SCOP3

A Rough Guide to Scientific Computing On the PlayStation 3

Technical Report UT-CS-07-595

Version 0.1

by Alfredo Buttari

Piotr Luszczek

Jakub Kurzak

Jack Dongarra

George Bosilca

Innovative Computing Laboratory
University of Tennessee Knoxville

April 17, 2007

Contents

1 **Introduction** 1

2 **Hardware** 3

2.1 CELL Processor 3

2.1.1 POWER Processing Element (PPE) 3

2.1.2 Synergistic Processing Element (SPE) 5

2.1.3 Element Interconnection Bus (EIB) 6

2.1.4 Memory System 7

2.2 PlayStation 3 7

2.2.1 Network Card 7

2.2.2 Graphics Card 7

2.3 GigaBit Ethernet Switch 8

2.4 Power Consumption 8

3 **Software** 9

3.1 Virtualization Layer: Game OS 9

3.2 Linux Kernel 9

3.3 Compilers 10

3.4 TCP/IP Stack 11
3.5 MPI 11

4 Cluster Setup 13

4.1 Basic Linux Installation 13
4.2 Linux Kernel Recompilation 15
4.3 IBM CELL SDK Installation 19
4.4 Network Configuration 21
4.5 MPI Installation 23
4.5.1 MPICH1 23
4.5.2 MPICH2 24
4.5.3 Open MPI 25

5 Development Environment 27

5.1 CELL Processor 27
5.2 PlayStation 3 Cluster 29

6 Programming Techniques 32

6.1 CELL Processor 32
6.1.1 Short Vector SIMD'ization 32
6.1.2 Intra-Chip Communication 35
6.1.3 Basic Steps of CELL Code Development 38
6.1.4 Quick Tips 40
6.2 PlayStation 3 Cluster 43

7 Programming Models 46

7.1 Octopiler 47
7.2 RapidMind 48
7.3 PeakStream 49
7.4 MPI Microtask 49
7.5 Cell Superscalar 50
7.6 The Sequoia Language 51
7.7 Mercury Multi-Core Framework 51

8 Application Examples 53
8.1 CELL Processor 53
8.1.1 Dense Linear Algebra 53
8.1.2 Sparse Linear Algebra 54
8.1.3 Fast Fourier Transform 57
8.2 PlayStation 3 Cluster 58
8.2.1 The SUMMA Algorithm 58
8.3 Distributed Computing 61
8.3.1 Folding@Home 61
9 Summary 62
9.1 Limitations of the PS 3 for Scientific Computing 62
9.2 CELL Processor Resources 64
9.3 Future 64
A Acronyms 68

Acknowledgements

We would like to thank Gary Rancourt and Kirk Jordan at IBM for taking care of our hardware needs and arranging for financial support. We are thankful to numerous IBM researchers for generously sharing with us their CELL expertise, in particular Sidney Manning, Daniel Brokenshire, Mike Kistler, Gordon Fossum, Thomas Chen and Michael Perrone.

Our thanks also go to Robert Cooper and John Brickman at Mercury Computer Systems for providing access to their hardware and software. We are also thankful to the Mercury research crew for sharing their CELL experience, in particular John Greene, Michael Pepe and Luke Cico.

Thanks to Adelajda Zareba for the photography artwork for this guide.

Plus everybody who makes important comments/corrections/contributions.

CHAPTER 1

Introduction

As much as the *Sony PlayStation 3* (*PS3*) has a range of interesting features, its heart, the *CELL* processor is what the fuss is all about. CELL, a shorthand for *CELL Broadband Engine Architecture*, also abbreviated as *CELL BE Architecture* or *CBEA*, is a microprocessor jointly developed by the alliance of Sony, Toshiba and IBM, known as *STI*.

The work started in 2000 at the STI Design Center in Austin, Texas, and for more than four years involved around 400 engineers and consumed close to half a billion dollars. The initial goal was to outperform desktop systems, available at the time of completion of the design, by an order of magnitude, through a dramatic increase in performance per chip area and per unit of power consumption. A quantum leap in performance would be achieved by abandoning the obsolete architectural model where performance relied on mechanisms like cache hierarchies and speculative execution, which those days brought diminishing returns in performance gains. Instead, the new architecture would rely on a heterogeneous multi-core design, with highly efficient data processors being at the heart. Their architecture would be stripped of costly and inefficient features like address translation, instruction reordering, register renaming and branch prediction. Instead they would be given powerful short vector SIMD capabilities and a massive register file. Cache hierarchies would be replaced by small and fast local memories and powerful DMA engines. This design approach resulted in a 200 million transistors chip, which today delivers performance barely approachable by its billion transistor counterparts and is available to the broad computing community in a truly of-the-shelf manner via a $600 gaming console.

As exciting as it may sound, using the PS3 for scientific computing is a bumpy ride. Parallel programming models for multi-core processors are in their infancy, and standardized APIs are not even on the horizon. As a result, presently, only hand-written code fully exploits the hardware capabilities of the CELL processor. Ultimately, the suitability of the PS3 platform for scientific computing is most heavily impaired by the devastating disproportion between the processing power of the processor and the crippling slowness of the interconnect, explained in detail in section 9.1. Nevertheless, the CELL processor is a revolutionary chip, delivering ground-breaking performance and now available in an affordable package. We hope that this *rough guide* will make the ride slightly less bumpy.

CHAPTER 2

Hardware

2.1 CELL Processor

Figure 2.1 shows the overall structure of the CELL processor. In the following sections we briefly summarize the main features of its most important element: the *Power Processing Element (PPE)*, the *Synergistic Processing Elements (SPEs)*, the *Element Interconnection Bus (EIB)* and the memory system.

2.1.1 POWER Processing Element (PPE)

The *Power Processing Element (PPE)* is a representative of the *POWER Architecture*, which includes the heavy-iron high-end POWER line of processors and as well as the family of PowerPC desktop processors. The PPE consists of the *Power Processing Unit (PPU)* and a unified (instruction and data) 512 KB 8-way set associative write-back cache. The PPU includes a 32 KB 2-way set associative reload-on-error instruction cache and a 32 KB 4-way set associative write-through data cache. L1 caches are parity protected, the L2 cache is protected with error-correction code (ECC). Cache line size is 128 bytes for all caches. Beside the standard floating point unit (FPU) The PPU also includes a short vector SIMD engine, VMX, an incarnation of the PowerPC Velocity Engine or AltiVec. The PPE's register fine is comprised of 32 64-bit general purpose registers, 32 64-bit floating-point registers and 32 128-bit vector registers.

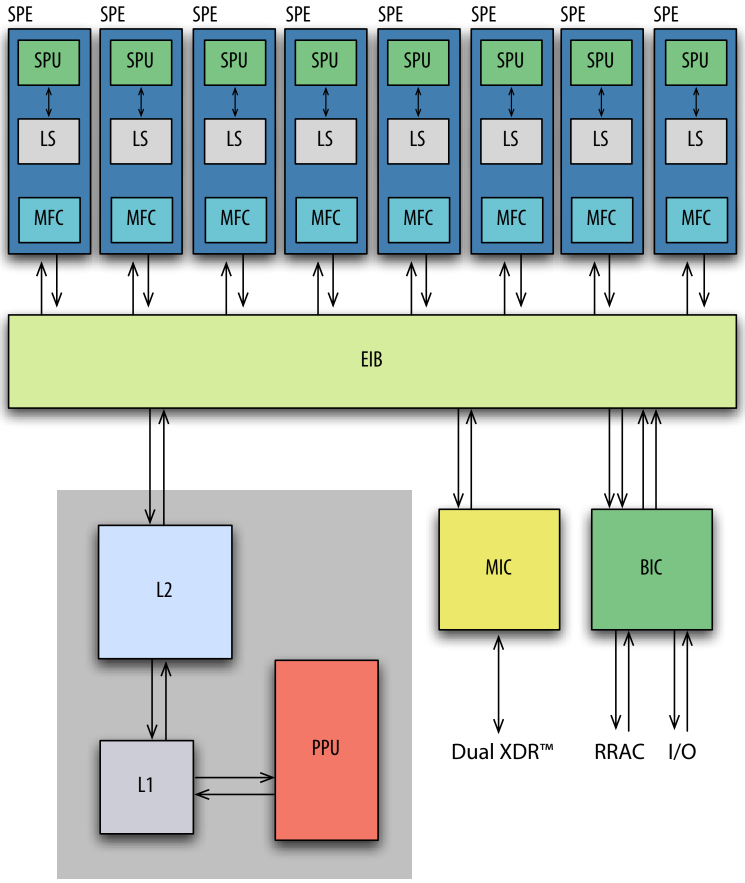

Figure 2.1: *CELL Broadband Engine architecture.*

The PPE is a 64-bit, 2-way *simultaneous multithreading (SMT)* processor binary compliant with the PowerPC 970 architecture. Although it uses the PowerPC 970 instruction set, its design is substantially different. It has a relatively simple architecture with in-order execution, which results in considerably smaller amount of circuitry than its out-of-order execution counterparts and lower energy consumption. This can potentially translate to lower performance, especially for applications heavy in branches. However, the high clock rate, high memory bandwidth and dual threading capabilities may make up for the potential performance deficiencies.

Especially important is the SMP feature, which to an extent corresponds to Intel's Hyper-Threading technology. The PPE seems to provide two independent execution units to the software layer. In practice the execution resources are shared, but each thread has its own copy of the architectural state, such as general-purpose registers. The technology comes at a 5% increase in the cost of the hardware and can potentially deliver from 10% to 30% increase in performance [1].

Clocked at 3.2 GHz, the PPE can theoretically deliver $2 \times 3.2 = 6.4$ Gflop/s of IEEE compliant double precision floating-point performance from its standard fully pipelined floating point unit using *fused multiply-add (FMA)* operation. It can also deliver $4 \times 2 \times 3.2 = 25.6$ Gflop/s of non-IEEE compliant single precision floating-point performance from its VMX unit using 4-way SIMD fused multiply-add operation.

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. 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 (see next section), rather than written from scratch.

2.1.2 Synergistic Processing Element (SPE)

The real power of the CELL processor does not lie in the PPE, but the other cores, eight of which is available in the current chip design, but only six enabled in the PlayStation 3. In the PlayStation 3, one is disabled for wafer yield reasons (If one is defective it is disabled, if none are defective a good one is disabled). The other is held hostage by the OS virtualization layer, the hypervisor, for internal purposes. These cores were originally code-named *Streaming Processing Elements (SPEs)* because of the suitability for steaming processing, and later renamed to *Synergistic Processing Elements*. The SPEs consist of the *Streaming Processing Unit (SPU)*, 256 KB of local memory for both code and data referred to as *local store (LS)*, and *Memory Flow Controller (MFC)*. Figure 2.2 shows the structure of the SPE.

The SPEs can only execute code residing in the local store and only operate on data residing in the local store. To the SPE the local store represents a flat 18-bit address space. Code and data can be moved between main memory and the local store through the internal bus (see next section) using *Direct Memory Access (DMA)* capabilities of the Memory Flow Controller. In principle the SPEs constitute a small distributed memory system on a chip, where data motion is managed with explicit messaging. The DMA facilities enable perfect overlapping of communication and computation, where some data is being processed while other is in flight.

The SPEs are short vector SIMD workhorses of the CELL. They possess a large 128-entry 128-bit vector register file, and a range of SIMD instructions which can operate simultaneously on 2 double precision values, 4 single precision values, 8 16-bit integers or 16 8-bit chars. Most of instructions are pipelined and can complete one vector operation in each clock cycle. This includes fused multiplication-addition in single precision, which means that two floating point operations can be accomplished on four values in each clock cycle, what translates to the peak of $2 \times 4 \times 3.2 = 25.6$ Gflop/s for each SPE, and adds up to the staggering peak of $6 \times 25.6 = 153.6$ Gflop/s for the PlayStation 3.

Unfortunately for scientific computing, equal emphasis was not put on double precision perfor-

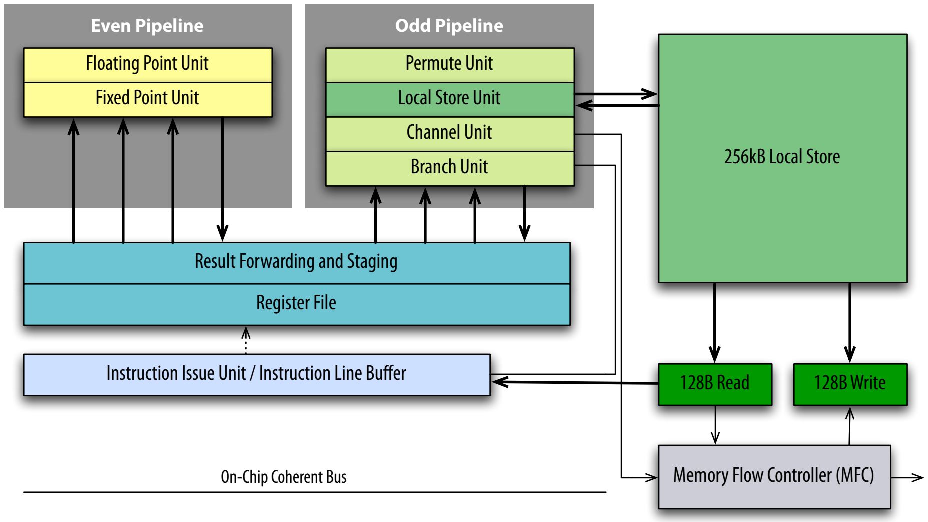

Figure 2.2: *Synergistic Processing Element architecture.*

mance. Unlike the PPE's VMV, the SPEs support double precision arithmetic, but the double precision instructions are not fully pipelined. In particular the FMA operation has a latency of seven cycles. As a result, the double precision peak of a single SPE equals $2 \times 2 \times 3.2/7 = 1.8$ Gflop/s what adds up to the peak of almost 11 Gflop/s for the PlayStation 3, which is still not bad comparing to other *common* processors.

SPEs have two pipelines with one being devoted to arithmetic and the other being devoted to data motion. Both issue instructions in-order and, if certain rules are followed, two instructions can be issued in each clock cycle, one to each pipeline.

2.1.3 Element Interconnection Bus (EIB)

All components of the CELL processors including the PPE, the SPEs, the main memory and I/O are interconnected with the *Element Interconnection Bus (EIB)*. The EIB is build of four unidirectional rings, two in each direction and an arbitration mechanism playing the role of *traffic lights*. Channel width and clock rates aside, each participant is hooked up to the bus with the bandwidth of 25.6 GB/s, and for all practical purposes you can assume that the bus cannot be saturated - you will not run out of internal bandwidth for any realistic workload.

2.1.4 Memory System

The memory system is built of dual-channel Rambus *Extreme Data Rate (XDR)* memory. The PlayStation 3 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 PlayStation 3

2.2.1 Network Card

The PlayStation 3 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. Linux as a guest OS has to use dedicated hypervisor call to access or setup the chip. This is done by a Linux driver called `gelic_net`. The network card has a dedicated DMA unit, which allows to make data transfer without PPE's intervention. To help with this, there is a dedicated hypervisor call to set up a DMAC.

One of many advantages of GigaBit Ethernet is possibility of increased frame size – so called Jumbo Frames. Many standard-compliant equipment pieces allow you to increase frame size from 1500 to 9000. It can increase available bandwidth by 20% in some case and significantly decreases processor load when handling network traffic. At this point, the PS3's built-in GigaBit NIC is limited by the kernel driver – the frame size can not be larger than 2308 bytes (see file `drivers/net/gelic_net.c`). It is not a recommended to change it from the default value of 1500 as it will not give enough performance increase to justify the hassle of configuring all the hardware connected to the card (switches, NFS servers, etc.)

2.2.2 Graphics Card

The PlayStation 3 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 with mutiheaded display walls.

2.3 GigaBit Ethernet Switch

Making a cluster out of separate PS3s requires an interconnectoin network such as a switch. It is possible to use a cheap switch to accommodate small number of units. However, PS3 features a relatively good NIC and the switch can quickly become the bottleneck in the cluster performance. Accordingly, a high-grade dedicated GigaBit Ethernet switch is recommended, which also gives the opportunity to use Jumbo Frames, although there are difficulties with this solution (see section 2.2.1).

