SPEEDUP - optimization and porting of path integral MC Code to new computing architectures

V. Slavnić, A. Balaž, D. Stojiljković, A. Belić, A. Bogojević
Scientific Computing Laboratory, Institute of Physics Belgrade, Serbia

Path integral formalism presents the concise and flexible formulation of quantum theories at different levels, providing also simple description of many other complex physical systems. Recently introduced analytical approach that systematically improves convergence of numerically calculated path integrals of a generic theory leads to a significant speedup of Path Integral Monte Carlo algorithms. This is implemented in the SPEEDUP code. Here we report on optimization, porting and testing of the SPEEDUP code on new computing architectures: latest Intel, and IBM POWER6 and PowerXCell CPUs. We find that the code can be highly optimized and take substantial advantage of the features of new CPU types.

1. Introduction

Path integral Monte Carlo code SPEEDUP [1] is used for various calculations mainly for studies of Quantum Mechanical systems and investigation of global and local properties of Bose-Einstein condensates. Porting of this code to new computing architectures will enable its use on a broader set of clusters and supercomputer facilities. The purpose of the code optimization is to fully utilize available computing resources, eliminating bottlenecks that may be located in different parts of the code, depending on the details of hardware implementation and architecture of the CPU. In some situations even compiling, linking or choosing more appropriate (optimized) libraries can lead to significant reduction in program execution times. However, the optimization must be performed carefully and the new code has to be verified after each change by comparison of its numerical results with the correct reference values.

In addition to obtaining highly optimized code, the above procedure can be also used to benchmark different hardware platforms and to compare their performance on a specific application/code. Such application-specific benchmarking, based on the assessment of hardware performance for the chosen set of applications, can be also used for the proper planning of hardware upgrades of computing centers supporting several user communities.

2. SPEEDUP code

Functional formalism in quantum theories naturally introduces Monte Carlo simulations as a method of choice for numerical studies of relevant physical systems. The discretization of the phase space (necessary in any numerical calculation) is already built in to the functional formalism through the definition of continuous (path) integrals, and can be directly translated into the Monte Carlo algorithm. A detail study of the relationship between discretization of different coarseness in the case of a general quantum theory leads to substantial increase in convergence of path integral to its continuum limit [2-4]. This study resulted in an analytic procedure for deriving a hierarchy of effective actions up to an arbitrary level *p*. We will illustrate the use of higher-level effective actions for calculation of the transition amplitude *A* for a quantum system that evolves from the initial state *i* to the final state *f* in time *T*. In the path integral formalism, this amplitude is given as $N\to\infty$ limit of the *(N-1)*-fold integral expression:

$$A_N(i;f;T)=\left(\frac{1}{2\pi\epsilon_N}\right)^{N/2}\int dq_1\cdots dq_{N-1}e^{-S_N},$$

where S_N is the discretized action of the theory and $\epsilon_N=T/N$ is the discrete time step. Using naively discretized action, the transition amplitude would converge to its continuum limit as

slow as $1/N$. Numerical simulations based on the use of effective action of the level p have much faster convergence, approaching the continuous limit as $1/N^p$. The effective discretized actions up to level p=18 are implemented in the Path Integral Monte Carlo SPEEDUP code [1] in C programming language. It is used for efficient calculation of transition amplitudes, partition functions, expectation values, as well as low lying energy spectra.

The algorithm of a serial SPEEDUP code can be divided to the following steps:

1. Initialize variables; allocate memory; set input parameters of the model, number of time and MC steps, and random number generator (RNG) seed.
2. Main Monte Carlo loop, which accumulates contributions of sampled trajectories to intermediate variables; each loop step consists of the following steps:
    a. Generate trajectory using bisection method [5]. The number of time steps is $N=2^s$, where s is the discretization level (input parameter),
    b. Calculate effective action for a generated trajectory and each sub-trajectory with smaller discretization level (s-1, ..., 1),
    c. Accumulate variables used to calculate observables and their error estimates at each discretization level,
3. Calculate observables and associated errors by averaging variables accumulated in the previous step at each discretization level,
4. Print the results, deallocate memory and exit the program.

Parallelization of the above Monte Carlo algorithm is very simple, since each loop step 2 is independent. Therefore, the total number of Monte Carlo steps can be easily and evenly divided to a desired number of CPU threads or parallel processes (in MPI or in other available parallelization environment).

