Efficiency analysis of a physical problem: Different parallel computational approaches for a dynamical integrator evolution.

Adriana Gaudiani[1], Alejandro Soba[2,3], and M. Florencia Carusela[1,3]

[1] Instituto de Ciencias, Universidad Nacional de General Sarmiento, Los Polvorines, Argentina
agaudi@ungs.edu.ar
[2] Comisión Nacional de Energía Atómica
Buenos Aires, Argentina
[3] Conicet, Argentina

Abstract. A great challenge for scientists is to execute their computational applications efficiently. Nowadays, parallel programming has become a fundamental key to achieve this goal. High-performance computing provides a solution to exploit parallel architectures in order to get optimal performance. Both parallel programming model and the system architecture will maximize the benefits if both together are suitable to the inherent parallelism of the problem.
We compared three parallelized versions of our algorithm when applied to the study of the heat transport phenomenon in a low dimensional system. We qualitatively analyze the obtained performance data based on the own characteristics of multicore architecture, shared memory and NVIDIA graphical multiprocesors related to the traditional programing models provided by MPI and OpenMP, and Cuda programming environment.
We conclude that GPUs parallel computing architecture is the most suitable programing model to achieve a better performance of our algorithm. We obtained an improvement of 15X, quite good for a program whose efficiency is strongly degraded by an integration process that essentially must be carried out in a serial way due to the dependence of the data.

1 Introduction

To analyze the dynamic evolution of a real system requires the development of a model that should include among other things the characteristic times of the phenomenon under study. Generally, these models are expressed as differential equations. Depending on the type, these equations are integrated using different numerical integration methods. This requires discretizing the equations, where the length of the integration step δT must be directly related to relevant timescales of the real physical problem, and the total time $t = \delta T.T$ (T total number of integration steps) must correspond to times longer than the typical duration of the phenomenon under study. This is a necessary condition to achieve an adequate insight of the problem. However, t has not necessarily a direct correlation with the real time of the simulation, which depends on the characteristics of the algorithm implemented.

In particular, parallel algorithms discussed in this paper are applied to study the phenomenon of heat transport in one-dimensional devices immersed in thermal environments. A serial execution of this algorithm is executed in times measured in days. Thus in order to optimize the computational resources and to reduce the large execution times required in the case of a simple serial integration, we propose different parallel implementations. We present schematically the main idea in Figure 1

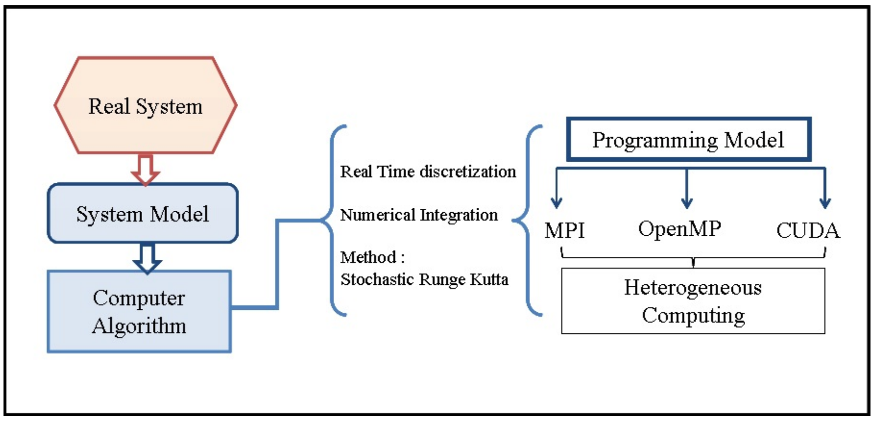


Fig. 1. The major steps from problem to a parallel computational solution.

In this paper, we describe the performance results that we obtained of our parallel algorithm implementation, using three parallel programming models and giving an overview of the heat algorithm behavior in each of our parallel implementations. They are message passing model with MPI on distributed memory systems, shared memory with OpenMP on multicore systems and finally, data parallelism on graphic processing units (GPUs) in a single instruction-multiple thread (SIMT) architecture.

