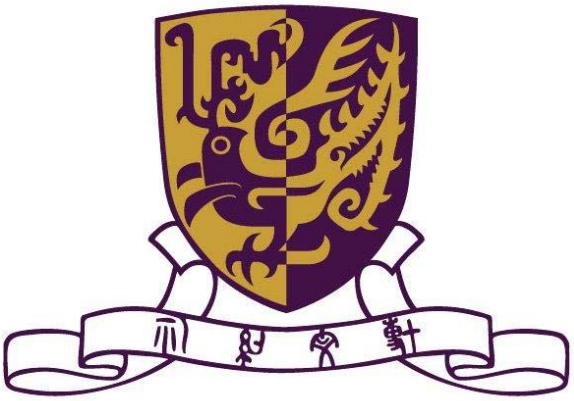
Department of Computer Science and Engineering
The Chinese University of Hong Kong



**2007-2008**
**Final Year Project Report (1st Term)**

LYU0703

# Parallel Distributed Programming on PS3

Huang Hiu Fung
05700512
hfhuang5@cse.cuhk.edu.hk

Wong Chung Hoi
05596742
chwong5@cse.cuhk.edu.hk

## Supervised by

Prof. Michael R. Lyu

# Abstract

This report covers our study and progress in parallel programming in this semester. It begins with some background information about multi-core processors, the motivation and objectives of our final year project.

Then it is followed by an overview of our developing environment. We give a brief description about the architecture of the Cell Broadband Engine, which is the multi-core processor used in PlayStation®3.

The next topic is an introduction of the principal of parallel programming we have studied. We will discuss some patterns for parallel programming, together with different types of algorithms we tried.

After this, we will present our experience in optimizing and porting a video comparison program to the PlayStation®3. We will compare the sequential and parallel approach, and demonstrate how we applied the features of parallel programming to have significant performance increase.

The last portion is a discussion about the project difficulties, project progress and our future works in the next semester.

# Table of Contents

# Chapter 1 Introduction

This chapter would briefly describe parallel programming. And discuss the reason why we have to use parallel programming instead of sequential one. The project motivation and objective are stated in this chapter also.

- Background Information
- Limitation of Single-Core Processor
  - Memory Access Latency
  - Wire Delays
  - Power Consumption
- Development of Multi-Core Processor
  - Reducing Power Consumption
  - Efficient Processing of Multiple Tasks
- Project Motivation
- Project Objectives

## 1.1 Background Information

In the computer industry, we are always looking for faster ways to solve a problem, both faster algorithms and faster computers. There have been tremendous advances in microprocessor technology in the past decades.

According to the Moore's Law, the computing power of processors doubles in every 18 months.

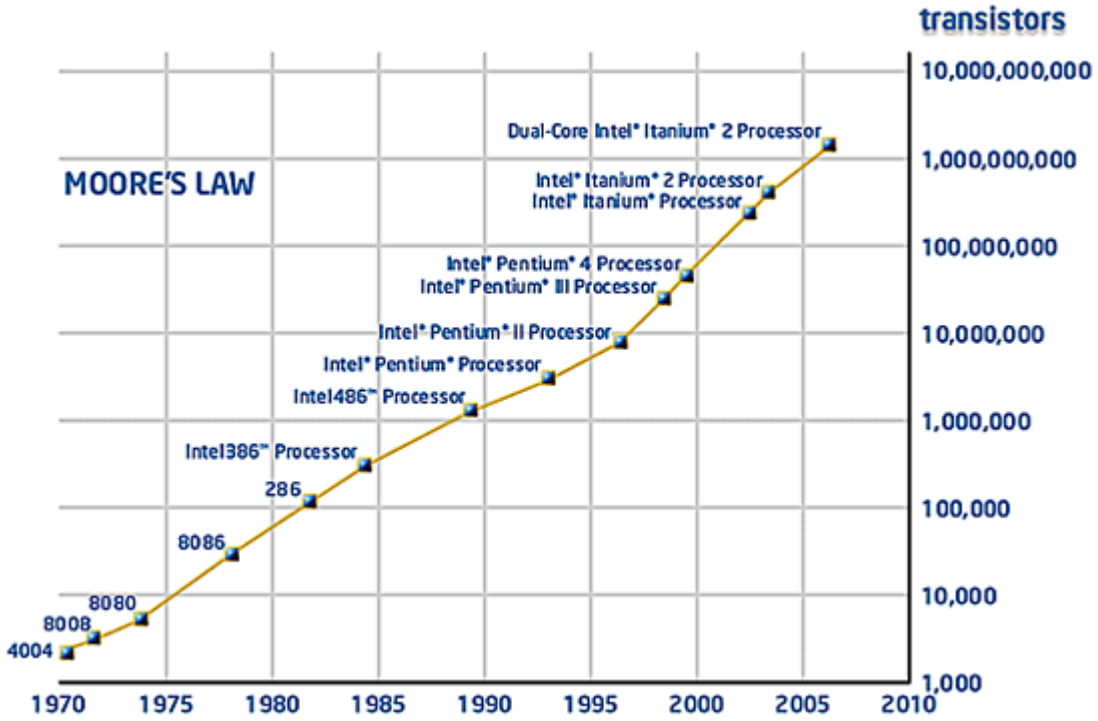



Fig 1.1 Moore's Law showing the growth of transistor.

In 1995, Intel's Pentium chip ran at 100 MHz, while in 2002, the clock rates of processors have increased to 3 GHz in an Intel Pentium 4 model.

These great advances are mainly achieved by frequency scaling, which means to increase the number of cycles per second (processor frequency) in a processor.

$$Runtime = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycles}$$

However, in May 2004, Intel announced the cancellation of its Tejas and Jayhawk processors, which are supposed to be the next 4 GHz Pentium chips. They decided to put more development effort on multi-core processors instead of single-core processors, which are the dual-core and quad-core chips we can see today.

## 1.2 Limitation of Single-Core Processor

The cancellation indicating that increasing the processor frequency of a single-core chip is no longer an efficient way to improve the performance of a processor. The wall of performance limit is hit because of 3 main reasons:

### *1.2.1 Memory Access Latency*

Firstly, the speed of memory is not increasing as fast as the CPU. The overall speed of computation is not only determined by the processor frequency, but also how fast it can access data in the memory. The access time to DRAM has been improving at 9~10% per year, while the performance of processor has been improving at 60% per year.

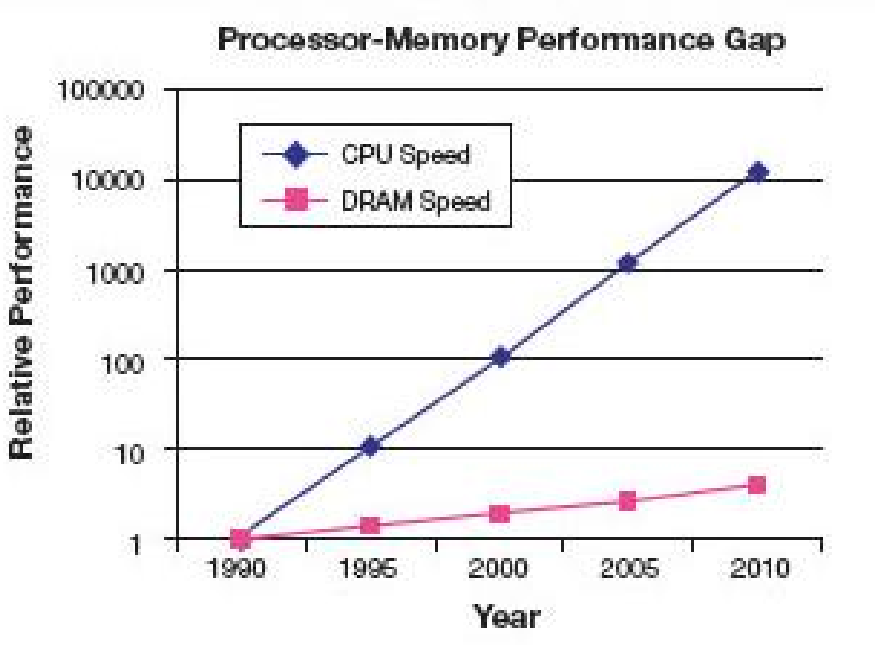



Fig. 1.2 Graph of Processor-Memory Performance Gap

The widening processor-memory performance gap can nullify the benefits in processor frequency increase, and hence is an important performance bottleneck.

### 1.2.2 Wire Delays

Second, the transistors on a single-core chip are becoming denser. This implies that longer wires are required to interconnect them. The path delay can cancel the speed increase of the transistors.

### 1.2.3 Power Consumption

The third one is the most important reason for the cancellation. The single-core processor has reached its performance limit for the amount of power it consumed.

For a processor, $P = C \times V^2 \times F$, where P is power, C is the capacitance being switched per clock cycle, V is voltage, and F is the processor frequency (cycles per second). The power consumption grows with the processor frequency. This increase in power density will produce more heat consequently.

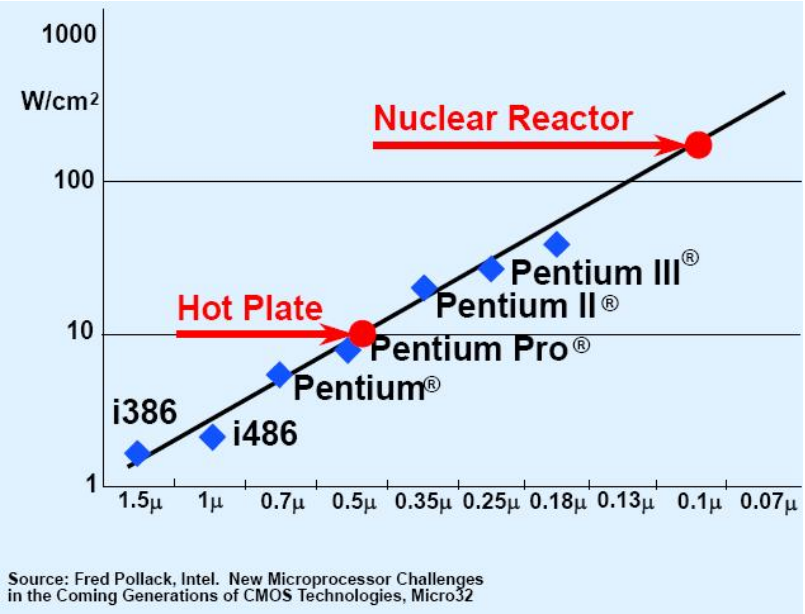



Fig. 1.3 Graph of Power Density over Minimum IC Feature Size

If the processor frequency in a single-core chip continues to grow, it may soon generate heat as much as a nuclear reactor. Therefore, the power consumption and hence the heat problem are the major obstacles that limit the frequency's increase.

## 1.3 Development of Multi-Core Processor

We have mentioned the limitation of single-core processors. In order to continue to improve the processor performance, new chip architectures – the multi-core processors, are developed.

