This article has been sub divided in to two parts. Part 1 gives introduction on basics of threads, different methods in which threads are used and Introduction to IBM CELL BE. Part 2 gives introduction to different programming techniques on IBM CELL BE and also shows difference in performance between different programming techniques on IBM CEL BE.

**Background:**

This section gives a short background to the reader on most common techniques used for parallel processing.

Threads, plays a critical role when we talk about developing applications/programs. Irrespective of the processor architecture and the operating system used every application program must be built using threads. Application may be Single threaded or Multi threaded, it is based on the requirements framed for the application program. Threads in an OS are run using a simple scheduling technique, Time Division Multiplexing (TDM). Modern operating systems support TDM and Multiprocessor threading.

![Figure 1: Single Threaded Sequential Program](image1)

Programming do not have control over how the operating system time slices the threads or how the multi processor scheduler works. Only control the programmer can get over this process is by passing parameters like which scheduling algorithm to use, at priority the process should run etc to the scheduler via system calls. The technique used to create the threads is similar, on most of the operating systems. Even the mechanisms used to control or manipulate the properties of these threads are similar.

Most commonly used libraries for threading are Pthread, MPI or Win32 threading library. These libraries help programmers create threads but do not give full control over their scheduling.
Introduction:

This section introduces readers to IBM Cell Broadband Engine (henceforth referred to as Cell B.E.).

The Cell B.E. processor is a multi-core processor based on IBM’s Power Architecture. The Cell BE processor is a chip consisting of nine processing elements (CPU Cores). A single chip contains a Power Processing Element (PPE) and eight Synergistic Processing Elements (SPE).

The main processing element is a fairly standard general-purpose processor. It is called the Power Processing Element, or PPE for short. PPE consists of 32KB instruction and 32KB data L1 cache.

The other 8 processing elements within the Cell BE are known as Synergistic Processing Elements, or SPEs. Each SPE consists of:

- Eight Vector Processors called Synergistic Processing Elements (SPE)
- A private memory area within the SPU called the local store (size of this area in PS3 is 256K)
- IO’s for external communication
- A set of 128 registers, each 128 bits wide (each register is normally treated as holding four 32-bit values simultaneously)
- A Memory Flow Controller (MFC) which manages DMA transfers between the SPU’s local store and main memory

SPE’s do not provide features that the PPE does or any general purpose processor does, making it incapable of running normal operating system tasks.

Therefore, the PPE acts as the resource manager and the SPEs act as the data crunchers. Programs on the PPE distribute tasks to the SPEs, and then they feed data back and forth to each other.

Connecting together the SPEs, the PPE, and the main memory controller is a bus called the Element Interconnect Bus. This is the main passageway through which data travels.

Cell B.E Block Diagram:

Figure 3: Cell B.E Block Diagram
Cell B.E in Sony PS3:

PS3 is one of the product which is built using IBM CELL BE that is low cost and easily available in the consumer market. PS3 is one of the resources I have used to perform all my experiments to make necessary measurements.

PS3 Hardware configuration:

1. 3.2 GHz, 64-bit nine-core processor ~200 GFLOPS
2. 64-bit Power Arch, Scalar Processor, PPE - Power Processing Element
3. 8 128-bit SIMD Vector Processor, SPEs - Synergistic Processing Elements
4. Elements connected via 200+ GB/s EIB - Element Interconnect Bus

Running Linux on PS3:

Terra Soft Solutions has developed Yellow Dog Linux 5.0.2/6.0 in cooperation with Sony specifically for the PS3. Apart from the Sony’s proprietary OS running on the PS3, it also provides flexibility to install different flavors of Linux OS enabling programmers to exploit the raw power of all hardware sub systems.

Typical Programming Techniques:

All the programming techniques we discuss in this document will be discussed with an example program written using different programming methodologies.

Example used to test the performance of different programming techniques on the Cell B.E performs Mathematical operations like Multiplication and Division on Integer and Floating point data of total size 1 Giga Bytes.

Before we proceed further I would like to answer a question which I feel almost every one reading this article may want to ask.

Why did I choose such a simple program for evaluating the performance?

Most of the applications/algorithms are built over basic mathematical operations which lays the foundation for them. If the foundation performs better only then the application built over it can be designed to perform better. So, Instead of using a complex algorithm I decided to use simple number crunching program which I believe will give us enough performance measures for evaluation.

Let’s talk about different programming techniques.

- Single Threaded Sequential Programming
- Multi Threaded Programming

Single Threaded Sequential Programming

This is a technique used to write programs/instructions in a sequential fashion and is the simplest programming technique. Figure shown below is a typical sequence used as in the example program.
Multi Threaded Programming

This is a technique used to write programs where program is broken into mutually exclusive different tasks which are then distributed among multiple threads for parallel (pseudo) execution. Threading is achieved by instantiating each thread using specific libraries based on the platform on which it is going to run on.

![Multi Threaded Programming](image)

**Figure 5: Multi Threaded Programming**

Cell B.E Programming Techniques:

- Parallel Processing on SPE
- Parallel SIMD Processing on SPE

Before we dive into the details of the above mentioned techniques, let’s understand how to distribute the processes to the SPE’s on the Cell B.E.

Programming SPE

The mechanism to distribute the process to the SPE is similar to the technique used to create threads, particularly Pthread library. Given below are the function and its parameter used to create a process and distribute to an SPE. Along with distributing the process, address of the application specific data in main memory is passed to the process running on the SPE. Process on SPE uses DMA to transfer application specific data from the PPE main memory to the SPE local store for processing.

