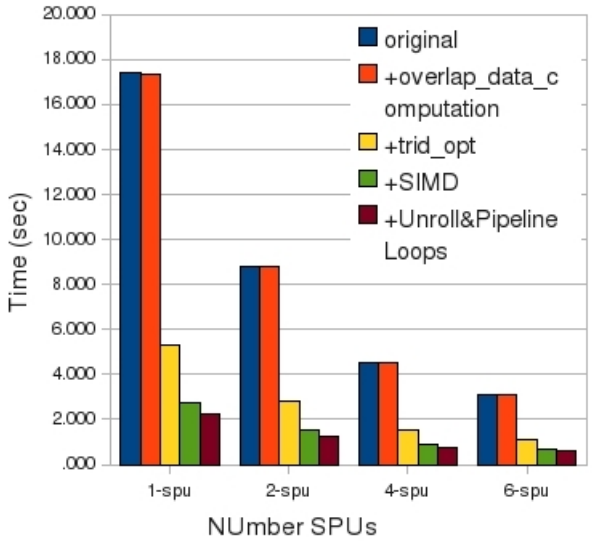

Figure 8.3: Performance impact of various optimizations.

Execution Time (sec)	**Original**	**+Overlap data comput**	**+Trid_opt**	**+SIMD**	**+Unroll&Pipeline Loops**
1-SPU	43.042	43.021	13.009	6.657	5.321
2-SPU	21.589	21.565	6.589	3.413	2.474
4-SPU	10.889	10.867	3.378	1.787	1.481
6-SPU	7.318	7.293	2.318	1.255	1.064

Table 8.4: Execution time of 360 CNPE for NZ=128 and NR=1005.

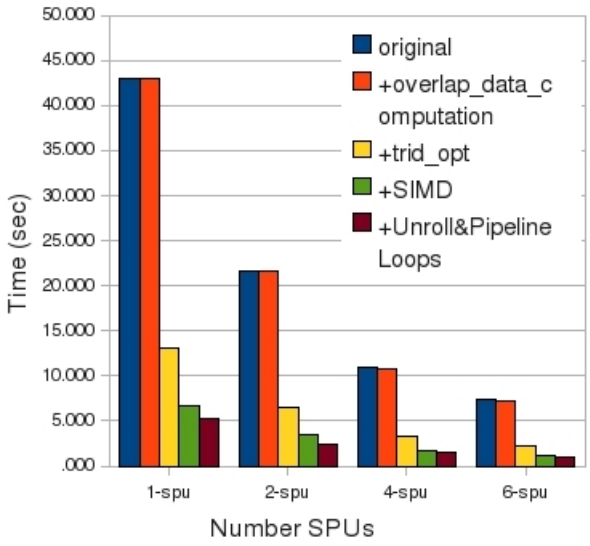

Figure 8.4: Performance impact of various optimizations.

- Table 8.4 shows the total execution time of the CNPE algorithm for 360 calls of the function CNPE with NZ=128 and NR=1005. Figure 8.4 shows the gradual improvement of the performance for this case.

- Table 8.5 shows the total execution time of the CNPE algorithm for 360 calls of the function CNPE with NZ=256 and NR=500. Figure 8.5 shows the gradual improvement of the performance for this case.

Execution Time (sec)	**Original**	**+Overlap data comput**	**+Trid_opt**	**+SIMD**	**+Unroll&Pipeline Loops**
1-SPU	43.547	43.528	12.885	6.496	5.249
2-SPU	21.862	21.845	6.523	3.337	2.706
4-SPU	11.002	10.984	3.341	1.751	1.443
6-SPU	7.395	7.378	2.285	1.245	1.039

Table 8.5: Execution time of 360 CNPE for NZ=256 and NR=500.

- Table 8.6 shows the total execution time of the CNPE algorithm for 360 calls of the function CNPE with NZ=512 and NR=1005. Figure 8.6 shows the gradual improvement of the performance for this case.

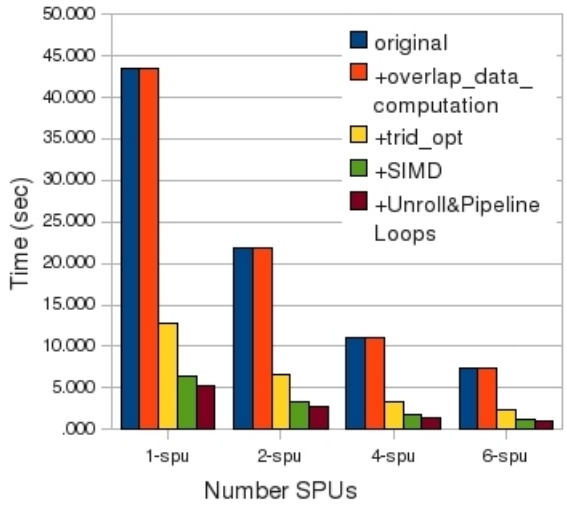

Figure 8.5: Performance impact of various optimizations.

