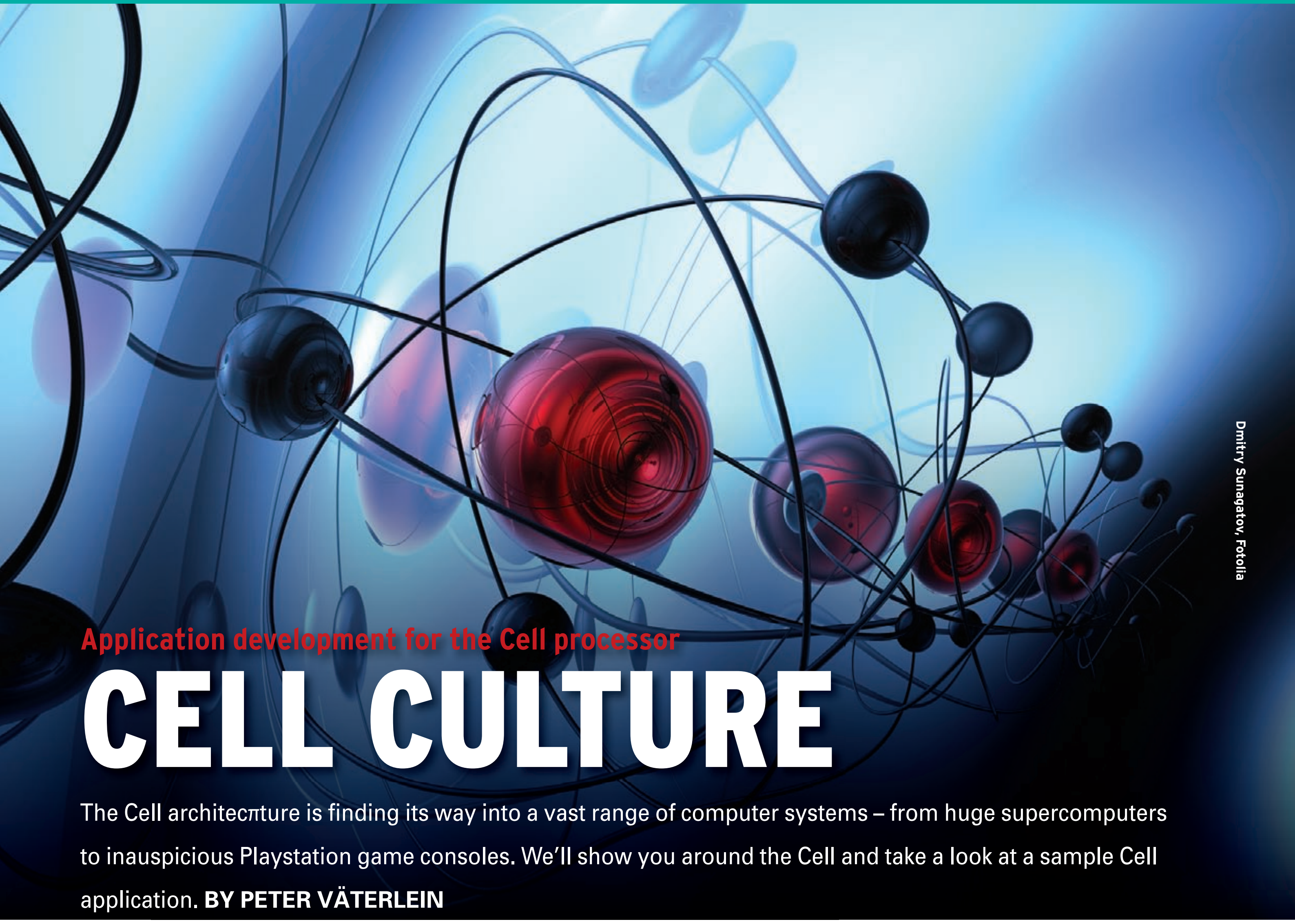


Application development for the Cell processor

CELL CULTURE

The Cell architecπture is finding its way into a vast range of computer systems – from huge supercomputers to inauspicious Playstation game consoles. We'll show you around the Cell and take a look at a sample Cell application. **BY PETER VÄTERLEIN**

Sony Computer Entertainment, Toshiba, and IBM started developing the innovative Cell Broadband Engine Architecture (CBEA) around 2001. The Cell architecture specializes in efficient processing of large data streams, such as the streams that occur in multimedia applications or computer games. The first implementation of the Cell architecture is the Cell Broadband Engine, also known as the Cell processor, which dates back to 2005 (Figure 1). Since it was introduced as the processor for the Sony PlayStation 3, the Cell CPU has attracted much attention. Although the Playstation (Figure 2) is certainly the most widespread application of the Cell architecture, the most spectacular application has to be the Roadrunner (Figure 3), which uses more than 12,000 Cell processors [1].