2.4 Power Consumption

An important aspect of a cluster installation is total power consumption and heat dissipation. The nominal power consumption of PlayStation 3 (as quoted on the back of each unit sold in North America) is 3.2 A at 120 V which is 384 Watts. It is only an estimate: a distributed computing workload (see section 8.3) uses about 200 Watts. This means that a standard electric outlet providing 20 A can handle not more than 6 PlayStations and a switch. Larger installation will require separate circuits. Similarly, heating is not a problem with small installations, but will likely become a consideration for larger ones.

CHAPTER 3

Software

3.1 Virtualization Layer: Game OS

Linux runs on the PlayStation 3 on top of a virtualization layer (also called hypervisor) that Sony refers to as *Game OS*. This means that all the hardware is accessible only through the hypervisor calls. The hardware signals the kernel through virtualized interrupts. They are used to implement callbacks for non-blocking system calls. Game OS permanently occupies one of the SPEs and controls access to 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.

3.2 Linux Kernel

At this point there are numerous distributions that have official or unofficial support for PS3. The distributions that are are currently known to work on PS3 (with varying level of support and end-user experience) include:

- Fedora Core 5 (section 4.1),
- YellowDog 5.0 (`http://www.ydl.net/`),
- Gentoo PowerPC 64 edition (`http://whitesanjuro.googlepages.com/`),

- Debian/etch (http://www.keshi.org/moin/moin.cgi/PS3/Debian/Live/).

The Fedora Core 5 distribution for PS3 features kernel version 2.6.16. Unfortunately, the official kernel has debugging enabled which results in a 30 MB kernel binary. The best advice is to recompile the kernel to make it smaller. A strong advantage of this distribution is compatibility with IBM's SDK for the CELL processor. If what you need is paid support then YellowDog 5.0 is the right choice. The distribution is specifically designed for PS3. It features the 2.6.16 kernel (already compiled without debugging support). Gentoo PowerPC 64 edition can also be installed and run on the console. Two hardware threads in the PPE should definitely help with Gentoo's constant compilation process. There also exists Live CD version of Debian/etch that can be installed on PS3 hardware.

All the distribution mentioned include Sony-contributed patches to the kernel to make it work on PS3 hardware and talk to the hypervisor. However, the kernel version 2.6.20 has the PS3 support already included in the source code without the need for external patches. Once this version of the kernel becomes more widespread it will be possible to use virtually any distribution on the PS3 hardware. And conversely, some of the PS3 hardware, like the game controller, will be usable under stock GNU/Linux installation.

3.3 Compilers

There is no compiler in existence today that will compile an existing C/C++ or Fortran code for parallel execution on the CELL processor and take full advantage of its performance.

All Linux distributions that support PlayStation 3 include a set of compilers that allow development in C/C++ and Fortran for execution on the PPE only. Fedora Core 5 comes with gcc version 4.1.0 which allows compilation of C (gcc), C++ (g++), and Fortran 95 (gfortran) programs. All these compilers will generate code only for the PPE. Accessing SPE will require the IBM's CELL SDK (section 5.1).

The Fedora Core 5 build of gcc 4.1.0 includes backport of OpenMP from version 4.2 of the gcc compiler. This means that adding the -fopenmp option will make the compiler recognize OpenMP directives and make the linker add the GOMP library to the binary. As mentioned earlier, PPE has direct support for two threads in hardware. This means that setting the environment variable OMP_NUM_THREADS to 2 will benefit some OpenMP-enabled codes. But OpenMP feature in the compiler will not use SPEs for computations. This can only be done by using IBM's CELL SDK (see section 5.1 for details).

In section 7 we discuss software technologies which allow for automatic parallelization of code to take advantage of the multi-core nature of the CELL without requiring the programmer to manually parallelize the code. However, right now, these efforts are still research projects rather than deployable solutions.

3.4 TCP/IP Stack

One way of accomplishing network programming on a cluster is by using the kernel's builtin socket interface. Without modifying the console's hardware the TCP/IP stack will in fact be the fastest way to communicate. Even programming interfaces geared more towards large scale parallel programming have to use sockets as the communicating medium and so the TCP/IP API's performance provides the upper bound of what can be achieved in terms of bandwidth and latency. Testing the socket interface is also probably one of the first things that can be done on a newly setup cluster.

The simplest TCP/IP network test can be performed using the `ping(8)` command. Here is an output from the flood mode (`ping -c 100000 -f host`):

```
100000 packets transmitted, 100000 received, 0% packet loss, time 32707ms
rtt min/avg/max/mdev = 0.084/0.307/0.689/0.100 ms, ipg/ewma 0.327/0.230 ms
```

and the standard mode with one second interval between packets (`ping -c 100 host`):

```
100 packets transmitted, 100 received, 0% packet loss, time 98998ms
rtt min/avg/max/mdev = 0.239/0.249/0.463/0.030 ms
```

One thing to note is the relatively high latency – on the order of 250μs – as compared to 60μs that can be obtained with the same NIC and GigE switch on a common x86 Linux machine. The main contributor to such high latency is the virtualization layer (see section 3.1).

3.5 MPI

Predominant model for programming numerical applications in cluster environment is set by the *Message Passing Interface (MPI)* standard. A few implementations of the standard are available in source code. Most popular are MPICH1, MPICH2, and OpenMPI. Chapter 4.5 includes more information about each of those packages along with installation instructions. Chapter 5.2 includes more information about PS3 code development using MPI.

Figure 3.1 compares these libraries using NetPIPE version 3.6.2. The figure might not directly indicate how each of the implementations will perform with a particular application but should give a general sense of what to expect and serve as a reference point quick sanity check of a particular installation.

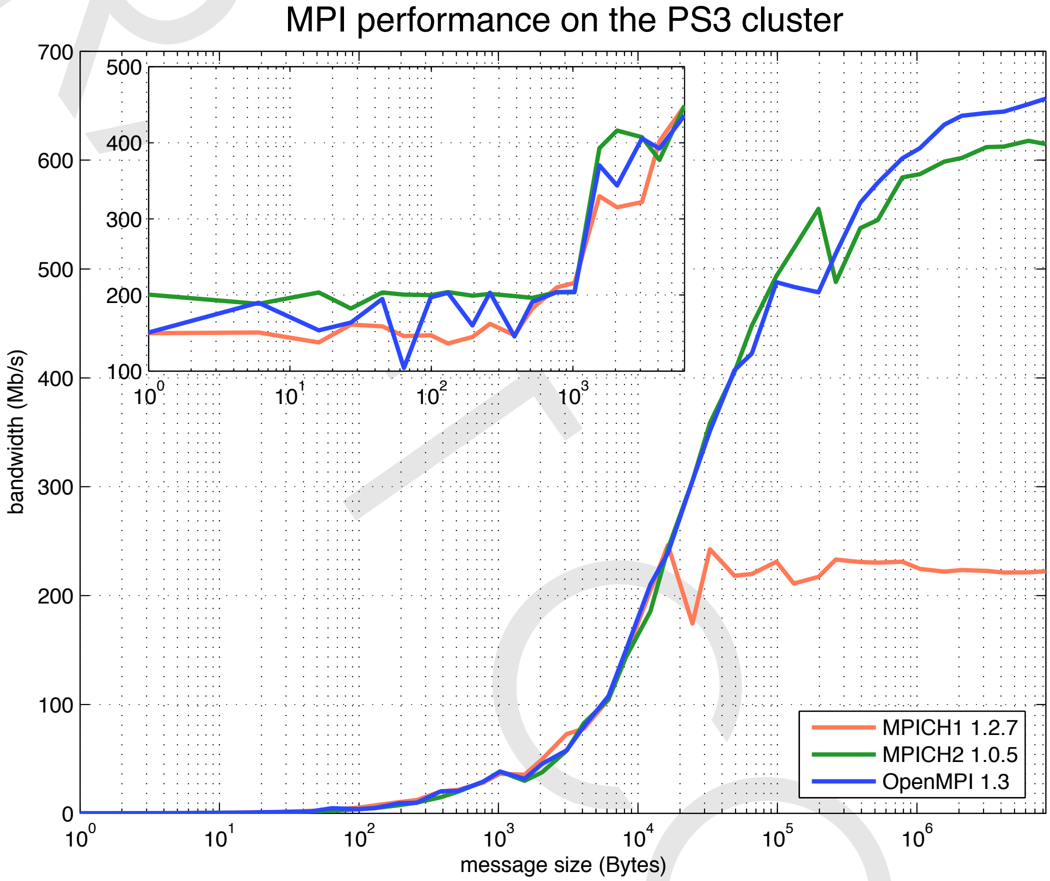

Figure 3.1: *Comparison of various MPI implementations using NetPIPE 3.6.2.*

CHAPTER 4

Cluster Setup

4.1 Basic Linux Installation

The native operating system of the PlayStation 3 (Game OS) includes provisions for smooth installation of another operating system. As of today, a number of PowerPC Linux distributions are available for installation on the PlayStation. Fedora Core, Gentoo and Yellow Dog can serve as examples. It is possible to make a permanent installation, as well as boot the system from a bootable CD (live-CD). We have successfully tried using Gentoo live-CD, as well as installing Fedora Core on the PlayStation's hard drive. Notably, a hard drive installation does not wipe out the native operating system, but enables coexistence of both systems in a dual-boot setup. If the PlayStation is meant for use in a cluster installation, a permanent installation is the obvious choice. Since Fedora Core 5 is IBM's system of choice for the SDK 2.0, it is probably the safest choice of Linux for the PS3, especially if the console will also host the development environment.

The Web contains plenty of resources for installing Linux on the PlayStation 3. However, we would like to point to the CellPerformance Web page (http://www.cellperformance.com/) as an excellent source of straightforward installation instructions, as well as a good starting point for further searches. Here, we just summarize the basic steps of the installation. In order to install Fedora Core 5 on the PlayStation 3 you need to follow these simple steps:

- Make necessary downloads:

  - ❑ Download Fedora Core 5 DVD ISO image **FC-5-ppc-DVD.iso** from a trusted mirror site, e.g. `ftp://mirror.linux.duke.edu/pub/fedora/linux/core/5/ppc/iso/`, and burn it to a DVD.
  - ❑ Download the PS3 add-on CD ISO image **CELL-Linux-CL_20061110-ADDON.iso** available at `ftp://ftp.uk.linux.org/pub/linux/Sony-PS3/`, and burn it to a CD.
  - ❑ If you plan to recompile the kernel, which is recommended, you should also download and burn the kernel source CD **CELL-Linux-CL_20061110-SRCCD.iso** available at the same location.
  - ❑ Copy the file **/kboot/otheros.bld** from the PS3 add-on CD to the location */ps3/otheros/otheros.bld* on a USB flash drive.
  - ❑ Go to the PlayStation Open Platform Web page, and follow instructions to download *Other OS Installer* **otheros.self**. Place it on the same USB flash drive in the location */ps3/otheros/otheros.self*.

- ❍ Prepare the PlayStation:
  - ❑ Repartition and format the PlayStation's hard drive by using the XMB (Cross Media Bar) menu of the Game OS. Go to **Settings** → **System Settings** → **Format Utility**. Preferably pick 10 GB for the Game OS, and the rest for Linux. This will create a Linux partition of around 43 GB in size.
  - ❑ Put the USB flash drive into one of the USB ports. Go to **Settings** → **System Settings** → **Install Other OS** and follow the instructions. The files on the USB drive should be detected automatically.
  - ❑ Go to **Settings** → **System Settings** → **Default System** and change the default system from *PS3* to *Other OS*. Restart the system.

- ❍ Install Linux:
  - ❑ When prompted by the *kboot* boot loader, type **install-fc sda**.
  - ❑ When prompted, insert the Fedora Core 5 DVD, and hit *Enter*.
  - ❑ When prompted, make the selection between the full and the minimal installation. Pick the minimal installation to avoid installation of unnecessary components. This will dramatically decrease the amount of disk consumed by the system and the length of the installation process.
  - ❑ When prompted, replace the Fedora DVD with the add-on CD, and hit *Enter*.
  - ❑ When prompted, enter the password for the root.

    ❑ Type **reboot** and hit *Enter*.

❍ Start network services by issuing the following sequence of commands as the root and reboot the system:

```
$ /sbin/chkconfig --level 345 NetworkManager on
$ /sbin/chkconfig --level 345 NetworkManagerDispatcher on
$ /sbin/service NetworkManager start
$ /sbin/service NetworkManagerDispatcher start
```

The PlayStation can be manually shut down at any time by pressing the power button for a couple of seconds. When Linux is running, it will initiate a shutdown, as if the **shutdown** command was issued. The PlayStation should never be powered down by using the switch on the back of the unit. Under Linux, the PlayStation can be booted to the Game OS by issuing the command **boot-game-os**. Also, pushing the power button for a couple of seconds (until the second beep) at the start time will boot the Game OS and set it as the default operating system.

When Linux boots up, the session starts in a low resolution video mode. This can be changed by using the **ps3videomode** command. For instance, issuing **ps3videomode -v 3 -f** will change the video mode to 720p. The available video modes are: 480i (1), 480p (2), 720p (3), 1080i (4) and 1080p (5) for NTSC territories, and 576i (6), 576p (7), 720p (8), 1080i (9) and 1080p (10) for PAL territories. The flag **-f** stands for full screen mode and it should be skipped if using it results in the monitor cropping the edges of the screen.

4.2 Linux Kernel Recompilation

In principle, every Linux distribution should be capable of running on the PlayStation 3. Since the operating system runs on the PPE and the PPE is binary compatible with the PowerPC processor, any distribution that comes in a PowerPC flavor can be installed on the PS3. In order to have access to the SPEs and to enable other devices, like the controller, a special, patched, Linux kernel has to be used. The source files for the patched kernel can be found in the compressed archive **linux-20061110.tar.bz2** on the Fedora kernel source CD (SRCCD). The recent *vanilla* kernel, version 2.6.20, provides full support for the CELL processor.

Since most of the common Linux distributions (apart from Gentoo) are based on binary packages, the Linux kernel image comes precompiled when the operating system is installed, which potentially relieves the user from the boring and sometimes complicated kernel configuration/compilation. Unfortunately, the recompilation is still necessary to enable features that are not included in the binary package by default and to fine tune the system performance.

Our experience in programming the CELL processor shows that using huge TLB pages improves performance in most cases. This feature is not enabled in the binary kernel package provided with any distribution for the PS3. By the same token, we have to reconfigure and recompile the kernel in order to enable huge TLB pages. The configuration step is quite easy, since we can apply the same configuration as was used for the binary package (usually saved in the */boot* directory) and change just what we need. Here is a quick procedure:

- Recompile the kernel with support for huge TLB pages:

  - Take the kernel source from the Fedora kernel source CD (**linux-20061110.tar.bz2**).
  - Unpack the archive to the directory */usr/src*.
  - Create the symbolic link to the directory containing the kernel source:

    ```
    $ ls -n /usr/src/linux-20061110 /usr/src/linux
    ```

  - Copy the kernel config file that comes with the Fedora installation to the directory */usr/src/linux*:

    ```
    $ cp /boot/config-2.6.16 /usr/src/linux/.config
    ```

  - Prepare for kernel configuration:

    ```
    $ make mrproper
    $ make oldconfig
    ```

    In this step, the old configuration file is analyzed and you are prompted whenever an option is encountered, which is not present in the old kernel. In this case the old and the new kernel are exactly the same, and no prompts should appear.

  - Enable huge TLB pages in the kernel configuration:

    ```
    $ make menuconfig
    ```

    Go to **File systems** → **Pseudo filesystems** and enable huge TLB pages by pressing the space bar on the **HugeTLB file system support** option. Select ”exit” repeatedly and answer ”yes” when asked to save the new kernel configuration.

  - Compile the kernel and the modules, and install the modules (It will take around 20 minutes):

    ```
    $ make all
    $ make modules install
    ```

- Install the new kernel:

```
$ cp /usr/src/linux/vmlinux /boot/vmlinux-2.6.16_HTLB
```

- Create a ramdisk image for the new kernel:

  ```
  $ mkinitrd /boot/initrd-2.6.16_HTLB.img 2.6.16
  ```

- Tell the bootloader (kboot) where the new kernel is located:

  ```
  $ vim /etc/kboot.conf
  ```

  Add the following line:

  ```
  linux_htlb='/boot/vmlinux-2.6.16_HTLB initrd=/boot/initrd-2.6.16_HTLB.img'
  ```

  If you want this kernel to be loaded by default, change the ”default” line into:

  ```
  default=linux_htlb
  ```

- Instrument the boot process to include huge TLB pages allocation:

  ```
  $ vim /etc/rc.local
  ```

  Add the following lines:

  ```
  mkdir -p /huge
  echo 20 > /proc/sys/vm/nr_hugepages
  mount -t hugetlbfs nodev /huge
  chown root:root /huge
  chmod 755 /huge
  ```

  Be sure to change the ”chown” line according to your system settings.