Execution Time (sec)	**Original**	**+Overlap data comput**	**+Trid_opt**	**+SIMD**	**+Unroll&Pipeline Loops**
1-SPU	171.605	171.585	51.281	25.513	20.411
2-SPU	85.998	85.976	25.791	12.944	10.393
4-SPU	43.209	43.187	13.085	6.663	5.391
6-SPU	28.963	28.942	8.858	4.622	3.781

Table 8.6: Execution time of 360 CNPE for NZ=512 and NR=1005.

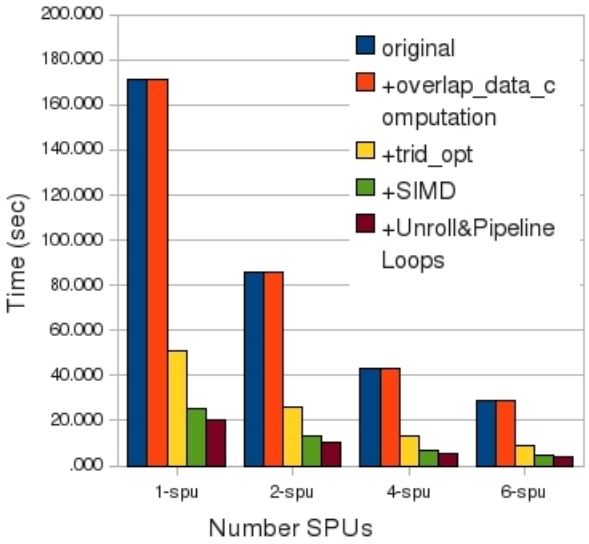

Figure 8.6: Performance impact of various optimizations.

As shown in the above Tables and Figures the performance of the code had a significant improvement with the use of multiple SPEs and the optimizations. The techniques *trid_opt* and *SIMD* contribute to increased efficiency considerably. The impact of the optimizations is gradually decreased as the number of SPEs increases. This is caused by the reduction of the percentage of the execution time that accepts the optimizations each time and the increase of the added overhead.

8.2.2 Final Comparison

This section presents the performance of the CNPE algorithm compared with the references machines P4, Xeon and with the execution of the application on the PPE only. For the machines P4 and Xeon, the CNPE algorithm was measured with -O3 option which had an important improvement in performance. First, it presents the performance of CNPE algorithm compared with the references machines P4, Xeon and PPE with same trid function (optimized) and afterwards with different trid. The same trid contains the *trid_opt* optimization which we applied at the spus. The summary of the results is shown in the next tables and figures.

- Table 8.7 shows the total execution time of the CNPE algorithm for 36 calls of the function CNPE with NZ=128, NR=1005 and same trid. Figure 8.7 shows the performance comparisons for this case.

Processor	**PPE**	**P4**	**P4 -O3**	**Xeon**	**Xeon -O3**	**Cell**
Execution Time (sec)	1.651	2.568	1.311	0.881	0.406	0.175

Table 8.7: Execution time of 36 CNPE for NZ=128, NR=1005 and same trid.

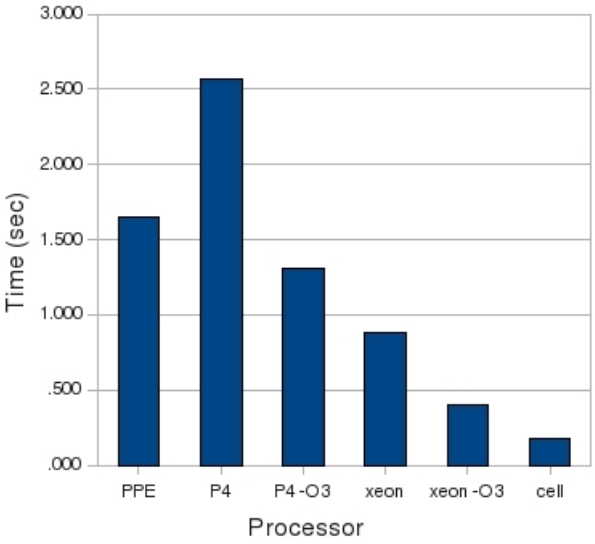

Figure 8.7: Performance comparisons for 36 CNPE with NZ=128, NR=1005 and same trid.

- Table 8.8 shows the total execution time of the CNPE algorithm for 36 calls of the function CNPE with NZ=256, NR=500 and same trid. Figure 8.8 shows the performance comparisons for this case.

Processor	**PPE**	**P4**	**P4 -O3**	**Xeon**	**Xeon -O3**	**Cell**
Execution Time (sec)	1.823	2.462	1.344	0.919	0.482	0.176

Table 8.8: Execution time of 36 CNPE for NZ=256, NR=500 and same trid.

