Technical Report CSTN-102

Comparing Intra- and Inter-Processor Parallelism on Multi-Core CellBE Processors for Scientific Simulations

K.A. Hawick, A. Leist and D.P. Playne and M.J. Johnson
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
email: k.a.hawick@massey.ac.nz
Tel: +64 9 414 0800   Fax: +64 9 441 8181

July 2010

ABSTRACT
The Cell Broadband Engine (Cell BE) multi-core processor from the STI consortium of Sony, Toshiba and IBM is a powerful but complex processing device that has attracted much attention since its inclusion in Sony PlayStation (PS3) gaming consoles. We report on some performance experiments using the multi-core Synergistic Processing Elements (SPE) concurrency capabilities of this chip. We compare performance and software implementation issues with conventional cluster computing techniques such as message-passing, in exploiting clusters of Cell BE processors for scientific simulations. We discuss performance and user programming issues for some hybrid solutions on clustered PS3 computers running Linux.

KEY WORDS
Cell BE processor; multi-core; SPE; PS3; cluster.

1 Introduction

The multi-core approach to CPU design is likely to remain very important for designers seeking to increase the available processing power while limiting thermal output. Programming such systems remains a challenge to applications developers however, and there is still great scope for experimentation with different optimisations for processors with small, large or medium numbers of cores, which may be symmetric or be arranged in a master slave combination [1]. Many of the present generation of processing devices have been driven by economic considerations and in particular by the computer games market. However scientific simulations share many of the software algorithms and optimisation goals of game software and particularly physics engines.

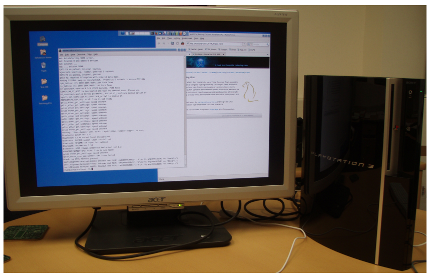
Figure 1: PlayStation PS3.

The consortium of Sony, Toshiba and IBM (STI) produced a multi-core processor chip that is known as the Cell Broadband Engine (CellBE) [2] and which is now widely known as the processor in the Sony PlayStation 3 (PS3) games machine. The CellBE comprises a PowerPC processing core (PPE) but is unusual in that it is supported by eight “synergistic processing elements” (SPEs) that provide additional performance acceleration [3]. A number of other products now have processors based on the PPE component of the CellBE including the Microsoft Xbox 360 games engine, and the CellBE itself is also available in blade systems. The CellBE design was apparently driven by silicon space and also power consumption considerations [4], and this design paradigm – of a relatively conventional main core supported by specialist processing accelerator cores – may become more prevalent as chip vendors attempt to combine large numbers of cores with other design criteria like minimal power consumption.

One notable game that exploits the power of the CellBE effectively for game physics and impressive explosions is Red Faction: Guerrilla, which is a third-person shooter game developed by Volition, Inc. and published by THQ. However there is a growing body of reported work on applications running on the CellBE, beyond their original games applications targets [5] and applications such as Lattice Quantum Chromodynamics [6] have achieved in excess of 45 GFlops on a CellBE in an IBM Bladecenter.

The CellBE as supplied on the Sony PlayStation 3 is accessible to a Linux programmer through the STI software development kit and a compilation system based on the Free Software Foundation GNU gcc compiler [7]. Scarpino's book [3] describes how to program the CellBE on the PS3 in more detail, but in brief it is possible to program both the PPE and the SPE using appropriate libraries called from what are essentially C/C++ application programs. The PS3 implementation uses a hypervisor arrangement so the Gaming Operating system supplied by Sony supervises a running Linux, but only six of the available eight SPEs are exposed to the Linux programmer. Similarly some additional features of the CellBE such as the graphics and storage interface are not directly controllable.

Nevertheless, the PS3/CellBE/Linux combination provides a remarkably cost effective developmental platform for development of application programs for the CellBE and has allowed us to investigate some of the relevant features for comparison with the other software and hardware platforms discussed in this article.