- Reboot. During the boot process, when presented the ”kboot:” prompt you will be able to choose your kernel using the ”tab” key.

- All the commands added to the rc.local file are executed at the end of the boot sequence. This means that the allocation of the huge TLB pages is performed when plenty of system memory has already been allocated to other processes. This results in allocation of only six or seven huge pages. In order to obtain a few more huge pages (eight or nine), we have to move the huge TLB pages allocation to an earlier stage in the boot sequence (i.e. to runlevel-1). In order to do that, create the */etc/init.d/htlb* script with the content shown in Figure 4.1 and add the service to runlevel-1:

  ```
  $ /sbin/chkconfig --add htlb
  ```

```
#!/bin/sh
#
# htlb: Start/stop huge TLB pages allocation
# chkconfig: 12345 80 20
# description: My script to run at runlevels 1,2,3,4 and 5
#

. /etc/rc.d/init.d/functions

start()
{
    mkdir -p /huge
    echo 20 > /proc/sys/vm/nr_hugepages
    mount -t hugetlbfs nodev /huge
    chown root:root /huge
    chmod 775 /huge
       }

stop()
{
    echo 0 > /proc/sys/vm/nr_hugepages
}

case "$1" in
  start)
start
;;
  stop)
stop
;;
  restart|reload)
        stop
        start
        ;;
  *)
        echo $"Usage: $0 {start|stop|status|restart|reload}"
        exit 1
;;
esac

exit 0
```

Figure 4.1: *The `/etc/init.d/htlb` script.*

Having huge TLB pages enabled can improve the performance of many applications. The reason for this improvement lies in the ability to perform fewer translations of the virtual addresses into physical addresses. In order to take advantage of this, memory as to be allocated in a special way. Suppose that four arrays have to be allocated each of size `array_size`; the code in Figure 4.2 shows how to perform these tasks.

```
#define FILE_NAME "/huge/hugefile"
#define HUGE_PAGE_SIZE  16*1024*1024

  ...

  char* addr;
  int fd, count, sv;
  double *array_1, *array_2, *array_3, *array_4;

  ...

  count  = 4*array_size*sizeof(double);
  count  = (count + HUGE_PAGE_SIZE-1) & ~(HUGE_PAGE_SIZE-1);

  printf("=== Using hugetlbfs ===\n");
  fd = open(FILE_NAME, O_CREAT|O_RDWR, 0755);
  assert(fd != -1);
  remove(FILE_NAME);

  addr  = (char *)mmap(0, count, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  assert(addr != MAP_FAILED);

  array_1    = (char  *)(addr);
  array_2    = (char  *)(addr+array_size);
  array_3    = (char  *)(addr+2*array_size);
  array_4    = (char  *)(addr+3*array_size);

  ...
```

Figure 4.2: *How to allocate memory on Huge TLB pages.*

4.3 IBM CELL SDK Installation

It is a remarkable fact that IBM provided the community with the CELL full system simulator [2, 3, 4], which enabled running CELL binaries on a PowePC or x86 host, before wide availability of CELL-based hardware. By the same token, the earliest method of CELL code development was cross-compilation on a non-CELL host. Despite the availability of the hardware today, there seems to be many advantages of being able to compile and build CELL code on a personal desktop or laptop. It is our method of choice, which we highly recommend to others.

Today, the third installment of the Software Development Kit, SDK 2.0, is available, following previous SDK 1.0 and 1.1. Although in principle the SDK can be used with any distribution of Linux, Fedora Core is IBM's distribution of choice. The initial version of the SDK required Fedora Core 4, current version 2.0 requires Fedora Core 5. The installation process for Fedora Core 5 is the most straightforward, and we also recommend it, especially for users without a thorough understanding of the differences between Linux distributions.

Over time, the installation process of the SDK simplified significantly. Software Development Kit 2.0 Installation Guide [5] provides excellent instructions for installing Fedora Core 5 on a host system, as well as instructions for installing the SDK 2.0 on the host system, and also the instructions for installing Eclipse IDE, for those users who are interested. Below, we summarize the steps for installing the SDK.

- Download the SDK ISO image **CellSDK20.iso** from the CELL SDK alphaWorks web site (`http://www.alphaworks.ibm.com/tech/cellsw/download`).
- As root, create a mount directory and mount the disk image on the mount directory:

```
$ mkdir -p /mnt/cellsdk
$ mount -o loop CellSDK20.iso /mnt/cellsdk
```

- Go to */mnt/cellsdk/software*. Invoke the installation script and follow the prompts:

```
$ cd /mnt/cellsdk/software
$ ./cellsdk/install
```

- Leave the mount directory and umount the disk image.

The last step is installing the *libspe* library on the PlayStation 3. *Libspe* is a PPE library for launching SPE executables and communicating with SPE threads. Two versions of the library are currently available, *libspe 1.2* and *libspe 2.0*. The former is a must-have. It introduced the original programming model for launching and controlling the SPE threads, and most of the code samples and tutorials are based on this model. The latter introduced a new programming model and is currently regarded as a prototype. We recommend that you install both libraries.

Both libraries are available on the web page of the *Barcelona Supercomputer Center* (BSC). Do the following to install the libraries:

- Go to BSC *Linux on CELL BE-based Systems* web page (`http://www.bsc.es/projects/deepcomputing/linuxoncell/`).
- Go to **Programming Models** → **Linux on Cell** → **Cell BE Components** → **libSPE**. Download **libspe-1.2.0-0.ppc.rpm**.
- Go to **Programming Models** → **Linux on Cell** → **Cell BE Components** → **libSPE 2**. Download **libspe2-2.0.1-1.ppc.rpm**.
- Move both RPMs to the PlayStation 3 and install them by issuing as root:

```
$ rpm -i libspe-1.2.0-0.ppc.rpm
$ rpm -i libspe2-2.0.1-1.ppc.rpm
```

After this step the installation is complete and the PlayStation 3 is ready to run your CELL code. Do not hesitate, however, to browse through the CELL repositories of the Barcelona Supercomputer Center. It is an excellent resource for valuable CELL code and documentation.

4.4 Network Configuration

In principle, configuring a cluster made of PS3 nodes isn't any different than configuring a cluster made of other kind of nodes. Network configuration consists of a few simple steps that are well known to anybody that has built a cluster at least once. Different approaches may be followed when building a cluster depending on the size of the cluster itself, the availability of resources and of software. Our choice is to put the cluster nodes behind a front-end machine (a *regular* Linux box). This will give us a number of advantages such as:

❍ **Better security.** Security policies can be enforced on the front-end and, since access to the nodes is only possible through the front-end, the whole cluster can be hidden from the rest of the network.

❍ **Cluster symmetry.** All the services needed by the cluster can be run on the front-end instead of one of the nodes. Services include shared volumes, Network Information Service, batch job schedulers, etc.

The front-end node must have two network cards; one will serve as an interface to the external network allowing users to remotely connect to the cluster, and the other will be connected to the internal cluster network. In the cluster internal network, each node, including the front-end, will have a static IP address. The network interface configuration can easily be done under Fedora Core Linux by editing the `/etc/sysconfig/network-scripts/ifcfg-eth0` file on each node; this script will configure both the network interface and the routing table. The content of this file is shown in Figure 4.3.

Once the `/etc/sysconfig/network-scripts/ifcfg-eth0` file is filled with the proper information, it is only necessary to edit the `/etc/resolv.conf` and the `/etc/hosts` files in order to set a name server (DNS) and a list of the nodes' hostnames. Since these files will be the same on every node of the cluster, they can be created on one and then copied to the others. Finally, on each node, the hostname must be set with the command `$hostname node01` (where `node01` must be replaced with the proper hostname).

An easier and more scalable approach would be to use a router that is capable of doing static DHCP. Once the router is configured, each node will acquire the IP address, hostname, and DNS server directly from the router through the DHCP protocol. In this case the content of the `/etc/sysconfig/network-scripts/ifcfg-eth0` file becomes that in Figure 4.4.

```
DEVICE=eth0
BOOTPROTO=static
HWADDR=xx:xx:xx:xx:xx:xx
IPADDR=192.168.1.10
NETMASK=255.255.255.0
NETWORK=192.168.1.0
BROADCAST=192.168.1.255
ONBOOT=yes
NAME=eth0
```

Figure 4.3: *The `/etc/sysconfig/network-scripts/ifcfg-eth0` script.*

```
DEVICE=eth0
BOOTPROTO=dhcp
HWADDR=xx:xx:xx:xx:xx:xx
ONBOOT=yes
NAME=eth0
```

Figure 4.4: *The `/etc/sysconfig/network-scripts/ifcfg-eth0` script when DHCP is used.*

Once the network is configured, it is possible to set up the services.

It is very important that all the users and user groups appear on every node with the same name and ID (UID for users and GID for groups). This can be done manually but it is very inconvenient. In fact, every time a new user must be added, his account has to be manually created on each node of the cluster. An easier and much scalable solution is to use the Network Information Service (a good alternative is LDAP). The NIS server has to be installed on the front-end node and a client will be installed on each node. A good guide on how to set up the NIS server and the client can be found at `http://tldp.org/HOWTO/NIS-HOWTO/index.html`. Once the NIS server and clients are set up, new users need only be added to the front-end node (with a slightly different procedure than usual) and they will be acquired by the nodes through the NIS service.

Another important service that is almost mandatory to set up in a cluster is the network file system. Such service provides shared disk volumes to the cluster users. These volumes (that usually correspond to the whole user's home directory) are visible from any cluster node, front-end included, and this means that if, for example, a file in this volume is edited, the changes do not have to be replicated on each node. There are many network file systems available, but the most common choice for small clusters is NFS. As for the NIS service, the NFS server has to be installed on the front-end and the clients on each node. The fact that the NFS server is on the front-end node also means that the shared volume has to physically be hosted there. Thus, the front-end must be

```
$ env RSHCOMMAND=ssh ./configure --with-device=ch_p4 --without-romio
   --disable-f77 --disable-f90 --disable-doc --disable-cxx
$ make -j2
$ make install
```

Figure 4.5: *The commands we used to set up, compile and build the MPICH1 library.*

equipped with a fast hard disk that is big enough to provide a reasonable quota to each user on the system. A good guide for the configuration of the NFS service can be found at http://tldp.org/HOWTO/NFS-HOWTO/index.html.

Finally, it can be a good choice to install a job scheduler. Job schedulers are used to manage the usage of a cluster by different users through a number of queues where jobs can be submitted. Once the job has been submitted, a priority is assigned to it based on the scheduler rules. The job is executed only when no other jobs in the queues have higher priority. This avoids multiple jobs running on the same node, which is a desirable condition in order to achieve high performance and predictability. Job schedulers are almost necessary for big clusters that are accessed by a large number of users but can give good advantages even on small sized clusters with few users. The most common and widely adopted choice for commodity clusters is the OpenPBS scheduler that can be obtained from http://www.openpbs.org/ along with documentation on how to set it up.

4.5 MPI Installation

4.5.1 MPICH1

MPICH1 from Argonne National Lab is a widely used implementation of the MPI 1.1 standard. The MPICH1 library can be downloaded by following instructions provided at http://www-unix.mcs.anl.gov/mpi/mpich1/. The version of the software that we tested was 1.2.7p1. It was released in November 4th, 2005. The authors strongly encourage the use of MPICH2 instead of MPICH1 for homogeneous clusters (such as a PS3 cluster). But for completeness, we decided to include our experiences with MPICH1 in this writing.

Figure 4.5 shows the commands we used to set up, compile and build the MPICH1 software. Since similar options and procedures apply to other MPI implementations, we give here a more detailed description of the meaning of the installation steps. The `env RSHCOMMAND=ssh` makes sure MPICH1 uses `ssh(1)` as the remote shell instead of the default `rsh(1)`. Almost all Linux distribution have better support for `ssh(1)` for security reasons. The options passed to the `configure` script were:

- `--with-device=ch_p4` – selects TCP/IP for messaging between nodes

```
$ ./configure --enable-fast --disable-f77 --disable-f90 --disable-cxx
  --disable-romio --disable-threads=single
$ make -j2
$ make install
```

Figure 4.6: *The commands we used to set up, compile and build the MPICH2 library.*

- `--without-romio` – disables ROMI/O – a well known implementation of the MPI I/O part of the MPI 2 standard. If the use of parallel I/O is not intended, the feature can be disabled.
- `--disable-f77` – disables FORTRAN 77 bindings.
- `--disable-f90` – disables FORTRAN 90 bindings.
- `--disable-cxx` – disables C++ bindings.
- `--disable-doc` – disables installation of some of the MPI documentation.

The `make -j2` can be used to take advantage of the two PPE hardware threads to speed up the compilation process.

4.5.2 MPICH2

MPICH2 is an MPI implementation written by the same team of developers that created MPICH1. It has many improvements over its predecessor with possibly only one drawback: lack of support for heterogeneity. MPICH1 allowed for free mixing of computer architectures in a single cluster. MPICH2 made a design decision to not support it which simplified the implementation and increased communication performance. This is a welcome feature if you plan to use MPICH2 for a cluster of only PS3s. Mixing PS3s and, say, PCs with x86 processors would require MPICH1 or Open MPI.

MPICH2 can be downloaded from http://www-unix.mcs.anl.gov/mpi/mpich2/. Just as was the case with MPICH1, setup, compilation and installation are very simple. It is worth mentioning that there is an option (`--enable-threads`) for the `configure` script that controls the threading mode of MPICH2. Depending on threading levels in your current application, it might be worthwhile to experiment with this option as there are two hardware threads on the PPE. Obviously, this option will not use SPEs to run threads inside the MPI library.

One distinct feature of MPICH2 is ability to set up daemons that will make launching jobs quicker (daemon support in MPICH1 was much more limited). Starting the daemons may be done by the privileged user or the end user. The daemon interface is much improved from the previous versions of MPICH and allows you to accommodate more than one MPI process per node.

Figure 4.6 the commands we used for the setup, compilation and installation of MPICH2 on our cluster. The options passed to the `configure` script were:

- `--enable-fast` – speeds up the build process for common configuration options
- `--disable-f77` – disables FORTRAN 77 bindings.
- `--disable-f90` – disables FORTRAN 90 bindings.
- `--disable-cxx` – disables C++ bindings.
- `--disable-romio` – disables ROMI/O – a well known implementation of the MPI I/O part of the MPI 2 standard. If the use of parallel I/O is not intended, the feature can be disabled.
- `--disable-threads=single` – disables threading support: only a single threaded applications will be allowed to link against the resulting library

4.5.3 Open MPI

Open MPI was created when developers of FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI joined their efforts and put together their experiences and invited contributors from other academic institutions, government labs, and vendors. It is a modern MPI implementation that is fully compliant with the MPI2 standard. It is available from http://www.open-mpi.org/. The older release (1.1.x series) is still available but users are strongly encouraged to download at least version 1.2 or even the nightly snapshots of the upcoming 1.3 release, as they are very stable.

Installation of Open MPI is also very simple. There is an option for enabling a set of features at configuration time like detailed debugging support (so that the application stack trace will be printed upon program failure without the need for a debugger) and threading support (to support threaded applications and have a progress thread inside the library for better support for non-blocking communication).

Launching of jobs is also simple. The start up is very fast with use of daemons (similarly to MPICH2) but the daemons are managed transparently by Open MPI without requiring any user intervention.

We tested two different builds of Open MPI:

- Version with debugging symbols and
- Version without debugging symbols.

The exact optioned that were used for both of these versions are given in Figure 4.7.

For completeness, here are the Open MPI equivalents of the options we mentioned while describing both MPICH implementations (with default values given in parentheses) as well as additional options that are specific to Open MPI:

Build with extensive debugging support:

```
$ ./configure --with-platform=ps3 --enable-debug --enable-picky
$ make -j2
$ make install
```

Build without debugging support:

```
$ ./configure --with-platform=ps3 --disable-debug --enable-picky
$ make -j2
$ make install
```

Figure 4.7: *The commands we used to set up, compile and build two configurations of the Open MPI library.*

- `--disable-mpi-f77` (default: enabled) – disables FORTRAN 77 bindings.
- `--disable-mpi-f90` (default: enabled) – disables FORTRAN 90 bindings.
- `--disable-mpi-profile` (default: enabled) – disables MPI's profiling interface.
- `--disable-mpi-cxx` (default: enabled) – disables C++ bindings.
- `--disable-mpi-cxx-seek` (default: enabled) – disables some C++ bindings related to I/O.
- `--enable-mpi-threads` (default: disabled) – enable threading support in the Open MPI library.
- `--enable-progress-threads` (default: disabled) – enable use of a separate thread for handling asynchronous communication.