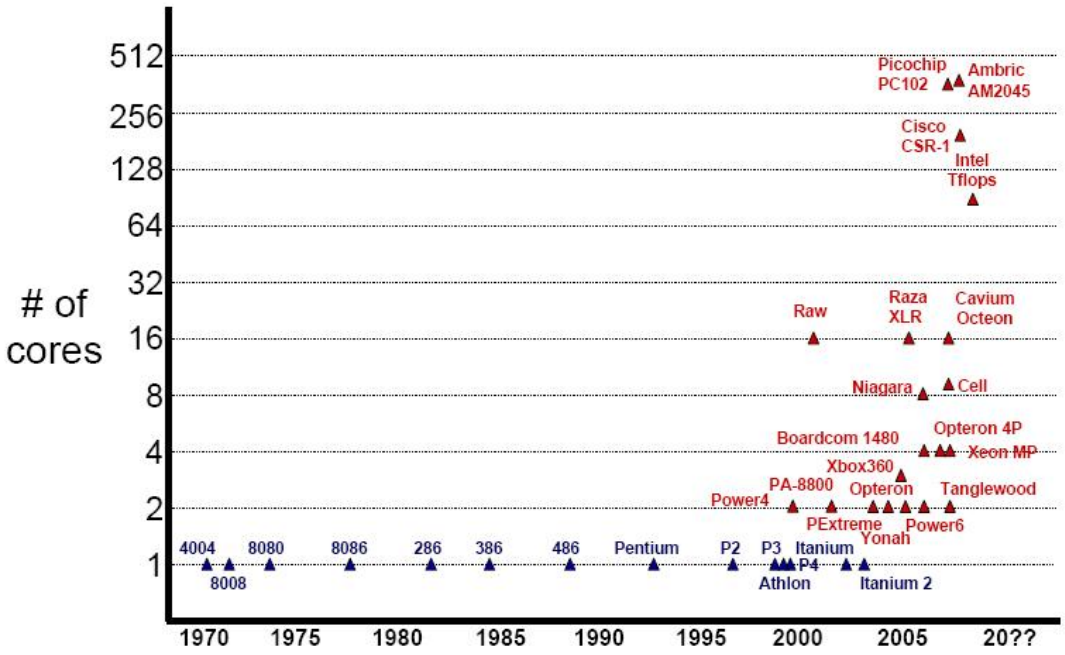



Fig. 1.4 Growth of No. of Cores in Processors

From the above graph, we can see that the trend of processor has changed from single-core to multi-core processor in recent years. Dual-core and quad-core chips for desktop machines are becoming popular today.

The development of has shifted to multi-core processor because there are several advantages over the single-core processor:

### 1.3.1 Reduce Power Consumption

Using multiple cores with low frequency instead of one with high frequency can reduce the power consumption, while still delivering better performance at the same time.

### 1.3.2 Efficient Processing of Multiple Tasks

Traditionally, we can only solve a problem through serial computation. While with parallel platform, we can divide the computation work into discrete parts and execute among the cores concurrently.

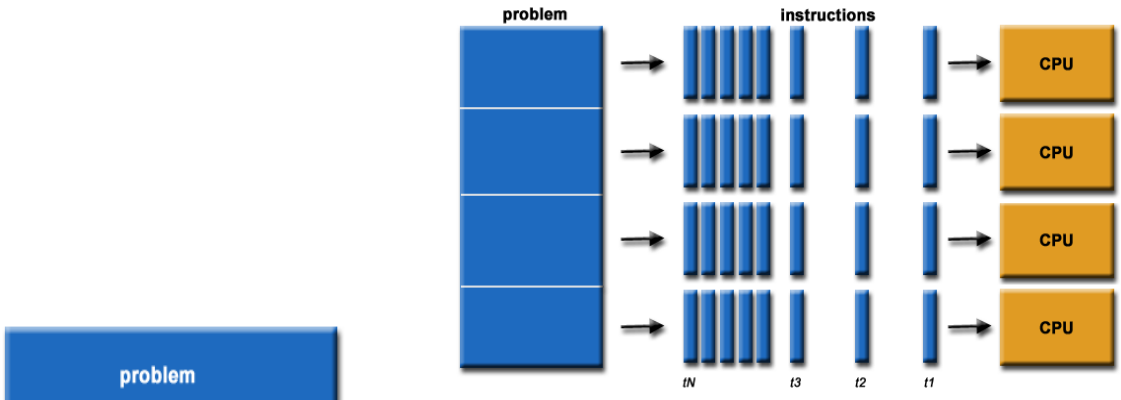



Fig. 1.5b Parallel Computation

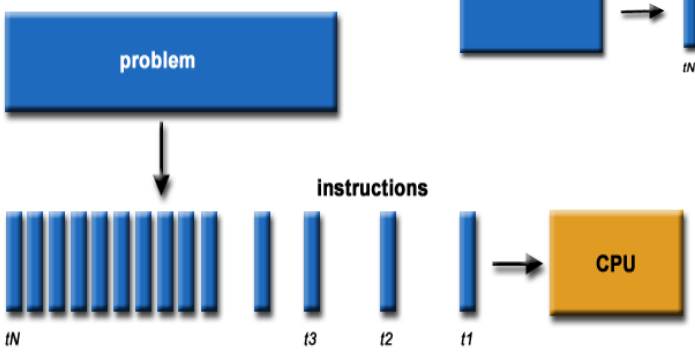
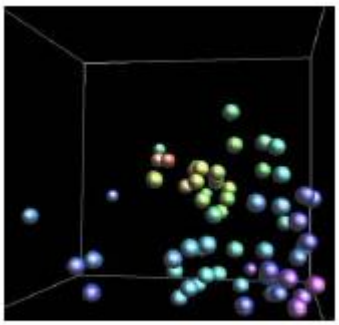


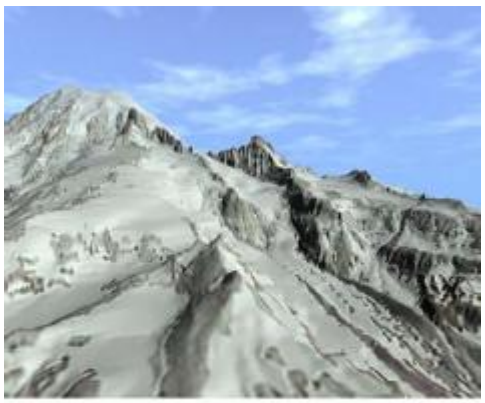

Fig. 1.5a Sequential Computation

## 1.4 Project Motivation

There are many applications that require large amount of data manipulation and computation, such as advanced graphics, virtual reality, simulation and multimedia processing.



Physics Simulation



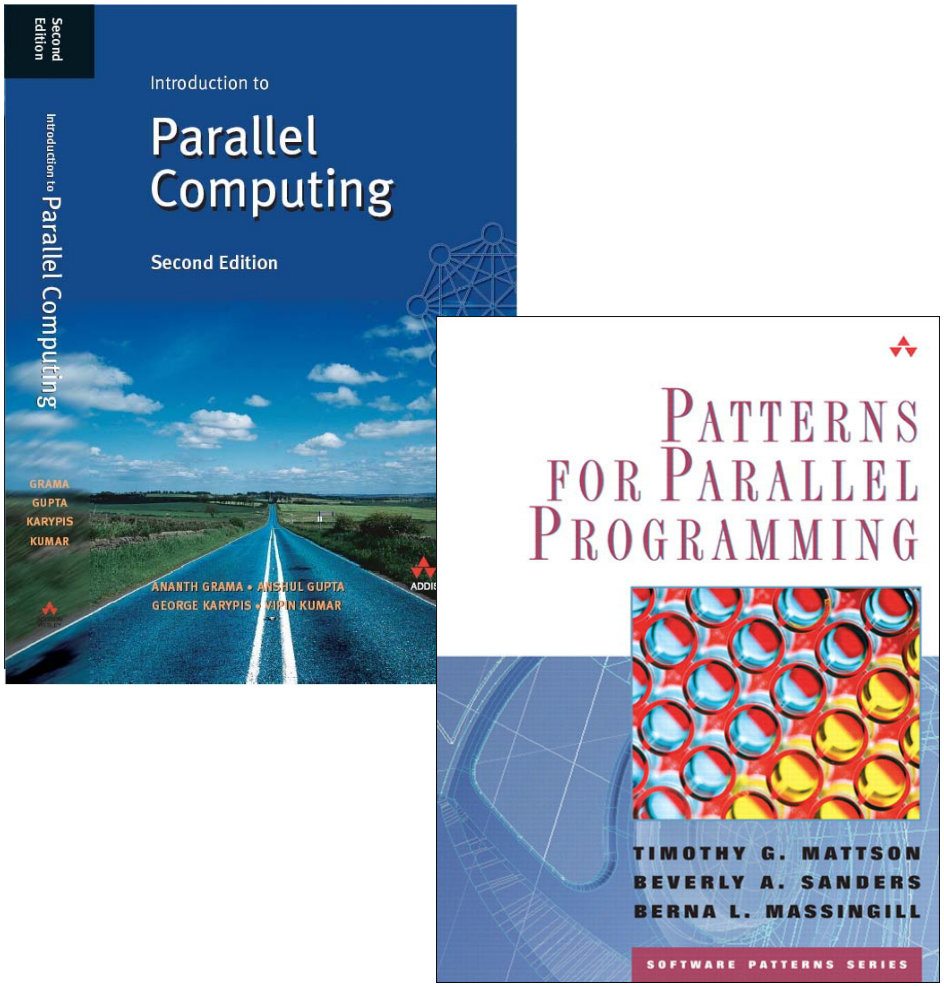
Terrain Rendering Engine

Fig. 1.6 Applications that require intensive computation

The multi-core technology has already brought us a hardware impact in improving the performance of processors. It will be the future hardware trend too. Although application that run on a single-core machine can still run on a multi-core one, it is useless to do such porting. In order to take advantages of multi-core architecture, software engineers have to optimize and parallelize the program.

Despite over 30 years of compilers and tools development in sequential programming, parallel programming is relatively new, compilers and tools are often not mature. Our group is greatly interested in parallel programming and finds this challenging.

We also realize that multi-core processors will be the dominant processors available that with high performance. And we believe that parallel programming will pay a more important role in future software engineering. Therefore we start this project, hoping to have some achievement in parallel programming.



## 1.5 Project Objectives

In this project, we will first study the features of parallel programming and have some hands-on experience on it. We will also compare and analyze the performance difference between sequential and parallel programs. To start with, we choose to program on a multi-core machine, i.e. PlayStation®3, rather than program over distributed machines.

After that, we will select an application which require large amount of data manipulation and computation. Modification from sequential approach to parallel approach will be made. We will use what we learnt to optimize the program to the largest extent, showing that great improvement can be made with parallel programming and a multi-core machine.

# Chapter 2 Development Environment

In this chapter we will introduce the development environment of our project, including both hardware and software.

For hardware, we have two PCs and a PlayStation®3 running Windows XP and Linux respectively.

For software, we use the IBM Cell Software development Kit, which provide the compiler and libraries for parallel programming on the Cell processor.

We will introduce one by one in this list:

- Personal Computer
- PlayStation®3
- Cell Broadband Engine
  - Power Processor Element
  - Synergistic Processor Element
  - Element Interconnect Bus
  - Memory Management
- Linux
- IBM Cell Software Development Kit

## 2.1 Personal Computer

We are provided with two PCs for our project use. Although we do our parallel programming on the PlayStation®3, the configuration of PCs is still worth mention. We also run programs on PCs as a reference, to compare the performance of a sequential program and its parallel version on PlayStation®3.

| Table 2.1 Major Specification of the PC | |
|---|---|
| CPU | Intel Pentium 4 3.0 GHz |
| Main Memory | 1 GB RAM |
| Operating System | Windows XP Professional Edition |

## 2.2 PlayStation®3

PlayStation®3 is the multi-core machine we used in this project. It is the third generation home video game console produced by the Sony Computer Entertainment, first released on November 2006. It uses the Cell Broadband Engine (Cell BE), which has great computation power, as the processor, giving high quality of game and graphics performance. The following is the basic hardware configuration of the PlayStation®3 we used.

Fig. 2.1 PlayStation®3 produced by Sony

| Table 2.2 Major Specification of the PlayStation®3 | |
|---|---|
| CPU | 3.2 GHz Cell Broadband Engine |
| Main Memory | 256 MB XDR DRAM |
| Hard Disk | 60 GB 2.5" SATA hard drive |
| PlayStation® System Software (i.e. the game OS) | Version 1.94 |
| Operating System | Fedora 7 (Linux Kernel 2.6.21) |

Although the major reason for designing PlayStation®3 is to be the next generation of video game console, its strong computation power and relatively low cost for a multi-core machine (about HK $3,000) making it possible to have different applications and development. Sony also opened the platform to allow other OS (typically Linux) to be installed on PlayStation®3.