The CellBE approach to multi core is to supply a relatively small number of accelerator cores to support the main CPU core. This is different from both mainstream multi-core CPU developments such as the Itanium where all the cores are full fledged CPU and are largely identical in their capabilities, but also from very-many core data-parallel systems such as Graphical Processing Units (GPUs) [8].

As Dongarra and others have reported [9,10], the PS3 is an appropriate platform for some but not all scientific applications. Although it is possible to use the PS3 network interface to cluster multiple PlayStations together, in this present paper we report only on performance observations using a single CellBE in a single PS3. Our main observation is that, at least at the time of writing, some considerable expertise, time and effort is required on the part of the applications programmer to make full use of the processing power of the CellBE. In fairness this is true of parallel programming generally. If programmer use of parallelism is necessary to attain optimal performance on future processors that have of necessity been designed to have low power considerations, then better software tools and/or programmer expertise will be required.

Figure 2 summarises the main points of the CellBE architecture. The PPE unit has some common ancestry with IBMs's other PowerPC chip designs and is a powerful core in its own right. It is supported by eight SPE cores each of which has some dedicated memory (256kB) and which are accessible through a bi-directional dual ring interface. The software development kit and associated library provide a great many library calls to communicate between different core components of the CellBE. Mechanisms include mailboxing and signalling as well as bulk data transfers using a Direct Memory Access (DMA) support infrastructure.

Generally, it is not difficult for an applications programmer to develop a master/slave software model to delegate computations to the SPE from the PPE. A Monte Carlo example for instance works well, providing that tasks for the slave SPE's fit in their own memory. Otherwise, some care in needed to barrier-synchronise their access to global memory to avoid contentions.

A number of software packages are relatively trivially ported to work on the PPE main core, treating it as any other PPC instructions set architecture. For example the Erlang parallel language system [11] worked on the PPE trivially. As a very rough indicator, we found the PPE by itself gave a performance equivalent to around half of that of a single CPU core on a contemporary desktop. This is quite impressive given the CellBE was designed with completely different optimisation criteria. To obtain high performance from the CellBE it is necessary to a) delegate computational work to the SPEs and b) make use of the vector instructions on both the PPE and SPEs.

The PPE and SPEs are capable of processing vector data types performing four instructions synchronously. To make full use of the SPEs, the programmer must use `spu_add, spu_mul, etc` to transform calculations into vector form. Other performance optimisation techniques available are enqueued DMA access (IE fetch memory while computing); bi-directional use of the IE ring; use of the memory flow controller.

We discuss some issues concerned with setting up a cluster of PS3s with the Linux operating system running on the CellBE processors in Section 2. We describe a random number generator algorithm (Section 3) and associated CellBE software optimisation issues in Section 4. We give some selected results covering a simple port to the CellBE and also the communications bandwidth attainable between CellBEs in a PS3 cluster in Section 5. We offer some discussion and areas for further work in Section 6.

2 PS3 Clusters

The platform used to run the CellBE implementations is a PlayStation 3 running Yellow Dog Linux 6.1. It uses a Cell processor running at 3.2GHz, which consists of 1 PowerPC Processor Element and 8 Synergistic Processor Elements, 6 of which are available to the developer. It has 256MB of system memory.

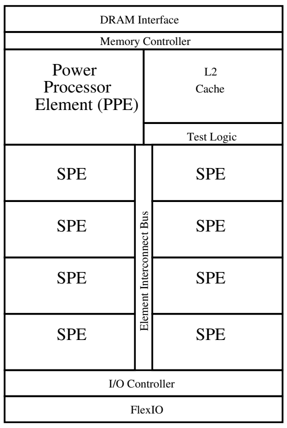

Figure 2: STI CellBE Processor Architecture.

Figure 2 shows the die layout of the CellBE processor and the main PPE unit and its supporting SPEs. For the work reported in this paper we only access 6 out of the 8 SPE units, the other two are dedicated to systems and graphical support on the PS3. At the time of writing Sony have withdrawn support for a second operating system such as Linux, on the new models of the PS3 although we were able to make use of extensive libraries and support code available for Fixstars Yellow Dog Linux installation packages [12]. Compilers and developer libraries for the CellBE/PPE/SPE combinations were also readily available through this Linux.