CHAPTER 5

Development Environment

5.1 CELL Processor

The CELL SDK provides the development environment for programming the CELL processor. The current release of the SDK is a great package offering a range of software tools including: a suite of compilers and debuggers, SIMD math libraries, parallel programming frameworks, full system simulator and Eclipse IDE.

As valuable as all those components are, the compiler suite is of the greatest importance to get you started running code on the CELL processor. Since the PPE and the SPEs are different architectures with disjoint address spaces, they require two distinct toolchains for software development. The most important component of the toolchains is the compiler suite including both the GNU GCC and the IBM XLC compilers. The compilers for both architectures produce object files in the standard *Executable and Linking Format* (ELF). A special format, *CBEA Embedded SPE Object Format* (CESOF), allows SPE executable object files to be embedded inside PPE object files.

Standard compilation steps include:

- Compilation of the SPU source code using either the GNU *spu-gcc* compiler or the IBM *spuxlc* compiler.

- Embedding of the SPU object code using the *ppu-embedspu* utility.

- ❍ Conversion of the embedded SPU code to an SPU library using the *ppu-ar* utility.
- ❍ Compilation of the PPU source code using either the GNU *ppu-gcc* compiler or the IBM *ppuxlc* compiler.
- ❍ Linking the PPU code with the library containing the SPU code and with the *libspe* library, to produce a single CELL executable file.

For example, given two source files *ppu_code.c* and *spu_code.c*, containing the PPU code and SPU code accordingly, the executable *cell_prog* can be built with the toolchain 3.3 by using the rudimentary makefile from Figure 5.1.

```
    TOOLCHAIN = /opt/cell/toolchain-3.3
    SPU_GCC = $(TOOLCHAIN)/bin/spu-gcc
    PPU_GCC = $(TOOLCHAIN)/bin/ppu-gcc
    PPU_MBD = $(TOOLCHAIN)/bin/ppu-embedspu
    PPU_AR  = $(TOOLCHAIN)/bin/ppu-ar
all:
    $(SPU_GCC) -O3 -c spu_code.c
    $(SPU_GCC) -o spu_code spu_code.o
    $(PPU_MBD) -m32 spu_code spu_code spu_code_embed.o
    $(PPU_AR)  -qcs spu_code_lib.a spu_code_embed.o
    $(PPU_GCC) -m32 -O3 -c ppu_code.c
    $(PPU_GCC) -m32 -o cell_prog ppu_code.o spu_code_lib.a -lspe
```

Figure 5.1: *Rudimentary CELL makefile.*

At this point, the development environment is still evolving, and writing your own makefile jeopardizes the portability of your compilation process to the next release of the SDK. IBM promotes the use of portable makefiles based on samples provided with the SDK. The samples, which can be found in the directory */opt/ibm/cell-sdk/prototype/src/samples/*, use the same makefile structure, where the main directory contains the main makefile and two subdirectories, *ppu* and *spu* for the PPU code and the SPU code, respectively. These subdirectories contain appropriate makefiles for compiling the PPU code and SPU code. The makefiles have a very simple structure, where the user's responsibility is to provide a few basic definitions and include the file */opt/ibm/cell-sdk/prototype/make.footer*, which handles aggregation in order to create one CELL executable.

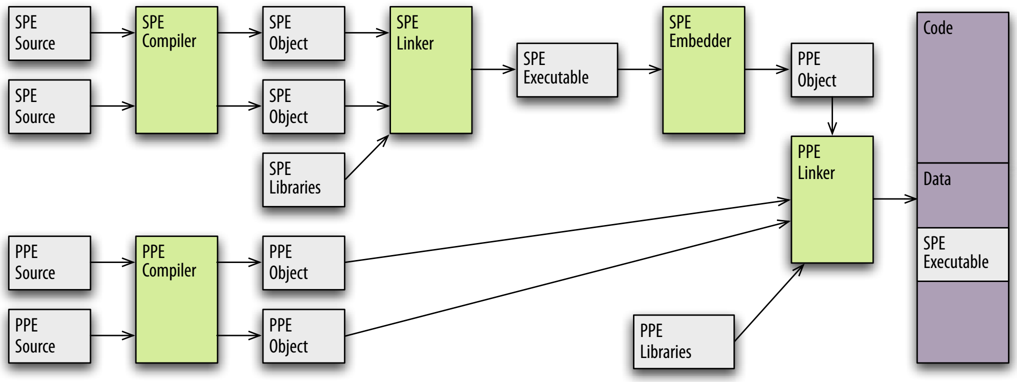

Figure 5.2: *Compilation/building.*

5.2 PlayStation 3 Cluster

The essential part of a cluster development environment is MPI. In a standard setting, an MPI implementation ships with a specialized command called `mpicc`. It invokes a standard compiler with extra flags that allow inclusion of the standard MPI header file `mpi.h`. But the `mpicc` command can also be invoked to link MPI programs. In this mode, `mpicc` adds additional flags to the real linker invocation to add libraries that implement the MPI standard. This is a sample sequence of commands to compile an MPI program from the file `foo.c`:

```
$ mpicc -c foo.c
$ mpicc -o foo foo.o
```

This is the preferred way of developing MPI applications, as it is independent of the MPI implementation being used as long as the right `mpicc` command gets invoked which can be adjusted with the `PATH` environment variable.

On a PlayStation 3 cluster, this mode of operation breaks down if IBM's CELL SDK is combined with MPI and they are not installed on the same machine (especially in cross-compiling scenario). In this case, standard CELL SDK tools should be used with additional options added to make them aware of location of files required by the MPI implementation. Unfortunately, this will make the process dependent on a particular MPI implementation and its supporting libraries. But once setup properly, it will work just as well as it would with the `mpicc` command.

To have the MPI header files available on the development machine they need to be copied from the `include` directory where the MPI implementation was installed. For MPICH1, the header files are `mpi.h`, `mpidefs.h` `mpi_errno.h`, `mpio.h`, and `mpi++.h` (the last two only if support for MPI I/O and MPI C++ bindings were enabled during the setup). For MPICH2 the header files are `mpi.h`,

Feature	MPICH1	MPICH2	OpenMPI

C API	`mpi.h`
	`mpidefs.h`
	`mpi_errno.h`
		`mpi.h`	`mpi.h`

MPI I/O	`mpio.h`	`mpio.h`	
C++ API	`mpi++.h`	`mpicxx.h`	`openmpi/`

Table 5.1: *Header files required for compilation using various MPI implementations.*

Feature	MPICH1	MPICH2	OpenMPI
C API	`libmpich.*`	`libmpich.*`	`libmpi.*`
C++ API	`libmpich++.*`	`libmpichcxx.*`	`libmpi_cxx.*`
FORTRAN API	`libfmpich.*`	`libfmpich.*`	`libmpi_f77`
Miscellaneous Support			`libopen-pal.*` `libopen-rte.*`

Table 5.2: *Library files required for linking with various MPI implementations.*

`mpio.h`, and `mpicxx.h` (the last two only if support for MPI I/O and MPI C++ bindings were enabled during the setup). For Open MPI, only the `mpi.h` file needs to be copied unless MPI C++ bindings are to be used, in which case the contents of the `openmpi` subdirectory should be copied as well. Table 5.1 summarizes the information on the header files.

Just as it is the case during compilation, linking MPI programs requires extra flags. The common command line flags to use for virtually all MPI implementations is the flag that adds a path to MPI libraries at link time and runtime so they can be found by the static and dynamic linker, respectively. The link time flag is `-L` for the static linker and an example use is `-L/path/to/mpi`, whereas to pass the path to the dynamic linker (usually referred to as a runtime path) so it can search for dynamic libraries at runtime one uses `-Wl,-rpath,/path/to/mpi`. As an alternative to the latter, the `LD_LIBRARY_PATH` environment variable can be set prior to running the program. After the path flags come the extra libraries to link. In MPICH1 and MPICH2, the library file is called `libmpich.a` and it can be linked in with `-lmpich` flag. In addition, if Fortran or C++ bindings are required, `-lmpich++` (`-lmpichcxx` in case of MPICH2) and `-lfmpich` have to be added. Open MPI link flags should include `-lmpi` for MPI C binding, `-lmpi_cxx` for MPI C++ binding, and `-lmpi_f77` for MPI Fortran binding. In addition, `libopen-pal.*` and `libopen-rte.*` will be linked into the executable and so they should be copied to the development environment machine. Table 5.2 summarizes the linking information for the three MPI implementation that were used.

To run an MPI application on a cluster, the `mpirun` or `mpiexec` commands should be used.

MPICH1 uses `ssh(1)` or `rsh(1)` commands for launching processes on the nodes of the cluster. The `RSHCOMMAND` environment variable controls which remote shell is used, but in general `ssh(1)` should be a preferred solution. The selection of nodes on which to run an MPI application, is done with a machine file. The default machine file usually resides in the installation directory called `share` but it can also be specified upon invocation of the `mpirun` command. The contents of the file for four PS3 nodes (whose IP names are `node1`, `node2`, `node3`, and `node4`, respectively) might look like this:

```
node1:2
node2:2
node3:2
node4:2
```

The `:2` suffix on each line indicates that there are two processors available on each node: this refers to the fact that the CELL processor's PPE has two hardware threads. With this file it will be possible to launch 8 MPI processes, two on each PS3. Since most of the work is usually done by the SPEs, it is enough to use only one MPI processes per PS3. Accordingly, the machine file for the sample 4 PS3 cluster will usually look like this:

```
node1
node2
node3
node4
```

The most common way of running MPICH2 applications is by launching an MPI daemon (`mpd`) on each node. Once the daemons are started, there is no need to specify a machine file any more because the `mpirun` (and `mpiexec`) command from MPICH2 will communicate to the daemons to discover the nodes and number of processes to spawn on each.

Open MPI also uses a machine file, but it has a slightly different syntax for this file. Here is a sample corresponding to the example given for MPICH1:

```
node1 slots=2
node2 slots=2
node3 slots=2
node4 slots=2
```

CHAPTER 6

Programming Techniques

6.1 CELL Processor

There is no substitute for getting your hands dirty with CELL programming, and unfortunately many people find the learning process to be quite hard at the beginning. Fortunately, the amount of high quality CELL programming literature freely available on the web today is quite impressive. Basically, there is at least one solid publication devoted to each major topic of CELL programming. The reader can choose from a big selection of guides, reference manuals, as well as very approachable beginner-level tutorials.

In this chapter, we will highlight the most important aspects of developing software on the CELL and point the reader to the appropriate literature where in-depth information can be found. Here we attempt to emphasize the aspects that distinguish the CELL processor from the majority of processors on the market today. This chapter is meant for newbies rather than seasoned veterans.

6.1.1 Short Vector SIMD'ization

The power of the CELL is in its SPEs. The SPEs are inherently vector processors, probably the most powerful short vector SIMD engines in existence today. At the same time they do not efficiently implement scalar (non-vector) operations. Non-vector operations are implemented using a so-called *preferred slot*, which is a location within a vector, which can be operated upon in a non-vector fashion.

Using the preferred slot requires the extra operations of shifting the element to the preferred slot and then shifting it back to its original location, which introduces costly overheads.

As a result, great speeds can be achieved using vectorization, and very poor performance can be expected when programming the SPEs using standard, scalar, C code. However, the fact that a standard C code will compile and run on the SPE gives the programmers a great starting point and allows for applying optimizations in a gradual manner.

Going to the level of assembly is not required. C language extensions, called intrinsics, are provided for convenience. Along with the intrinsics come vector type definitions and vector literals. Most of the intrinsics have a one to one translation to assembler instructions. Coding using the intrinsics is very close to coding in assembly, while leaving room for the compiler to do instruction reordering and register coloring. The *Cell Broadband Engine Programming Tutorial* [6] is an excellent place to start learning vectorization. *C/C++ Language Extensions for Cell Broadband Engine Architecture* [7] is the ultimate reference. The *SPU Assembly Language Specification* [8] can occasionally come in handy as well.

The goal of vectorization is to eliminate scalar operations, which means eliminating insertions and extractions of elements to and from a vector. This goal can be achieved by using shuffle operations to rearrange elements within a vector. The dual issue pipelines of the SPE enable performing shuffles in parallel with arithmetic operation, which minimizes their impact. In many situations, shuffles can be almost completely hidden behind arithmetic.

The technique of loop unrolling goes hand in hand with vectorization. Compilers can do a great job scheduling operations and optimizing for dual issues, if they are provided with many *independent* operations. *Independent* is the keyword here, since inexperienced programmers tend to attempt unrolling by putting a number of instructions in the loop, which carry dependencies from one instruction to another. As always, loop unrolling also plays its traditional role of minimizing the overhead of loop branches.

Unroll heavily. Take advantage of the enormous set of 128 vector registers at the SPE's disposal. Create big chunks of straight line code (big basic blocks). Keep your source small and readable by using defines and nested defines. Big basic blocks are a common practice for implementing fast Fourier transforms (FFTs), where such chunks are called *codelets*. There is a limit to the usability of the technique. At some point you can exhaust the register file and experience a performance drop. Also, the code size may become an issue due to the size of the local store. Nevertheless, do not be shy when unrolling. Do not unroll just a few iterations. Unroll tens of iterations. In case of nested loops, think if it is feasible to unroll the inner loop completely. If such techniques seam odd to you, look at the file */opt /ibm /cell-sdk /prototype /src /workloads /matrix_mul /spu /block.c* or */opt /ibm /cell-sdk /prototype /src /workloads /FFT16M /spu /fft_spu.c*.

If you are new to the concept of vectorization in the SIMD sense, optimize your code gradually following this simple scheme:

- Start with implementing your computation on the SPE by writing a standard (scalar) C code.
- Start gradually introducing vector operations. You can freely mix vector and scalar code. Create vector aliases to your scalar data structures, e.g., declare a vector pointer and assign it the beginning of your scalar array. Use that pointer to implement some of your computation.
- Over time try to eliminate scalar operations completely. Eliminate extractions and insertions of vector elements by using shifts, rotations and shuffles.
- Unroll loops. This step takes some experience, since the right amount of unrolling is a function of the algorithm being implemented, size of the register file and desired size of the binary code. The good news is that even a moderate amount of effort will result in satisfactory performance in most cases.