For example, the Cell BE of PlayStation®3 is utilized to achieve great performance in the Folding@Home project launched by the Stanford University. Folding@Home is a distributed computing project designed to perform simulation of protein folding and other molecular dynamics which require highly intensive computation. Nowadays, with about 50,000 active PlayStation®3, the console accounts for over 60% of all the computation, making it the most powerful distributed computing network in the Guinness World Record.

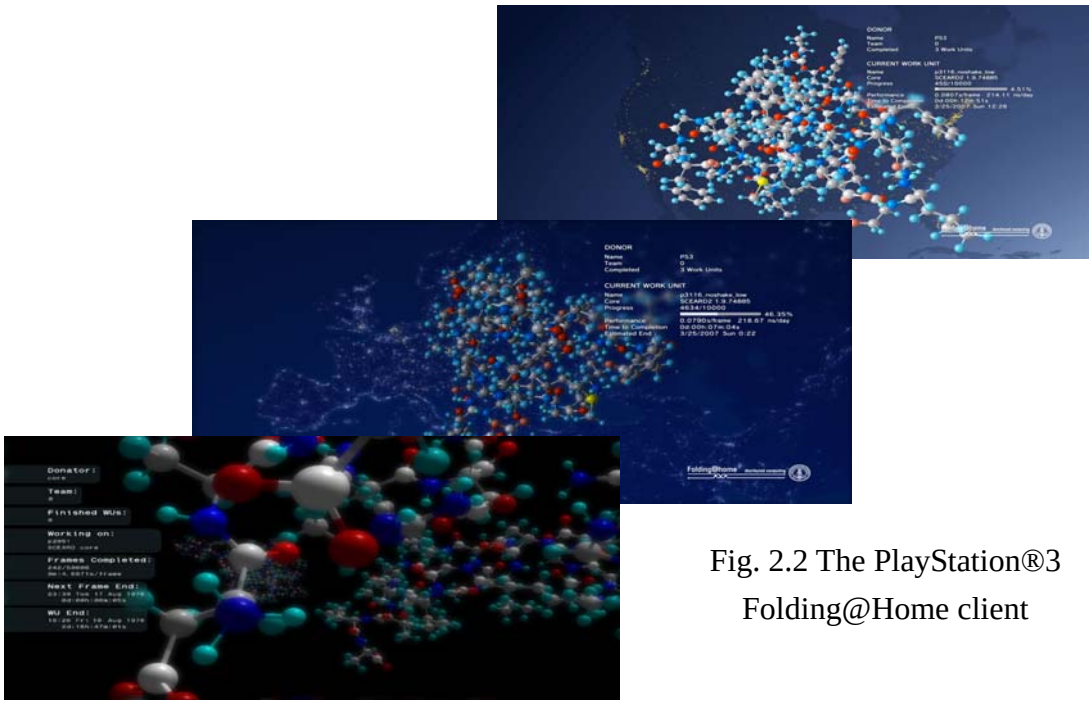


Fig. 2.2 The PlayStation®3 Folding@Home client

## 2.3 Cell Broadband Engine

As mentioned above, the Cell processor is the soul of the PlayStation®3. Cell, with full name Cell Broadband Engine Architecture, is jointly designed by Sony, Toshiba and IBM, started in 2001.

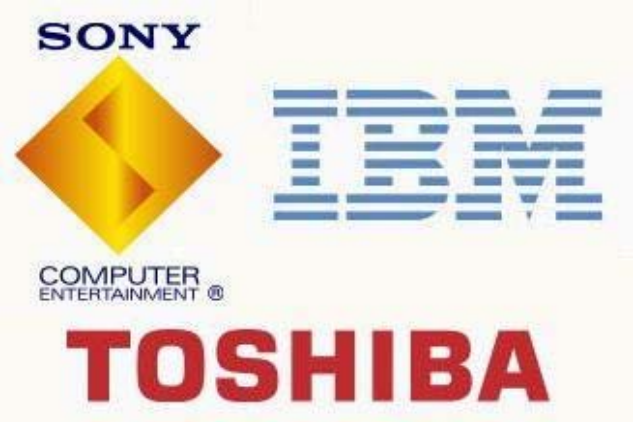



Fig. 2.3 STI alliance that developed Cell

An overview of the Cell architecture is shown as follow:

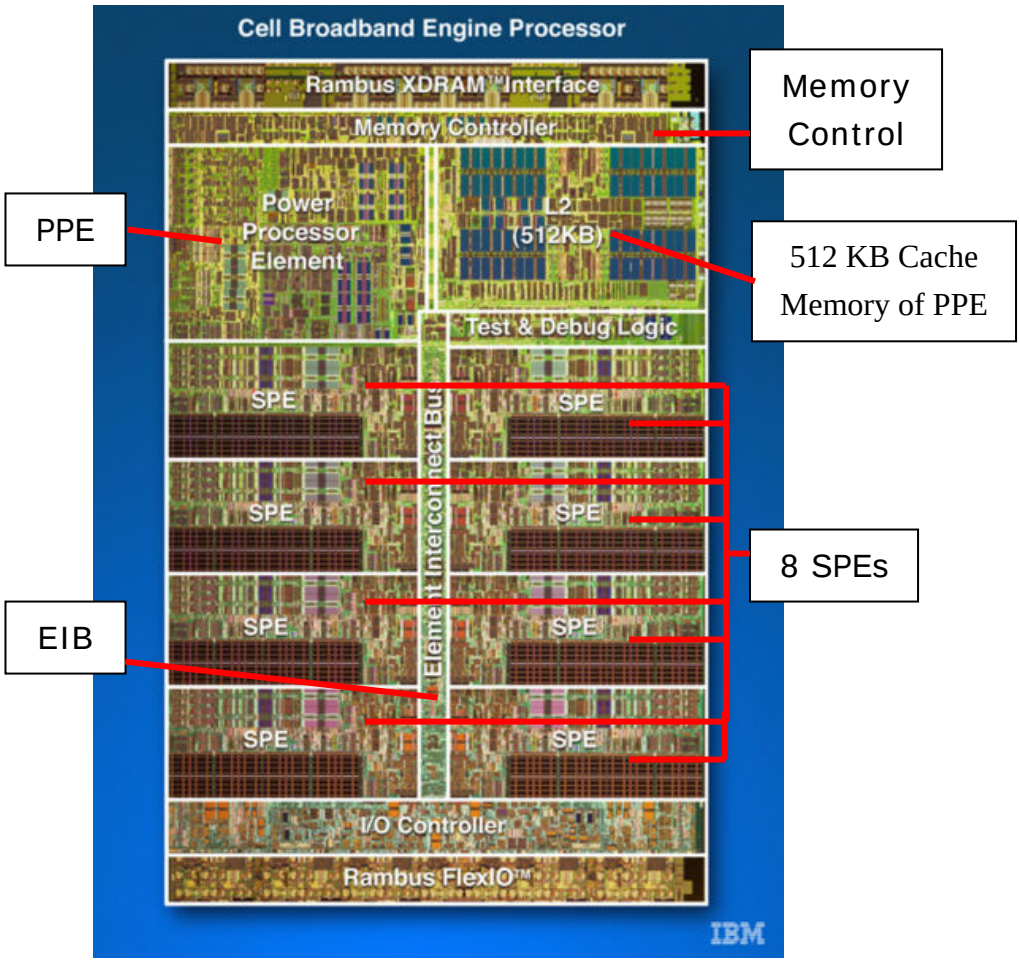



Fig. 2.4a Real architecture of Cell

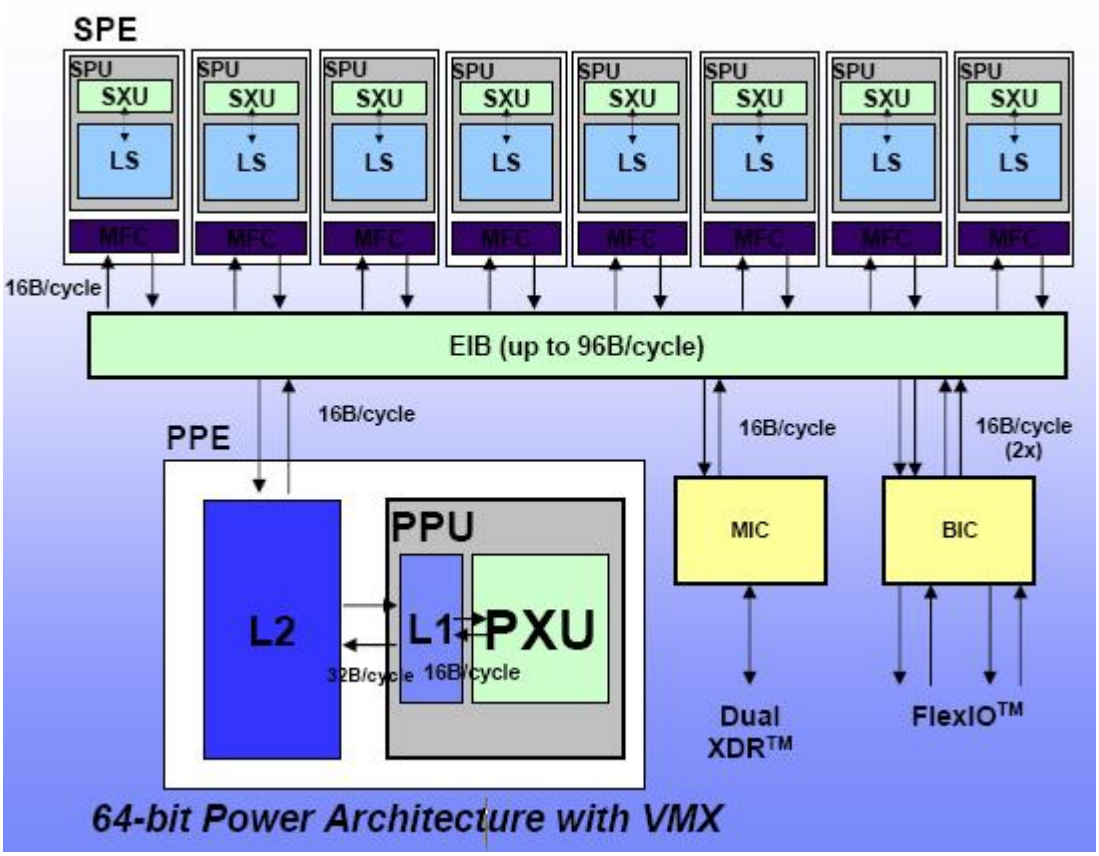



Fig. 2.4b Overview of Cell Architecture

Figure 2.4a and 2.4b has already shown the architecture and some of the major components of Cell BE. We are going to give more details for each of them.

### 2.3.1 Power Processor Element (PPE)

This is a general purpose, 64-bit PowerPC architecture based and two-way multi-threaded processor. It has 32 KB L1 cache and 512KB L2 cache. The PPE acts like a 64-bit PowerPC processor, allowing the execution of both operating system and applications.