Some of th work we report below made use of a single CellBE/PS3, but we also experimented with message passing communications amongst several PS3s arranged in a cluster.

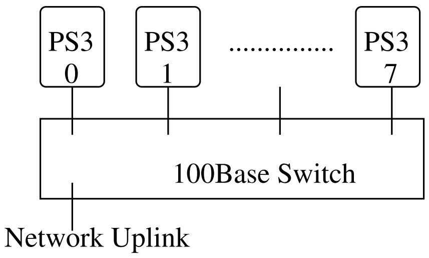

Figure 3: Cluster Architecture.

Figure 3 shows the standard cluster architecture for a small group of PS3s running Linux. W e evaluated raw network communication parameters as well as that available from application program to application program, via a software stack such as the MPI message passing interface [13].

Figure 4: PlayStation PS3 Proto Cluster

Figure 4 shows the scale of the PS3 next to a typical switch. These games consoles stack very neatly on their end and dissipate heat quite well through a grille on the top. When running scientific computations as described below, the graphics system was inactive and this further reduced thermal output. Our cluster was able to be run in a normal laboratory environment without special air conditioning.

3 Random Number Generation

Random number generation [14] is one of the most widely used facilities in computer simulations, finding uses in applications such as spin models [15] amongst many others. A number of different algorithms are widely used [16–19] ranging from fast but low quality system supplied generators such as the rand()/random() generators available on Unix [20] systems to slower but high quality 64-bit algorithms such as the Mersenne Twistor generator [21].

Marsaglia's lagged-Fibonacci generator [22] is a 24-bit algorithm that produces good quality uniform deviates and which has been widely used in Monte Carlo work [23]. It is convenient for our purposes in this present paper as not all target accelerator hardware platforms uniformly support 64-bit floating point calculations.

The Marsaglia lagged-Fibonacci random number generator (RNG) has been described in full elsewhere [22], but in summary the details are given in Algorithm 1, which we provide for completeness.

Algorithm 1 Marsaglia Uniform Random Number Generator, where an initialisation procedure sets the values as given below, and fills the lag table with deviates.

```
declare $u[97]$
initialise($u, seed$)
declare $i \leftarrow 96$
declare $j \leftarrow 32$
declare $c \leftarrow 362436.0/16777216.0$
declare $d \leftarrow 7654321.0/16777216.0$
declare $m \leftarrow 16777213.0/16777216.0$
for $n \leftarrow 0$ to $N$ do
    uniform($i, j, c, d, m, u$)
end for
```

Where i, j index a lag table which is shown here of 97 deviates, but which can be any suitable prime, subject to available memory and where c, d, m are suitable values.

A number of optimisations for this sort of random number generation algorithm are possible on the various implementation platforms. One obvious one is to synchronise a separate thread that can produce an independent stream of random deviates that are consumed by the main application thread. Other algorithms, whose descriptions are beyond the space limitations of our present paper, generate whole vectors or arrays of deviates together using a SIMD approach which can be used in applications that have similarly shaped work arrays or objects such as images or model data fields.

4 CellBE Software Architecture

It was relatively easy to create a quick port implementation of the LFG on the CellBE. However the performance was far from optimal and implementing the lagged-Fibonacci generator on the Cell processor efficiently requires some not inconsiderable programming effort.

Algorithm 2 Marsaglia Uniform Random Number Generator, each call will generate a single random number.

```
function uniform($i, j, c, d, m, u$)
    declare $result \leftarrow u[i] - u[j]$
    if $result < 0$ then
        $result \leftarrow result + 1$
    end if
    $u[i] \leftarrow result$
    $i \leftarrow i - 1$
    if $i < 0$ then
        $i \leftarrow 96$
    end if
    $j \leftarrow j - 1$
    if $j < 0$ then
        $j \leftarrow 96$
    end if
    $c \leftarrow c - d$
    if $c < 0$ then
        $c \leftarrow c + m$
    end if
    $result \leftarrow result - c$
    if $result < 0$ then
        $result \leftarrow result + 1$
    end if
    return $result$
end function
```

4.1 Rapid Port Implementation

We give code listings showing typical code harness and library calls for programming the CellBE with control code running on the PPE 1; slave code running on the SPE 2 and some common data structures and application specific code 3. Generally an application can be initially ported using these library calls and relatively minor data packaging changes to ensure efficient transfers between PPE and SPE.