Organization: This paper is organized as follows. In section 2 is given an overview of heat transport algorithm and parallel programming models used to improve the algorithm, along with the desired features on applications that will benefit from these High Performance Computing Systems (HPCS). In section 3 we discuss the performance of our parallel implementations and compare the programming effort and resulting performance. In section 4 and 5 we present the experimental results and conclusions of our work respectively.

2 Background

In this section we first present the general features of the heat transport algorithm, then we highlight those desired aspects for efficient execution of parallel algorithms as a function of the target architecture.

2.1 The heat transport algorithm

The device is modeled with two chains of $N/2$ atoms with harmonical nearest neighbors interactions with a strengh constant $ki, i = 1, N + 1$. x_i denotes the displacement from the equilibrium position of each particle. The system properties are obtained in the stationary regime, that is when the system thermalize. If the integration is made with the stochastic Runge-Kutta (SRK) [3] algorithm and for the chosen time step (discussed below), the last condition is fulfilled for $T > 10^8$ integration steps [1]. Moreover, we are also interested in the effect that the size system N has on the thermal properties of the physical system. But, as the thermalization time also depends on the number of atoms N, the size study requires to increase the integration time and the memory resources.

The dynamical equations can be written in a non dimensional form as:

$$\begin{aligned} &\ddot{x}_i = F(x_i) + k_i(x_{i+1} + x_{i-1} - 2x_i); \qquad i = 2...N-1 \\ &\ddot{x}_i = F(x_i) + k_i(x_{i+1} + x_{i-1} - 2x_i) - \gamma \dot{x}_i + \sqrt(2\gamma K_B T_{L,R})\xi_i(t); \qquad i = L, R \end{aligned} \quad (1)$$

$\dot{x}_i, \ddot{x}_i$ are the velocity and acceleration of the atom i respectively, $F(x_i)$ is a periodic on-site potential that model a substratum and the last term corresponds to the harmonic interaction.

The system is driven out of equilibrium by two mechanisms:

a) The ends of the segments are in contact with Langevin type thermal reservoirs with zero mean and variance $< \xi_{L/R}(t), \xi_{L/R}(t') > = 2\gamma K_B T_{L,R} \delta(t - t')$, where γ is the strengh of the coupling between the system and the baths. The temperatures of the L/R (left/right) thermal baths are simultaneously modulated in time with frequency ω_{temp}: $T_{L,R}(t) = T_{0,i}(1 + \Delta sgn(sin(\omega_{temp} t))$, $i = L, R$, where $T_{0,i}$ is the reference temperature of each reservoir and $T_{0,i}\Delta$ is the amplitude of the modulation. Only the atoms at the ends are inmersed in the thermal reservoirs.

b)The modulation of the coupling between the two segments is given by $k_{N/2}(t) = K_0(1 + sin(\omega_K t))$, with frequency ω_K.

The dynamical evolution is obtained integrating the system given in Eq.1 with a stochastic Runge-Kutta algorithm (SRK), with an integration steps $\delta T = 0.005$. This time step corresponds to a physical time greater than the relaxation time of the reservoirs.

The thermal properties of the system are obtained in the stationary regime, that is when the system thermalize. This condition is fulfilled if the integration with the SRK method and the chosen time step is made for $T > 10^8$ integration steps. More over, we are also interested in the size effect (N) on the thermal properties. However as the thermalization time also depends on the number of oscillators N, this study requires an increase of the integration time and the memory resources.

We tailor the heat transport algorithm for the different parallel computing architectures, taking into account that for every time stage of the numerical method, the data for the x_i element depends on x_{i-1} and x_{i+1} at the same time. These interactions are first neighbors type and they are schematically shown in Figure 2

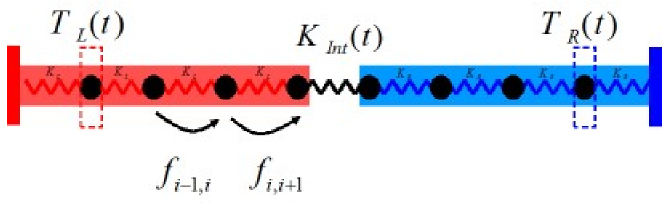


Fig. 2. Schematic diagram of the physical model

For the purpose of study the system behavior we must run several scenarios by combining different values of frequency (ω_K) and temperature (ω_{temp}) parameters. This means investing a longer execution time than for a single pair $(\omega_K, \omega_{temp})$ of parameters. We must consider that the complexity order of the heat algorithm for a single pair is $O(T * 2^N)$, as shown by the author of this paper [4]. Notwithstanding we use chains of N=256 to N=2048 oscillators for this work, we need to increase these values in the physical problem.