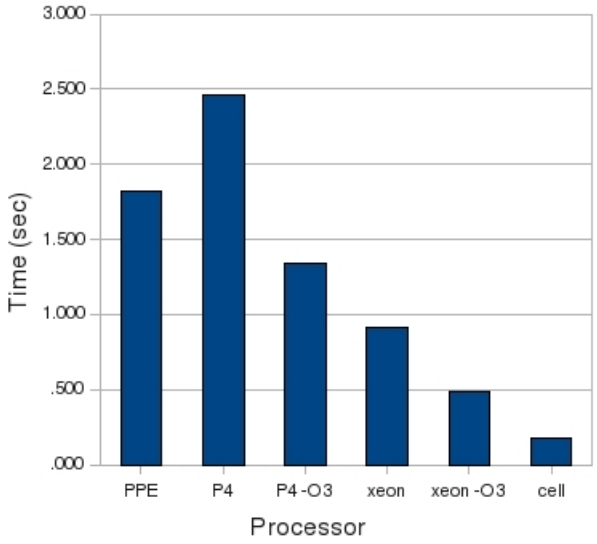

Figure 8.8: Performance comparisons for 36 CNPE with NZ=256, NR=500 and same trid.

- Table 8.9 shows the total execution time of the CNPE algorithm for 36 calls of the function CNPE with NZ=512, NR=1005 and same trid. Figure 8.9 shows the performance comparisons for this case.

Processor	**PPE**	**P4**	**P4 -O3**	**Xeon**	**Xeon -O3**	**Cell**
Execution Time (sec)	7.897	7.089	4.171	4.133	2.455	0.634

Table 8.9: Execution time of 36 CNPE for NZ=512, NR=1005 and same trid.

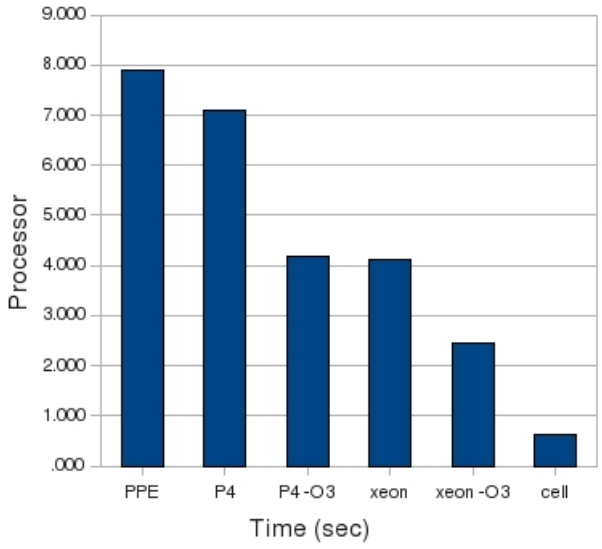

Figure 8.9: Performance comparisons for 36 CNPE with NZ=512, NR=1005 and same trid.

From the above results, we observe that in the first two cases we achieve the same speedup over P4-O3 and Xeon-O3. In the first case we achieve speedup 7.49x over P4-O3 and 2.32x over Xeon -O3 and in the second 7.63x and 2.74x respectively. Finally, in the third case we achieve 6.58x speedup over P4-O3 and 3.81x over Xeon-O3.

- Table 8.10 shows the total execution time of the CNPE algorithm for 360 calls of the function CNPE with NZ=128, NR=1005 and same trid. Figure 8.10 shows the performance comparisons for this case.

Processor	**PPE**	**P4**	**P4 -O3**	**Xeon**	**Xeon -O3**	**Cell**
Execution Time (sec)	16.102	24.845	13.602	8.588	4.037	1.064

Table 8.10: Execution time of 360 CNPE for NZ=128, NR=1005 and same trid.

- Table 8.11 shows the total execution time of the CNPE algorithm for 360 calls of the function CNPE with NZ=256, NR=500 and same trid. Figure 8.11 shows the performance comparisons for this case.

- Table 8.12 shows the total execution time of the CNPE algorithm for 360 calls of the function CNPE with NZ=512, NR=1005 and same trid. Figure 8.12 shows the performance comparisons for this case.

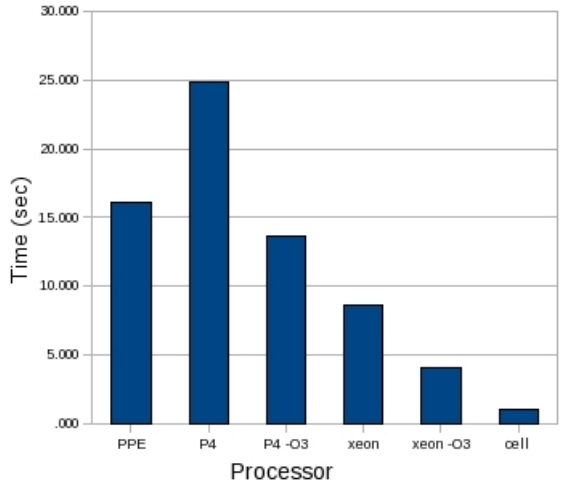

Figure 8.10: Performance comparisons for 360 CNPE with NZ=128, NR=1005 and same trid.