Cell blades are available from both IBM and Mercury Computer Systems. Mercury has even built a PCI Express card with a full-fledged Cell processor computer. Toshiba uses a variant of the Cell processor in its Qosmio notebooks.

In addition to its power and flexibility, the Cell is also known for energy efficiency. Cell-based systems currently hold the top seven spots in the Green 500 List [2] of the most energy-efficient supercomputers. In this article, I explore the Cell architecture and describe an example application that will help you get started with programming for the Cell.

The Cell computer specializes in handling problems that need a large amount of computer power but are easily split into separate tasks. The individual Cell processor cores then process these separate tasks in parallel.

The Cell processor consists of a conventional processor core (Power Processing Element, PPE) with 64-bit IBM Power Architecture and eight Synergistic Processing Elements (SPE; see Figure 4). Each of the eight SPEs has 256KB of local memory and a DMA controller (Memory Flow Controller, MFC). All nine processors are linked by a data bus (Element Interconnect Bus, EIB) to each other, the main memory, and the peripheral devices.

While the operating system on the PPE manages system resources, the SPEs handle algebraic operations. Their 128-bit registers either manipulate four 32-bit numbers per operation (short integers or single-precision floating points), or two 64-bit figures (long integers, or double-precision floating points). This SIMD architecture (Single Instruction, Multiple Data) is similar to the PC processor's MMX extension.

One special thing about the SPEs is that they only work with code and data stored in their local memory; they do not access main memory or peripherals. Applications must ensure that the right code and data are available locally. The data transfer operations between main and local SPE memory are organized by the SPE's DMA controllers and do not cause SPU overhead.

Developing for the Cell

Of course, developing applications for the Cell processor is more appealing for those who have access to a Cell-based machine. If you work on a Cell blade

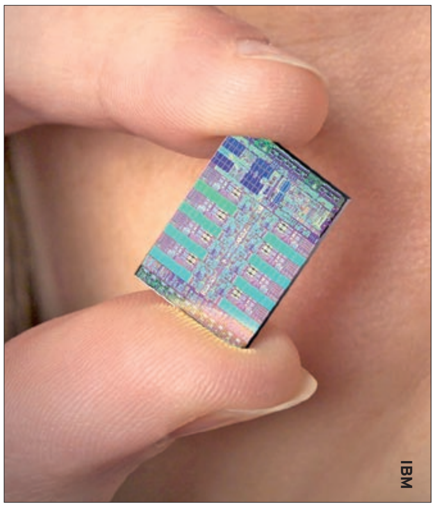


Figure 1: The Cell CPU is manufactured using a Silicon On Insulator (SOI) approach.

server, you will probably develop your applications directly on the Cell platform. If you have a Playstation 3, it makes more sense to use a Linux PC as your development platform. The Playstation doesn't have much in the way of RAM – just 256MB – and the low memory becomes fairly obvious when you work with an X11 interface.

IBM provides a free Software Development Kit (SDK) for the Cell architecture [3]. The Cell SDK will run on the x86, x86_64, and PowerPC platforms, as well as on Cell-based Linux machines. The latest version of the Cell SDK (3.1) supports Fedora 9 and Red Hat Enterprise Linux 5.2. The kit includes the *Developer* and *Extras* CD images and an RPM package with the installation script. Up to version 3.0, the Cell SDK for Fedora included a system simulator, which would let programmers test and optimize applications without physical Cell hardware. As of Version 3.1, the simulator is available separately from the IBM website [4]. The new Version 3.1 is still in beta, but it works perfectly on Fedora 9.

