Optimization and Porting of the Path Integral MonteCarlo SPEEDUP Code to New Computing Architectures

V. Slavnić, A. Balaž, D. Stojiljković, A. Belić, A. Bogojević
Scientific Computing Laboratory
Institute of Physics Belgrade
Pregrevica 118, 11080 Belgrade, Serbia
http://www.scl.rs/

Abstract— **Many complex quantum physical systems can be most effectively described by the path integral formalism. The SPEEDUP code implements recently introduced analytical approach that systematically improves convergence of numerically calculated transition amplitudes of a generic quantum non-relativistic theory that leads to significant speedup of Path Integral Monte Carlo algorithms. In this paper we report on the optimization, porting and testing of the SPEEDUP code, done on several new computing architectures: IBM POWER6, PowerXCell and the latest Intel CPUs. We give overview of optimization techniques used and present results on the use of advanced features of new CPU types, which can be efficiently applied for the optimization of the SPEEDUP code.**

Keywords- Monte Carlo; SCL; SPEEDUP; Path Integral; Optimization; Porting;

I. 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.

II. 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 detailed 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 $\varepsilon_N=T/N$ 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 the 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.

III. 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 and a separate 1U server based on latest Intel Nehalem Xeon processors:

- HX21XM blade Server based on Intel Xeon technology. It features two Intel Xeon E5405 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 1+8 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.

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.

- Intel Server System SR1625UR based on latest Intel Xeon processors with Nehalem micro-architecture. Two quad-core Xeon X5570 processors are present within the system. These CPUs run on 2.93GHz with triple channel DDR3 memory subsystem with support of latest SSE4.2 extensions. They are equipped with 256 Kb of Mid-Level cache per core and 8MB of cache shared between cores (L3). With this micro-architecture Intel reintroduced its Hyper-Threading technology that supposed to enhance parallelization of computational tasks. Beside GCC, Intel ICC compiler, as for the other Intel system, was used for obtaining results of testing described in this paper.

IV. 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 *A* 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 *B* gives results for SPEEDUP MPI code tested on Intel platform, while Section *C* presents the threaded SPEEDUP code and results obtained with Intel and POWER architectures. In Section *D* we give results for the Cell SPEEDUP code, and in Section *E* we compare all results.

A. Serial SPEEDUP code

For Intel Xeon 5405 Blade server we compiled the serial code with GCC C compiler using optimization flag -O1, which turns to give the best performance. Along with GCC, we also used ICC compiler with optimization flag –fast, equivalent to the combination -O3 -xHOST -ipo -no-prec-div -static.

Intel Nehalem platform shows best results with GCC flags -O1 –funroll-loops (loop unrolling), and with the -fast flag for Intel's ICC.

On POWER6 and Cell Blades the code was compiled with both GCC and IBM XLC compiler. On Cell Blade we used the flags -O1 -funroll-loops with GCC, and with XLC flags -O5 -qaltivec -qenablevmx. Appropriate versions of GCC and XLC binaries were used (ppu-gcc and ppuxlc). On POWER6 Blade -q64 -O5 –qaltivec -qenablevmx flags were used with XLC and -O3 -funroll-loops with GCC. Results for the serial program benchmarking are presented in Table 1.

Platform/Compiler	GCC	ICC	XLC
Intel Xeon 5405	(6280±20) s	(1600±20) s	-
Intel Nehalem	(3520±10) s	(920±10) s	-
POWER6	(8980±10) s	-	(1830±10) s
Cell	(25350±50) s	-	(12550±50) s

Table 1: Average times of execution of a serial SPEEDUP code on all tested platforms with different compilers. The flags used are given in the text.

Table 1 demonstrates the significant increase in the speed of the execution of the code when platform-specific compiler is used. New Nehalem platform in conjunction with ICC compiler gives the best performance compared to all other platforms. 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 additional available SPEs are used.

B. MPI SPEEDUP code

On the Intel Blade Xeon 5405 and Intel Nehalem platform, we tested the performance of the SPEEDUP code with MPI implementation, compiled by the ICC compiler with -fast flag. Also we tested the behavior of Nehalem CPUs with Hyper-Threading feature enabled and disabled. The results are shown in Figure 1.

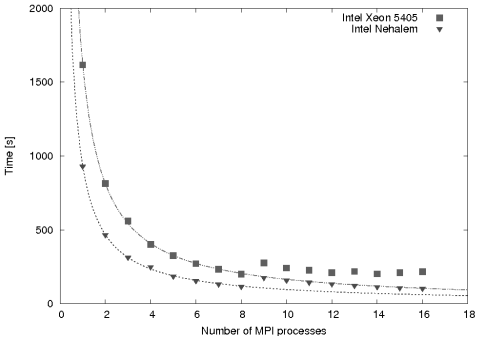

Figure 1: Average times of execution of the MPI SPEEDUP code on Intel Xeon 5405 and Intel Nehalem platforms compiled with ICC (-fast flag). The curves give fits to the expected dependence A + B / (Number of MPI processes).