Listing 1: PPE Random Number Generator Call Code harness

```
  //aligned for transfer:
  program_data pd[NSPE] __attribute__((aligned(16)));

  for(int s=0;s<NSPE;s++){
    pd[s].slaveid = s;
    // seed by SPE number
    pd[s].seed = 12347 * (s+1);
  }

  initialise_mg1(12345678);  // RNG seed value

  //Create SPE Task invoke:
  speid_t  spe_id[NSPE];

  for(int s=NSPE-1;s>=0;s--){
    spe_id[s] = spe_create_thread(0, &randomize_handle,
              &pd[s], NULL, -1, 0 );

    if(spe_id[s] == 0) {
      fprintf(stderr, ``Error creating SPE thread %d!\n'', s);
      return 1;
    }
  }

  //Wait For Completion:
  for(int s=0;s<NSPE;s++){
    spe_wait(spe_id[s], NULL, 0);
  }
```

Listing 2: SPE Random Number Generator Code

```
int main(unsigned long long spe_id,
         unsigned long long program_data_ea,
         unsigned long long env) {

  program_data pd __attribute__( (aligned(16)) );

  int tag_id = 0;

  // Read data and Initiate copy:
  mfc_get( &pd, program_data_ea, sizeof( pd ), tag_id, 0, 0 );

  // Wait for completion:
  mfc_write_tag_mask( 1<<tag_id );
  mfc_read_tag_status_any();

  // Process:
  initialise_mg1( pd.seed );
  for( int i=0;i<100000000;i++){ // 100 million deviates
    pd.u = uniform_mg1();
  }

  // Write results and Initiate copy:
  mfc_put( &pd, program_data_ea,
           sizeof( program_data ), tag_id, 0, 0);

  // Wait for completion:
  mfc_write_tag_mask( 1<<tag_id );
  mfc_read_tag_status_any();

  return 0;
}
```

Listing 3: Lagged Fibonacci Random Number Generator Code

```
// struct for communication with the PPE
typedef struct {
  int seed;
  float u;
  long long int slaveid;

  //char _padding[4]; // multiple of 16 bytes
} program_data;

#ifndef LAG_TABLE_LENGTH
#define LAG_TABLE_LENGTH 97
#endif

static float mg1_u[LAG_TABLE_LENGTH+1];
static float mg1_c, mg1_cd, mg1_cm;
static int mg1_i;
static int mg1_j;

float uniform_mg1(){
    float result;
    result = mg1_u[mg1_i] - mg1_u[mg1_j];
    if( result < 0.0 )
        result = result + 1.0;
    mg1_u[mg1_i] = result;
    mg1_i = mg1_i - 1;
    if( mg1_i == 0 )
        mg1_i = LAG_TABLE_LENGTH;
    mg1_j = mg1_j - 1;
    if( mg1_j == 0 )
        mg1_j = LAG_TABLE_LENGTH;
    mg1_c = mg1_c - mg1_cd;
    if( mg1_c < 0.0 )
        mg1_c = mg1_c + mg1_cm;
    result = result - mg1_c;
    if( result < 0.0 )
        result = result + 1.0;
    return( result );
}
```

As can be seen from these code fragments, for the most part programming accelerators such as the PPE/SPE combinations is readily done in languages like C/C++ with appropriate calls to systems level libraries.

4.2 Optimised Vector Processing Implementation

There are six separate SPEs each of which can process a vector for four elements synchronously. Vectors types are used to make full use of the SPEs processing capabilities. Thus for each iteration, each SPE will generate four random numbers (one for each element in the vector).

To ensure that unique random numbers are generated, each element in the vector of each SPE must have a unique lag table. Six SPEs with four elements per vector results in twenty-four lag tables. These lag tables are implemented as a single lag table of type `vector float` but each element of the vectors is initialised differently. Care should be taken when initialising these lag tables to make certain that the lag tables do not have correlated values and produce skewed results.