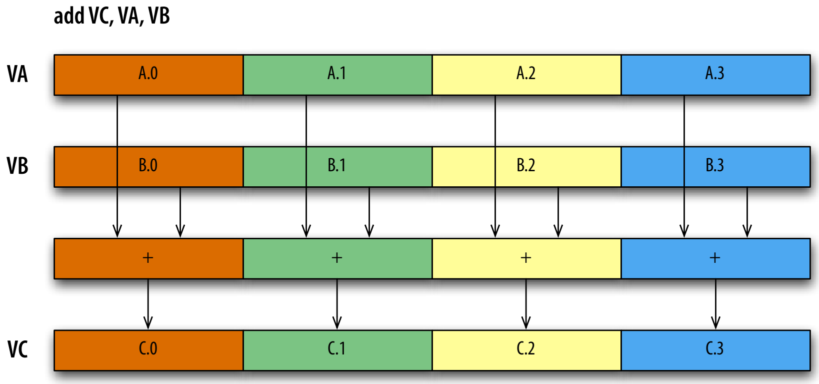

Figure 6.1: *SIMD processing.*

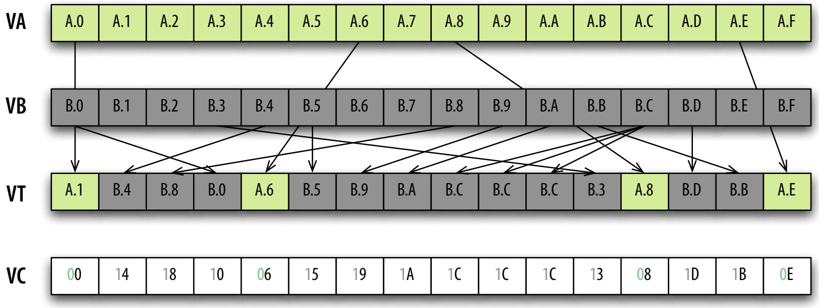

Figure 6.2: *Shuffle.*

6.1.2 Intra-Chip Communication

The CELL processor provides three basic methods of communication - DMA transfers, mailboxes and signals. The *Cell Broadband Engine Programming Tutorial* [6] is good starting point to learn about the CELL communication mechanisms. The *Cell Broadband Engine Programming Handbook* [1] is the ultimate reference. Here we will focus on the first two mechanisms, DMAs and mailboxes, since these are most useful in programming any kind of numerical applications. DMA transfers are the most important means of communication on the CELL processor, facilitating both bulk data transfers and synchronization. In case of SPE to SPE communication, the other two communication mechanisms are actually implemented by means of multiple DMA transfers. Nevertheless, the most important function of DMAs is bulk data movement, equivalent to message passing with MPI. Here we briefly summarize the capabilities of DMA transfers:

- DMA transfers enable exchange of data between the main memory and the local stores of the SPEs, as well as transfers from one local store to another.
- The messages can be of size 1, 2, 4, 8, and 16 bytes, and multiplicities of 16 bytes up to 16KB. Source and destination addresses of messages 16 bytes and larger have to be 16 bytes aligned, and addresses of messages shorter than 16 bytes require the same alignment as the message size. Additionally, messages of subvector sizes (less than 16 bytes) have to have the same alignment of source and destination addresses within the vector.
- Messages larger than 16KB can only be achieved by combining multiple DMA transfers. DMA lists are a convenient facility to achieve this goal, as well as to implement strided memory access. A DMA list can combine up to 2048 DMA transfers.
- DMA transfers are most efficient if they transfer at least one cache line and if they are aligned to the size of a cache line, which is 128 bytes.
- By default, DMA messages are not ordered. Ordering of DMAs can be enforced by the use of *barriers* and *fences*. A barrier orders a message with respect to messages issued before as well as after a given message. A fence orders a message only with respect to messages issued before the given message (Figure 6.3).
- DMA transfers are non-blocking in their very nature. While DMAs are in progress, the SPE should be doing some useful work and only check for DMA completion, when it comes to processing of the transferred data.
- DMA engines are parts of the SPEs. Each SPE can queue up to 16 requests in its own DMA queue. Each DMA engine also has a proxy DMA queue, which can be accessed by the PPE and other SPEs. The proxy queue can hold up to eight requests.

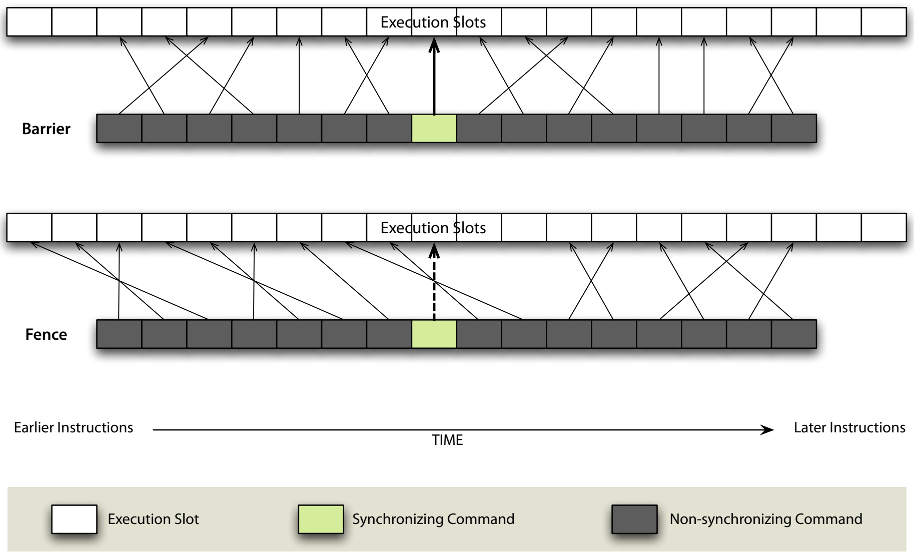

Figure 6.3: *Barrier/fence.*

Both the SPEs and the PPE are capable of initiating DMAs, but the SPE-initiated DMAs are more efficient and should be given preference over the PPE-initiated DMAs. Nevertheless, if the need arises to use the PPE-initiated DMAs, it can be accomplished by means of the MFC SPE proxy command functions described in *SPE Runtime Management Library*, chapter *SPE MFC Proxy Command Functions*.

Although each single SPE has a theoretical bandwidth of 25.6 GB/s, which is equal to the peak bandwidth of the main memory, a single SPE will have a hard time saturating this bandwidth. In order to get good utilization of the bus, you should initiate many requests from many SPEs, and also restrain from ordering the messages, if possible, to give the arbiter the most room for traffic optimization.

An important aspect of the CELL communication system is the efficiency of local store to local store communication. The main memory offers considerable bandwidth of 25.6 GB/s. At the same time, however, the bus connecting the PPE, the SPEs and the main memory possesses much greater internal bandwidth. Different numbers are reported by different sources. Commonly, the bus is rated at 204.8 GB/s. The important aspect here is that the bus is almost impossible to saturate by communication between the interconnected elements. It means that the SPEs, when accessing the main memory heavily, will exhaust the memory bandwidth. At the same time, when communicating between one another, they will never encounter a communication bottleneck. By the same token, if an application has the potential for SPE to SPE communication, such communication should definitely be given preference over main memory communication. An example of such patterns would be stream processing, where data is passed from one SPE to another in a pipeline fashion. Although local store to local store transfers may seem a little less straightforward than main memory transfers, in practice they are not difficult to implement at all. Given that the CELL processor implements a global addressing scheme, in which each local store can be accessed by its *effective address*, local store to local store communication can be implemented as follows:

- The PPE retrieves the effective address of each local store by calling the *spe_get_ls()* function.
- The PPE passes the list of addresses of all local stores to all SPEs, through a DMA transfer.
- On the SPE side, a communication buffer is declared as a global variable and as a result has the same physical addresses within the local store on all SPEs.
- An SPE sums the physical buffer address with the effective address of the local store of another SPE to get the address of the remote buffer. It uses this address as the source address to pull data from the other SPE, or as a destination address to push data to the other SPE.

Local store to local store communication may prove invaluable not only for bulk data transfers, but also for synchronization between SPEs. One thing to remember here is the subvector alignment of source and destination for subvector length transfers.

Mailboxes are a convenient mechanism for sending short, 32-bit messages from the PPE to the SPEs and between the SPEs. The mailboxes are First-In-First-Out (FIFO) queues, meaning the messages are processed in the order of their issue. Each SPE has a four-entry mailbox for receiving incoming messages from the PPE and other SPEs, and two one-entry mailboxes for sending outgoing messages to the PPE and other SPEs - one of which serves the purpose of raising an interrupt on the receiving device. Mailbox operations have blocking nature on the SPE. An attempt to write to a full outbound mailbox will stall until the mailbox is cleared by a PPE read. Similarly, an attempt to read from an empty inbound mailbox will stall until the PPE writes to the mailbox. The same does not apply to the PPE. Neither an attempt to write to a full mailbox nor an attempt to read an empty mailbox will stall the PPE. Mailboxes are useful to communicate short messages, such as completion flags or progress status. They can also serve the purpose of communicating short data, such as storage addresses and function parameters.

The blocking nature of the mailboxes on the SPE side makes them perfect for the PPE to initiate actions on the SPEs. However for two reasons they should not be used by the SPEs to acknowledge completion of operations to the PPE. DMA completion has a local meaning on the SPE. In other words, completion of a DMA on the SPE means that the local buffers are available for reuse, but not that the data made it to the memory. If a DMA transfer is immediately followed by an acknowledgment mailbox message, the message can make it to the PPE before the data. Also, the PPE continuously reading the SPE's outbound mailbox will flood the bus causing loss of bandwidth. A better way of acknowledging completion of an operation or a data transfer from an SPE to the PPE is to use an acknowledgment DMA protected by a fence with respect to the data transfer DMA. The PPE can periodically test the memory location (variable) written to by the SPE, or even *spin* (busy wait) on the variable, whichever is appropriate. You should remember to declare the variable as *volatile* to prevent the compiler from optimizing it out.

A standard communication scenario could have the following structure:

- The PPE sends a mailbox message to the SPE, waiting on a mailbox.
- The SPE receives the mailbox message and interprets the command.
- The SPE pulls data for processing from the main memory, performs appropriate actions, and returns the result to the main memory.
- The SPE sends an acknowledgment message to the PPE by placing a value in the acknowledgment variable in the main memory.
- The PPE tests the acknowledgment variable and receives the completion notification.

6.1.3 Basic Steps of CELL Code Development

A programmer who takes one of the SDK samples and introduces changes by taking a random walk through the code is doomed to failure and endless hours of debugging. The most effective method of debugging is to not produce bugs. We recommend that you take CELL programming step-by-step and follow these simple stages of code development:

- Start by working in 32-bit mode. In this mode, addresses can be cast to unsigned integers and easily operated upon. They can also be passed around as single mailbox messages. There is no system in existence today possessing more memory than can be addressed by 32 bits. In particular, the 256 KB of the PlayStation 3 can be addressed by 32 bits. In scientific parallel computing, you are supposed to be getting more memory by grabbing more processors, not trying to plow through endless amounts of memory on a single CPU.

  If you really need 32/64-bit portability, keep in mind that SPEs always treat effective memory addresses as 64-bit values. When the PPE is running in 32-bit mode the upper 32 bits of an effective address must be set to zero. Also, the *long* type is not portable between addressing modes. Use *int* or *unsigned int* to always get 32-bit values, and *long long* or *unsigned long long* to always get 64-bit values.

- Save yourself the time of learning how to vectorize your code on the AltiVec/VMX and porting it to the SPE. Code for the SPE right away. It is the real thing. It has a massive register file and a range of powerful instructions. Its dual-issue capability will allow you to get close to the peak.

  The indispensable *spu-timing* tool will let you understand exactly what is going on in your code. All you have to do is to compile the SPE code to assembly by adding the **-S** flag to the usual set of compilation flags, and then run the spu-timing tool on the assembly file. The tool will produce an output file containing the scheduling information for the SPE code.

- Perform all your initializations and setup on the PPE. Prepare your data for processing on the SPE in a continuous memory buffer. There is a caveat to this approach, which is that initializations performed by the PPE will move the data to the L2 cache, which slows down the access by the SPEs. Ignore this fact when learning the ropes of CELL programming.

- Write a simple DMA. Implement a copy operation. Make the SPE read a chunk of data from one place in main memory to another. Verify the operation for correctness on the PPE.

- Introduce a processing stage on the SPE between the read and write operation. Do not vectorize yet, but use standard, scalar, C code. It will compile and run out of the box. Copy-paste the same code in the PPE correctness check code. Do not expect bit-wise correctness. SPEs do not implement IEEE compliant floating point arithmetic. In most cases, expect errors on the order of the precision used (machine epsilon).

- Start vectorizing the SPE code. Keep checking for correctness against the scalar code on the PPE. Learn the process gradually. Introduce more and more vectorized code, while keeping some scalar code.

- Start measuring your execution time on the SPE by using the decrementer. Start looking at the output of the spu-timing tool. Your ultimate goal is to eliminate loop overhead via unrolling and maximizing the dual issue rate by mixing arithmetic and data manipulation instructions.

- Try to introduce double-buffering to hide the communication. It is analogous to the similar technique in computer graphics, where the contents of one frame is being displayed, while another is being produced in the memory. Fetch the data for the upcoming loop iteration $N+1$ while processing the data in the current loop iteration N.

- Devise a data or work partitioning strategy, and distribute your task to many SPEs. At the beginning you can use the PPE to synchronize the SPEs. Later on you can try to make the SPEs synchronize between each other. The main synchronization mechanisms are mailboxes, DMAs and signals.

  One straightforward method of synchronization is a DMA transfer of a single variable, which can be polled by the PPE or the SPE in order to determine if an operation completed. Remember to declare such synchronization variables as *volatile* to avoid the compiler optimizing them out.

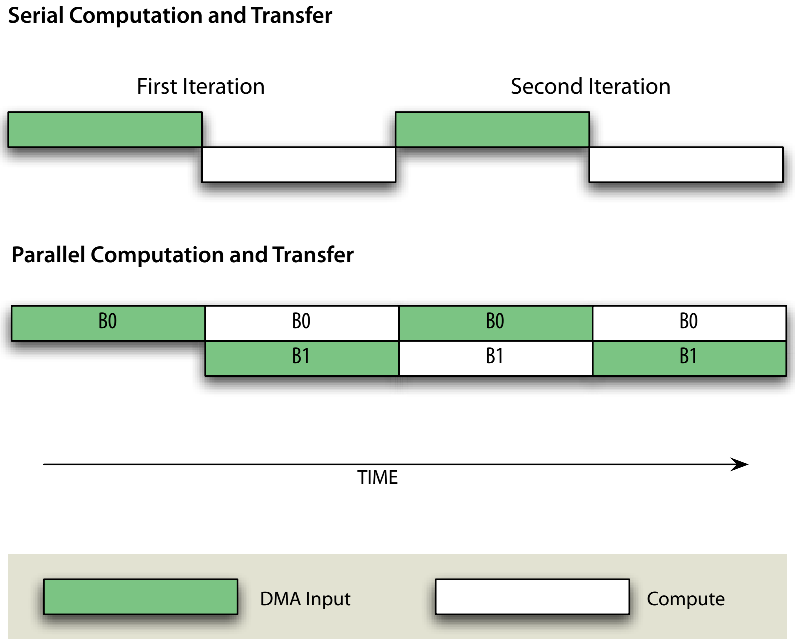

Figure 6.4: *Double buffering.*

6.1.4 Quick Tips

In this section we give a few more programming tips that you may find useful:

- **Tweak performance using static analysis and the decrementer.** Traditionally, programmers rely on performance counters to identify performance problems. The values typically measured include cache misses (L1, L2, ...), TLB misses, and the number of floating point operations. From the SPE standpoint, there is little use in trying to measure these values. Cache misses are nonexistent, TLB misses can be eliminated by using huge TLB pages, and since the SPE's pipelines implement in-order execution, code behavior is precisely defined by the object code. By the same token, the performance of the SPE code operating on the local store can be analyzed by looking at the object/assembly code. The *spu-timing* comes in handy here.

  If you desire to measure the actual execution time, or measure time of operations that exhibit variable performance, like DMA data transfers, the indispensable tool is the SPE decrementer, a hardware register that ticks with a fixed frequency, which can be read and written by the user.

- **Do not use PPU for computational tasks.** The AltiVec/VMX engine on the PPU may appear to you as having the power of a ninth SPE. Do not be misled. The unit will not deliver performance equivalent to the SPE because of the combined effect of two factors: unlike the

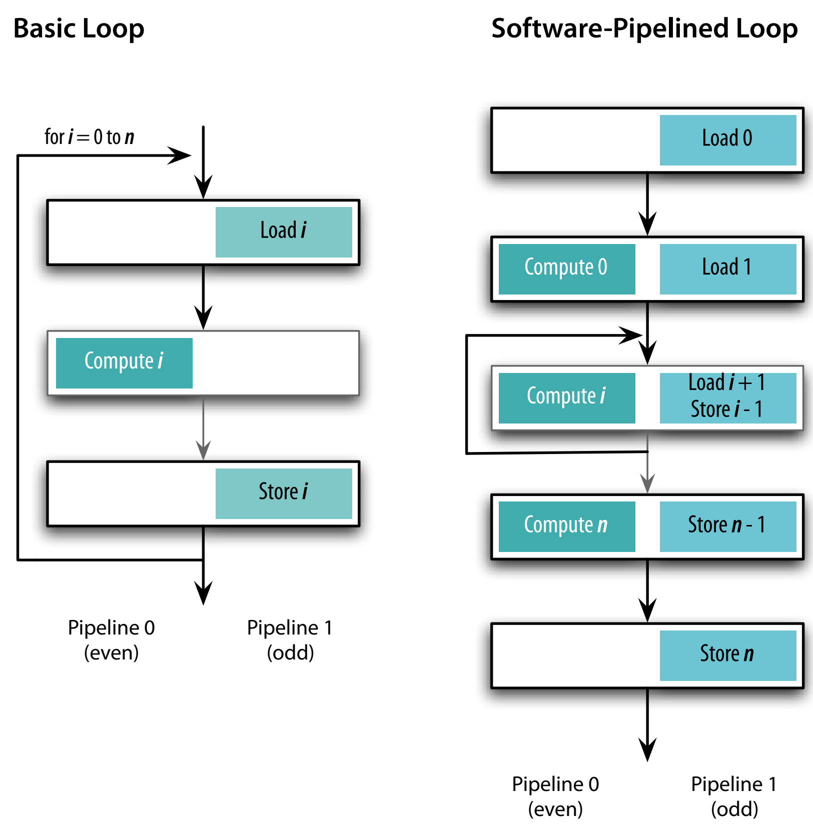

Figure 6.5: *Software pipelining.*

SPE, the VMX resides on top of cache hierarchy, not its private local store, and the data traffic through the cache levels will be limiting the performance (the memory wall). This effect will be aggravated by the tiny size of the VMX register file of 32 versus the mighty SPE with 128 registers. Your number crunching code will be subjected to OS jitter and similar phenomena absent on the SPE.