The SPEEDUP code generates large numbers of random trajectories and relies on the MC theory to achieve no correlations between the generated trajectories. This necessitates high-quality RNG, able to produce large numbers of uncorrelated random numbers from the uniform probability density distribution, in a form suitable for parallel simulation. For the SPEEDUP code we have used SPRNG - Scalable Parallel Random Number Generator [6], which is verified to satisfy all of the above criteria. SPRNG can generate large numbers of separate uncorrelated streams of random numbers, making it ideal for parallel applications.

3. Tested hardware architectures

The hardware platform used for the testing reported in this paper was IBM BladeCenter with 3 kinds of servers within the H-type chassis commonly used in high performance computing:

- HX21XM blade Server based on Intel Xeon technology. It features two Intel Xeon 5405 processors that run on 2.0 GHz with front side bus of 1333MHZ and level two cache (L2) of 12MB with support for Intel SSE2, SSE3, SSE4.1 extensions. Along with standard GCC (GNU Compiler Collection) compiler (gcc version 4.1.2), Intel C++ Compiler Professional Edition 11.1 by Intel Corporation (ICC) [7] that includes advanced optimization, multithreading, and processor support, as well as automatic processor dispatch, vectorization, and loop unrolling was used for testing in this paper.
- The BladeCenter JS22 server is a single-wide, 4-core, 2-socket with two cores per socket, 4.0 GHz POWER6 [8] SCM processors. Each processor includes 64 KB I-cache and 32 KB D-cache L1 cache per core with 4 MB L2 cache per core. Processors in this blade server are based on POWER RISC instruction set architecture (ISA) with AltiVec, a single-instruction, multiple-data (SIMD) extensions. IBM provides XL C/C++ compiler solution (XLC) [9] that offers automated SIMD capabilities for application code that can

be quite help for programmers. Beside GCC compiler IBM XLC/C++ is used for benchmark purposes in this paper.

- The IBM BladeCenter QS22 is based on 2 multi-core IBM PowerXCell 8i processors, based on Cell Broadband Engine Architecture (Cell/B.E.) [10]. The Cell Broadband Engine is a single-chip multiprocessor with nine processors specialized into two types:
    1. The PowerPC Processor Element (PPE) is a general-purpose, dual-threaded, 64-bit RISC processor fully compliant with the 64-bit PowerPC Architecture, with the Vector/SIMD Multimedia Extension operating at 3.2 GHz. It is intended primarily for control processing, running operating systems, managing system resources, and managing SPE threads.
    2. The SPE (Synergetic Processing Element) is core optimized for running compute-intensive applications. SPEs are single-instruction, multiple-data (SIMD) processor elements that are meant to be used for data-rich operations allocated to them by the PPE. Each SPE contains a RISC core, 256 KB software-controlled locale storage (LS) for instructions and data, and a 128-bit, 128-entry unified register file. The SPEs provide a deterministic operating environment. An SPE accesses both main memory and the local storage of other SPE's exclusively with DMA commands. They do not have caches, so cache misses are not a factor in their performance and programmer should to avoid branch intensive code.

    The Cell Broadband Engine has one PPE and eight SPEs.

Such a heterogeneous multi-core architecture of the Cell CPU requires that a developer adopts several new programming paradigms in order to fully utilize the full potential of Cell B/E processor. In addition to the GNU tools (including C and C++ compilers) which are provided with the Software Developer's Kit for Multicore Acceleration [11], one can also use IBM XL C/C++ Compiler [9] for Multicore Acceleration, specialized for Cell Broadband Engine solution.

4. Results

Here we describe the performed optimization and the obtained benchmarking results. In all benchmarks in this paper we have executed the code with *Nmc*=5120000 MC samples for the quantum-mechanical amplitude of the quartic anharmonic oscillator with the boundary conditions $q(t=0)=0$, $q(t=T=1)=1$, with zero anharmonicity and with level $p=9$ effective action. We always used the same seed for SPRNG generator so that the results can be easily compared. Section 4.1 gives results for a serial SPEEDUP code on each platform with different compilers. These results are later used as a reference in benchmarking and in verification of the optimized code. Section 4.2 gives results for SPEEDUP MPI code tested on Intel platform and Section 4.3 presents the threaded SPEEDUP code and results obtained with Intel and POWER architectures. In Section 4.4 we give results for the Cell SPEEDUP code, and in Section 4.5 we compare all obtained results.

4.1. Serial SPEEDUP code

For Intel Blade server we compiled the serial code with GCC C compiler using optimization flags *O1* and *funroll-loops* which give the best performance (better than the *O3* flag, with or without loop unrolling). Along with GCC, we also used ICC compiler with maximal optimization flag *O2*.