Function:

```
spe_create_thread
```

Parameters:

- `spe_gid_t gid`
  
  Identifier of the SPE group that the new thread will belong to. SPE group identifiers are returned by `spe_create_group`. The new SPE thread inherits scheduling attributes from the designated SPE group. If `gid` is equal to `SPE_DEF_GRP (0)`, then a new group is created with default scheduling attributes, as set by calling `spe_group_defaults`.

- `spe_program_handle_t *spe_program_handle`
  
  Indicates the program to be executed on the SPE. This is a pointer to an SPE ELF image which has already been loaded and mapped into system memory. This pointer...
is normally provided as a symbol reference to an SPE ELF executable image which has been embedded into a PPE ELF object and linked with the calling PPE program.

- **void *argp**
  Pointer to application specific data. This is passed as the second parameter to the SPE program.

- **void *envp**
  Pointer to environment specific data. This is passed as the third parameter to the SPE program.

- **unsigned long mask**
  Mask is used to select the SPE to execute the process on. If -1 is used, program is loaded onto the available SPE.

- **int flags**
  NULL.

### Parallel Processing on SPE

The API explained above is used to create four processes for the mathematical operations on the SPE. By doing this all four operations can be run in parallel.

To understand how the distribution of process and data transfer was done on the CELL BE, let’s go through the paragraph below.

![Figure 6: Parallel Processing on SPE](image)

Main program responsible for process distribution is resident on the PPE. Distribution of the process to the SPE is done by using `spe_create_thread` API in the main thread.

PPE then loads the program on a selected SPE or an available SPE based on the parameter passed and passes the address of the main memory at which the source processing data resides. Process on SPE then initiates a DMA transfer from the main memory to Local memory and uses it for processing. On completion of data processing, it uses DMA and transfers the processed data back to the PPE.
Parallel SIMD Processing on SPE

In this technique mathematical operations are distributed to the SPE as explained above. Difference between this technique and previously explained technique is, apart from running the process in parallel, data is processed using the vector co-processors.

Let’s look into how the vector co-processor can be used.

What are Vector Co-Processors?

A vector processor, or array processor, is a CPU design that is able to run mathematical operations on multiple data elements simultaneously. This is in contrast to a scalar processor which handles one element at a time.

For Example:

If we have two arrays of elements of type short with 5 elements each, all elements from both the arrays can be multiplied to each other in one go. Let’s say multiplication of two elements takes 1 clock cycle. Using scalar processor, multiplication of 5 elements will take 5 clock cycles but using vector processor, it gets done in 1 clock cycle.

Let’s have a look at the picture below to understand this:

**Multiplication of Elements on Scalar Processor:**

|------------|---|------------|---|------------|

On a scalar processor, all five elements from both arrays are multiplied one by one.

**Multiplication of Elements on Vector Processor:**

|------------|---|------------|---|------------|

On a vector processor, all five elements from both arrays are multiplied at once.

**Vector Data Types on Cell B.E**

Cell B.E supports vector data types of size 16 bytes (128 bits). So, one vector of type char can hold 16 char elements in it. Let’s have a look at all the vector types supported by Cell B.E.

<table>
<thead>
<tr>
<th>Vector Data Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>qword</td>
<td>128-bit quad word vector of unspecified type</td>
</tr>
<tr>
<td>vector unsigned char</td>
<td>16 8-bit unsigned integer characters (bytes)</td>
</tr>
<tr>
<td>vector signed char</td>
<td>16 8-bit signed integer characters (bytes)</td>
</tr>
<tr>
<td>vector unsigned short</td>
<td>8 16-bit unsigned integer half words</td>
</tr>
<tr>
<td>vector signed short</td>
<td>8 16-bit signed integer half words</td>
</tr>
<tr>
<td>vector unsigned int</td>
<td>4 32-bit unsigned integer</td>
</tr>
<tr>
<td>vector signed int</td>
<td>4 32-bit signed integer words</td>
</tr>
<tr>
<td>vector unsigned long long</td>
<td>2 64-bit unsigned integer double words</td>
</tr>
</tbody>
</table>
Performance on Cell B.E

So let’s have a look at some measurements taken using the example programs.

<table>
<thead>
<tr>
<th>Vector Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector signed long long</td>
<td>2 64-bit signed integer double words</td>
</tr>
<tr>
<td>Vector float</td>
<td>4 32-bit single precision floats</td>
</tr>
<tr>
<td>Vector double</td>
<td>2 64-bit double precision floats</td>
</tr>
</tbody>
</table>

Figure 7: PPE – Wall Clock Time – Processing Time

Chart above shows the total wall clock time taken by the programs written using the four techniques explained earlier. Values are in milliseconds.

Chart below shows the total Processor time taken by the programs written using the four techniques explained earlier. Values are in milliseconds.

When the processes were distributed to the SPE, total time spent by the PPE on the process execution drastically dropped down to 10 milli seconds. Also, only 4 out of 8 SPE were used.
So, this opens a door for running few more processes in parallel from PPE on the SPE.

**Conclusion**

From the above calculations following conclusions can be derived at:

- With advancement in the processor architecture programmer must adopt programming techniques best suited to the processor architecture in order to get optimum performance

- Best programming technique (out of the ones discussed above) to use on the Cell B.E for processing large chunks of data is “Parallel Processing with SIMD”

**Bibliography**