Processor	**PPE**	**P4**	**P4 -O3**	**Xeon**	**Xeon -O3**	**Cell**
Execution Time (sec)	18.136	24.606	13.602	8.725	4.228	1.039

Table 8.11: Execution time of 360 CNPE for NZ=256, NR=500 and same trid.

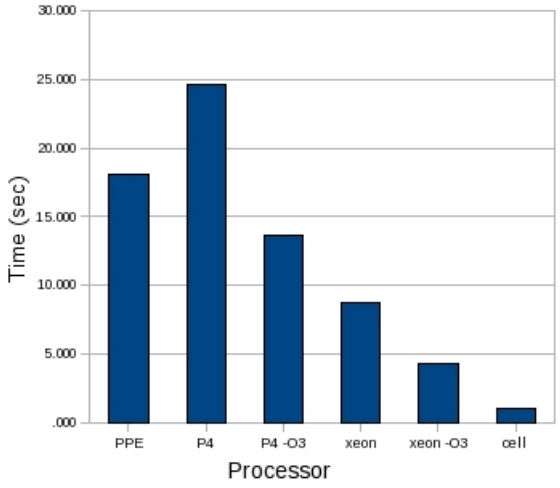

Figure 8.11: Performance comparisons for 360 CNPE with NZ=256, NR=500 and same trid.

Processor	**PPE**	**P4**	**P4 -O3**	**Xeon**	**Xeon -O3**	**Cell**
Execution Time (sec)	76.281	77.051	46.817	40.282	24.192	3.781

Table 8.12: Execution time of 360 CNPE for NZ=512, NR=1005 and same trid.

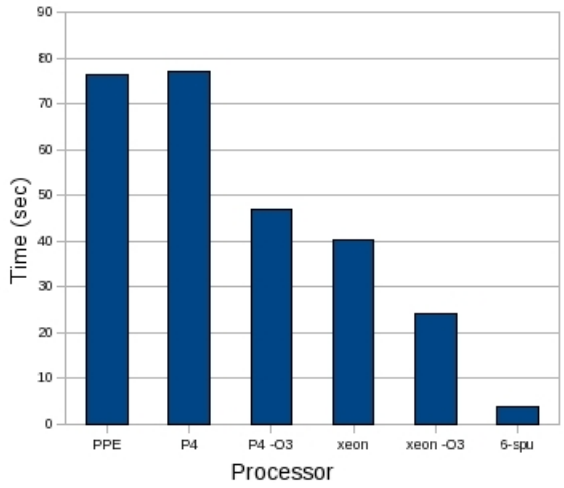

Figure 8.12: Performance comparisons for 360 CNPE with NZ=512, NR=1005 and same trid.

From the above results, we observe that in the first case we achieve speedup 12.78x over P4-O3 and 3.86x over Xeon -O3 and in the second 13.1x and 4.07x respectively. Finally, in the third case we achieve 12.38x speedup over P4-O3 and 6.39x over Xeon-O3.

Below, the section presents the results of the comparison for different trid.

- Table 8.13 shows the total execution time of the CNPE algorithm for 36 calls of the function CNPE with NZ=128, NR=1005 and different trid. Figure 8.13 shows the performance comparisons for this case.

Processor	**PPE**	**P4**	**P4 -O3**	**Xeon**	**Xeon -O3**	**Cell**
Execution Time (sec)	10.513	5.831	4.332	3.150	2.056	0.175

Table 8.13: Execution time of 36 CNPE for NZ=128, NR=1005 and different trid.

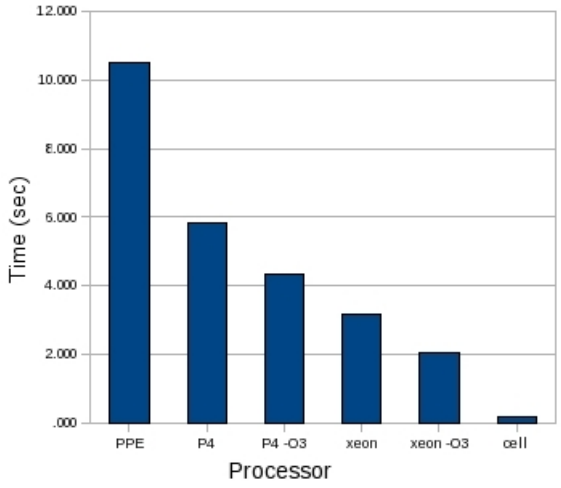

Figure 8.13: Performance comparisons for 36 CNPE with NZ=128, NR=1005 and different trid.