Leave the PPE free from computational tasks and utilize it to your benefit for handling the MPI communication between the PlayStations. It is almost like having a dedicated communication processor.

- **Avoid the operating system.** Maintain processor control and avoid passing it to the operating system by using convenient OS facilities. The operating system code may not necessarily be optimized for performance, and you have to keep in mind that the PPE is two orders of magnitude weaker than the combined power of the SPEs. Usually innocent tasks can introduce unpleasant overheads.

- **Avoid TLB misses - use huge TLB pages.** You pass the control to the OS when incurring a TLB miss. On the SPE, a TLB miss will be caused by a DMA transfer involving an effective memory address of a page not present in its TLB. For those users who hope to get close to the peak performance, we recommend the use of huge TLB pages. With huge pages and for the memory sizes offered by today's hardware, once touched, the address will never be paged-out from the TLB. It should be also pointed out that local store to local store transfers are performed using effective addresses and are subject to TLB misses as well.

- **Avoid UNIX timers.** Other operating system tools not to be trusted are standard Unix time facilities. We highly recommend that you use SPE decrementer to measure time. It is extremely straightforward to use, under direct control of the user, and has just about perfect granularity for most computational tasks. One thing to be pointed out is that the decrementer ticks with different frequencies on a CELL blade (around 14 MHz) and on a PlayStation 3 (around 80 MHz).

- **Do not debug using printf() in the SPU code.** The text will travel to the screen by a DMA transfer and potentially introduce false synchronization that alters your program behavior. Additionally, the code of the standard C library will have to be linked into your SPE code, consuming plenty of local store and leaving little space for your code.

- **Be ready for compiler oddities.** The compilers for the CELL processor have not fully matured yet. In the earlier versions, serious bugs were quite common. Although the situation has improved, it is still likely to encounter issues. In particular, throughout the text we recommend practices like mixing scalar and vector code, the use of explicit type casts, and explicit pointer arithmetic on scalar and vector pointers - techniques which we experienced that choke compilers. Fortunately, there are two compilers available, the XLC and the GCC, and when one has problems with a particular piece of code, the quick workaround is to try the other.

  The same applies to performance. There can be serious performance differences between XLC and GCC. There can also be serious performance differences between compilers in different releases of the SDK, unfortunately often meaning worse performance in the newer versions. Those who do not have access to the older versions can hope for the situation to improve in the future. Also, one should not blindly believe that compiling with the **-O3** flag will always deliver the best results. Quite often, **-Os** (optimization for object code size) performs better.

- **Use Unholy Practices.** Forget what they taught you in freshmen programming courses, and throw political correctness out the window when getting up to speed on the CELL. You want to write code that is well structured, readable, and maintainable, but do yourself a favor and free yourself from the burden of thinking about portability at this time. Portability is a feature of well established programming models defined by standardized APIs, which are in their infancy on the CELL processor.

In particular:

- **Use explicit type casts.** Save yourself the effort of creating unions to operate on array elements in a vector as well as scalar way. Explicitly cast vector and scalar types to each other. Do not be intimidated by casting addresses (pointers) in 32-bit mode to unsigned integers. They are both just numbers. That is why you can pass them around in mailboxes. The CELL architecture keeps you conscious about type sizes anyway by enforcing alignment and size restrictions on DMA communication.
- **Use pointer arithmetic.** Pointer arithmetic is a common practice when programming for performance and indispensable with aggressive loop unrolling. Pay attention to the type when incrementing a pointer. Incrementing a vector pointer by one moves it forward by a number of scalar elements (e.g., four floats).

6.2 PlayStation 3 Cluster

Distributed memory programming on a cluster of CELL/PS3 is not conceptually different than programming for any other cluster. MPI is always the method of choice for the SPMD programming paradigm. The most important difference comes from the fact that, locally, computations are offloaded to the SPEs. In order to take advantage of this situation, it is worth making a distinction between two cases:

1. The PPE is absolutely not involved in local computations: this is usually the case where elementary operations are submitted to the SPEs.
2. The PPE is involved in local computations: this is the case where, for example, complex local computations have to be performed, and the PPE has to schedule the elementary tasks that compose the operation.

In both cases, it is possible to overlap MPI communications with local computations but different approaches have to be used.

Let us start with the easiest case where PPE is not involved. In this case the PPE is idle while the SPEs carry on the local computations and thus can take care of performing the MPI communications. Take as an example the simple code in Figure 6.6. This code contains a simple loop where some data is copied inside a buffer by means of the `copy_into_buf`. Then a communication is performed (the `MPI_comm` routine denotes a general MPI communication routine), once the data has been sent/received, the local computations are started on the SPEs with the `start_SPEs` routine; the last step of the loop consists of waiting until the SPE computations are completed in the `wait_SPEs_compl` routine.

```
...
for(i=0; i<n; i++){

   copy_into_buf(buf, data[i]);

   MPI_comm(buf,...);

   start_SPEs(buf,...);

   wait_SPEs_compl();

}
...
```

Figure 6.6: *MPI example code.*

Overlapping communications and computations in this case can be done using a method that is conceptually equivalent to applying the double-buffering technique (well known to CELL programmers) at a higher level. In order to achieve this overlapping we need to take advantage of MPI non-blocking communications and double the number of buffers needed for the communications. The general idea is that, while the SPEs carry on the computations related to step N of the loop, the PPE performs the communications related to step $N+1$. The code in Figure 6.6 is transformed into the code in Figure 6.7

In the case where the PPE is involved in local computations, it is still possible to (partially) hide MPI communications. In fact, it is possible to take advantage of the fact that the PPE is a two-way, hyperthreaded processor. It is thus necessary to create two threads on the PPE, one that exclusively manages MPI communications while the other handles the local computations.

```
...
copy_into_buf(buf1, data[0]);

MPI_Icomm(buf1,...);

for(i=0; i<n-1; i++){

   copy_into_buf(buf2, data[i+1]);

   MPI_Icomm(buf2,...);

   start_SPEs(buf1,...);

   MPI_wait_on_buf(buf2,...);

   wait_SPEs_compl();

   swap_buffers(buf1, buf2);

}

start_SPEs(buf1,...);
wait_SPEs_compl();
...
```

Figure 6.7: *MPI example code with communication/computation overlapping.*

CHAPTER 7

Programming Models

The basic taxonomy for the CELL programming models was introduced by Kahle et at. [9]. Six models were distinguished, some of which can be qualified as PPE-centric and some as SPE-centric:

Function offload model is one where the main application executes on the PPE and offloads performance-critical functions to the SPE by using calls to a library, possibly provided by a third party.

Device extension model is a form of the function offload model, where the SPE provides services previously delivered by a device or acts as an intelligent front-end for a device.

Computational acceleration model is an SPE-centric model, where standard parallel programming techniques are used to implement most computationally intensive sections of code on the SPEs and the PPE acts mostly as a system service facility. Both shared and distributed memory programming techniques apply here.

Streaming model is one where the SPEs are arranged in a pipeline, where each of them applies a particular computational kernel to the data that passes through it. The model is very attractive due to the fact that the internal bandwidth greatly exceeds the main memory bandwidth. Load balancing may become an issues if the pipeline stages do not have a near equal amount of work.

Shared memory multiprocessor model can be utilized thanks to the DMA cache coherency capabilities. A conventional shared memory store is replaced by a combination of a store to the local store and a DMA to shared memory with the PPE and all SPEs assigned to the same address space. Atomic update primitives can be used by utilizing the DMA lock line commands.

Asymmetric thread runtime model is extremely flexible and widespread on conventional SMPs. On the SPEs, however, it would be very costly to implement full preemptive task switching, and some other model would have to be implemented, e.g., FIFO run-to-completion.

Aside from this taxonomy, it is important to notice that the advent of a multi-core processor brings different communities together. In particular, the CELL processor seems to ignite similar enthusiasm in both the HPC/scientific community and the DSP/embedded community. By the same token, the world of programming techniques proposed for the CELL is as diverse as the involved communities and includes shared-memory models, distributed memory models, and stream processing models, to name the most prominent ones. In this chapter, we present a brief overview of a few emerging frameworks for programming the CELL processor.

7.1 Octopiler

The Octopiler [10] is a code name for a compiler research project at IBM (`http://www.research.ibm.com/cellcompiler/`) targeted for the CELL processor. The two most important features of the Octopiler is automatic SIMD'ization of code and automatic user-directed exploitation of shared-memory parallelism. The compiler is capable of generating, within single compilation, the code for both the PPE and the SPEs. The PPE path includes VMX support and tuning for the PPE pipeline. The SPE path supports specific architectural features of the SPEs, including automatic exploitation of their vector architecture.

The compiler employs established methods in the area of automatic SIMD'ization. The *unroll-and-pack* and *loop-based* approaches are combined in order to extract both SIMD parallelism within basic blocks (e.g., manually unrolled loops), as well as between loop iterations (by using *loop blocking*). The compiler also attempts to minimize data reorganization due to compile-time or run-time data misalignment. Strong emphasis is put on efficient instruction scheduling with the goal of minimizing the length of the critical path, and the technique of bundling is utilized to satisfy constraints to enable dual issue. Special attention is devoted to specific architectural constraints of the SPEs. One example is the prevention of the instruction fetch starvation due to the fact that a single local memory port is shared between memory instructions and instruction fetch. Another example can be efficient branch hinting with constraints on the distance between the hint and the branch.

The compiler relies on the existing infrastructure of the IBM XL compiler and includes a high-level optimizer called the Toronto Portable Optimizer (TPO), which applies both intraprocedural optimizations and interprocedural optimizations. Automatic parallelization is based on the OpenMP programming model, which provides the programmer with the abstraction of single shared-memory address space. OpenMP directives can be used to identify parallel code regions, and the compiler takes care of producing code sections for the PPE and the SPEs. The PPE executes the OpenMP master thread, which uses the runtime library to distribute the work to the SPEs. The SPEs use DMAs to fetch tasks from a queue. The compiler-controlled software cache mechanism is utilized to improve performance of the single shared-memory abstraction. Automatic code partitioning and the mechanism of overlays are employed collaboratively by the compiler and the runtime environment to reduce the impact of local store limitations.

7.2 RapidMind

Both the RapidMind [11, 12] and PeakStream [13] (see next section 7.3) development environments provide extensions to common programming languages like C or C++ for the development of high performance applications that can be run on multi-core processors like the CELL Broadband Engine, GPUs, etc. The source code that makes use of the provided APIs is translated by means of a Just-in-Time compiler into a parallel program that is capable of exploiting the multiple execution units present on the target host.

RapidMind provides a software development platform that allows the developer to use standard C++ programming to create high performance and massively parallel applications that run on the GPU, the CELL BE and multi-core CPUs. The RapidMind platform acts as an embedded programming language inside C++. It is built around a small set of types that can be used to capture and specify arbitrary computations. Arbitrary functions, including control flow, can be specified dynamically. Parallel execution is primarily invoked by applying these functions to arrays, which generates new arrays. Access patterns on arrays allow data to be collected and redistributed. Collective operations, such as scatter, gather, and programmable reduction, support other standard parallel communication patterns and complete the programming model.

The RapidMind platform includes an extensive runtime component as well as interface and dynamic compilation components. The runtime component automates common tasks such as task queuing, data streaming, data transfer, synchronization, and load-balancing. It asynchronously manages tasks executing on remote processors and manages data transfers to and from distributed memory. This runtime component provides a framework for efficient parallel execution of the computation specified by the main program.

7.3 PeakStream

PeakStream [13] is a comprehensive application development platform designed to maximize the floating point power of multi-core processors, such as the CELL Broadband Engine. It consists of four major components: the PeakStream APIs, the PeakStream VM, the PeakStream Profiler and the PeakStream Debugger. The application is coded to use the PeakStream APIs and linked against the PeakStream Virtual Machine (VM) libraries. The libraries handle all of the interaction details with the processor. When the application uses the PeakStream APIs to perform mathematical operations, e.g., addition or use of math library functions like *exp*, those API calls are processed by the Virtual Machine. The Virtual Machine then creates optimized parallel kernels that are executed on the processor on a Just-in-Time (JIT) basis. The application must use explicit I/O calls (read and write) to move data into and out of the VM. Use of arrays as the fundamental data type in the PeakStream Platform, coupled with dynamic translation of programs, has the effect of decoupling the application programming model from the programming model of the processor being used. The VM has detailed knowledge of the specific processor being used (GPU/CELL/CPU). It performs optimizations necessary to make the application's array-based code perform well, providing results with adequate precision and accuracy.

The PeakStream Platform supports use of the C and C++ languages for application development. Language bindings to platform operations are provided by a set of header files and shared libraries.

7.4 MPI Microtask

The MPI Microtask [14] is a research project at IBM. The approach is based on the fact that the MPI message passing model expresses parallelism (both data and pipeline parallelism) through messaging and allows identification of tasks and their dependencies by analyzing communication patterns. The system consists of a preprocessor and a runtime environment, where the preprocessor responsibilities include task graph generation, task clustering, and scheduling of clusters of tasks. At the same time, the runtime environment takes care of synchronization, context switching and message buffer management. The model allows the preprocessor and runtime system to optimize the scheduling of computation and communication by taking advantage of the communication explicitly expressed in the code.

The MPI Microtask approach eliminates the burden of managing the local store and only requires the programmer to partition the application into a collection of small microtasks that fit into the local store. A microtask can be compared to a *virtual SPE*, which uses the MPI to communicate with other microtasks. The execution of microtasks is optimized by exploiting the explicit communication of the MPI model. A special, supportive microtask runs on the PPE, where code size restrictions are nonexistent, and calls supportive OS services, like I/O operations. The dynamic process model of

MPI-2 is adopted, where the supportive microtask spawns smaller microtasks in a recursive process, until the microtasks fit into the local store. MPI communicators are used to identify communication contexts.

The model also introduces the concept of a *basic task*, which is a unit of computation that causes communication only at its beginning and end, and corresponds to a computation kernel in stream programming languages. Basic task is analogous to the concept of a *basic block*, which is a portion of code with no jump or jump target in the middle. It is the preprocessor's responsibility to divide each microtask into a collection of basic tasks and group basic tasks with strong dependencies together in clusters in an attempt to minimize the number of context switches. The system implements a static scheduling algorithm with a dynamic programming method. A task precedence graph is constructed in the form of a direct acyclic graph (DAG), where nodes represent basic tasks and edges correspond to pairs of sends and receives. The scheduler identifies basic tasks that can be executed independently (in parallel) by examining their dependencies and then applies various optimizations.

7.5 Cell Superscalar

Cell Superscalar (CellSs) [15] is a framework being developed at the Barcelona Supercomputer Center (http://www.bsc.es/cellsuperscalar) and is a derivative of a former project, Grid Superscalar, targeted for Grid computing and a precursor for a companion project, SMP Superscalar aimed at generic, multi-core architectures.

CellSs employs techniques borrowed from the field of computer architecture, such as data dependency analysis, data renaming and locality exploitation, to perform automatic parallelization of sequential code at the time of execution. The programmer writes sequential code, where SPE functions are annotated with compiler directives (pragmas - somewhat similar to OpenMP), which identify input and output arguments. The execution environment consists of a source-to-source compiler and a runtime library. Execution is based on a data dependency graph, where each node represents an instance of an SPE function and edges denote data dependencies. At runtime, the PPE takes care of task generation by completely unrolling the loops in the code and dynamic scheduling of the tasks. Parallelism is exploited by scheduling independent nodes to different SPEs at the same time. Dependent tasks are scheduled to the same SPE to facilitate data reuse. Priority hints can be used to help identify the critical path for more efficient scheduling and tasks are scheduled in batches to enable communication overlapping by double buffering.