The lagged-Fibonacci generator algorithm has a two conditional statements that affect variables of vector type. These conditional statements both take the form of `if( result < 0.0) result = result + 1.0;` (See Algorithm 1). As each element in the vector will have a different value depending on its unique lag table, different elements in the vector may need to take different branches.

There are two ways of overcoming this issue. The first method is to extract the elements from the vector and process them individually. This method is not

Algorithm 3 Pseudo-code for Marsaglia Lagged-Fibonacci algorithm implemented on the CellBE using vectors.

```
declare vector float $u[97]$
initialise(u)
declare $i \leftarrow 96$
declare $j \leftarrow 32$
declare $c \leftarrow 362436.0/16777215.0$
declare $d \leftarrow 7654321.0/16777215.0$
declare $m \leftarrow 16777213.0/16777215.0$
function uniform()
  declare vector float $zero \leftarrow$ spu_splats(0.0)
  declare vector float $one \leftarrow$ spu_splats(1.0)
  declare vector float $result \leftarrow u[i]$ - $u[j]$
  declare vector float $plus1 \leftarrow result + one$
  declare vector unsigned $sel\_mask \leftarrow result > zero$
  $result \leftarrow$ select($result$, $plus1$, $sel\_mask$)
  $u[i] \leftarrow result$
  $i = i - 1$
  if $i == 0$ then
    $i \leftarrow 96$
  end if
  $j = j - 1$
  if $j == 0$ then
    $j \leftarrow 96$
  end if
  $c = c - d$
  if $c < 0$ then
    $c \leftarrow c + m$
  end if
  $result \leftarrow result -$spu_splats($c$)
  $plus1 \leftarrow result + one$
  $sel\_mask \leftarrow result > zero$
  $result \leftarrow$ select($result$, $plus1$, $sel\_mask$)
  return $result$
end function
```

ideal as it does not use the vector processing ability of the cell, instead the `spu_sel` and `spu_cmpgt` instructions can be used.

The `spu_cmpgt` instruction will compare two vectors (greater than condition) and return another vector with the bits set to 1 if the condition is true and 0 if the condition is false. The comparison is performed in an element-wise manner so the bits can be different for each element. The `spu_sel` can then select values from two different values depending on the bits in a mask vector (obtained from the `spu_cmpgt` instruction).

Using these two instructions the conditional statement `if( result < 0.0) result = result + 1.0;` can be processed as vectors with different branches for each element. The pseudo-code for this process can be seen in Algorithm 3.

5 Results

We report on two approaches to programming the CellBE for random number generation. The first is to treat the SPE units as simplified slaves to the PPE without making any particular attempts at source code optimisation. This can be achieved readily enough but to make good use of the SPE (and indeed the PPE itself) properly some extra optimisation is worthwhile.

5.1 Random Number Generation

A rapid port of the Lagged Fibonacci code was made initially, so that each SPE could be given a task thread consisting of multiple calls to the generator. Figure 5 shows an interesting pattern of timing data resulting from increased levels of work (multiples of 100 million generated random deviates) being given to the SPEs. There is an initial overhead in invoking any SPEs at all, followed by relatively flat time plateaus as all six SPEs are equally loaded, followed by a jump to the next plateau as more thread tasks are added.

The fitted average slope indicates approximately 1.73 seconds per task, or a speed rating of approximately 58 million generated deviates per second. This is not particularly impressive compared with single core CPU performance. The optimised implementation makes use of the vector processing units on the SPEs and therefore provides a useful performance improvement. To illustrate how CellBE performance relates to other parallel architectures we show results

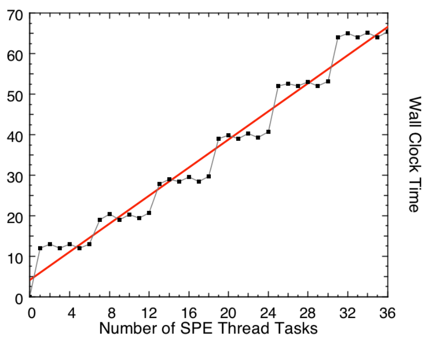

Figure 5: Timing to generate 100 million Lagged Fibonacci Random Number Generator on the 6 accessible SPEs on the PS3. Time per deviate-task is therefore in effective microseconds.

from both versions compared to LFG implementations on: one CPU core; four CPU cores; one GPU and two GPUs. These results are shown in Table 1 and are discussed further in [1].