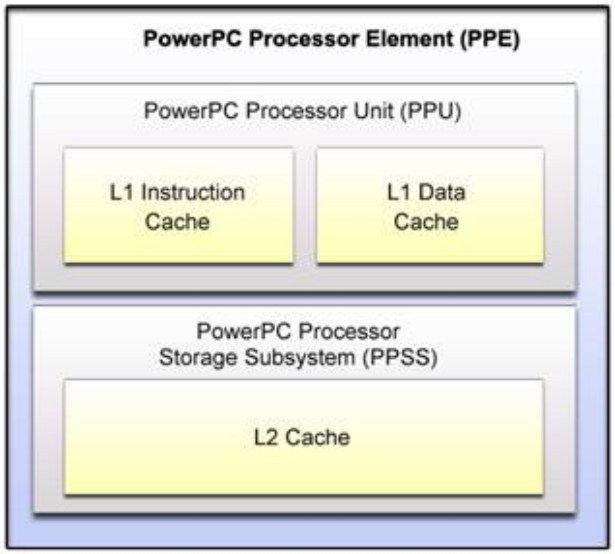



Fig. 2.5 Design of PPE

The PPE is designed as a control-intensive processor in Cell which mainly process control, including:

1. The I/O of accessing the main memory and other external devices requested by the operating system
2. The control over all 8 SPEs

### 2.3.2 Synergistic Processor Element (SPE)

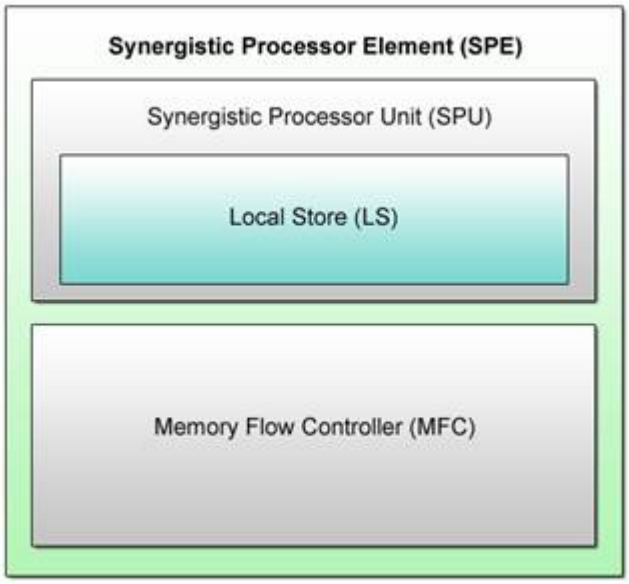



Fig. 2.6 Design of a SPE

The SPE is less complicated than the PPE, since it is target to provide computation performance, not control-intensive one. Every SPE consists of three main parts:

1. A Synergistic Processor Unit (SPU), to perform its allocated task.
2. A 256KB Local Store (LS), which is the only memory accessible by the SPU.
3. A Memory Flow Controller (MFC), to control data transfer between the SPE's LS and the main memory or other SPEs.

There totally 8 SPEs in the Cell BE. While for the Cell BE inside PlayStation®3, 1 SPE is disabled and 1 is reserved for the system software (i.e. the game OS), meaning that only 6 SPEs are accessible for programming under Linux.

### 2.3.3 Element Interconnect Bus (EIB)

The EIB is an internal communication bus built on Cell so as to connect different elements on the chip, such as PPE, SPE and memory controller. It is implemented as a circular ring of four channels, supporting multiple simultaneous transfers. Hence faster data or message exchange between elements is achieved.

### 2.3.4 Memory Management

The figure on the right shows how an SPE get data from the main memory. It is a typical example of accessing main memory, during programming the Cell BE. The process is described as follow:

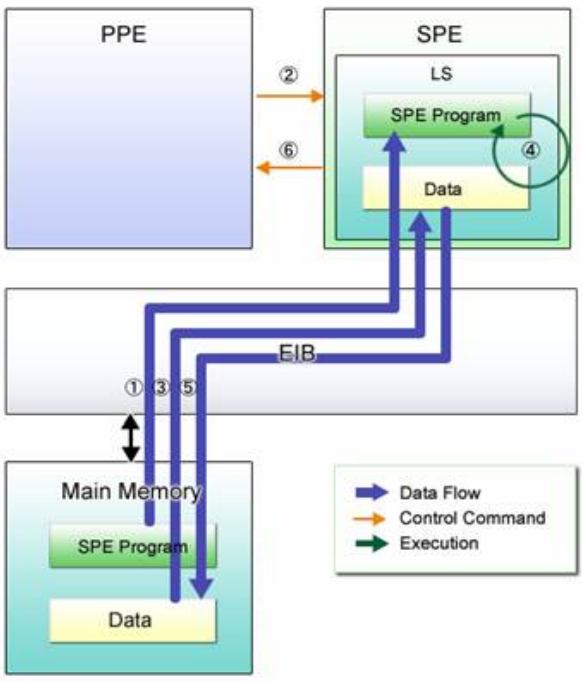



Fig. 2.7 Data Flow and Program

1. PPE ask the SPE to run its program.
2. SPE need to access data in the main memory
3. MFC in SPE handle the situation and get data from the main memory with Direct Memory Access (DMA)
4. The result data is store in the LS of the SPE.
5. All the communication is done through the EIB.

## 2.4 Linux

Although PlayStation®3 has its own system software, Sony has also opened the platform for third-party OS. The most common one is Linux. People have tried to install different distributions of Linux on it, such as Yellow Dog, Fedora, Ubuntu. All these are tested and can operate on the console.



Among them, we finally chose Fedora 7, with kernel updated to 2.6.23, as our developing environment. This is mostly because IBM officially declared that Fedora 7 is compatible with the Cell SDK, which is an important tool for our parallel programming.

## 2.5 IBM Cell Software Development Kit

The IBM Cell Broadband Engine SDK provides a complete Cell BE development environment. The SDK contains important tools for our parallel programming development, including:

- Libraries (SPE Run-time Management Library, SIMD math library)
- Samples source code
- IBM XL C/C++ Alpha Edition for Cell BE Processor
- IBM Full-System Simulator for the Cell BE Processor
- An Eclipse-based Integrated Development Environment
- GNU GCC compilers for PPU and SPU etc.

We use the version 2.1 to do our project. While in late October 2007, version 3.0 is released. We may shift to newer version in future.

# Chapter 3 Principals of Parallel Programming

## Parallel Algorithm vs. Serial Algorithm

Parallel algorithm is different from traditional serial algorithm. Traditional serial algorithm makes use of 1 CPU, executing command one by one and computes the final result. Parallel algorithm tries to make use of more than 1 processing unit for computing at a time. The result from each processing unit has to be put back together for the final result.

| Parallel algorithm | Serial algorithm |
|---|---|
| multiple processing units | single processing unit |
| communication overhead | no communication overhead |
| higher complexity in code | straight forward code |
| ensure load balance between PU | everything is done by CPU |

The above table shows some different between parallel algorithm and serial algorithm. It is useful when we have to decide whether a problem should be solved by parallel algorithm or serial algorithm.

Simply speaking, for problem which required heavy computation, parallel algorithm would be ideal as the communication overhead becomes negligible. For problem which has simple algorithm, it is not necessary to parallelize it as it increases the complexity of the code. Besides, the overhead in communication also becomes dominant.

# Concept of Load Balance

Load balance is an important concept. How the mappings of task to processing units are done can have a significant impact on the overall performance of a parallel algorithm. It is crucial to avoid the situation in which a subset of the processing units is doing most of the work while others processing units are idle most of the time.

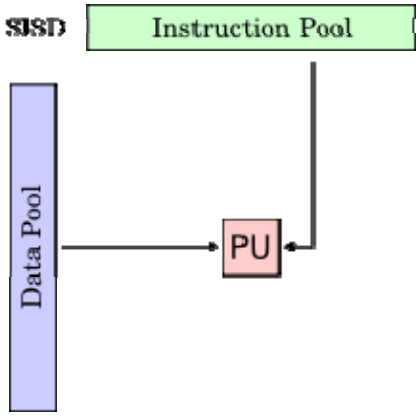
If load balance is not ensured, the total runtime of the program will be the runtime of the busiest processing unit. Furthermore, the computation time of idling processing units is wasted. Thus jobs should be distributed as evenly as possible.

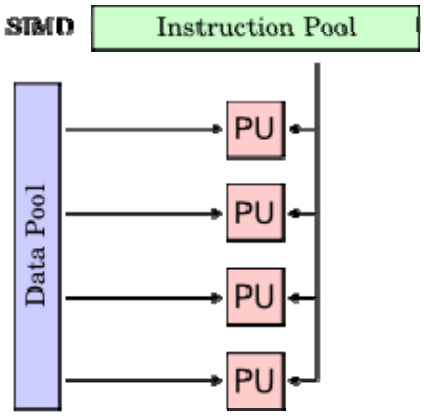
# Parallel Architecture

| Flynn's taxonomy | | |
|---|---|---|
| | **Single Instruction** | **Multiple Instruction** |
| **Single Data** | SISD | MISD |
| **Multiple Data** | SIMD | MIMD |

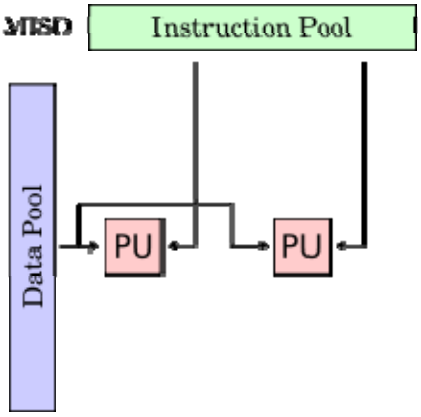
Flynn's taxonomy is the most common way to classify parallel architecture. Flynn categorizes all computers according to number of data streams and number of instruction streams they have. As shown in the above table, there are 4 types of them: SISD, MISD, SIMD and MIMD.

SISD – Single instruction single data
This the most common von Neumann model with a single processer, an instruction stream and a data stream. The instructions are carried out one by one.

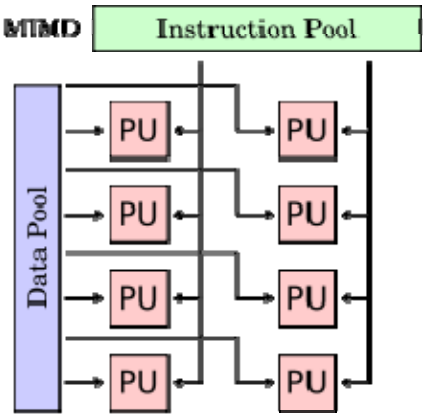



SIMD – Single instruction multiple data
In this case, the instruction stream is concurrently broadcast to multiple processers. Different stream of data are processed by different processers with the same instruction. For example, the SPE intrinsic functions provided by CellSDK are SIMD. They can apply same operation on every element of the input vector at the same time.




MISD – Multiple instruction single data
There are no well-known systems fits in this category. It is mentioned for completeness.

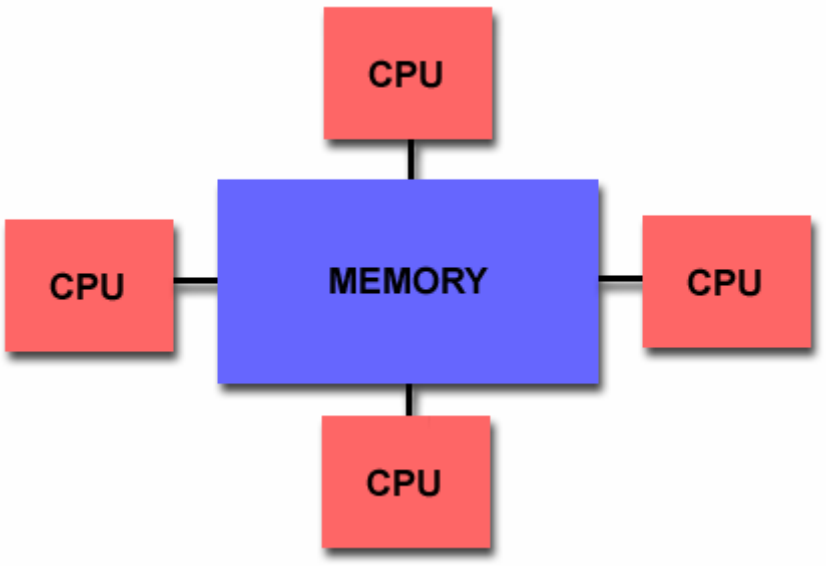



MIMD – Multiple instruction multiple data
This system has its own stream of instruction operating on its own stream of data. This is the most common type of parallel system. However, MIMD is a rather general description of a system. Therefore it can be further break down into the following.

# Shared-Memory System and Distributed-Memory System

Systems we use to implement parallel program are classified into 2 types according to their distribution of physical memory, namely shared-memory system and distributed-memory system.