The CellSs tool does not perform automatic SIMD'ization of the SPE code. Other tools have to be used for this purpose, like the Octopiler. Manual SIMD'ization is another option. CellSs also enables for collection of execution traces and their visualization using the Paraver package.

7.6 The Sequoia Language

Sequoia [16] is a programming language whose goal is portable programs that can efficiently use various memory hierarchies. Program execution is prescribed in the form of tasks that are to be implemented as functions. Parallel execution happens at special loop constructs that map the execution to smaller tasks. A task is associated with a node of the memory hierarchy tree. And invoking a task makes a copy of data between levels of the hierarchy, which is the only way that tasks communicate with each other. The memory hierarchy can be presented to the Sequoia compiler as having virtual levels not present in the actual hardware. This allows adding additional channels of communication between tasks. Such could be the case for distributed memory systems, which need explicit message passing to communicate data. Adding a virtual global shared memory in the Sequoia's hierarchy allows generation of the runtime code that explicitly exchanges messages rather than make local memory copies.

Sequoia has an implementation for the IBM CELL blade. The CELL hardware is presented to the Sequoia compiler as a three-level hierarchy with main memory, local stores and the register file. The computational kernels such as **SGEMM**, **FFT3D**, **GRAVITY**, and **HMMER** scale relatively well with the number of SPEs. And bandwidth intensive kernels such as **SAXPY** and **SGEMV** scale with the available bandwidth. The CELL implementation of these kernels uses the SPE intrinsincs for efficient vectorization.

7.7 Mercury Multi-Core Framework

The Mercury Multi-Core Framework (MCF) library [17] is an API for programming the CELL processor using the function offload engine (FOE), with emphasis on exploiting data parallelism. The PPE plays the role of the *manager* responsible for running the control plane code, while the SPEs are the *workers* responsible for running the processing plane code. The SPEs run a tiny 12KB kernel, which facilitates synchronization, communication, shared allocations, and quick loading of worker tasks. The processing units are organized into a network represented by a network handle and consist of one manager and a set of workers. The network handle is a container for tasks, plugins, messages, queues, allocations, teams, and synchronization objects. The members of the network can be partitioned into teams, represented by a team handle. An MCF team of workers can be directed by the manager to perform a *task*. A worker can only perform a task upon the manager command. Form the manager standpoint, tasks are non-blocking. After launching a task, the manager is free to perform other activities while the workers perform their assigned tasks. Workers, however, run tasks to completion, and each worker may perform only one task at a time. Plugins, on the other hand, are functions that cat be dynamically loaded, run, and unloaded by a worker without any interaction from the manager. Also, plugins can be loaded in a non-blocking fashion, which allows for communication and computation overlapping for code in the same way as it is usually applied to data. In other words, the technique of double buffering can be applied to code as well. Workers can be synchronized using barriers or semaphores. Bulk data transfers are facilitated by the DMA API, similar to the one in the standard SDK. MCF also provides mailbox messages and message queues, implementing two-sided communication similar to the MPI model. An interesting concept in the library is the channel abstraction, which significantly simplifies DMA communication by relying on simple *put* and *get* operations and automatically takes care of tiling the data in any number of dimensions as well as facilitating double buffering at the same time. MCF includes a classic histogram profiling mechanism for collecting execution statistics. It also facilitates collection of execution traces, which can be visualized using Mercury's tool, TATL.

CHAPTER 8

Application Examples

8.1 CELL Processor

8.1.1 Dense Linear Algebra

A fundamental capability of computers is to solve dense systems of linear equations. Also, it is the established way of benchmarking most powerful computers, which are currently ranked on the TOP500 list (http://www.top500.org/) according to their score on the Linpack benchmark [18], a linear system solver based on Gaussian elimination with partial pivoting. Unfortunately for the CELL processor, Linpack is defined in double precision, where the CELL delivers respectable but not astonishing performance. Results for the initial implementation of the Linpack benchmark in double precision were reported by Chen et al. [19]. For a matrix of size 2Kx2K they achieved 11.05 Gflop/s, which is around 75% of the double precision peak. They have also implemented a single precision version of the code, which achieved 155 Gflop/s (again around 75%efficiency) for a matrix of size 4Kx4K. Unfortunately, a single precision algorithm does not legitimately implement the Linpack benchmark,. This is where the idea of iterative refinement comes into play.

Iterative refinement is a well known method for improving the solution of a linear system of equations of the form $Ax = b$. First, the coefficient matrix A is factorized using LU decomposition into the product of a lower triangular matrix L and an upper triangular matrix U. Partial row pivoting is used to maintain numerical stability resulting in the factorization $PA = LU$, where P is the row permutation matrix. The system is solved by solving $Ly = Pb$ (*forward substitution*) and then solving $Ux = y$ (*backward substitution*). Due to the roundoff error, the solution carries an error related to the condition number of the coefficient matrix A. In order to improve the computed solution, an iterative refinement process is applied, which produces a correction to the computed solution, x, at each iteration, which then yields the basic iterative refinement algorithm (Algorithm 1).

Algorithm 1 Iterative refinement of the solution of a system of linear equations using Matlab™notation.

```
repeat
  $r = b - Ax$
  $z = L\backslash(U\backslash Pr)$
  $x = x + z$
until $x$ is accurate enough
```

The algorithm can be modified to use a mixed-precision approach. The factorization $PA = LU$ and the solution of the triangular systems $Ly = Pb$ and $Ux = y$ are computed using single precision arithmetic. The residual calculation and the update of the solution are computed using double precision arithmetic and the original double precision coefficients. The most computationally expensive operations, including the factorization of the coefficient matrix A and the forward and backward substitution, are performed using single precision arithmetic and take advantage of the single precision speed. The only operations executed in double precision are the residual calculation and the update of the solution. It can be observed that all operations of $O(n^3)$ computational complexity are handled in single precision, and all operations performed in double precision are of at most $O(n^2)$ complexity. The coefficient matrix A is converted to single precision for the LU factorization. At the same time, the original matrix in double precision is preserved for the residual calculation.

Our initial implementation of the mixed-precision Linpack benchmark [20] placed the CELL processor on the Linpack Report [21] with performance close to 100 Gflop/s (Figure 8.1). More recently, we have implemented a similar mixed-precision solver based on the Cholesky factorization for symmetric positive definite systems, where both the computation intensive steps of the factorization, as well as the memory intensive steps of the refinement, were much better optimized and achieved performance in excess of 155 Gflop/s (Figure 8.2).

8.1.2 Sparse Linear Algebra

The CELL processor is not best suited for sparse matrix operations. These operations are memory bound. This means that the computational power of the processor cannot be fully exploited. In this respect, the CELL is not different than any other processor since, on modern architectures, the bus bandwidth still represents a major bottleneck. This is the reason why it is not possible to

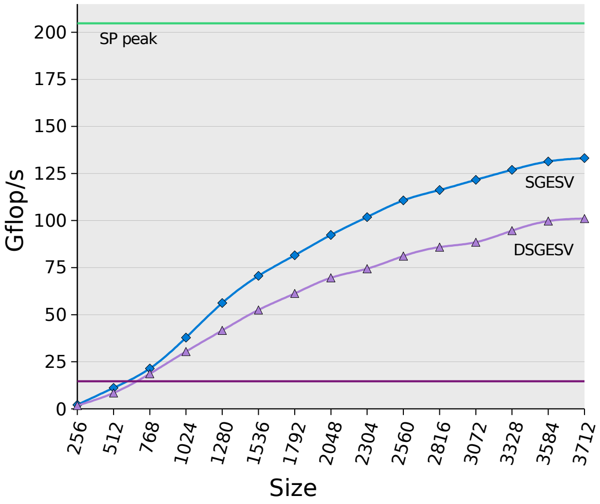

Figure 8.1: *Mixed-precision solver based on LU factorization.*

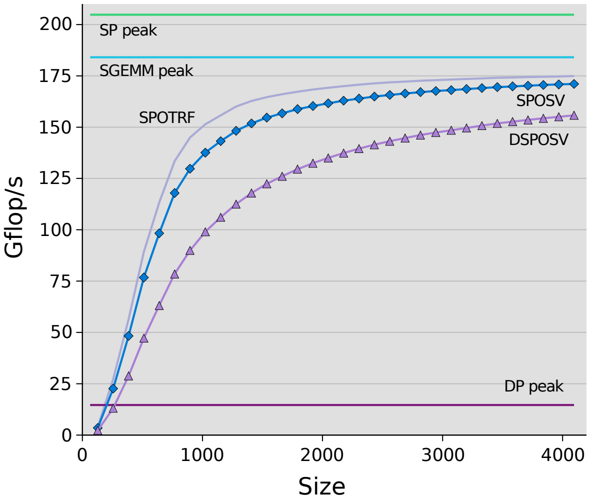

Figure 8.2: *Mixed-precision solver based on Cholesky factorization.*

achieve high performance even for BLAS-2 dense operations (though they are relatively much easier to implement).

The phenomenon that makes the dense operations run close to the peek floating point performance is the *surface-to-volume* effect. Typically, in case of a dense problem of size n, n^3 operations are performed on n^2 amount of data. If only the problem size is large enough, the communication will take less time than the computation, and can be completely hidden. Unfortunately, the *surface-*

to-volume effect does not apply to sparse operations.

Since, for sparse operations, communication is more expensive than computation, a perfectly optimized code will only run, at most, at the speed of the memory bus. In a sparse matrix-vector product operation $2 \cdot nnz$ floating-point operations are performed (where nnz is the number of nonzeroes in the matrix) and $nnz + O(n)$ values are transferred from memory to the processor (the $O(n)$ value comes from the movement of the source and destination vectors). Thus, assuming that the CELL bus speed is 25.6 GB/s and that a single precision floating-point value is four bytes, the upper-bound speed for this operation is:

$$perf = \frac{2 \cdot nnz}{(nnz + O(n)) \cdot 4/25.6} < 12.8 Gflop/s$$

This upper bound means that, at best, it is possible to achieve an efficiency of $12.8/204.8 \simeq 0.06 = 6\%$. For other *standard* architectures (like the x86 family), the efficiency can be 30% or more. It is, of course, a higher fraction of a much lower peak.

Even though 12.8 Gflop/s is a remarkable speed (higher than conventional processors), in practice the maximum possible performance can be much lower because, in general, the $O(n)$ term is not negligible at all. Moreover, it is very difficult to transfer and crunch the data at full speed because it can be very hard to apply the programming rules described in section 6.1. The main problems come from:

- **Vectorization:** It is a pretty hard task to vectorize operations on sparse data. Conventional approaches are the usage of block storage formats (like BCSR), diagonal storage formats (like JAD), or the use of techniques like the *segmented scan*. In the first case, there is substantial overhead due to the presence of fill-in elements that have to be introduced when the matrix does not have an intrinsic block structure. The diagonal storage formats and the segmented scan technique rely on the availability of gather operations in the ISA that are not included in the CELL instruction set.

- **Memory alignment:** The highest transfer rates are achieved on a CELL platform only when both the source (on the global storage) and destination (on the SPE local storage) addresses are 128-bit aligned and when the size of the transfer is a multiple of 128 bits. Because of the sparse nature of the data, and of the heavy use of indirect addressing, it is very hard to follow a regular access pattern to memory on aligned locations.

- **Double Buffering:** Due to the nature of the data, it is not possible to know a priori how much data will be moved from the global memory to the SPE local memory at each step of an algorithm/operation, and in the general case this quantity will be different at every step. This may result in poor overlapping between communications and computations.

- **Unrolling:** In the general case, it is not possible to know the length of loops, which makes it hard to unroll them.

- **Reduction of the number of branches:** It is not always possible because of the use of indirect addressing.

There are some lucky cases, like tridiagonal matrices (see Figure 8.3), where it is possible to follow a very regular memory access pattern and, thus, reduce all the problems listed above. In this case, the $O(n)$ term is not negligible since $nnz \simeq 3n$. However, when implementing an algorithm like the Preconditioned Conjugate Gradient (PCG), it is possible to group operations in order to reuse the data that has been fetched into the SPEs local memories, which increases the ratio between floating point operations and data movement. Based on this approach, we developed a version of the PCG algorithm, with Jacobi preconditioner, for tri-diagonal matrices capable of running at around 6.6 Gflop/s.

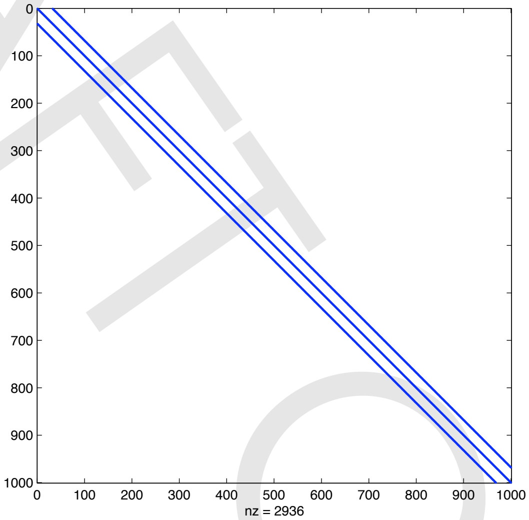

Figure 8.3: *A tridiagonal (sparse) matrix.*

8.1.3 Fast Fourier Transform

One way of looking at the CELL processor is to treat it as eight digital signal processors (DSP), augmented with a control processor, on a single chip. Along with the fact that the processing units are interconnected with an extremely powerful bus, the features make it ideal for signal processing applications, which mostly translates to fast Fourier transform (FFT) code.

An early implementation of an FFT algorithm was reported by IBM for a fixed size input data set of 16 million (2^{24}) single precision complex samples [22]. The algorithm used is a modified *stride-by-1* algorithm, proposed by David Bailey in 1986 and based on Stockham self-sorting FFT. The stride-by-1 approach allows for natural partitioning of the array without data rearrangement, which simplifies parallel execution by SIMD instructions. Performance relies on SIMD'ization and manual unrolling. The transposition needed in the algorithm is performed with a combination of shuffles and DMAs. Double buffering is used to hide communication latencies. The performance of 46.8 Gflop/s was reported using the standard $5N\log N$ metric. The code is distributed along with the SDK and resides in the location `file:///opt/ibm/cell-sdk/prototype/src/workloads/FFT16M/spu`.

Mercury Computer Systems investigated an implementation of an FFT algorithm of a fixed size of 64K complex elements [23, 24]. The size was chosen such that the data set does not fit in a single local store, but does fit in all the local stores combined. The algorithm is based on a classic 2D decomposition of a 1D FFT, where an N-element FFT is viewed as an 2D $NR \times NC$ matrix. NC NR-point column FFTs are followed by element-wise multiplication by an $NR \times NC$ matrix of *twiddle factors*, NR NC-point row FFTs, and a transposition of the $NR \times NC$ matrix to the $NC \times NR$ matrix.

Each SPE processes a 256 × 32 region with a SIMD-vectorized code. First, 32 column FFTs are performed. Then each 32 × 32 tile is multiplied by the twiddle factors and transposed, with the twiddle factors generated on the fly. Processing of the 32 × 32 tiles is pipelined, so that communication between SPEs is overlapped with computation. Then, 32 row FFTs are performed. At the end, the output dataset is assembled in the main memory, also subject to overlapping of processing and communication.

Using the $5N\log N$ metric, the theoretical model of the algorithm estimates performance of the algorithm for almost 60 Gflop/s in the sense of latency (single computation from the beginning to the end) and 100 Gflop/s for the throughput (consecutive execution of many repetitions). In an actual run, the throughput of 90 Gflop/s was achieved, which means around 30 times higher speed than can be achieved by a standard processor.

8.2 PlayStation 3 Cluster

8.2.1 The SUMMA Algorithm

The SUMMA [25, 26] algorithm for parallel matrix multiplication is very flexible and very effective. Since SUMMA can be defined as a block algorithm, high performance is achieved using BLAS-3 operations to carry out local computations (in particular using the `_GEMM` operation). Block-cyclic storage formats can be used with SUMMA, which allows a better exploitation of the fast BLAS-3 operations on each node. Moreover, communication and computation can be easily overlapped, providing even more performance and scalability. In the following discussion, we are assuming a non-cyclic but block storage for the matrices. In fact, even though the block storage is not a strict requirement on *standard* architectures (but is still recommended to achieve higher data locality and, thus, higher performance), it is almost mandatory on the CELL since non-block storage may result is quite poor performance.