The implementations of the lagged-Fibonacci generators have been tested by generating 24 billion random numbers and measuring the time taken. In the performance measures (See Table 1) the random numbers have not been used for any purpose as the only intention was to measure the generation time. This is obviously not useful in itself but it is assumed that any application generating random numbers such as these will make use of them on the same device as they were generated. Otherwise the random values can simply be written to memory and extracted from the device for use elsewhere.

Table 1: Comparison between implementations of the time taken to generate 24,000,000,000 random numbers using the lagged-Fibonacci generator.

Device	Time (seconds)	Speed-up
CPU	256.45	1.0x
PThreads	66.72	3.8x
TBB	95.40	2.7x
Cell	415.20	0.6x
Cell (Vector)	23.60	10.9x
CUDA	8.40	30.5x
Multi-GPU	4.33	59.2x

5.2 Cluster Latency and Bandwidth

Some applications can make use of multiple CellBEs running as part of a cluster. The PS3 implementation is limited by the (entirely adequate for gaming) 100base network interface. Timing using point-to-point between pairs of PS3s gives the data shown in figure 6.

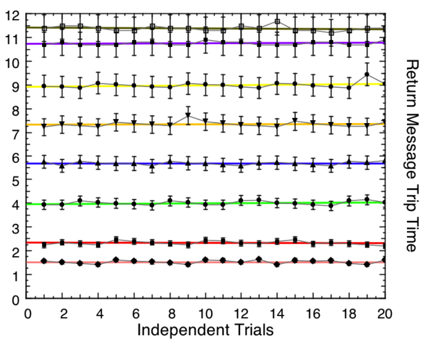

Figure 6: Timing for Different sized message payloads.

Twenty independent trials were made and an average for each message payload size obtained. These average values are shown in figure 7.

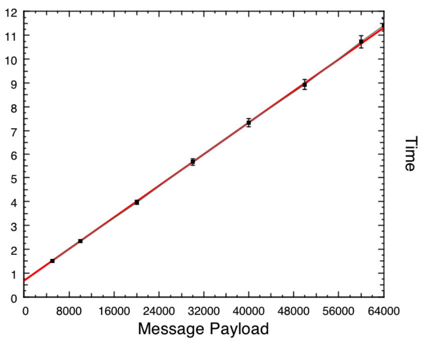

Figure 7: Latency of Communications as y-intercept and effective bandwidth as slope.

The data indicate a reasonably accurate average bandwidth of approximately 46 MBit/sec with 0.7 ms latency between PS3s. This is adequate for throughput style parallel job distribution but the latency is too high and the bandwidth is too low for tightly coupled applications that distribute data arrays such as field values in a partial differential equation solver [24].

6 Discussion and Conclusions

We have shown how the CellBE available in the PS3 is both interesting and accessible to developers of scientific simulations. The Cell Broadband Engine has been shown to provide a significant speed-up over the simple single-core CPU in our experiments. This is particularly impressive also when taking device cost and heat production into account. However, the complexity of code development made this a somewhat less favourable option, especially considering the results produced are not significantly faster than the multi-core CPU applications. We can obtain relatively interesting performance if we optimise code at the SPE level. However, developing code to correctly use the vector processing abilities of the Cell and to reduce the waiting time for memory requests to be completed by the MFCs proved to be a challenge. We would expect that ongoing compiler developments will do more of this instruction-level and vector-level work in the future.

This sort of hybrid architecture has great promise not least due to its low thermal consumption, but programming it is non trivial. This might not matter if there was already a good paradigmal code base around. It seems likely that as this sort of architecture becomes more common more such code and optimisation experience will become available. We might hope that compilers will incorporate more suitable optimisers well, but one promising approach is to generate appropriately optimised application code fragments directly. We are working on tools for semi automatic code generation that can in principle target optimisations for any platform for which we know the architecture [25].