On POWER6 and Cell Blades the code was compiled with both GCC and IBM XLC compilers. On Cell Blade we used the flags *O3*, *funroll-loops*, *mabi=altivec*, and *maltivec*

with GCC, and *O5*, *qaltivec* and *qenablevmx* with XLC. Appropriate versions of GCC and XLC binaries were used (ppu-gcc and ppuxlc). On POWER6 Blade the *O5* flag was used with XLC and *O3* and *funroll-loops* with GCC. Results for serial program testing are presented in Table 1.

Table 1. Average time of execution of a serial SPEEDUP code on all tested platforms with different compilers

Compiler / Platform	GCC	ICC	XLC
Intel	(13760±50) s	(10160±30) s	-
POWER6	(17000±10) s	-	(1900±10) s
Cell	(49410±50) s	-	(14020±20) s

Table 1 demonstrates the significant increase in the speed of the code when platform-specific compiler is used. We also see that in this specific case the POWER6 platform in combination with the XLC compiler shows order of magnitude improvement in the speed compared to the Intel platform. On the other hand, it is clear that Cell version, running only on the PPE is no match for other two platforms. Real utilization of the Cell platform can be achieved only when SPEs are used.

4.2. MPI SPEEDUP code

On the Intel Blade multicore, we tested the performance of the SPEEDUP code with MPI implementation, compiled with GCC and ICC compliers. The results are shown in Fig. 1.

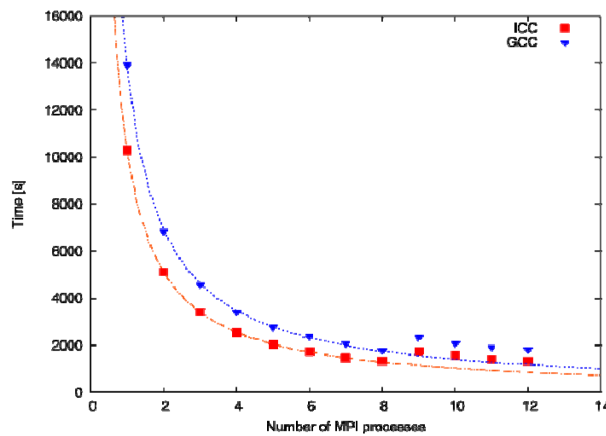

Fig. 1. Average times of execution of the MPI SPEEDUP code on Intel platform compiled with ICC (*O2* flag) and GCC (*O1* and *funroll-loops*). The curves give fits to the expected dependence $A + B$ / (Number of MPI processes).

As we can see, the MPI version of the code shows excellent scalability with the number of MPI processes. When the number of MPI processes exceeds the number of physical cores in the system, the operating system is trying to distribute the load among

already fully loaded cores, which creates additional overhead. This implementation gives minimal execution time of 1320s.

4.3. Modified SPEEDUP code

To fully optimize the parallel SPEEDUP code, instead of using MPI API we implemented its threaded version using POSIX threads (pthreads). Each thread calculates *Nmc/Nth* of Monte Carlo samples where *Nth* is the number of threads. Also, some minor additional modifications of the code were performed, focusing on specific improvements for p=9 effective action. The Intel version was compiled with ICC, while the POWER version was compiled with XLC. The obtained numerical results are shown in Fig. 2.

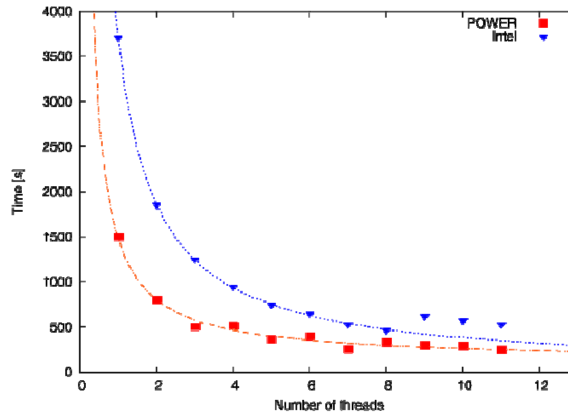

Fig. 2. Average times of execution of the threaded SPEEDUP code on Intel and POWER platforms

With the threaded code we obtained a significant increase in the speed of the code, even without implementing specific vector instructions (AltiVec on POWER6 or SSE on Intel). Again, POWER6 Blade in conjunction with XLC compiler was faster than Intel Blade. However, the relative increase in the speed of threaded code was larger on the Intel platform. The minimal execution time was 460s on Intel and 250s on POWER Blade. This gives relative increase in the speed of the code of 2.8 (threaded vs. MPI) for Intel and 1.3 (one thread vs. serial) for POWER6 platform.