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 (eight), the operating system is trying to distribute the load among already fully loaded cores, which creates additional overhead. This is less pronounced at the Nehalem platform, with the Hyper-Threading enabled. In that case, as shown in Figure 2, slightly better results are achieved when the numbers of MPI instances exceeds the number of physical cores. Below this threshold the results are identical.

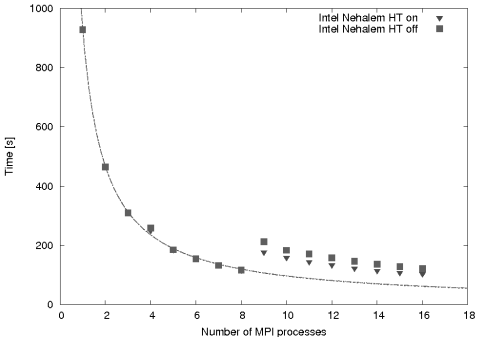

Figure 2: Average times of execution of the MPI SPEEDUP code on Intel Nehalem platform compiled with ICC (fast flag) with Hyper-Threading technology enabled (HT on) and disabled (HT off).

MPI implementation gives minimal execution time of around 100s for Nehalem platform and around 200s for Intel Xeon 5405.

C. Modified SPEEDUP code

To fully optimize the parallel SPEEDUP code, instead of using MPI API, we implemented its threaded version using the POSIX threads (pthreads). Each thread calculates *Nmc/Nth* of Monte Carlo samples, where *Nth* represents the number of initiated threads. Also, some minor additional modifications of the code were performed, focusing on specific improvements for *p*=9 effective action. The Intel version of the code was compiled with ICC, while the POWER version was compiled with XLC. The obtained numerical results are summarized in Figure 3.

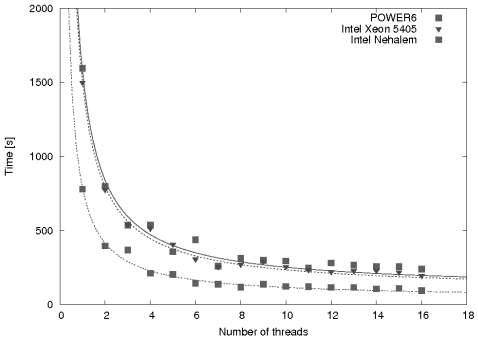

Figure 3: Average times of execution of the threaded SPEEDUP code on Intel and POWER6 platforms.

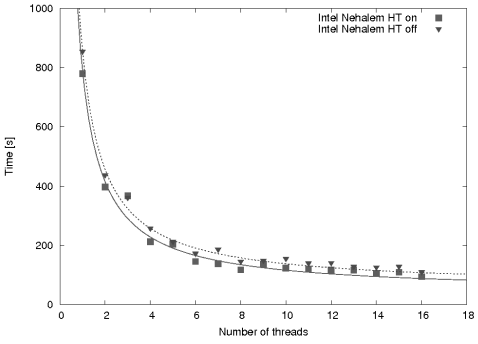

Figure 4: Average times of execution of the threaded SPEEDUP code on Intel Nehalem platform with Hyper-Threading enabled (HT on) and disabled (HT off).

With the threaded code we obtained non-negligible increase in the speed of the code compared to previous implementations. Again, Intel Nehalem with the ICC compiler was much faster than all other platforms. If we compare the increase in the speed gained by implementing the threaded code, the POWER6 platform shows a 12% performance gain (threaded vs. the serial code), while we get around 6% gain for Intel platforms (threaded vs. MPI code).

The minimal execution time with the threaded code was 190s on Intel Xeon 5405 Blade, 95s on Intel Nehalem and 235s on the POWER Blade. Again, we can see a small impact on the execution speed when Hyper-Threading technology is enabled on the Intel Nehalem CPUs (Figure 4.).

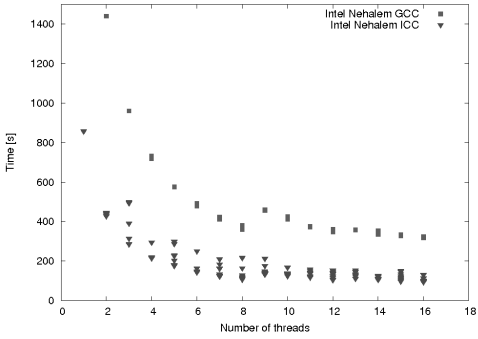

Figure 5: Times of execution of the threaded SPEEDUP code on Intel Nehalem platform with ICC and GCC compiler.

We have also observed an interesting behavior on Intel platforms, which is presented in Figure 5. Although the threaded code gives better performance when compiled with ICC compared to the code compiled with GCC, the times of execution of the ICC-complied code for the same parameters and the same number of threads differ significantly for several consecutive runs. Such relatively large scattering of execution times around the average might be accredited to the low-level hardware implementation details of Intel CPUs, as well as to the aggressive optimization techniques used by the –fast flag. On the other hand, the execution of the same code compiled with GCC did not exhibit such behavior. This might point to the load-balancing issues when aggressive optimization is used with ICC, while GCC is not able to achieve such level of optimization and thus is not affected. The similar behavior was also observed on Intel Xeon 5405 platform.