Linux on the Playstation

In contrast to other console manufacturers, Sony officially supports the installation of Linux on the Playstation, and you will find many howtos on the web [6]. There are two things to note about running Linux on the PS 3. First, direct access to the hardware is not supported; to protect its proprietary firmware, Sony added a virtualization layer. Second, only six of the Cell processor's eight SPEs are available to Linux programs.

According to IBM, the minimum hardware requirement is an Intel Pentium 4 with 2GHz clock speed or an AMD Socket F Opteron. On top of this, the SDK needs 1GB RAM and 5GB free disk space. To install the Cell SDK on Linux, you also need the *rsync*, *sed*, *TCL*, and *wget* packages. Because the installation script downloads various packages from the Barcelona Supercomputer Center [5], you will need continuous Internet access throughout the installation.

The *cell-install-3.1.0-0.0.-noarch.-rpm* RPM creates an */opt/cell* directory for the developer environment and documentation. The installation script expects the path to the CD images as an option:

```
/opt/cell/cellsdk ⮐
--iso path install
```

This variant has the advantage that you can install the content of both images in a single process. If you have the Cell SDK CD images on separate CDs, you need to insert the Developer and Extras CD one after another and launch the installation separately by typing */opt-/cell/cellsdk install*. If you have installed the system simulator, you can initialize it using the */opt/cell/cellsdk_sync_simulator* script, which installs some required SDK elements. The ISOs contain several libraries that are not open source. For installation documentation, check out */opt/cell/sdk/docs/install*.

π on the Cell

Applications for the Cell processor consist of at least two parts: a program that

Figure 2: The most popular application for the Cell processor is Sony's Playstation 3.

Figure 3: The Roadrunner supercomputer by IBM, which currently holds the number one spot in the Top 500 list, uses around 12000 units of the Cell chip.

runs on the Power PC core (PPE program), and at least one program that keeps the SPEs busy (SPE program). To allow the PPE program to control the SPE software, the PPE source code must include the *libspe2.h* header file from the Libspe2 library. The SPE program contains the actual calculating routines. An SPE program must include the *spu_intrinsics.h* and *spu_mfcio.h* header files for SIMD calculation functions and for communication with the PPE and the other SPEs.

The example program described in this article provides an approximation of π using the Shotgun algorithm (see the box titled "The Mathematical Shotgun"). The program expects command-line pa-

The Mathematical Shotgun

Several methods exist for calculating an approximate value for π. The Shotgun algorithm involves the computer calculating pairs of random numbers between 0 and 1 (Figure 5). Each pair represents a point in a square with an edge length of 1, where the bottom left corner has the coordinates (0,0) and the top right corner the coordinates (1,1).

Assuming that the dots are spread evenly across the square, the ratio between the number of dots that lie inside a circle of radius 1 and the total number of dots is approximately equal to the ratio between the areas of a quarter circle with a radius of 1 and a square with an edge length of 1, which is exactly π/4.