We also note an interesting scaling issue on POWER6 system. While the threaded code scales perfectly on Intel Blade, POWER6 Blade shows strange behavior for even number of threads, where execution times are slightly higher than expected. When the code is compiled with GCC, the same behavior is observed for odd number of threads. Such throttling may be related to low-level hardware details that are not properly implemented in different compilers.

4.4. Cell SPEEDUP code

The heterogeneity of the Cell architecture required the slight rearrangement of the SPEEDUP code. We used MPI version of the code as a basis, and modified it so as to separate parts that are executed on the PPE and parts that are executed in parallel on SPEs. Our implementation was to create a number of pthreads on the PPE that will pass control and start execution of the code on the dedicated SPE for each pthread. Each SPE performs *Nmc/Number_of_SPEs* MC steps, running the same code, only with different parameters passed by the PPE. After all SPEs finish their work, the final processing of gathered data is done on the PPE.

The main problem in a proper porting of the SPEEDUP code to the Cell architecture was missing Cell SPRNG code that can be compiled for the SPU. For this reason, we have compiled SPRNG for the PPE and performed all RNG operations only on the PPEs. This was done in parallel through several pthreads, distributed between both PPU processors of a QS22 Blade. Each pthread is associated with one of SPEs and synchronizes with it using mailbox technique, one of the simplest, hardware based, ways of communication within Cell CPU. The PPE mailbox checking is implemented through the interrupt, without active waiting (such as polling through the loop). Access to the main memory by all SPEs is realized through the Direct Memory Access (DMA) transfers. We have one initial transfer where control data from the PPE are received, one final transfer where computation results are sent back to the main memory and intermediate transfers of generated random numbers for each MC step. The XLC-compiled code was superior in the performance compared to the GCC-compiled code. The results for the XLC-compiled code are shown in Fig. 3.

As we can see, the fact that only PPEs are used for generation of random trajectories leads to a saturation of the performance when we increase the number of used SPEs to around 4. In the ideal case, when PPEs would be able to produce enough random trajectories for all SPEs, the simulation execution time would be around 250s, as can be seen in Fig. 3 for the code without random number generation). We also tested the code with the communication part disabled (no DMA memory transfers). From Table 2 we see that the communication does not have significant impact on the execution time and does not represent a bottleneck. To confirm this, we tested also the code that only generates random trajectories on PPEs, and observed the saturation in its performance at about 750s for the given *Nmc* number. This clearly corresponds to the minimal execution time for the full version of the Cell code in Fig. 3.

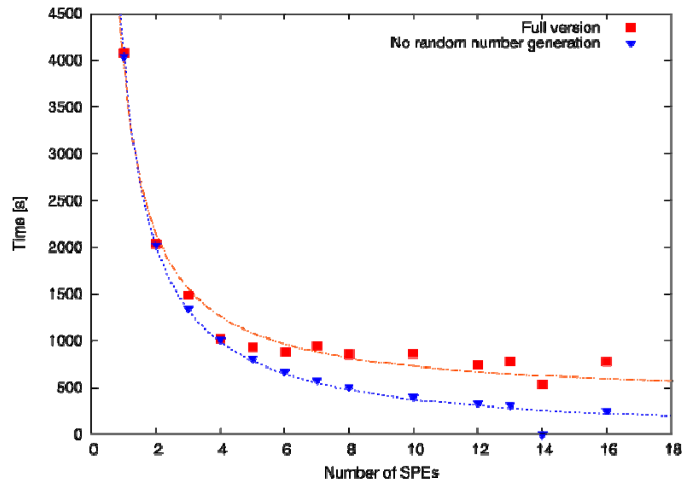

Fig. 3. Average times of execution of the Cell SPEEDUP code (full version) and for the code without generation of random numbers

Table 2. Average times of execution of the Cell SPEEDUP code without random trajectories generation and without PPE-SPE communication

Number of SPEs	No random trajectories generation	No communicati on
1	(4040±5) s	(4020±5) s
2	(2020±5) s	(2010±5) s
4	(1010±5) s	(1000±5) s
6	(675±5) s	(670±5) s
8	(505±5) s	(500±5) s
10	(405±5) s	(400±5) s
16	(255±5) s	(250±5) s

Therefore, as we can see, the missing implementation of the SPRNG library was limiting factor in fully utilizing the capabilities of all SPEs of the Cell Blade. This would not be the case if individual MC step calculation would require more time to complete, since then PPEs would be able to generate random trajectories at a sufficient rate. Such situation can be easily achieved e.g. if one uses higher effective action level *p* code. We have demonstrated similar situation in Fig. 4, where we have used unoptimized Cell SPEEDUP code, and where we observe perfect scaling of the code with the number of SPEs. Note that we used only 5120 MC samples for these tests since the code is now executed much slower.