2.2 Parallel Programming Overview

Developing a parallel application is strongly conditioned by the system on which this will be deployed and the programming model chosen. But the choice of the model is made in terms of the available parallel computational resources and the type of parallelism inherent in the problem. In parallel computation the most common alternatives today are message passing, data parallelism and shared memory. In the following, we describe some features of the three standard parallel programming models[2].

MPI is a standard for parallel programming on distributed memory systems, as are clusters. MPI communication libraries provide an interface for writing message passing programs. The most important goals of this model are: achievable performance, portability and network transparency. However, some MPI applications do not scale when the problem size is fixed and the number of core is increased, also they perform poorly when require large shared memory. Our MPI program uses message passing for communications and it was designed in a Single Program Multiple Data (SPMD) way as parallel paradigm. SPMD applications compute a set of tasks and then communicate the results to their neighbors. Just as in our application, tasks need information of their neighbors before proceeding with the next iteration.

The two major hardware trends impacting the parallel programming today are: the rise of many-core CPU architectures and the inclusion of powerful graphics processing units (GPUs) in every desktop computer [6].

OpenMP is a portable approach for parallel programming on shared memory systems that offer a global view of application memory address space, helping to facilitate the development of parallel programs. OpenMP on shared memory systems has become a solution to solve a wide range of large-scale scientific problems which can be solved in parallel to decrease the execution time [8].

GPUs are an inexpensive commodity technology that is already ubiquitous. These technology is highlighted by massively many-core multiprocessors and data level concurrency. It provides a general purpose parallel computing architecture, in the case of modern NVIDIA GPUs it is called Compute Unified Device Architecture (CUDA). In general purpose computing the GPU is used as CPU co-processor in order to accelerate a specific computation. CUDA enables to divide the parallel program execution in tasks that can run across thousands of concurrent threads mapped to hundreds of processor cores. The application benefit with GPUs parallelism when code is written in a Single Instruction Multiple Data (SIMD) mode, this means many elements can be processed in lockstep running the exact same code [9].

3 Parallel Implementations

Our experiences were carried out in a 56 nodes multicluster Intel(R) Dual Core Xeon(TM) 5030 of 2.66GHz processors with a infiniband switch, a multicluster 32 cores Intel Xeon(R) E5-2680 of 2.70GHz and 20MB L2 cache and 64 GB of RAM memory and a GPU Geforce GTX 560TI Fermi GF114 with capability 2.1 and 1Gb de RAM.

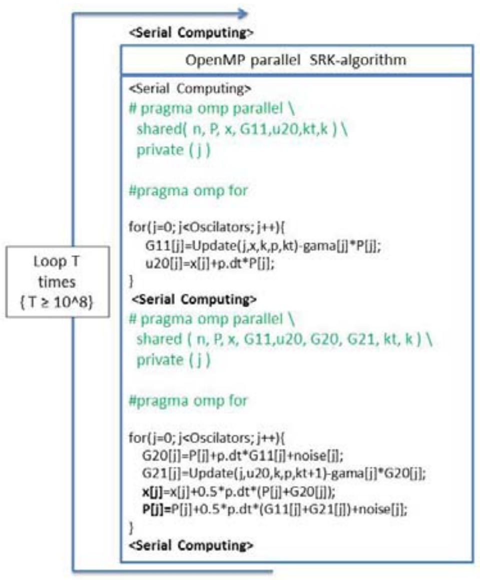
The main computation in the heat transport problem is the SRK algorithm. The numerical method simulates the system evolution over time and it computes each oscillator status at each iteration. This type of computational approach forces a serial integration with no possibility of distributing task between several processes. As we need $T > 10^8$ time steps and oscillator chains with more than 2000 elements, it becomes crucial to minimize the SRK execution time. We address this problem by writing the three versions of our algorithm and then evaluate the performance achieved.