- Table 8.14 shows the total execution time of the CNPE algorithm for 36 calls of the function CNPE with NZ=256, NR=500 and different trid. Figure 8.14 shows the performance comparisons for this case.

- Table 8.15 shows the total execution time of the CNPE algorithm for 36 calls of the function CNPE with NZ=512, NR=1005 and different trid. Figure 8.15 shows the performance comparisons for this case.

Processor	PPE	P4	P4 -O3	Xeon	Xeon -O3	Cell
Execution Time (sec)	12.038	5.474	4.274	3.206	1.953	0.176

Table 8.14: Execution time of 36 CNPE for NZ=256, NR=500 and different trid.

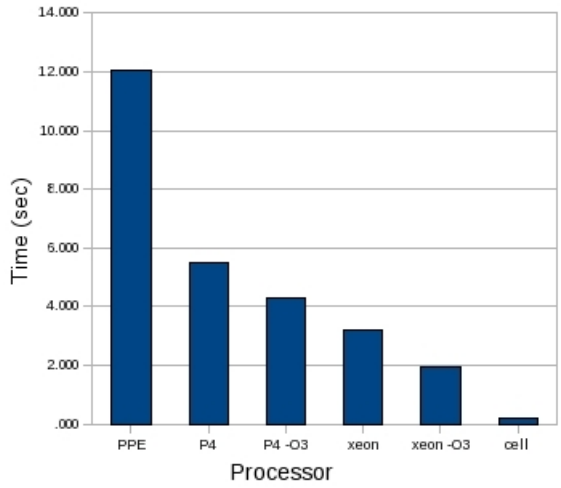

Figure 8.14: Performance comparisons for 36 CNPE with NZ=256, NR=500 and different trid.

Processor	PPE	P4	P4 -O3	Xeon	Xeon -O3	Cell
Execution Time (sec)	60.851	21.675	14.304	12.851	7.908	0.634

Table 8.15: Execution time of 36 CNPE for NZ=512, NR=1005 and different trid.

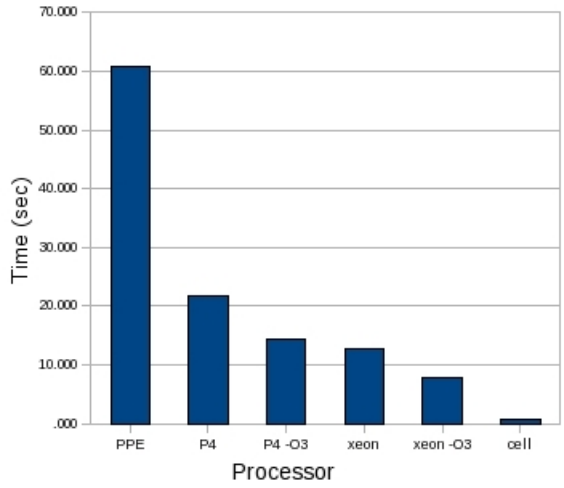

Figure 8.15: Performance comparisons for 36 CNPE with NZ=512, NR=1005 and different trid.

From the above results, we observe that in the first case we achieve speedup 24.75x over P4-O3 and 11.75x over Xeon -O3 and in the second 24.28x and 11.16x respectively. Finally, in the third case we achieve 22.56x speedup over P4-O3 and 12.47x over Xeon-O3.

- Table 8.16 shows the total execution time of the CNPE algorithm for 360 calls of the function CNPE with NZ=128, NR=1005 and different trid. Figure 8.16 shows the performance comparisons for this case.

Processor	**PPE**	**P4**	**P4 -O3**	**Xeon**	**Xeon -O3**	**Cell**
Execution Time (sec)	104.849	57.927	42.833	30.874	19.971	1.064

Table 8.16: Execution time of 360 CNPE for NZ=128, NR=1005 and different trid.

- Table 8.17 shows the total execution time of the CNPE algorithm for 360 calls of the function CNPE with NZ=256, NR=500 and different trid. Figure 8.17 shows the performance comparisons for this case.

- Table 8.18 shows the total execution time of the CNPE algorithm for 360 calls of the function CNPE with NZ=512, NR=1005 and different trid. Figure 8.18 shows the performance comparisons for this case.

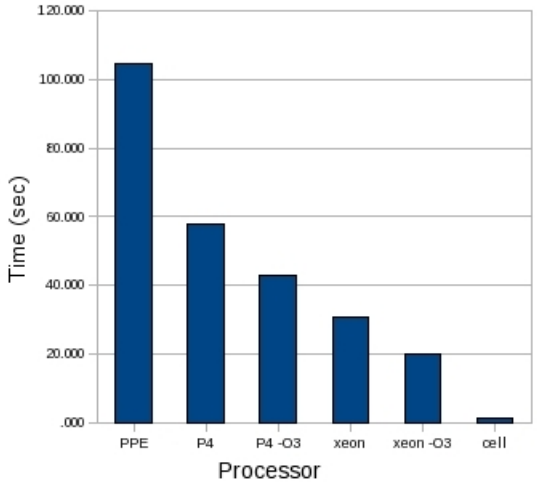

Figure 8.16: Performance comparisons for 360 CNPE with NZ=128, NR=1005 and different trid.