The SUMMA algorithm proceeds through successive steps, where at step i the block column A_{*i} and the block row B_{i*} are multiplied in order to update C. The steps of the SUMMA algorithm are described in Algorithm 2.

Algorithm 2 SUMMA

```
 1: for $i = 1$ to $n/nb$ do
 2:   if I own $A_{*i}$ then
 3:     Copy $A_{*i}$ in buf1
 4:     Bcast buf1 to my proc row
 5:   end if
 6:   if I own $B_{i*}$ then
 7:     Copy $b_{i*}$ in buf2
 8:     Bcast buf2 to my proc column
 9:   end if
10:   $C = C + buf1 \times buf2$
11: end for
```

The performance of the SUMMA algorithm can be greatly improved by overlapping communications with computations. On each node (each PlayStation) all the computation is offloaded to the SPEs. By the same token, the PPE (otherwise idle) can be devoted entirely to handling the communication. An almost perfect overlap can be achieved by means of a simple *double buffering* where, the data for step $N+1$ of Algorithm 2 are broadcast in advance at step N.

Our implementation of the SUMMA algorithm offloads the local computation at step 10 of Algorithm 2 to the SPEs. The local computation scales almost linearly with the number of SPEs, and achieves more than 97% of the peak of all six SPEs. While the local _GEMMs for step k of the algorithm 2 are executed on the SPEs, the PPE handles the communication broadcasting the data needed for step $k+1$. This yields an almost perfect overlapping of communications and computations, what allows to hide the less expensive of the two phases.

Figure 8.2.1 shows part of the execution log for the SUMMA algorithm on a 2x2 processors grid for problem size $n = 6144$. The blue blocks show the MPI data exchange steps while the yellow blocks are the local _GEMM operations; arrows show the message flows in the processors' grid. From this figure it is possible to see that, thanks to the overlapping technique implemented, the less expensive of the communication and computation phases can be completely hidden. For the problem size represented in Figure 8.2.1, the local computations cost (i.e. the yellow blocks) can be hidden since communications are almost three times more expensive (see Figure 8.5 (*right*)).

Since the algorithm is very simple, it is possible to easily produce a very accurate performance

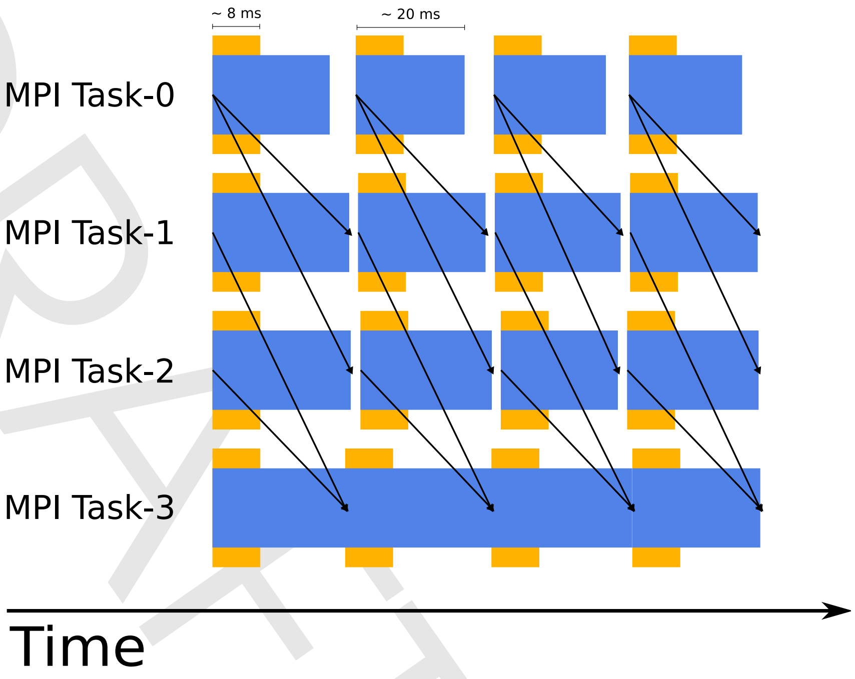

Figure 8.4: *Gantt chart of the execution of the SUMMA algorithm on a four PlayStation 3s using a* 2×2 *processor grid and six SPEs per node.*

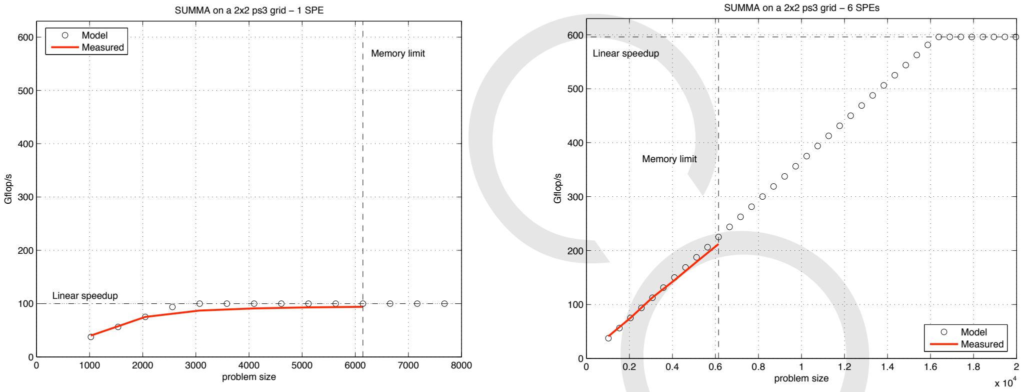

Figure 8.5: *The measured performance for the SUMMA algorithm on a* 2×2 *PlayStation 3 processor grid. A comparison is made with the results obtained from a theoretical model.*

model under the assumption that, if many decoupled communications are performed at the same time, they do not affect each other performance-wise. Figure 8.5 shows the experimental results obtained running our SUMMA implementation on a 2×2 PlayStation 3 processor grid. A comparison with the model that we developed is presented in the case where one SPE is used for the local computations (*left*) and in the case where all six available SPEs are used (*right*). The cost of the local computations is very small when all six SPEs are used; this means that the surface-to-volume effect only comes into play for very big problem sizes (see Figure 8.5 (*right*)). Due to the system limitations in memory (only 256 MB available on each node), it is not possible to have matrices bigger than 6144, which is pretty far from the point where linear speedup can be achieved (around 16000). When only one SPE is used on each node, the cost of the local computations is relatively much higher (a factor of six), and thus the surface-to-volume effect is reached at problem sizes within the limits of the system, as shown in Figure 8.5 (*left*). Since the CELL processor can perform double precision arithmetic at a much lower rate than single precision (a factor of 14), it is possible to achieve linear speedup within the system memory limits with a double precision SUMMA implementation. Figure 8.5 also shows that the model is very accurate.

8.3 Distributed Computing

8.3.1 Folding@Home

Folding@Home is a distributed computing project that simulates protein folding in order to understand and find cures for many well known diseases. One way of joining the Folding@Home effort is to upgrade the PS3 firmware to version 1.6. The upgrade creates a Folding@Home icon in the Network column of the PS3 menu. Clicking on the icon will setup the client and perform computations when the console is idle. Much more information about the project can be found on their website (http://folding.stanford.edu/). PS3 is a good fit for such a project. At the time of this writing PS3 consoles accounted for as much as 75% of the combined computational power (measured in Tflop/s) donated to the project.

Obviously, in the context of distributed computing, a cluster of consoles is better than a single console as far as computational power is concerned. However, it can become a problem if not automated. The Folding@Home website currently does not provide an automated way of enabling the computation on the entire cluster with a single step – each console in the cluster has to be enabled manually. Also, the Folding@Home client for PS3 does not run on Linux.

CHAPTER 9

Summary

9.1 Limitations of the PS 3 for Scientific Computing

Although the PlayStation 3 seems to be an attractive solution for high performance computing due to its high peak performance and low cost (close to $4 per Gflop/s), there are many serious limitations that have to be taken into account. We list some of the difficulties in achieving high performance.

Main memory access rate. The problem applies to the CELL processor and frankly speaking most modern processors, and is due to the fact that execution units can generate floating point results at a speed that is much higher than the speed at which the memory can feed data to the execution units. The main memory of the CELL processor is capable of a 25.6 GB/s peak transfer rate, while each SPE has a peak performance of 25.6 Gflop/s in single precision. For a 4 byte long single precision floating point value, a single SPE has to perform four floating point operations in order to hide the communication. If six SPEs operate concurrently, $4 \times 6 = 24$ operations have to be performed on each single precision value in order to hide the communication. As explained in section 8.1.2, the ratio cannot be achieved for sparse operations, which can be performed, at best, at the speed of the memory, which results in the efficiency of only 12.5% of the theoretical peak.

Network interconnect speed. The PlayStation 3 is equipped with a GigaBit Ethernet network interface. The capacity of this interconnect is out of balance if compared with the theoretical peak performance of each node. Let us assume that: the theoretical peak performance (153.6 Gflop/s in single precision using the 6 available SPEs) can be achieved on each node, the full network bandwidth (1 Gbit/s = 0.125 GB/s) can be achieved between the nodes, and it is possible to completely overlap communication with computations. Linear scaling can be obtained when local computations are more expensive than network communications. Under these assumptions, this will only happen when:

$$n = \frac{153.6 * 4}{0.125} = 4915.2$$

floating point operations are performed on every single precision word (four bytes of data) that is sent/received by a node. This limitation is further exacerbated by the fact that only a poor fraction of the bandwidth can be achieved. See section 8.2.1 for a practical example. The bottom line is that only computationally intensive applications can benefit from connecting multiple PS3s together to form a cluster. Computation, even as floating point intensive as dense matrix multiply, cannot be effectively parallelized over many PS3s.

Main memory size. The PlayStation 3 is equipped with only 256 MB of main memory. This represents a serious limitation when combined with the slowness of the network interconnect. As an example, consider the SUMMA parallel algorithm for dense matrix-matrix multiplication described in section 8.2.1. If n^2 is the number of matrix elements, then n^3 operations are performed. It turns out that, in order to achieve linear speedup, the size of the local matrix on each node has to be at least 4915.6 (see previous bullet). In perfect circumstances this would require around 288 MB of memory per node (see section 8.2.1 for details).

Double precision performance. Peak performance of double precision floating point arithmetic is a factor of 14 below the peak performance of single precision. Computations which demand full precision accuracy will see a peak performance of only 14 Gflop/s, unless mixed-precision approaches can be applied.

Programing paradigm. One of the most attractive features of the CELL processor is its simple architecture and the fact that all its functionalities can be fully controlled by the programmer. In most other *common* processors performance can only be obtained with a good exploitation of cache memories whose behavior can be controlled by the programmer only indirectly. As a result, the performance of applications developed for the CELL architecture is very predictable and straightforward to model; it is possible to get very close to the peak performance of the processor, much more than what can be done on cache based processors. This high performance and predictability, however, comes at a cost. Writing efficient and fast code for the CELL processor is, in fact, a difficult task since it requires a deep knowledge of the processor architecture, of the development environment and some experience. High performance can only be achieved if very low level code is produced that is on the border line between high level languages and assembly. Besides requiring a lot of effort, source code developed for the CELL processor is not portable at all on other architectures. Even though higher level programming models have been proposed for the CELL processor (see section 7), it is only possible to achieve a small portion of the processor's performance relying on them.

The bottom line is that the PlayStation 3's attractiveness for scientific computations will depend on the application's characteristics. For many applications, the limitations noted above will render the PlayStation 3 ineffective for clustering and severely limited for many scientific computations.

9.2 CELL Processor Resources

9.3 Future

Bibliography

[1] IBM. *Cell Broadband Engine Programming Handbook, Version 1.0*, April 2006.

[2] IBM. *IBM Full-System Simulator User's Guide, Modeling Systems based on the Cell Broadband Engine Processort Version 2.0*, November 2006.

[3] IBM. *Performance Analysis with the IBM Full-System Simulator, Modeling the Performance of the Cell Broadband Engine Processor, Version 2.0*, November 2006.

[4] IBM. *IBM Full-System Simulator Command Reference, Understanding and Applying Commands in the IBM Full-System Simulator Environment, Version 0.01*, October 2005.

[5] IBM. *Software Development Kit 2.0 Installation Guide, Version 2.0*, December 2006.

[6] IBM. *Cell Broadband Engine Programming Tutorial, Version 2.0*, December 2006.

[7] IBM. *C/C++ Language Extentions for Cell Broadband Engine Architecture, Version 2.2.1*, November 2006.

[8] IBM. *SPU Assembly Language Specification, Version 1.4*, October 2006.

[9] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. & Dev.*, 49(4/5):589–604, 2005.

[10] A. J. Eichenberger et al. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Sys. J.*, 45(1):59–84, 2006.

[11] RAPIDMIND. http://www.rapidmind.net/.

[12] RapidMind. Real-time ray tracing: CELL BE development made simple. http://www.rapidmind.net/pdfs/RapidMindRaytracer.pdf, 2006.

[13] PEAKSTREAM. http://www.peakstreaminc.com/.

[14] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Sys. J.*, 45(1):85–102, 2006.

[15] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[16] K. Fatahalian et al. Sequoia: Programming the memory hierarchy. In *Proceedings of SC2006*, Tampla, Florida, November 2006.

[17] M. Pepe. *Multi-Core Framework (MCF), Version 0.4.4*. Mercury Computer Systems, October 2006.

[18] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present and future. *Concurrency Computat.: Pract. Exper*, 15(9):803–820, 2003.

[19] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine architecture and its first implementation, A performance view. http://www-128.ibm.com/developerworks/power/library/pa-cellperf/, November 2005.

[20] J Kurzak and J. J. Dongarra. Implementation of mixed precision in solving systems of linear equations on the CELL processor. *Concurrency Computat.: Pract. Exper*, 2007. in press, DOI: 10.1002/cpe.1164, http://www3.interscience.wiley.com/cgi-bin/abstract/114046331/ABSTRACT, http://www.cs.utk.edu/~library/TechReports/2006/ut-cs-06-580.pdf.

[21] J. J. Dongarra. Performance of various computers using standard linear equations software, (Linpack benchmark report). Technical Report CS-89-85, Computer Science Department, University of Tennessee, 2006. www.netlib.org/benchmark/performance.ps.

[22] A. C. Chowg, G. C. Fossum, and D. A. Brokenshire. A programming example: Large FFT on the Cell Broadband Engine, 2005.

[23] J. Greene and R. Cooper. A parallel 64K complex FFT algorithm for the IBM/Sony/Toshiba Cell Broadband Engine processor. http://www.mc.com/literature/literature_files/GSPx-64KFFT-Paper-28Sep05.pdf, September 2005.

[24] J. Greene, M. Pepe, and R. Cooper. A parallel 64K complex FFT algorithm for the IBM/Sony/Toshiba Cell Broadband Engine processor. In *Summit on Software and Algorithms for the Cell Processor*. Innovative Computing Laboratory, University of Tennessee, October 2006. http://www.cs.utk.edu/~dongarra/cell2006/cell-slides/19-Luke-Cico.pdf.

[25] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM J. Res. Dev.*, 38(6):673–681, 1994.

[26] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency Computat.: Pract. Exper*, 9(4):255–274, 1997. http://www3.interscience.wiley.com/cgi-bin/abstract/13861/ABSTRACT, citeseer.ist.psu.edu/vandegeijn95summa.html.

APPENDIX A

Acronyms

BLAS Basic Linear Algebra Subprograms

CBE CELL Broadband Engine

CISC Complex Instruction Set Computer

DMA Direct Memory Access

EIB Element Interconnect Bus

Gflop/s Giga floating-point operations per second

GPR General Purpose Register

GPU Graphics Processing Unit

FFT Fast Fourier Transform

IEEE 754 IEEE Standard for Binary Floating-Point Arithmetic

ISA Instruction Set Architecture

MFC Memory Flow Controller

MMU Memory Management Unit

MPI Message Passing Interface

NFS Network File System

NIC Network Interface Card

NUMA NonUniform Memory Access

PPE Power Processor Element

PPU Power Processor Unit

PS3 PlayStation 3

PTE Page-Table Entry

RAM Random Access Memory

RISC Reduced Instruction Set Computer

SDK Software Development Kit

SMT Simultaneous MultiThreading

SPE Synergistic Processing Element

SPU Synergistic Processing Unit

SUMMA Scalable Universal Matrix Multiplication Algorithm

TLB Translation Look-aside Buffer

VLIW Very Long Instruction Word

XDR high-performance RAM from Rambus (also: eXternal Data Representation)