Our first approach was a MPI parallelization when we had to deal with inter-process communication to reduce the overhead. The SPMD implementation works with a simple mapping method by assigning identical amount of data - chain elements - for each process. The features inherent to the problem are the cause of a compulsory data transfer between the neighbours in the chain, because in each integration steps we need data of the last step. This fact is responsable of the bad performance of the MPI implementations (not showed here), where a severe degradation of the speed-up even with two process are obtained, no matters the increment up to N=2000 and despite of the low latency of the network (¡180ns). Expected result, because the kind of algorithms used and to the large number of explicit time steps needed, is neccesary mention that the amount of time needed to perform a complete integration for the whole set is aproximately 0.3 ms.

The next stage was to develop a parallel program for a shared memory architecture with OpenMP. It was a fairly simple task taking advantage that data are read from memory shared by all processes. By avoiding inter-processes message passing we reduced the overhead and got a good performance, as explained in the next section. The pseudocode is shown in Figure 3. The difficulty with OpenMP is that it is often hard to get decent performance, especially at large scale[5].

Finally we present our GPU implementation for the integrator SRK algorithm. We wrote a CUDA kernel to carry out the parallel version taking into account the data dependencies inherent to the problem. The thread tasks are represented by the kernel code and kernels are mapped over a structured grid of threads. The program structure is sketched in Figure 4.

The whole SRK operations performed on each chain element are carried out concurrently by the kernel code launched on every thread. The threads are mapped to processors depending on the grid configuration, that is how many blocks and threads per block are specified when invoking the kernel. It should be noted that the results of the intermediate steps need to be copied to host



```
<Serial Computing>
<Serial Computing>
# pragma omp parallel \
  shared( n, P, x, G11,u20,kt,k ) \
  private ( j )

#pragma omp for

for(j=0; j<Oscilators; j++){
   G11[j]=Update(j,x,k,p,kt)-gama[j]*P[j];
   u20[j]=x[j]+p.dt*P[j];
}
<Serial Computing>
# pragma omp parallel \
  shared ( n, P, x, G11,u20, G20, G21, kt, k ) \
  private ( j )

#pragma omp for

for(j=0; j<Oscilators; j++){
   G20[j]=P[j]+p.dt*G11[j]+noise[j];
   G21[j]=Update(j,u20,k,p,kt+1)-gama[j]*G20[j];
   x[j]=x[j]+0.5*p.dt*(P[j]+G20[j]);
   P[j]=P[j]+0.5*p.dt*(G11[j]+G21[j])+noise[j];
}
<Serial Computing>
```

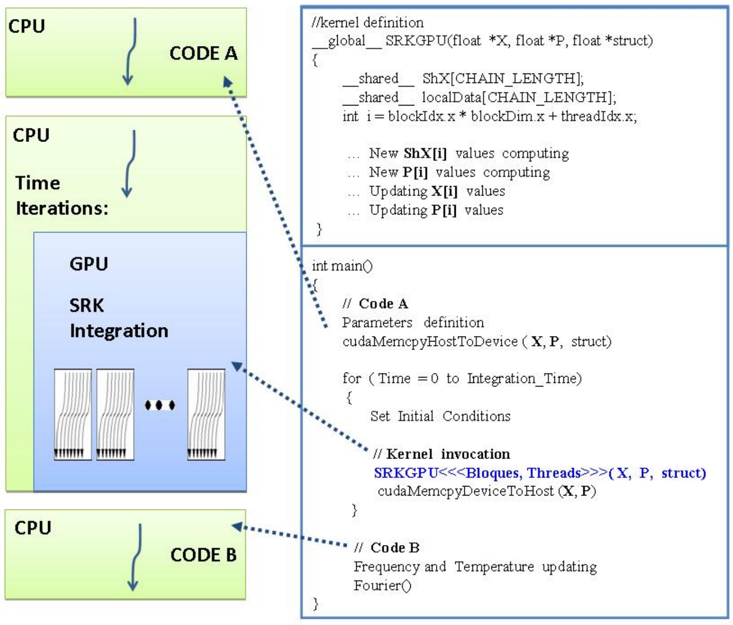
Fig. 3. OpenMP pseudocode program

memory. This copy is performed in ocassion to store partial results, and it is an overhead source. We used Occupancy Calculator spreadsheet to select the best data layout and to maximize Stream Multiprocessors occupancy [7]. In our program, we achieve the best performance when each block has 256 threads.