Processor	**PPE**	**P4**	**P4 -O3**	**Xeon**	**Xeon -O3**	**Cell**
Execution Time (sec)	119.995	57.310	42.595	31.653	19.240	1.039

Table 8.17: Execution time of 360 CNPE for NZ=256, NR=500 and different trid.

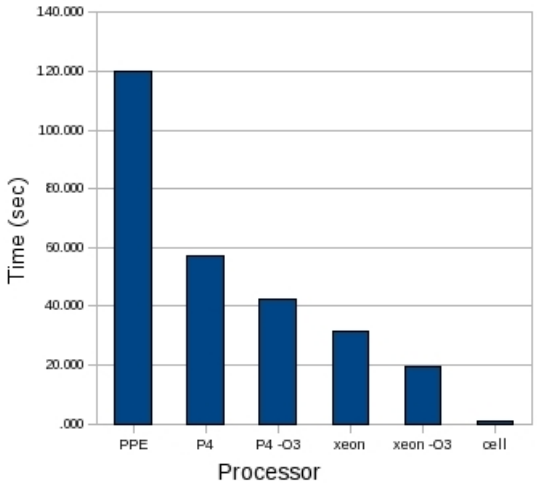

Figure 8.17: Performance comparisons for 360 CNPE with NZ=256, NR=500 and different trid.

Processor	PPE	P4	P4 -O3	Xeon	Xeon -O3	Cell
Execution Time (sec)	491.191	214.944	153.385	127.141	77.216	3.781

Table 8.18: Execution time of 360 CNPE for NZ=512, NR=1005 and different trid.

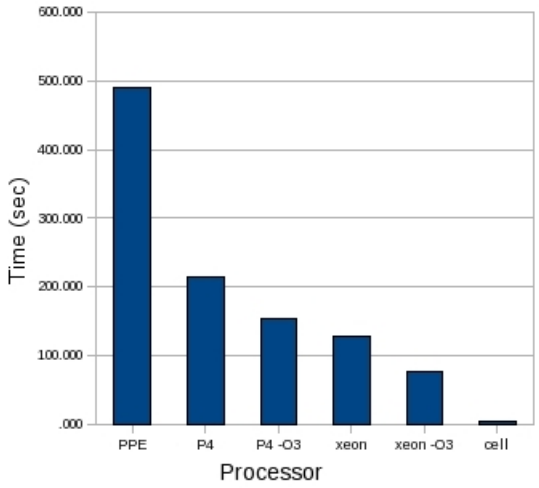

Figure 8.18: Performance comparisons for 360 CNPE with NZ=512, NR=1005 and different trid.

From the above results, we observe that in the first case we achieve speedup 40.26x over P4-O3 and 18.77x over Xeon -O3 and in the second 41x and 18.52x respectively. Finally, in the third case we achieve 42.15x speedup over P4-O3 and 20.42x over Xeon-O3.

Furthermore, we observe that the speedup over Xeon -O3 and P4-O3 raises as the size of problem increases from 36 to 360 CNPE. This happens because the rest of code which runs on the PPE consumes a significant amount of time compared with the total execution time in the case of 36 CNPE.

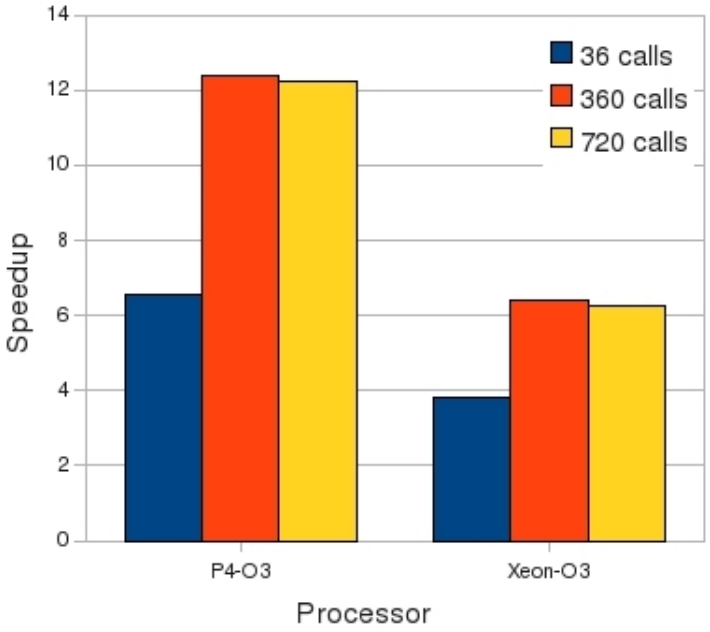

Figure 8.19: Speedup over P4-O3 and xeon-O3 for NZ=512 and NR=1005 with same trid

Finally, we take measurements for the 720 CNPE but we did not have improvement in the performance, the speedup was same as the 360 CNPE.