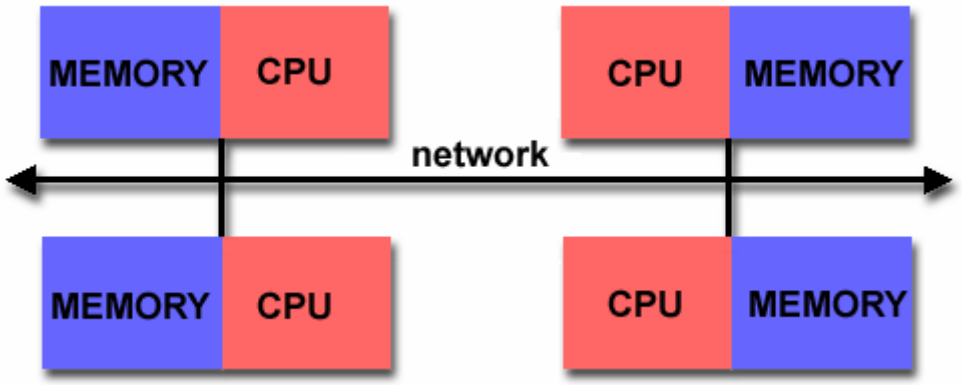
Share-memory system refers to a system with large block of random access memory that can be access by several different central processing units in a multiple-processor computer system.




This type of system is relatively easy to program as every process access to same piece of data. They need a fast method to access central memory. Also, the memory coherence has to be maintained carefully.

In PlayStation®3, each SPE can access the main memory via Memory Flow Controller (MFC) by issuing Direct Memory Access (DMA) command. Therefore PlayStation®3 can be regard as a shared-memory system.

On the other hands, distributed-memory system refers to a multiple-processor computer system in which each processing has its own private memory. Task has to be distributed to different processors for processing. After that, data has to be reassembled to generate a meaningful output.




In PlayStation®3, each SPE has a local store (LS) of size 256 KB. LS are the working space for SPE. With LS, SPE can be assigned with task or data by the PPE, achieving parallel programming.

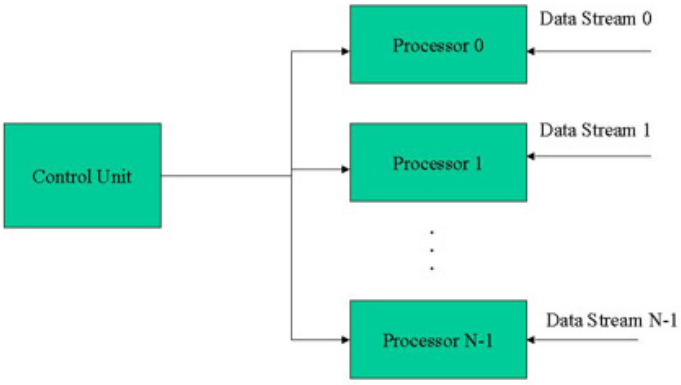
Actually, PlayStation®3 has both shared-memory and distributed-memory features. Thus, it can be classified as hybrid distributed-shared memory architecture.

This gives programmer a high degree of freedom when trying to parallelize a serial program or designing a parallel program in PlayStation®3 platform. It makes PlayStation®3 an ideal environment for parallel program development.

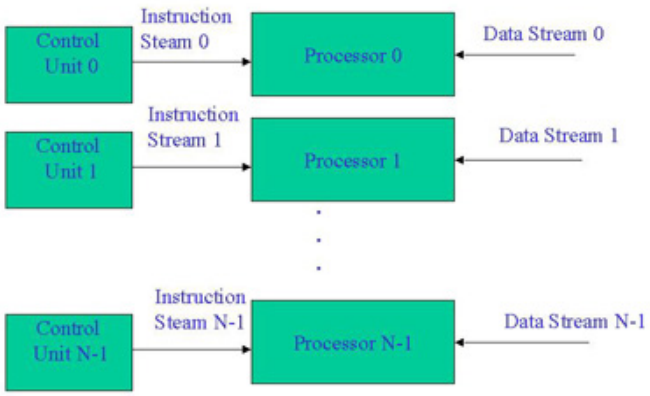
# Data Parallelism and Task Parallelism

Data parallelism and task parallelism are different way to write parallel program.

Data parallelism is achieved by splitting data into smaller parts. Then distribute the smaller parts to different processing unit for computation. Usually, it happens within SIMD system where single set of instruction is used to operate on multiple set of data.




Task parallelism is achieved by splitting program into different task. These tasks are assigned to different processing units. They can operate on same or different data. They can also communicate with each other and passing data around. Task parallelism usually occurred in MIMD system.



# Synchronization

Synchronization controls order of task execution. There are 2 types of synchronization: process synchronization and data synchronization.

Process synchronization means multiple tasks are depending on each other in order to complete the own task. They have to wait for input, respond from the other task. When designing this type of program, we must be carefully looking out for deadlock.

Data synchronization means there are multiple tasks accessing the same data, we must ensure that the value read by the tasks is the most up-to-date one. Whenever there are updates on the data, we need to make sure every other task are using the new value.

Synchronization problem is fairly common in parallel program. Tools like semaphores, monitors, mailboxes are useful to write such kind of program.

# Chapter 4 Optimization of the ADVISE Program

So far we have studied several principals of parallel programming. We also have some hands-on experience on writing programs on the PlayStation®3 with the Cell SDK. Thus we select a program which requires intensive data manipulation and computation. And we try to optimize it to see how much performance we can improve, making it a more efficient application. The details of the program and the process are described in the following parts.

- Introduction of PC Version ADVISER Program
- Porting PC Version to PlayStation®3 Platform
  - Inconsistent Representation in Different Platform
  - Working out the Algorithm
  - Communication Between PPE and SPE
  - The Flow of the Parallel Program
- Time Attack and Optimization
  - Making Use of SIMD Intrinsic
  - Changing the Data Type
  - Implementing Double Buffering
  - Parallel Reading for All Files
  - Distributing Job to Idling PPE
  - Applying SIMD for Loop Counter
  - Optimizing by Loop Unrolling

The program we selected is called ADVISER. ADVISER is a program which compares 2 video clips and looks for similarity. It is useful for applications like locating the advertisement that appears every night. However, since video clip contains a large amount of data, the comparison procedure is not as easy as comparing texts or pictures.

Therefore, the program is divided into 3 parts:

1. Generating meaningful data (in form of numbers) of frames from the video
2. Comparing and looking for the most similar frames
3. Locating the similar segment which consist of a series of very similar frames

We are given the PC version of the 2nd part of the program. Our objective is to speed up the program by porting it to the PlayStation®3 system and making use of its parallel computation power.

## 4.1 Introduction of PC Version ADVISER Program

Here is a brief introduction of the PC version program. Given 2 folders named “Repository” and “Target”. Each contains many “*.hl3” files which are video frame’s data extracted from the video clips by another program. These “*.hl3” files are actually consists of 1024 double precision value.

For each “*.hl3” file in “Target” folder, we need to map it to another “*.hl3” file in “Repository” directory such that square of their Euclidean distance is smallest. Mathematically, given a “Target” file P $(p_1, p_2, \cdots, p_{1024})$, we try to look for another file Q $(q_1, q_2, \cdots, q_{1024})$ in “Repository” such that the value $\sum_{i=1}^{1024}(p_i - q_i)$ is minimum. By doing this, each frame in the “Target” folder is mapped to the most mathematically resemble frame in the “Repository” folder.

The PC program is written in a traditional sequential way. First, it reads in every file in the “Target” directory and assigns an array for each file for later computation.

Second, it reads in files in “Repository” directory one by one. For each read in file in the “Repository” directory, the square of Euclidean distance with every file in “Target” directory is computed. If the square of Euclidean distance computed is smaller than the minimum

value found so far, it is updated.

Finally, the target file name, the corresponding repository file name, and the minimum score are outputted. Assume there are *m* files in the "Target" directory and *n* files in the "Repository" directory, the above algorithm is $O(m \times n)$.

There will be 2 output files of this ADVISER program.
First one is a text file containing a list of most matched files.
That is:

target hl3 1 most match repository A difference value = ??
target hl3 2 most match repository B difference value = ??
target hl3 3 most match repository C difference value = ??
etc.

Also in the second half of the text file, it will show the segment that is probably the same video segment (e.g. advertisement)

With these, the second output file, which is a xml file, is made.
It will get the most similar segment from the video and allow you to play it.

## Result of the sequential PC version

The input for this program is:

| Input | No. of hl3 files |
|---|---|
| Target directory | 5473 |
| Repository directory | 7547 |

Each hl3 file is a binary file of 1024 integers

In order to have a reasonable comparison, all the result shown here, including those for PlayStation®3 version, use this set of data as input.

Time to read both directory, without computation = 25 sec

Total elapsed time of the program = 658 sec

Thus, for the sequential ADVISER program on PC:

The net elapsed time = 658 – 25 = **633 sec**

This result will be used in the rest part to compare the performance with PlayStation®3.

## 4.2 Porting PC Version to PlayStation®3 Platform

After studying the PC version program and understanding how it works, we start to design how the program should be ported to the PlayStation®3 platform.

### *4.2.1 Inconsistent Representation in Different Platform*

First difficulty we encountered was the endianness problem. The "*.hl3" file in PC are 1024 double precision values in binary format. Due to the different of representation (big-endian and small-endian representation) under different computer architecture (Intel X86 V.S. PowerPC 64), the "*.hl3" files from PC cannot be read directly at PlayStation®3.

Therefore, we have written 2 programs. One program converts binary representation of double precision values to ASCII representation in the PC. Another program converts ASCII representation of double precision values to binary representation in PlayStation®3. This solves the inconsistency between binary representations in two different systems.

### 4.2.2 Working out the Algorithm

In our parallel distributed computation program, we decide to read in the files in reverse order, that is first read in files in “Repository” directory and then “Target” directory. In this way, we can let each SPE handle a “Target” file and send back only a minimum score and its match to the PPE. Rather than letting each SPE handle a “Repository” file and send back an array of square of Euclidean distance value to PPE.

Now once we have read all files in “Repository”, for each “Target” file we read, we can immediately send them to SPE for processing rather than waiting every files to be read first like the PC version. This could save the read input time, i.e. time is required to read “Repository” only, reading time for each “Target” file will overlap with the computation time.

If we are using all the 6 SPEs, then each SPE just take 1/6 of the total “Target” files for computation. In this way, we can apply data parallelism.

### 4.2.3 Communication Between PPE and SPE

PPE and SPE communicate in 2 ways. One is via Direct Memory Access (DMA). Another way is via mailboxes. For large pieces of information, DMA is used. Mailboxes are used to transfer a few byte of information, like an integer or a flag.

CONTROL_BLOCK is transfer by DMA. It contains information a SPE need to know. Such as the address of data of “Repository” files, address of data of “Target” files, number of “Repository” files, address for sending back result to PPU, etc. With all these addresses and an index, it can access any files in the main memory by DMA.

```
typedef struct _CONTROL_BLOCK {
  double** repos_data_addr;
  char* target_file_addr;
  double** target_data_addr;
  unsigned int repos_num;
  OUTPUT_BLOCK* output_addr;
  unsigned int start;
  unsigned int padding[2];
} CONTROL_BLOCK;
```

Similar to CONTROL_BLOCK, OUTPUT_BLOCK is transfer by DMA. It contains only 2 integers. One is the index of mapped “Repository” file. Another is the score of it.

```
typedef struct _OUTPUT_BLOCK {
  int min_file_idx;
  int min_diff;
  int padding[2];
} OUTPUT_BLOCK;
```

### *4.2.4 The Flow of the Parallel Program*

Here is the brief introduction on the flow of the parallel program.

Program flow of PPE:

1. All files in “Repository” are read and stored in a structured array.
2. Control blocks for each SPE are generated.
3. 6 SPE threads are created and they read in their own control block by DMA.
4. A “Target” file is read and stored in array.
5. If any SPE is not busy, send the index of read file to SPE, else stall the PPE.
6. Loop back to 4 until all “Target” files are read.
7. Issue a “Finish” signal to all SPE.
8. Wait all SPE thread to terminate.
9. Write out the result to a file.