D. 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 serial sections to be executed on the PPE from the parallel sections that can be executed on as many SPEs as available in the system. Our main implementation idea 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* Monte Carlo 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 library code that can be compiled and executed on each SPU. For this reason, we have compiled SPRNG for the PPE and performed all random number generation operations only on PPEs. This was done in parallel through several pthreads, distributed between both PPE processors of a QS22 Blade. Each pthread was associated with one of SPEs and synchronized with it using the mailbox technique. It is 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 Figure 6.

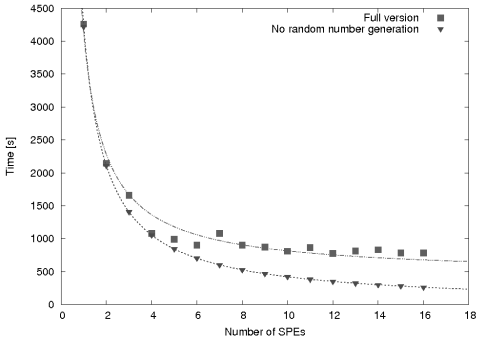

Figure 6: Average times of execution of the Cell SPEEDUP code. For comparison, we also give execution times for the code without generation of random numbers.

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. After that, PPEs are not able to generate random numbers sufficiently enough, and further increase in the number of SPEs used does not lead to any improvement in the performance. In the ideal case, if PPEs would be able to produce enough random trajectories for all SPEs, the simulation execution time would be around 260s, as can be seen in Figure 6 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 770s for the given number of Monte Carlo samples *Nmc*. This clearly corresponds to the minimal execution time for the full version of the Cell code in Figure 6.

Number of SPEs	No random trajectories generation	No communication
1	(4220±5) s	(4200±5) s
2	(2110±5) s	(2100±5) s
4	(1055±5) s	(1050±5) s
8	(530±5) s	(525±5) s
16	(265±5) s	(260±5) s

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

Therefore, as we can see, the missing implementation of the SPRNG library was a limiting factor in fully utilizing the capabilities of all SPEs of the Cell Blade. However, this problem would not be even seen in the case when individual MC steps take more time to finish their calculation, 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 [1-4]. We have demonstrated similar situation in Figure 7, where we have used unoptimized Cell SPEEDUP code, and where we observe perfect scaling of the code with the number of SPEs. Disabling the optimization leads to a much slower execution of the code, and each MC step takes much more time to be completed, thus giving enough time to PPEs to generate needed random trajectories according to the bisection algorithm. This also demonstrates the fact that specific details of the optimal porting of an application to the Cell architecture can significantly depend on the execution run-time parameters. Such situation is not frequently encountered on other computing platforms, and is here due to the current limitations of the Cell SPUs, as well as limitations in their communication model.

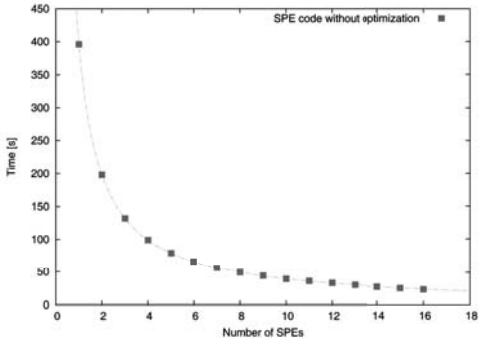

Figure 7: Average times of execution of the Cell SPEEDUP code compiled without optimization. The number of Monte Carlo steps is decreased to *Nmc*=5120, since the code without optimization is executed much slower.

E. 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.

Intel Xeon 5405	Intel Nehalem	POWER6	Cell	Cell ideal
190s	95s	235s	770s	260s

Table 3: Minimal average execution time for each tested platform for the fully optimized SPEEDUP code. For each platform we have selected the optimal implementation.

The difference in performance of two tested Intel platforms can be partially explained by the higher clock frequency of 2.93 GHz for the Nehalem CPU, compared to only 2.0GHz frequency for the Intel Xeon 5405. However, even if we rescale the performances of both platforms to the same frequency, we still see a 30% better performance of the Nehalem platform. Such significantly better performance is due to the improved architecture of the newer CPU.

V. CONCLUSIONS

We have ported and optimized Path Integral Monte Carlo SPEEDUP code to four different computing architectures (Intel Xeon 5405, Intel Nehalem X5570, IBM 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 sections 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 both Intel platforms, especially on Intel Nehalem where it achieves superior performance compared to all other hardware platforms. Contrary to our expectations based on previous experiences with the Hyper-Threading technology, it did not improve the performance of the code significantly.

The Cell platform is demonstrated to be able to achieve respectable 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*. However, the tested Cell CPU is no match for POWER6 or latest Intel CPUs. The Nehalem platform also significantly outperforms POWER6 CPU, despite its much higher frequency of 4.0 GHz.

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.

ACKNOWLEDGMENTS

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 and Intel Corporation (UK) Ltd.

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] POWER, http://www-03.ibm.com/technology/power/
[10] Cell B/E, http://www-03.ibm.com/technology/cell/
[11] IBM SDK for Multicore Acceleration, http://www.ibm.com/developerworks/power/cell/