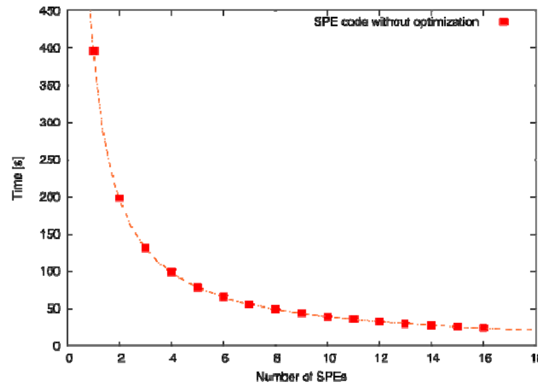

Fig. 4. Average times of execution of the Cell SPEEDUP code compiled without optimization

4.5. Comparison of hardware performance results

The overview of the obtained performance results for all tested hardware platforms is presented in Table 3. For Intel and POWER6 platform we give the results for the fully optimized threaded SPEEDUP code. For the Cell platform we give the minimal obtained execution time, as well as the execution time obtained with random trajectories generation disabled, which corresponds to the full utilization of all SPEs.

Table 3. Minimal average execution time per Blade of the fully optimized SPEEDUP code for each tested platform

Intel	POWER6	Cell	Cell Ideal
460s	250s	750s	250s

5. Conclusion

We have ported and optimized Path Integral Monte Carlo SPEEDUP code to three different computing architectures (Intel, POWER6 and Cell) and used the obtained code for benchmarking of these hardware platforms. For Intel and POWER6 platforms full optimization was obtained with the straightforward threaded version of the code, while the Cell platform required more complex changes of the code (implementation of separate PPE and SPE parts of the code). For benchmarking purposes we have also used different available compilers for each of architectures, and our results clearly show that platform-specific compilers always give much better performance.

The SPEEDUP code was most easily optimized on the POWER6 platform, where it also achieves superior performance (per Blade server) compared to all other hardware platforms. The Cell platform is demonstrated to be able to achieve the same level of performance in the case when individual MC steps take more time to complete. In the current implementation, due to the missing Cell SPRNG library, SPEEDUP code can fully utilize all Cell SPEs only for higher effective action levels p. The Intel platform shows also very good performance and excellent scalability, without any glitches for certain (odd or even) number of cores, observed on other platforms.

The plans for further development and testing include porting of SPRNG library to SPEs and implementation of platform-specific instructions (vectorization) for each tested platform.

Acknowledgements

This work was supported in part by the Serbian Ministry of Science, under project No. OI141035, and the European Commission under EU Centre of Excellence grant CX-CMCS. Numerical simulations were run on the AEGIS e-Infrastructure, supported in part by FP7 projects EGEE-III and SEE-GRID-SCI. The authors also acknowledge support by IBM Serbia.

References

[1] SPEEDUP, https://viewvc.scl.rs/viewvc/speedup/

[2] A. Bogojevic, A. Balaz, A. Belic, Phys. Rev. Lett. 94 (2005) 180403.

[3] A. Bogojevic, I. Vidanovic, A. Balaz and A. Belic, Phys. Lett. A 372 (2008) 3341-3349.

[4] A. Balaz, A. Bogojevic, I. Vidanovic and A. Pelster, Phys. Rev. E 79 (2009) 036701.

[5] D. M. Ceperley, Rev. Mod. Phys. 67 (1995) 279.

[6] http://sprng.cs.fsu.edu/

[7] ICC, http://software.intel.com/en-us/intel-compilers/

[8] POWER, http://www-03.ibm.com/technology/power/

[9] XLC, http://www-01.ibm.com/software/awdtools/xlcpp/

[10] Cell B/E, http://www-03.ibm.com/technology/cell/

[11] IBM SDK for Multicore Acceleration, http://www.ibm.com/developerworks/power/cell/