Program flow of SPE:

1. A SPE thread is created with address of Control Block as parameter.
2. Fetch the control block by DMA.
3. Check mailbox for index of “Target” file.
4. Process the “Target” file.
    A. *Fetch a “Repository” file’s data via DMA.*
    B. *Compute the square of Euclidean distance.*
    C. *Update the value if it is smaller.*
    D. *Loop back to A until all “Repository” file is processed.*
    E. *Fill in an Output Block.*

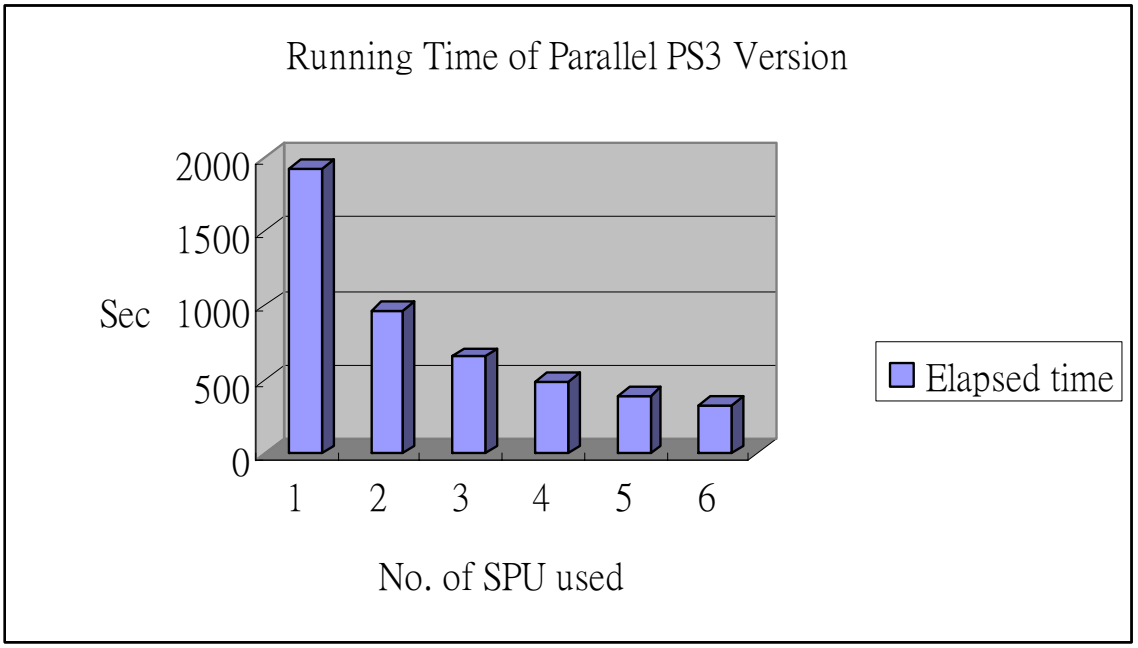
*F. Send back result to PPU via DMA.*

5. Loop back to 4 until "Finish" signal is received.
6. Notify PPE via mailbox that it is terminating.
7. Thread terminated.

## Result of the parallel PS3 version

| No. of SPU used | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Read input time (sec) | 3 | 4 | 4 | 5 | 3 | 4 |
| Total Elapsed time (sec) | 1931 | 968 | 661 | 491 | 394 | 334 |
| **Net Elapsed time (sec)** | **1928** | **964** | **657** | **487** | **391** | **330** |

**Control**




As we can see from the graph, the elapsed time decreases with increasing number of SPU used. The relationship can be expressed as:

$$\frac{\text{Elapsed Time of 1 SPU}}{\text{Elapsed Time of N SPU}} = N$$

Since there is no parallelism at all when the program is execute with 1 SPU, thus the result of 1 SPU (1928 sec) is used as a control. It represents the performance of the sequential version on the same platform.

Hence with all 6 SPU used for computation, we have performance **6 times** faster than the control

The best result we got so far is **330 sec**.

As a reference, we have also taken the elapsed time for the program to execute on PPU only, i.e. a sequential program

The result is:

Elapsed Time of PPU = **3119 sec**

When compared to the elapsed time for 1 SPU, which means no parallelism in fact, the one of PPU is much larger (3119 sec to 1928 sec). We can conclude that SPU, which is designed for intensive computation, has greater computation power than the control-intensive PPU.

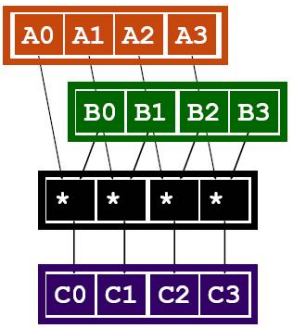
## 4.3 Time Attack and Optimization

Through out the semester, we try to apply different method to speed up out program. At first, we expect our program to have a speed up of about 6X due to parallel distribution of works to 6 SPE.

However the result is not that satisfying. The program can only achieve a speed up of 2X (330 sec to 633 sec). This is due to the relative high processing power of CPU in PC than the PPU in PlayStation®3.

Also PC has access to more memory than PlayStation®3. Yet, after more and more trial and different technique applied, we can finally obtain a speed up of about 12X.

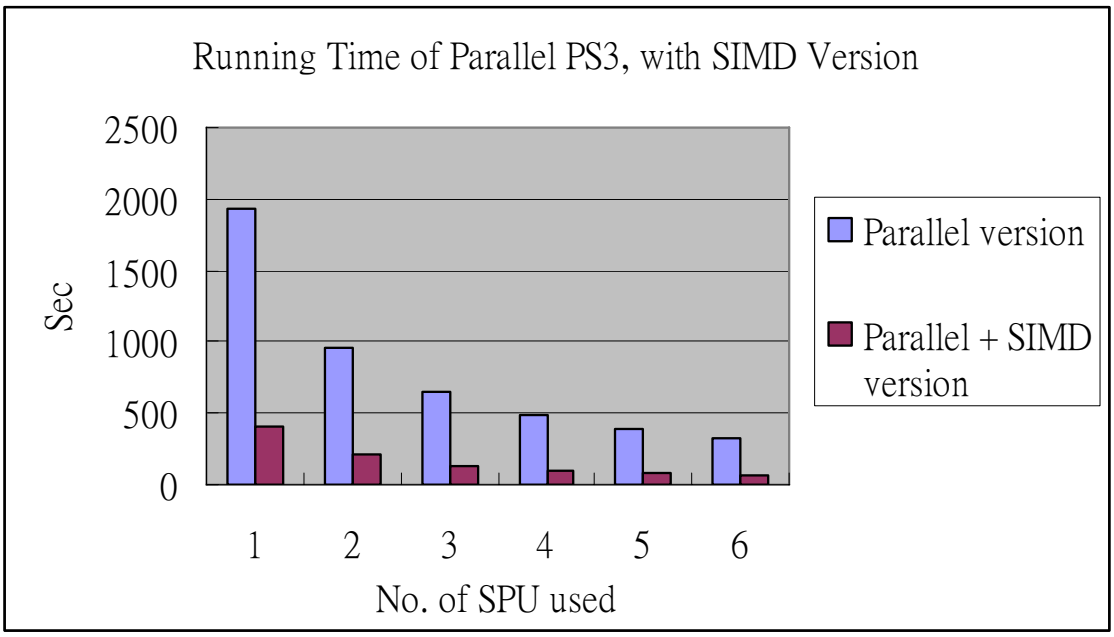
### 4.3.1 Making Use of SIMD Intrinsic (Major Improvement)




SIMD is short forming of single instruction multiple data. It means applying single operation on multiple operands at the same time. CellSDK provide SIMD intrinsic instruction for SPE and PPE. These SIMD instructions take 128 bits register as input. Then apply operation on every element of the vector at the same time. For double precision number we used as our data type, they are 64 bits. Therefore each vector can contain at most 2 double precision values. Simply replacing the operator in formula calculating the square of Euclidean distance yielded a speed up of 2X easily.

## Result of the parallel, with SIMD PS3 version

| No. of SPU used | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Read input time (sec) | 5 | 4 | 4 | 4 | 5 | 6 |
| Total Elapsed time (sec) | 414 | 208 | 140 | 106 | 87 | 77 |
| **Net Elapsed time (sec)** | **409** | **204** | **136** | **104** | **82** | **71** |




After SIMD is applied, the elapsed is further decreased. The SIMD version runs at approximately **4 times** faster than the one without using SIMD. Total 4X6 = 24 times faster than the control.

The best result we got so far is **71 sec**.

### 4.3.2 Changing the Data Type (Major Improvement)

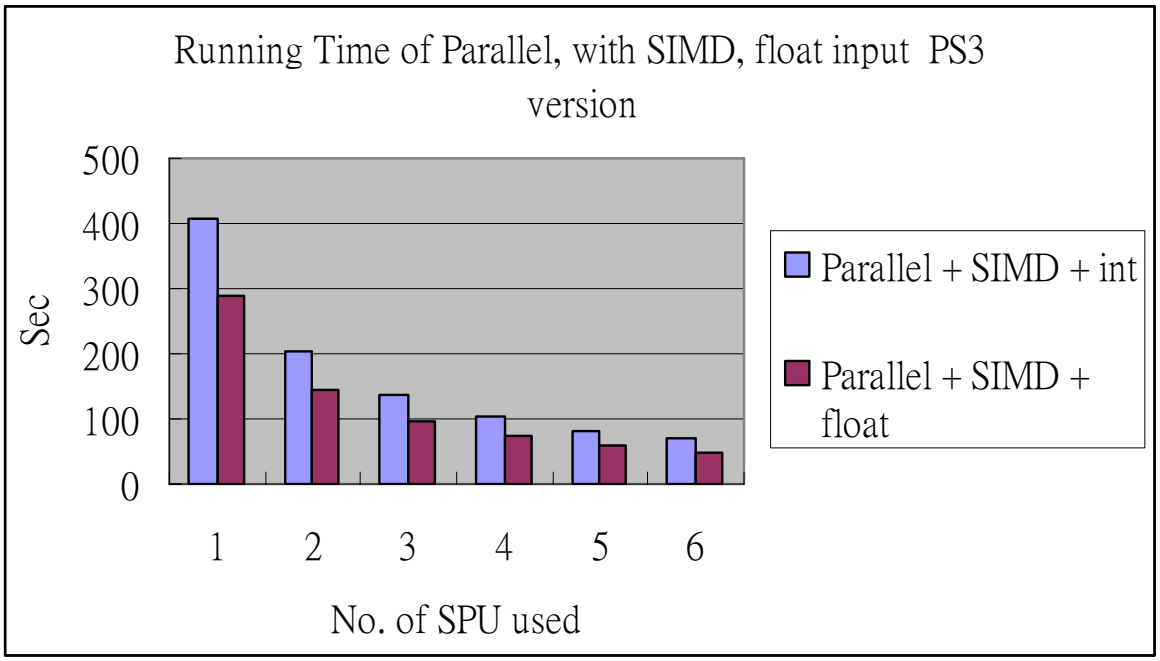
We had great success in speeding up the program by using SIMD instructions. We want to push the SIMD instructions to limit. After enquiring our coordinator, we all agreed that double precision is not needed in this problem is not that important. Firstly, the number used is relatively small. Secondly, the number is acceptable even with errors as long as it does not overflow. Therefore our coordinator suggests us to change the data type from double (64 bits) to integer (32 bits). By doing so, we can now pack 4 integer values in the 128 bits register.

However, the original design of SPE is to handle floating point number. Hence, there are nearly no SIMD instructions for integer data type. We finally decided to use float, which is also 32 bits, as our data type. Now by applying SIMD instructions again, we can save one step in converting the integer to float, obtaining another 30% speed up than using double as data type.

Therefore the choice of data type is very important when writing parallel program in PlayStation®3.