From Figures 8.19 and 8.20, we observe that the maximum achieved speedup over P4-O3 and Xeon-O3 is 42.15x and 20.42x respectively with different trid, and speedup 12.38x and 6.39 over P4-O3 and Xeon-O3 respectively with same trid.

8.3 Verification

The last but not least step of the design was the overall verification of the application. As it was mentioned in section 4.6.2 the output of the application is the biggest received sound pressure by the receiver, so a comparison of this value was

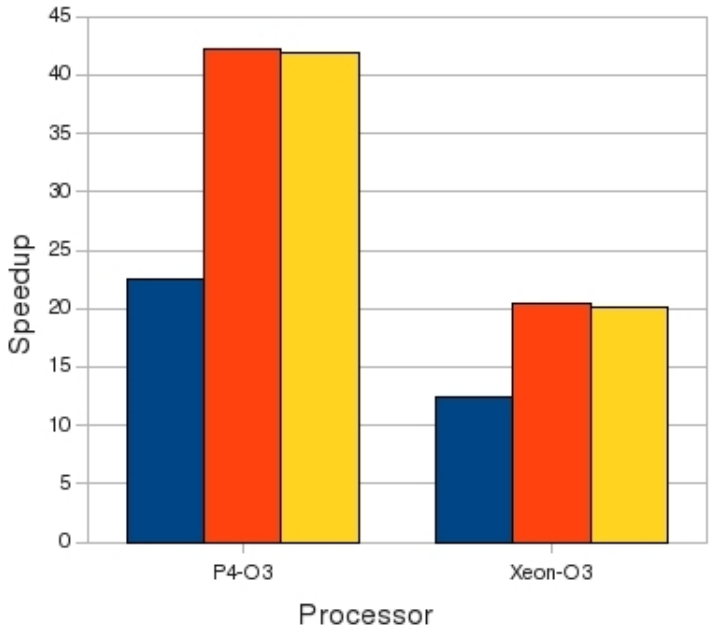

Figure 8.20: Speedup over P4-O3 and xeon-O3 for NZ=512 and NR=1005 with different trid

made to verify the result. A result produced by the execution of the original code was compared with a result produced from the execution on PS3. The verification process was successful and all the results were the same.

Chapter 9

Conclusions and Future Work

This chapter presents the conclusions from this work and proposes some ideas for future work.

9.1 Conclusions

The main contribution of this work was the parallelization of the Glimmer and CNPE algorithm, and their execution on the Cell processor.

The first algorithm is a biological application and the second is a applied mathematics application. Furthermore, the offloaded function of the Glimmer application had different characteristics in relation with the offloaded function of the CNPE. The offloaded function of the Glimmer contained many control statements and few calculations, on the other hand the offloaded function of the CNPE contained many calculations, as additions, multiplications, substractions and divisions between in single-precision floating point numbers.

From the final results, we concluded that the SPUs are highly effective at computations, but not optimally efficient at *gcc/TPCC* (load-compare-add- branch) type codes. This happens because the SPUs use a software controlled prediction of branches.Thus, correctly predicted branches are executed in one cycle, but a mispredicted branch (conditional or unconditional) incurs a penalty of approximately 18-19 cycles. Considering the typical SPU instruction latency of two- to-seven cycles, mispredicted branches degrade program performance.

9.2 Future Work

In the current thesis, we have applied most of the possible improvements that could be done with the use of the function offload model. Below, we propose some ideas for future work.

For both applications, the complete redesign of the application, the vectorization of the code running on the PPE and the use of a different model would probably increase their performance. Furthermore, the solution of the tridiagonal system with a different method would result better performance for the CNPE application.

Bibliography

[A. 05] A. Eichenberger et al. "Optimizing Compiler for the Cell Processor". *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT 2005), IEEE CS Press*, pages 161–172, 2005.

[A. 07] A. Arevalo, et al. "Programming the Cell Broadband Engine: Examples and Best Practices". *1st ed. IBM*, 2007.

[BD95] McIninch J. Koonin E. Rudd K. Medigue C. Borodovsky M. and Danchin. *Nucleic Acids Res., 23*, pages 3554–3562, 1995.

[BM93] M. Borodovsky and .D. Mcininch. *Comp. Chem.,17*, pages 123–133, 1993.

[C. 00] C. D. Cantrell. "Modern Mathematical Methods for Physicists and Engineers". *ISBN: 978-0521598279*, 2000.

[Cen05] STI Design Center. *"A Remote Procedure Call Implementation for the Cell Broadband Architecture"*, 1st edition, 2005.