In summary, we believe thermally efficient hybrid multi-core designs with a mix of high and low capability cores such as the CellBE will indeed find a continued market in both games applications as well as scientific simulations, providing improved programming tools can be made available for them.

References

[1] Hawick, K.A., Leist, A., Playne, D.P.: Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software. Technical Report CSTN-091, Computer Science, Massey University (2009)

[2] Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., Shippy, D.: Introduction to the cell multiprocessor. IBM Journal of Research and development **49** (2005) 589–604

[3] Scarpino, M.: Programming the Cell Processor for Games, Graphics and Computation. Number ISBN 978-013-600886-6. Prentice Hall (2009)

[4] Shippy, D., Phipps, M.: The Race for a New Game Machine. Number ISBN 978-0-8065-3101-4. Citadel Press (2009)

[5] Biberstein, M., Harel, Y., Heilper, A.: Clock synchronization in cell/b.e. traces. Concurrency and Computation: Practice & Experience **21** (2009) 1760–1774

[6] Spray, J.C.: Lattice QCD on the Cell Processor. Master's thesis, Edinburgh University (2007)

[7] Free Software Foundation: The GNU compiler collection. (2007)

[8] Leist, A., Playne, D., Hawick, K.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Concurrency and Computation: Practice and Experience **21** (2009) 2400–2437 CSTN-065.

[9] Kurzak, J., Buttari, A., Luszcek, P., Dongarra, J.: The playstation 3 for high-performance scientific computing. Computing in Science and Engineering **10** (2008) 84–87

[10] Buttari, A., Dongarra, J., Kurzak, J.: Limitations of the playstation 3 for high performance cluster computing. Technical report, University of Tennessee - Knoxville (2007)

[11] Armstrong, J., Williams, M., Wikstrom, C., Virding, R.: Concurrent Programming in Erlang. Prentice Hall (1996) ISBN 0-13-285792-8.

[12] Fixstars: A Guide to Installing Yellow Dog Linux 6.1. Technical report, Fixstars (2008)

[13] Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press (1994) ISBN 0-262-57104-8.

[14] Hawick, K., Leist, A., Playne, D.: Some Parallel Random Number Generators and Implementation Issues on GPUs, Multi-Core and Cell Processors. Technical Report CSTN-103, Computer Science, Massey University (2009)

[15] Hawick, K., Leist, A., Playne, D.: Damaged Lattice and Small-World Spin Model Simulations using Monte-Carlo Cluster Algorithms, CUDA and GPUs. Technical Report CSTN-093, Computer Science, Massey University (2009)

[16] L'Ecuyer, P.: Software for uniform random number generation: distinguishing the good and the bad. In: Proc. 2001 Winter Simulation Conference. Volume 2. (2001) 95–105

[17] Marsaglia, G.: Random Number generation. (Florida Preprint, 1983)

[18] Marsaglia, G.: A Current view of random number generators. In: Computer science and statistics: 16th symposium on the interface, Atlanta (1984) Keynote address.

[19] Marsaglia, G., Tsay, L.H.: Matrices and the Structure of Random Number Sequences. Linear algebra and its applications **67** (1985) 147–156

[20] BSD: Random - Unix Manual Pages. (1993)

[21] Matsumoto, M., Nishimura, T.: Mersenne twistor: A 623-diminsionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation **8 No 1.** (1998) 3–30

[22] Marsaglia, G., Zaman, A.: Toward a universal random number generator. FSU-SCRI-87-50, Florida State University (1987)

[23] Binder, K., Heermann, D.W.: Monte Carlo Simulation in Statistical Physics. Springer-Verlag (1997)

[24] Hawick, K., Playne, D.: Automated and parallel code generation for finite-differencing stencils with arbitrary data types. In: Proc. Int. Conf. Computational Science, (ICCS), Workshop on Automated Program Generation for Computational Science, Amsterdam June 2010. Number CSTN-106 (2010)

[25] Hawick, K.A., Playne, D.P.: Automatically Generating Efficient Simulation Codes on GPUs from Partial Differential Equations. Technical Report CSTN-087, Computer Science, Massey University (2010) Submitted to Springer J. Sci. Computing.