## Result of the parallel, with SIMD, float input PS3 version

| No. of SPU used | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Read input time (sec) | 4 | 4 | 3 | 3 | 4 | 5 |
| Total Elapsed time (sec) | 294 | 149 | 100 | 77 | 62 | 54 |
| **Net Elapsed time (sec)** | **290** | **145** | **97** | **74** | **58** | **49** |




As we can see, after using floating point number input, we save the step to convert every integer to float before applied SIMD. This float type version runs **30% faster** than the integer version.

This makes the best performance becomes **49 sec**.

### 4.3.3 Implementing Double Buffering (Minor Improvement)

Originally, there is only one buffer for storing “Repository” file.

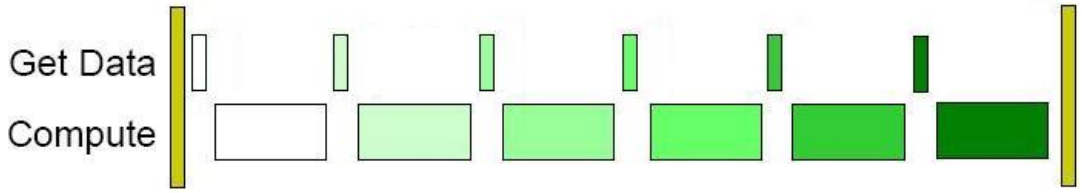
Therefore the flow of program is like:

A. Fetch a “Repository” file’s data via DMA.

B. Compute the square of Euclidean distance.

C. Update the value if it is smaller.

Loop back to A until all “Repository” file is processed.




Since there is a Memory Flow Controller (MFC) in each SPE for DMA, therefore DMA commands can be execute simultaneously while SPE is doing computation. Double buffering is a technique that makes use of this property.

By allocating one more buffer for “Repository” file in SPE’s local store, it allows “Repository” file to be fetched first and get ready in the buffer. Shorten the runtime used by DMA.

The new flow of the program becomes:

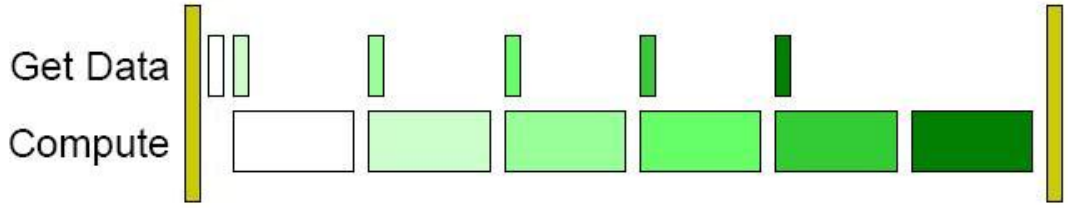
A. Fetch the first “Repository” file’s data to buffer 1 via DMA.

B. Do the following simultaneously

1. Compute the square of Euclidean distance using “Repository” file from buffer 1.
2. Fetch the next “Repository” file in buffer 2 via DMA.

C. Update the value if it is smaller

D. Loop back to B with usage of buffer 1 swapped with buffer 2.




As can be observe from the figure, if communication is smaller than computation, there won’t be much speed up. This is the case in our program unfortunately.

### 4.3.4 Parallel Reading for all Files (Failed Attempt)

At first, we thought that reading files in both “Target” and “Repository” directory can be done simultaneously. This is because files can be read in any order, as long as they are read by the program. Hence, an attempt is to try to let SPE share the file reading job to reduce the run time.

After implementing the program, we found that the program is actually slowed down. The explanation to this phenomenon is that we made a wrong assumption at start. In algorithm level, the files can be read in parallel way is true. However, in the hardware level, the hard disc cannot handle the parallel reading request by 6 SPE. That is why this is a failed attempt in speeding up the program.

### 4.3.5 Distributing job to idling PPE (Minor Improvement)

Another thought to speed up the program is base on the fact that PPE stalls for long time waiting SPEs to finish their jobs throughout the program's runtime. Therefore we tried to assign a file to the PPE for processing. When the PPE is stalling and waiting for busy SPEs to finish their job, instead of stalling, it is assigned a "Target" file for processing.

However, PPE is not specified for heavy computation. The time PPE takes to compute a minimum score and a minimum score match is about 5-6 times to the time SPE takes. The speed up is thus negligible for the following reasons.

For example, there is total of 1000 "Target" files for processing. Each SPE will handle 1000/6 = about 167 files. Assume SPE works 6 times faster than PPE. PPE can only handle 1000/(6*6+1) = about 27 files. That's mean for each SPE, PPE helped out with 27/6 = 4.5 files. So the newest total runtime will be the time for one SPE to process 167 - 4.5 = 162.5 files, which is negligible.

On the other hands, this method increases the complexity of the program code by a lot. So we finally decided not to apply this method.

### 4.3.6 Applying SIMD for Loop Counter (Major Improvement)

Observing that for each SPE, most of its time is spent on running the through a for-loop which loop though a file and computing the square of Euclidean distance.

Here is the flow of the for-loop:

1. initialize i = 0, diff = (0, 0, 0, 0).
2. for i < Number of float numbers in a file / Number of floats packed in a register
   A. temp = SIMD subtraction on vector i in “Target” and “Repository” file.
   B. diff = SIMD addition (SIMD multiplication (temp, temp) , diff).
3. i = i + 1.
4. Loop back to 2.

As can be observed above, A and B make use of SIMD command, while 2 and 3 does not. Since this for-loop is small, line 2 and 3 actually occupied quite a large proportion runtime in the for-loop even it is just Boolean comparison and integer addition operation. Hence, speeding up 2 and 3 should yield some speed up to the final runtime. We try to apply SIMD command to the loop counter i. Loop counter i now has a data type of short (16 bits).

Therefore we can pack 8 short values in the register in the following way:

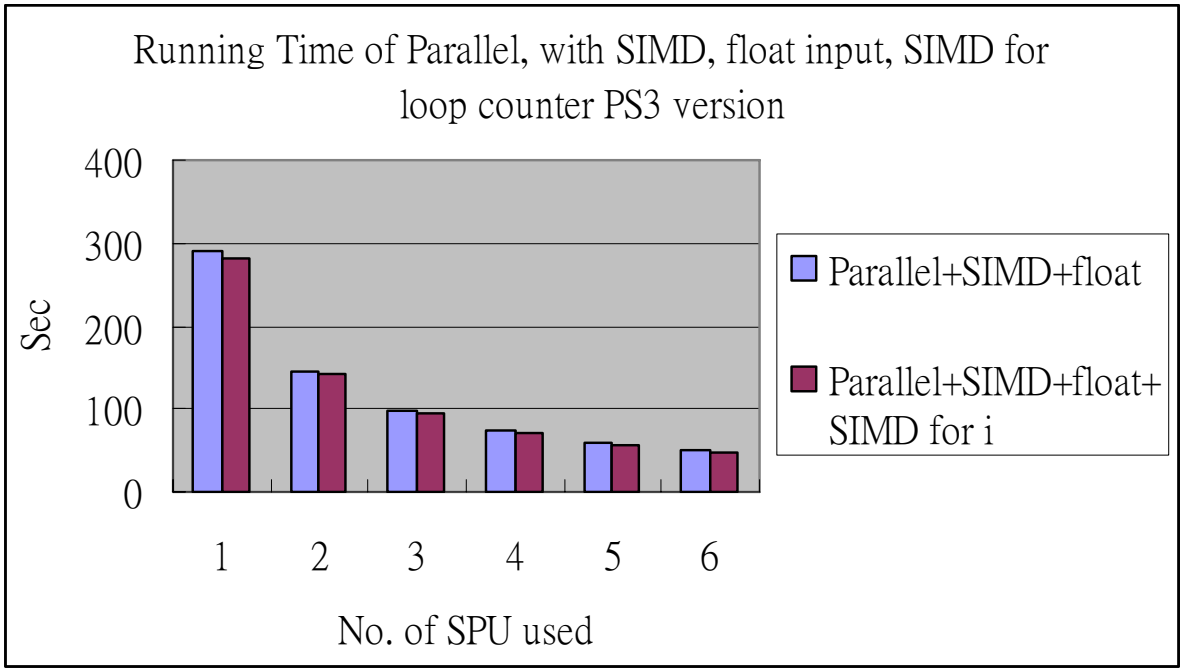
1. initialize i = (0,1,2,3,4,5,6,7) , diff = (0, 0, 0, 0).
2. for i[0] < Number of float numbers in a file / Number of floats packed in a register
   A. *temp = SIMD subtraction on vector i[0] in "Target" and "Repository" file.*
   B. *diff = SIMD addition (SIMD multiplication (temp, temp) , diff).*
   C. *temp = SIMD subtraction on vector i[1] in "Target" and "Repository" file.*
   D. *diff = SIMD addition (SIMD multiplication (temp, temp) , diff).*
   E. *temp = SIMD subtraction on vector i[2] in "Target" and "Repository" file.*
   F. *diff = SIMD addition (SIMD multiplication (temp, temp) , diff).*
   G. *temp = SIMD subtraction on vector i[3] in "Target" and "Repository" file.*
   H. *diff = SIMD addition (SIMD multiplication (temp, temp) , diff).*
   I. *temp = SIMD subtraction on vector i[4] in "Target" and "Repository" file.*
   J. *diff = SIMD addition (SIMD multiplication (temp, temp) , diff).*
   K. *temp = SIMD subtraction on vector i[5] in "Target" and "Repository" file.*
   L. *diff = SIMD addition (SIMD multiplication (temp, temp) , diff).*
   M. *temp = SIMD subtraction on vector i[6] in "Target" and "Repository" file.*
   N. *diff = SIMD addition (SIMD multiplication (temp, temp) , diff).*
   O. *temp = SIMD subtraction on vector i[7] in "Target" and "Repository" file.*
   P. *diff = SIMD addition (SIMD multiplication (temp, temp) , diff).*
3. i = SIMD addition (i, (8, 8, 8, 8, 8, 8, 8, 8)).
4. Loop back to 2.

The above algorithm replaces loop counter i by a short vector i. Thanks to SIMD instructions. The addition and comparison operations are reduced by 8 times.

Notice that this method does not make use of any parallel programming technique. Instead, it is just a method for optimization. Further optimization is described below.

## Result of the parallel, with SIMD, float input, SIMD for loop counter PS3 version

| No. of SPU used | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Read input time (sec) | 4 | 5 | 3 | 4 | 4 | 4 |
| Total Elapsed time (sec) | 286 | 146 | 97 | 75 | 60 | 51 |
| **Net Elapsed time (sec)** | **282** | **141** | **94** | **71** | **56** | **47** |




With this new approach, we reduce the computation of the loop counter to gain a little improvement (about **4%**). However, it shows the possibility to have faster performance by further loop unrolling.

The best performance becomes **47 sec**.