4 Performance Evaluation

In this section we evaluate the computational performance of heat transport algorithm on the three platforms.

1. MPI parallel algorithm: we analize the system behavior for a number of oscillators between 256 up to 2000. As we mentioned in the earlier section a dramatic performance degradation with the increase in number of processors is visible. Due to the domain division and the requirement in the integration process of interchange the information related to first neighbors in the chain, the amount of communications increases as a result of continually updating the state of those elements. As a consequence of this the performance is not better even in case of add more particles in the chain, because the bottle neck in the code still are the neighbors communications and this number



```
//kernel definition
__global__ SRKGPU(float *X, float *P, float *struct)
{
      __shared__  ShX[CHAIN_LENGTH];
      __shared__  localData[CHAIN_LENGTH];
      int i = blockIdx.x * blockDim.x + threadIdx.x;

       ... New ShX[i] values computing
       ... New P[i] values computing
       ... Updating X[i] values
       ... Updating P[i] values
}
int main()
{
      // Code A
      Parameters definition
      cudaMemcpyHostToDevice ( X, P, struct)

      for ( Time = 0 to Integration_Time)
      {
           Set Initial Conditions

            // Kernel invocation
            SRKGPU<<<Bloques, Threads>>>( X, P, struct)
             cudaMemcpyDeviceToHost (X, P)
        }
        // Code B
        Frequency and Temperature updating
        Fourier()
}
```

Fig. 4. High Level Structure of CUDA program.

increases with the numbers of domains used. The numerical integrator SRK does not allow the possibility of parallelize the main loop, that is why the performance of this technique is limited to only a few processors. Comparing the estimated latency time (aprox. 180 ns) and the integration of a one single time step of the whole set (aprox. 0.3ms) we are sure than the introduced overhead in the comunication is responsable of this degradation in the speed-up.

2. OpemMp parallel algorithm: We used two multicores cluster, a 32-Core IntelXeon and a 8-Core Xeon. Figure 5 displays the speed-up achieved in both systems for 256 and 1024 chain elements. In this approach, data structures are allocated in shared memory and stay there for every loop during the execution, so we launched a team of threads to parallelize the SRK algorithm. We parallelized those loops in which computation is independently of one another, but still endures the overhead imposed by the serialization of the integration process. The process of updating oscillators states at each iteration was benefited with this programming model. We get a maximum efficiency of 41% for 8 processors and 256 chain oscillators and 66% for 1024 oscillators. The scheduling scheme was delegated to the compiler and runtime system, the results were similar to static and dynamic scheduling.

Figure 5

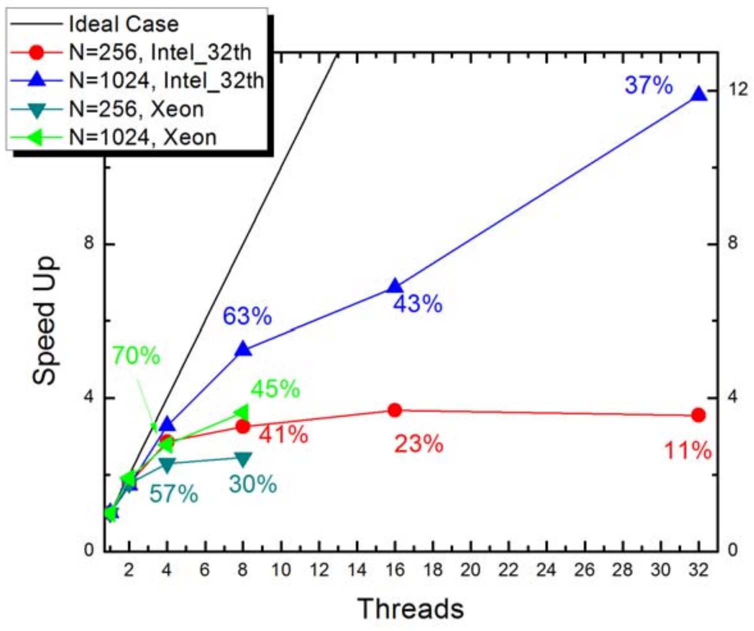


Fig. 5. OpenMP implementation results. The percentages represent the quality of the result with respect to the ideal case i.e. the amount of resources used.