[Cra] Crank, J. and P. Nicolson. "A practical method for numerical evaluation of solutions of partial differential equations of the heat conduction type". *Proc. Camb. Phil. Soc. 43: 50-67, 1947.*

[D. 06] D. A. Brokenshire. "Maximizing the Power of the Cell Broadband Engine Processor: 25 Tips to Optimal Application Performance". *IBM developerWorks technical article*, 2006.

[Dav05] David A. Bader, Yue Li, Tao Li and Vipin Sachdeva. "BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture on Bioinformatics Applications". *The IEEE International Symposium on Workload Characterization (IISWC 2005), Austin, TX*, October 6-8, 2005.

[Deb] Debian Team. "Debian". `http://www.debian.org/`.

[et 05] J. Kahle et al. "Introduction to the Cell Multiprocessor". *IBM J. Research and Development*, pages 589–604, September 2005.

[F.] F. D. Tappert. "The parabolic approximation method,in wave Propagation and Underwater Acoustics, eds J. B. Keller and J. S. Papadakis, Lecture Notes in Physics". *Vol. 70 (Springer-Verlag, Heidelberg,1977)*, pages 224–287.

[FOR] FORTH Research. "Long-Range Noise Propagation and Helicopter path Optimization for Noise Reduction". *http://www.forth.gr*.

[G. 89] G. Barton. "Elements of Green's Functions and Propagation: Potentials, Diffusion, and Waves". *ISBN: 978-0198519980*, 1989.

[Gen] Gentoo Linux. "Gentoo". *http://www.gentoo.org/*.

[IBM] IBM. "IBM BladeCenter QS20". *http://www-03.ibm.com/technology/splash/qs20/*.

[IBM07a] IBM. *"Cell Broadband Engine Programming Handbook"*, 1.1 edition, Apr. 2007.

[IBM07b] IBM. *"Full-System Simulator for the Cell Broadband Engine Processor"*, 3.0 edition, Oct. 2007.

[IBM07c] IBM. *"Performance Analysis with the Full-System Simulator"*, 3.0 edition, Oct. 2007.

[IBM08a] IBM. *"Cell Broadband Engine Programming Handbook"*, 3.1 edition, 2008.

[IBM08b] IBM. *"Cell Broadband Engine Programming Tutorial"*, 3.1 edition, 2008.

[IBM08c] IBM. *"SPE Runtime Management Library"*, 3.1 edition, 2008.

[IBM08d] IBM. *"SPU C/C++ Language Extensions"*, 3.1 edition, 2008.

[J. 09] J. A. Trangenstein. "Numerical Solution of Hyperbolic Partial Differential Equations". *ISBN: 978-0521877275*, 2009.

[M. 06a] M. Gschwind et al. "Synergistic Processing in Cell's Multicore Architecture". *IEEE Micro*, pages 10–24, Mar./Apr. 2006.

[M. 06b] M. Kistler et al. "Cell Multiprocessor Communication Network: Built for Speed". *IEEE Micro*, pages 10–23, May/June 2006.

[MS07] Michael Gschwind, IBM T.J. Watson Research Center and Sid Manning, Mark Nutter and David Erb, IBM Austin. "An Open Source Environment For Cell Broadband Engine System Software", June 2007.

[N. 08] N. A. Kampanis, V. Dougalis, J. A. Ekaterinaris. "Effective Computational Methods for Wave Propagation (Numerical Insights)". *ISBN: 978-1584885689*, 2008.

[Red] Red Hat. "Fedora Project". *http://fedoraproject.org*.

[S. 08] S. Marburg, B. Nolte. "Computational Acoustics of Noise Propagation in Fluids - Finite and Boundary Element Methods". *ISBN: 978-3540774471*, 2008.

[Sca09] Matthew Scarpino. *"Programming the Cell Processor"*, prentice hall,1st edition, 2009.

[Son] Sony Computer Entertainment. "Playstation 3". *http://gr.playstation.com/ps3/index.html*.

[Ste98] Steven L. Salzberg, Arthur L. Delcher, Simon Kasif and Owen White. "Microbial gene identification using interpolated Markov models". *Nucleic Acids Research, Vol. 26, No. 2*, pages 544–548, 1998.

[Ste99] Steven L. Salzberg, Arthur L. Delcher, Douglas Harmon, Simon Kasif and Owen White. "Improved microbial gene identification with GLIMMER". *Nucleic Acids Research, Vol. 27, No. 23*, pages 4636–4641, 1999.

[Ter] TerraSoft. "Yellow Dog Linux". *http://www.terrasoftsolutions.com/products/ydl/*.

[The] The Board of Regents of the University of Wisconsin System. "Translation and Open Reading Frame". *http://www.npac.syr.edu/projects/cpsedu/summer98summary/examples/hpf/hpf.html*.

[Twe05] S. P. Meyn and R.L. Tweedie. *"Markov Chains and Stochastic Stability"*. Cambridge University Press, 2005.