### 4.3.7 Optimizing by Loop Unrolling (Major Improvement)

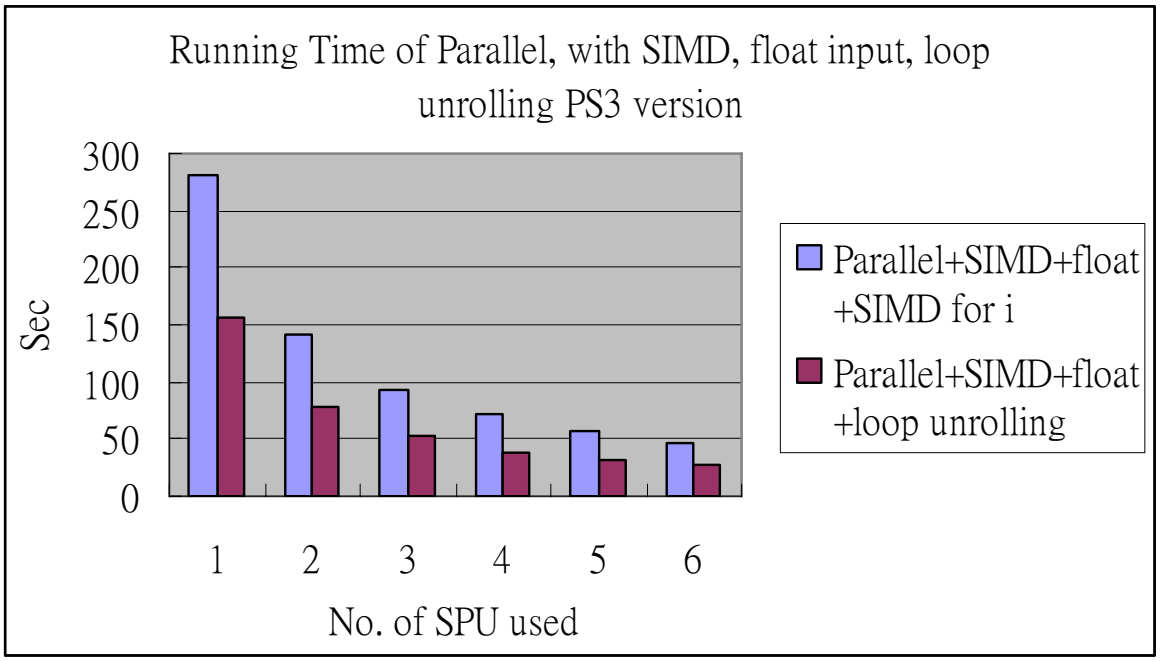
As using SIMD operation to compute loop counter can reduce the runtime by many. We then tried to break down the for-loop completely. For-loop is unrolled by repeating the statements inside for 256 times. It is done by writing another program to generate the code.

```
int main(){
    int i;
    for (i=0;i<HL3_SIZE/INT_VEC_ELE;i++)
    {
        printf("\ttemp = spu_sub(hl3_a[%d].vec, hl3_b[%d].vec);\n", i, i);
        printf("\tdiff.vec = spu_madd(temp, temp, diff.vec);\n");
    }
    return 0;
}
```

Then the code generated by the above program are copied and pasted into the source code. Now we have completely get rid of the loop counter i. The speed up becomes more obvious.

## Result of the parallel, with SIMD, float input, loop unrolling PS3 version

| No. of SPU used | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Read input time (sec) | 3 | 4 | 3 | 3 | 4 | 3 |
| Total Elapsed time (sec) | 159 | 82 | 55 | 42 | 35 | 30 |
| **Net Elapsed time (sec)** | **156** | **78** | **52** | **39** | **31** | **27** |




With total loop unrolling by hard coding some codes, we remove the computation of the loop counter, which contribute a certain portion to the computation time since our for loop is relatively small.

This give us about **45%** faster in the running time of the program

The ultimate best performance becomes **27 sec**.

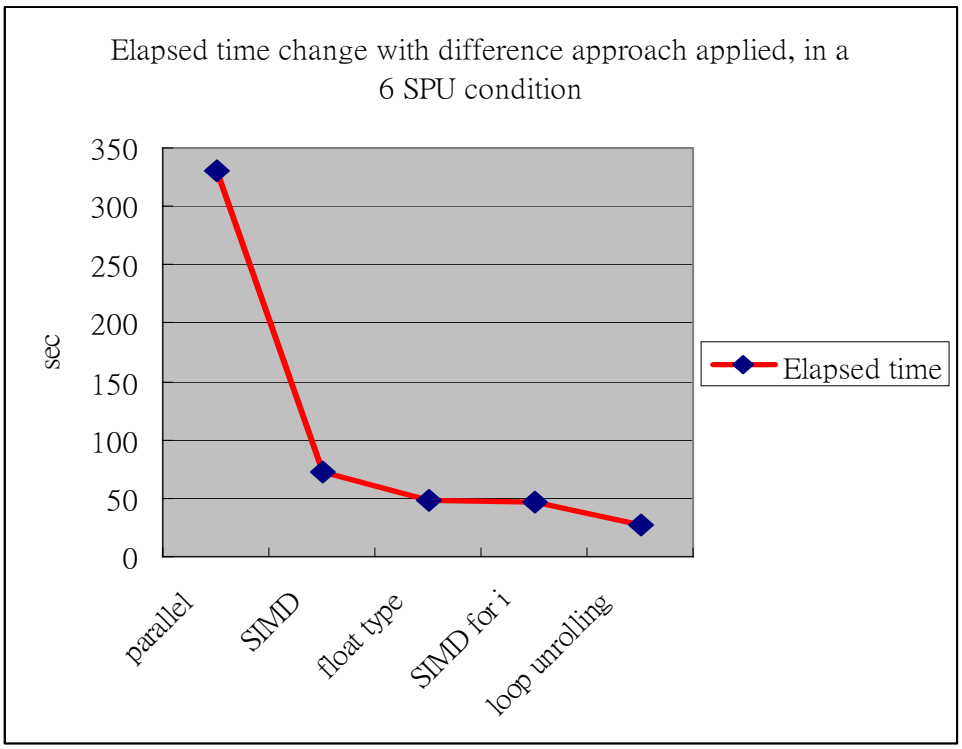
## 4.4 Conclusion of Optimization

With the original sequential approach, it takes 633 sec to execute on the PC, and 1928 sec to execute on 1 SPU only (without counting the read input time).

Our final result is 27 sec, which is:

23 times faster than the sequential algorithm on PC

71 times faster than the sequential algorithm on PlayStation®3, using 1 SPU




We can see the parallelizing, changing type to floating point number and loop unrolling are the major reasons for performance improvement.

After applying the above approaches, we believe we have made a great improvement in the performance of the ADVISER program.

# Chapter 5 Project Difficulties

## Incompatibilities of PlayStation®3 with Linux

Sony PlayStation®3 is a relatively new platform. Not much OS actually support PlayStation®3. We found that most people use Yellow Dog Linux and Fedora 7 as their OS on PlayStation®3.

First, we tried Yellow Dog Linux as suggested by the MIT course webpage. Unfortunately, Cell SDK 2.0 which is suggested by the course webpage is no longer available at that time. So, we tried to download the newest Cell SDK 2.1. However, it required Fedora Core 6 for it to function normally. To make things worse, Fedora Core 6 is no longer available too. We can only download the most up-to-date Fedora 7 OS.

Luckily, Cell SDK 2.1 works fine with Fedora 7. This is the environment we used in this project in this semester.

One problem exists though, that is PlayStation®3 frozen every time we try to reboot. We solve this problem by downloading a custom kernel for PlayStation®3 and recompiling it.

# Limited Resources on the Internet

Our researching resources mainly come from 3 ways. Books, MIT course website and IBM resources centre. Since parallel programming on PlayStation®3 is relatively new. Tools, OS support are still under development and not yet stable. Therefore not many resources can be relied on when we try to study parallel programming.

For example, we tried to debug our program by using profiling tools as suggested by our research staff. However, this is not available until the full release of Cell SDK 3.0 in late October. We dare to try it. This is because removing and reinstalling Cell SDK can take very long time. Also, we are not sure if anything will crash or not functioning with the new Cell SDK 3.0. As a result, we chose to stay with Cell SDK 2.1 eventually.

## Rapid Update of OS and Cell SDK

As mentioned above, both OS support and Cell SDK are still under development. Update and new packages are released time by time.

Takes the example of Cell SDK, version 3.0 includes many new libraries like the Accelerated Library Framework, SPU Timer library, Basic Linear Algebra Subprogram, etc. Fedora 8 is also released recently. They are so new that we cannot make full use of these libraries yet.

# Burning down of PlayStation®3!!

During the semester, we try to take a look at the OpenCV library for Cell BE architecture as suggested by our research staff. Some sample program in OpenCV library requires the use of webcam.

First we tried to connect Logitech QuickCam® Pro 3000. However, there is insufficient support of webcam driver. Even the camera can be detected by the PlayStation®3 system, no images can be viewed.

Fig. 5.1 Logitech QuickCam® Pro 3000

Next we tried to connect Logitech QuickCam® Sphere ™ MP and see if it works. Out of our expectation, the web cam actually burnt all the 4 USB ports in the PlayStation®3 system. With all the USB ports down, we can use keyboard, mouse. We cannot even format the whole system. So, finally we have to take the PlayStation®3 to the Sony Support Centre for repair. Luckily, the warrant hasn't expired. We are able to get a brand new console.

Fig. 5.2 Logitech QuickCam® Sphere ™ MP

# Chapter 6 Project Progress

The following is the progress of our Final Year Project:

| | |
|---|---|
| Summer Holiday 2007 | ■ Try to install Linux on PlayStation®3<br>■ Some background study about architecture of Cell and parallel programming |
| September 2007 | ■ Set up the developing environment: install Fedora 7 and Cell SDK<br>■ Study the programming with Cell SDK<br>■ Write simple programs on PlayStation®3 using Cell SDK, implement parallel features |
| October 2007 | ■ Try to install OpenCV and connect web cam to PlayStation®3, which blew the PlayStation®3<br>■ Decide to optimize the ADVISER program<br>■ Start to parallelize ADVISER on PlayStation®3 |
| November 2007 | ■ Make further improvement on ADVISER by trying more features, some succeed some failed<br>■ Try to use IBM Visual Performance Analyzer, but it is not compatible with Cell SDK 2.1<br>■ Collective data to analysis the performance of the program<br>■ Have our PlayStation®3 repaired<br>■ Write report |

# Chapter 7 Future Works

We have some ideas about the next step of our Final Year Project in the coming semester.

First, we will update the Cell SDK from 2.1 to 3.0, so that we can have newer tools to help our project, especially those performance analysis tools which are not compatible with Cell SDK 2.1

Second, for the ADVISER program, we will try to port the whole application on PlayStation®3 and have optimization throughout the whole program, not just the comparison part.

Last but not least, we will continue to learn new parallel programming features, so we can try more different ways to improve performance.

# Chapter 8 Acknowledgement

We would like to thank our project supervisor, Professor Michael R. Lyu. He gives us useful advices and provides the resources we need for our project. In addition to this, he also reminds us the importance of concrete statistics and good scheduling.

Besides, we would like to thank Mr. Edward Yau and Mr. Un Tze Lung, who are the research staff in VIEW Lab. They give valuable advices, both conceptual and technical, to our project. The original ADVISER program they provided has started an important part of our project too.

# Chapter 9 Reference


1. MIT Multicore Programming Primer: PS3 Cell Programming
   http://cag.csail.mit.edu/ps3/index.shtml
2. Patterns for Parallel Programming. Mattson, Sanders, and Massingill (2005)
3. IBM Cell Broadband Engine resource center
   http://www-128.ibm.com/developerworks/power/cell/
4. Introduction to Parallel Computing, Grama, Gupta, Karypis, Kumar, Addison-Wesley(2003)
5. Discovering Multi-Core: Extending the Benefits of Moore's Law, Technology@Intel Magazine, July 2005
6. PlayStation®3 Technical Specification
   http://www.us.playstation.com/ps3/about/specs
7. Intel Halts Development of 2 New Microprocessors, Laurie J.Flynn
   http://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007
8. Intel decides two cores are better than one, Tom Krazit, IDG News Service
   http://www.infoworld.com/article/04/05/07/HNintelcores_1.html
9. The Processor-Memory bottleneck: Problems and Solutions, Nihar R. Mahapatra, Balakrishna Venkatrao
   http://www.acm.org/crossroads/xrds5-3/pmgap.html
10. Parallel Computing, Wikipedia
    http://en.wikipedia.org/wiki/Parallel_computing

11. Programming Models for Scalable Multicore Programming, Michael D. McCool
http://pccluster.nchc.org.tw/xoops/modules/newbb/viewtopic.php?forum=2&topic_id=1016
12. Folding@Home on the PS3
http://www.stanford.edu/group/pandegroup/folding/FAQ-PS3.html
13. Basics of Cell Architecture
http://ps3.keshi.org/dsk/20061208/doc/CellProgrammingTutorial/BasicsOfCellArchitecture.html
14. Barcelona Supercomputer Center (BSC)
http://www.bsc.es/projects/deepcomputing/linuxoncell/
15. Cell(microprocessor), Wikipedia
http://en.wikipedia.org/wiki/Cell_%28microprocessor%29