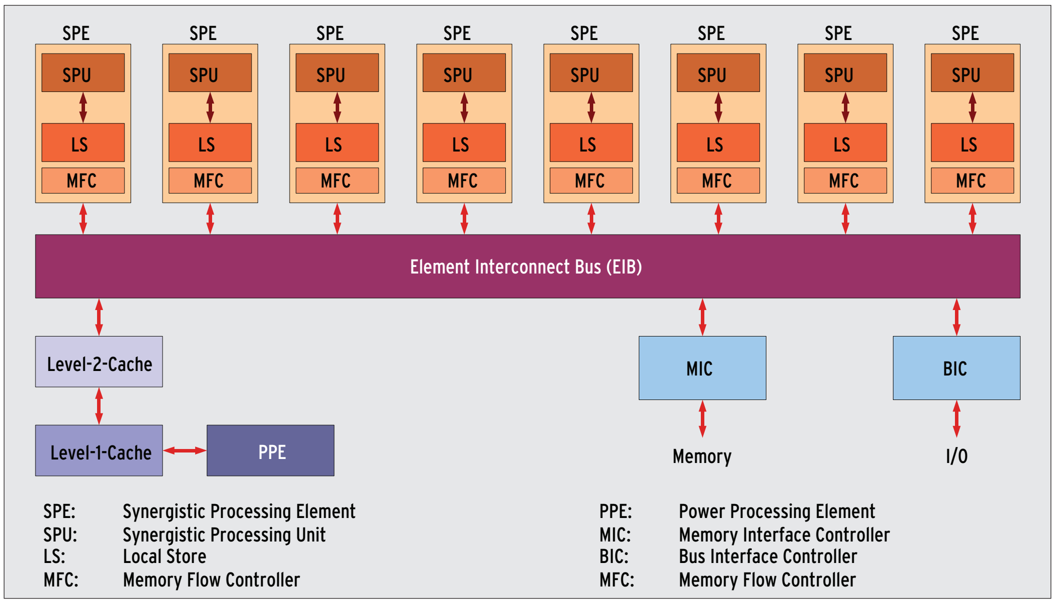

SPE
SPU
LS
MFC
Element Interconnect Bus (EIB)
Level-2-Cache
Level-1-Cache
PPE
MIC
BIC
Memory
I/O
SPE: Synergistic Processing Element
SPU: Synergistic Processing Unit
LS: Local Store
MFC: Memory Flow Controller
PPE: Power Processing Element
MIC: Memory Interface Controller
BIC: Bus Interface Controller
MFC: Memory Flow Controller

Figure 4: Each Synergistic Processing Element (SPE) has a Synergistic Processing Unit (SPU), local memory (LM), and a Memory Flow Controller (MFC). A Memory Interface Controller (MIC) sits in front of the main memory, and a Bus Interface Controller (BIC) is in front of the input/output interface.

rameters for the number of random pairs of figures to generate and the number of SPEs. After the *main()* routine in *pi_libspe_ppe.c* has parsed the command line for this information, the program dynamically allocates three memory areas. The first array stores a structure with the parameters that the PPE and SPE exchange for each SPE. The *spe_par_t* SPE structure type is declared in the *pi_libspe.h* header (Listing 1). The second array stores a structure with the SPE context for each SPE. This data contains everything the PPE needs to know about a program running on an SPE. The data type for this is declared in *libspe2.h*.

Addressing

The start addresses for variables that the PPE and SPE need to exchange later must be integer multiples of 16, or even integer multiples of 128 for best possible data transfer. Programmers can achieve this by using the *posix_memalign()* function instead of the conventional *malloc()*. The size of the individual blocks exchanged by the PPE and SPE also must be a multiple of 16. If inexplicable bus errors occur when you test the application, this is often a result of incorrect start address alignment or illegal block sizes in the data blocks transferred. The third array is only used internally by the PPE program and does not have to fulfill any special requirements with regard to start addresses or sizes.

Random

The PPE program contains a loop (Listing 2), which distributes the workload over the SPEs involved and sets a seed for creating pseudo random numbers from the current system time. To launch a program on an SPE, three steps are required. First, the *spe_context_create()* function (line 7) needs to create an SPE context. Second, the *spe_program_load()* function (line 8) needs to specify the program to execute; the programmer needs to declare the *spe_program_handle_t* variable in the PPE program header for this. This variable is always declared externally, that is, outside of the function. The name is identical to the name that the SPE program will be given later when you compile it.

The third step is for the *spe_context_run()* function to launch the program you want to execute. Normally, this function would block the PPE program while the SPE program is running, thus preventing any other SPE programs from launching parallel to it. A Posix thread helps to avoid this by executing the *spu_pthread()* function (line 10), which in turn launches an SPE program without interrupting the PPE program flow.

Now the SPE program needs to know where the parameters for the forthcoming calculations are located. Each SPE has a mailbox for incoming messages (four 32-bit words) and a mailbox for outgoing messages (one 32-bit word). Another mailbox triggers a software interrupt when data is available. In this case, the PPE program calls *spe_in_mbox_write()* (line 13) to pass in the start address of the array in which the parameters for the calculations are

Listing 1: Header File pi_libspe.h

```
#ifndef PI_LIBSPE_H_
#define PI_LIBSPE_H_

typedef struct {
        float value;
        uint64_t rounds;
        uint64_t seed;
        char reserved[4];
} spe_par_t;

#endif /*PI_LIBSPE_H_*/
```

Listing 2: For Loop for Controlling the SPEs

```
01 for ( i = 0; i < numspe; i++ ) {
02   spe_par[i].rounds = rounds /
   numspe;
03   gettimeofday( &tv, NULL );
04   spe_par[i].seed = tv.tv_sec *
   1000000 + tv.tv_usec;
05   spe_par[i].value = 0.0;
06
07   spe_ctx[i] = spe_context_
   create(0,NULL);
08   spe_program_load( spe_ctx[i], &pi_
   libspe_spe );
09
10   pthread_create( &spe_thread_
   handle[i], NULL, &spu_pthread, &spe_
   ctx[i] );
11
12   myaddr = (uint64_t) &spe_par[i];
13   spe_in_mbox_write( spe_ctx[i], (
   unsigned int * ) &myaddr, 2, SPE_
   MBOX_ANY_NONBLOCKING );
14 }
```

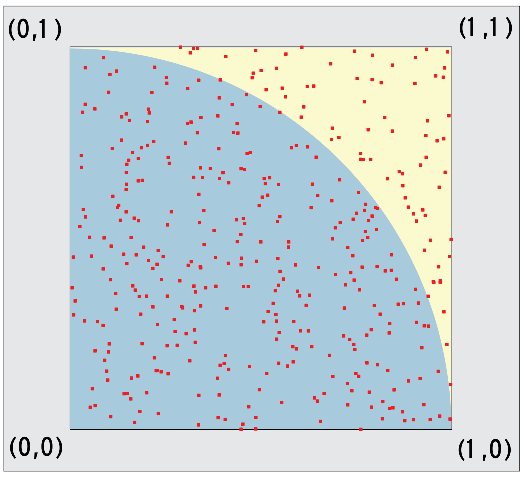


Figure 5: Approximating π with the Shotgun algorithm: each red dot represents a pair of random figures. If you count the dots in the blue circle and divide this number by the total number of dots, the result is an approximate value for π/4.

stored. The SPE context defines which SPE receives the message; its start address is the first function argument.

When all SPE programs have terminated, the PPE program releases the memory for the SPE context in question. Finally, the PPE program outputs the SPE's results on the console.

SPE Culture

The SPE's work starts with the *compute_pi()* function (Listing 3). *compute_pi()* expects a seed as an argument, which it will use to generate random numbers, and the number of pairs of numbers to calculate. The function returns an approximate value for π as a function value. To allow this to happen, the *main()* function (Listing 4) reads the main memory address at which the structure with the parameters for the current SPE program is located. This address is also referred to as an effective address.

Because the *spu_read_in_mbox()* function can only read single 32-bit words, it must be called twice to retrieve the full 64-bit address (lines 7 and 8). The variables declared inside the SPE program all lie within the SPE's local memory space. Pointers also reference memory addresses in the local memory. Because the Cell processor uses Big Endian architecture, the first word contains the higher value, and the second word contains the lower value bits.

Next, the SPE program must reserve a tag ID to distinguish DMA data transfers between main and local memory (line 10). An SPE can manage up to 32 tag IDs. Following this, the *spu_mfcdma64()* function transfers the parameter block that points to the main memory address previously retrieved from the mailbox to the *spe_par* variable in local memory (line 12). This function can handle both read and write DMA transfer. The sixth argument defines the transfer direction, as a comparison with line 18 shows.

The *spu_mfcdma64()* function does not wait for the memory transfer to complete. To ensure data integrity, the SPE program must wait until the DMA controller (Memory Flow Controller, MFC) has finished; the *mfc_read_tag_status_all()* (line 14) makes sure of this. The *mfc_write_tag_mask()* function (lines 19 and 20) tells us which of the 32 possible parallel DMA transfers it is waiting for.

Now the calculations can start, and the results, which are again stored in the *spe_par* structure, make their way back into main memory. Finally, line 22 releases the tag ID.

Instilling Life

Creating the object files is the next step. Because the PPE SPE processor cores use different instruction sets, two different compilers must be used to build the source:

```
/opt/cell/toolchain/bin/↩
spu-gcc -o ↩
pi_libspe_spe.spuo ↩
pi_libspe_spe.c
/opt/cell/toolchain/bin/↩
ppu-gcc -c pi_libspe_ppe.c
```

The .spuo suffix indicates an object file based on the SPE instruction set. To create a single executable, the *ppu-embedspu* tool converts the SPE program's object code into a format that the PPE can read:

```
/opt/cell/toolchain/bin/↩
ppu-embedspu pi_libspe_spe ↩
pi_libspe_spe.spuo ↩
pi_libspe_spe.o
```

The first parameter is the name used by the PPE to address the SPE program; it is identical to the name of the *spe_program_handle_t* type variable, which is declared in the *pi_libspe_ppe.c* source file.

The second parameter is the name of the file containing the SPE object code, and the third refers to the file where *ppu-embedspu* will write the PPE-readable object code. Finally, the developer must link the PPE and SPE programs with the *libspe2* library to create an executable:

```
/opt/cell/toolchain/bin/↩
ppu-gcc -o pi_libspe ↩
pi_libspe_ppe.o ↩
pi_libspe_spe.o -lspe2
```

If you have access to a computer with Cell hardware, you can simply copy the *pi_libspe* executable to it and execute the program. If you are using the simulator, you will need to take a small detour.

Simulated Entity

Before you can launch the Cell Full System Simulator, you must store the path to the simulator in the *SYSTEMSIM_TOP* environmental variable, which is */opt/ibm/systemsim-cell* by default.

The following command wakes up the simulator:

Listing 3: compute_pi Function

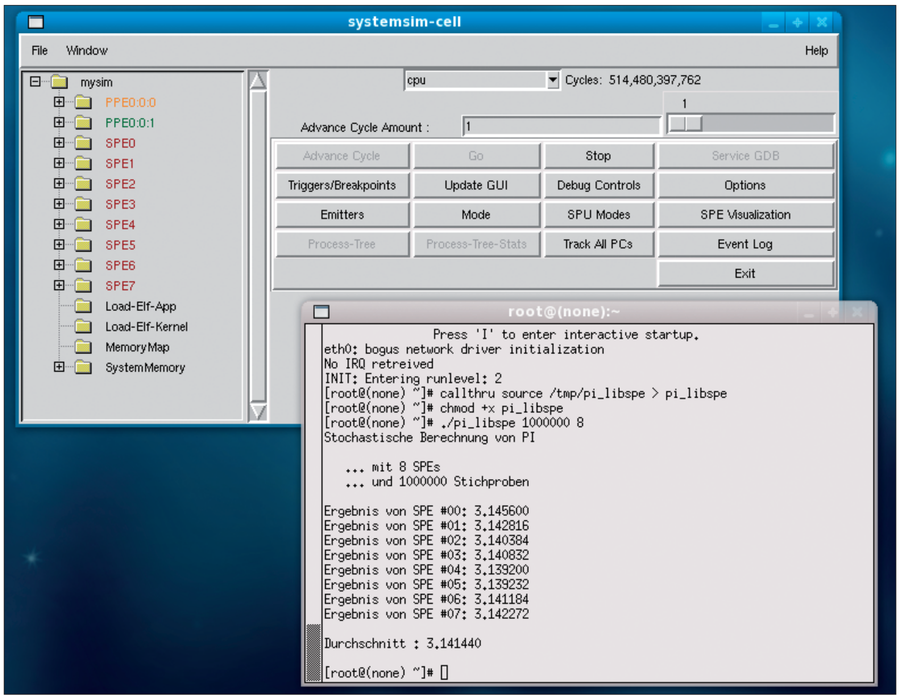
```
float compute_pi( long int seed,
  uint64_t rounds )
{
  uint64_t i;
  uint64_t in = 0;
  float x, y;
  unsigned long int h;

  srand48( seed );

  for ( i = 0; i < rounds; i++ ) {
    x = (float) lrand48()/RAND_MAX;
    y = (float) lrand48()/RAND_MAX;

    if (( x * x + y * y ) < 1.0 ) {
      in++;
    }
  }

  return ( float ) 4.0 * in / rounds;
}
```



```
                    Press 'I' to enter interactive startup.
eth0: bogus network driver initialization
No IRQ retreived
INIT: Entering runlevel: 2
[root@(none) ~]# callthru source /tmp/pi_libspe > pi_libspe
[root@(none) ~]# chmod +x pi_libspe
[root@(none) ~]# ./pi_libspe 1000000 8
Stochastische Berechnung von PI

   ... mit 8 SPEs
   ... und 1000000 Stichproben

Ergebnis von SPE #00: 3.145600
Ergebnis von SPE #01: 3.142816
Ergebnis von SPE #02: 3.140384
Ergebnis von SPE #03: 3.140832
Ergebnis von SPE #04: 3.139200
Ergebnis von SPE #05: 3.139232
Ergebnis von SPE #06: 3.141184
Ergebnis von SPE #07: 3.142272

Durchschnitt : 3.141440

[root@(none) ~]# 
```

Figure 6: The Cell Full System Simulator by IBM makes physical Cell hardware unnecessary.

```
/opt/ibm/systemsim-cell/bin/systemsim -g
```

The *-g* option launches a Tcl/Tk-based graphical interface (Figure 6). To see the various modes the simulator offers, press the *Mode* button. For a simple function test, *Fast Mode* is probably your best choice. Clicking *Go* launches the simulator. Now the console window will show you the operating system booting on the simulated Cell machine.

To load the program you want to run on the simulator, use the *callthru* command. If you run the command without any parameters, it will just show a help text. To import an executable file stored in the path */tmp/pi_libspe* on the physical machine, use the command:

```
callthru source /tmp/pi_libspe > pi_libspe
```

After modifying the permissions, as in *chmod u+x pi_libspe*, you can then finally launch the program:

```
./pi_libspe 1000000 8
```

Running the program tells the simulation machine to create a million pairs of random numbers using eight SPEs. The precision with which the result matches the accepted value of π depends on the quality of the pseudo-random numbers, but also on the number of attempts. The statistical error is approximately identical to the reciprocal value of the square root of the number of attempts. Given one-million attempts, the deviation between the approximated value and the actual value of π is about one thousandth, that is, about 0.003.

Another programming tool is the Data Communication and Synchronization (DaCS) library. Dacs abstracts a number of the Cell processor's special features, which means that it potentially could be ported to other accelerator architectures. In contrast to this, the Accelerator Library Framework (ALF) implements a programming model that swaps out individual functions to the SPEs. DaCS and ALF are included in the IBM developer environment.

The Multicore Application Runtime System (Mars) is an open source project spearheaded by Sony [7]. Mars installs miniature kernels on the SPEs, and the kernels autonomously manage the execution of programs on "their" SPEs. Released in November 2008, version 1.0.1 is available as either an RPM package or Tar archive. ■

INFO

[1] Top 500: *http://www.top500.org*

[2] Green 500: *http://www.green500.org*

[3] Cell SDK: *http://www.ibm.com/developerworks/power/cell*

[4] Cell system simulator: *http://www.alphaworks.ibm.com/tech/cellsystemsim*

[5] Barcelona Supercomputer Center: *http://www.bsc.es*

[6] Linux on the PS 3: *http://en.wikipedia.org/wiki/Linux_for_PlayStation_3*

[7] Mars software and documentation: *ftp://ftp.infradead.org/pub/Sony-PS3/mars*

THE AUTHOR

Professor Peter Väterlein teaches at the University of Esslingen's Faculty of Information Technology. His specialties are operating systems – preferably Linux – and parallel computating from multicore processors to grid computing. His homepage is *http://www.hs-esslingen.de/mitarbeiter/Peter.Vaeterlein*.

Listing 4: Main Function in the SPE Program

```
int main ()
{
  uint32_t ea_block_h, ea_block_l;
  uint32_t tag_id;
  spe_par_t spe_par __attribute__
((aligned(16)));

  ea_block_h = spu_read_in_mbox();
  ea_block_l = spu_read_in_mbox();

  tag_id = mfc_tag_reserve();

  spu_mfcdma64( &spe_par, ea_block_h,
ea_block_l, sizeof( spe_par_t ),
tag_id, MFC_GET_CMD );
  mfc_write_tag_mask( 1 << tag_id );
  mfc_read_tag_status_all();

  spe_par.value = compute_pi( spe_
par.seed, spe_par.rounds );

  spu_mfcdma64( &spe_par, ea_block_h,
ea_block_l, sizeof( spe_par_t ),
tag_id, MFC_PUT_CMD );
  mfc_write_tag_mask( 1 << tag_id );
  mfc_read_tag_status_all();

  mfc_tag_release( tag_id );

  return 0;
}
```