3. As can be seen in Figure 4, its central part, CUDA SRK kernel is invoked within the main loop, running in CPU, and mapped on threads configuration. This main loop is governed by the number of integration steps. It means, 10^8 times or more. We were able to achieve a significant improvement in overall system performance. The timed computation is represented in the central part of the figure, including both CPU main loop and memory transfer overhead between memory RAM and GPU device. Chain oscillator interactions as we see in Figure 2 represent a problem to solve. So, we used shared memory into kernel code when accessing data structures to improve performance. The maximum speedup achieved is 15X for 8192 oscillators, and it was measured as the ratio between serial and parallel time. Serial run was done on a CPU Intel 8 Core i7-2600 - 3.4 GHz. Kernels execution time remain constant between 16 to 256 oscillators, then between 1024 and 8192 oscillators.The warps threads don't diverge and they keep maximum GPUs

occupancy. In Figure 6, we show this results when the kernel is launched with 256 threads per block. As can be seen, we increase the load, but time does not increase, and we can process 8192 oscillators at the same time as 2048!, meanwhile CPU execution time grows exponentially with load. For this reason, GPU approach is a very good choice.

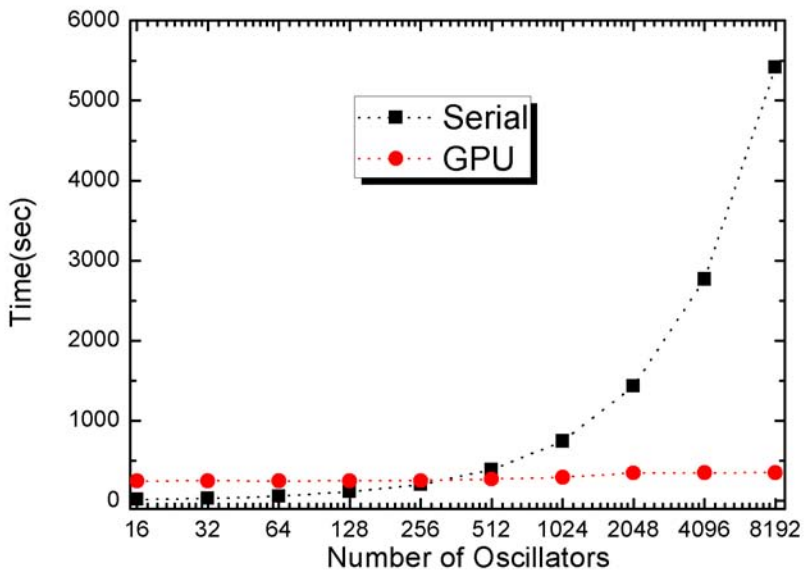


Fig. 6. GPU and Serial runtimes. Results represent execution time, using Time Steps=8000000 and Threads per Block=256.

5 Conclusions and Future Work

This work has applicability in the modelling of many low-dimensional physical systems and technological applications in different scales. In particular our proposal is the study of the heat transport phenomenon along one-dimensional nanodevices. Despite the model studied here is rather simple, it is usually a good first approach to achieve a qualitative and quantitative physical insight of the phenomenon of energy transfer. On the other hand, the computational model of this problem consumes a high runtime, prompting the use of parallel computing resources.

We have shown the suitability of commodity GPUs and a parallel CUDA-based algorithm for solving this problem, in particular when long chains are considered. The MPI implementation is inefficient in this case. We are working to further improve these results taking advantage of the availability of hibrid computers for high performance computing.

6 Acknowledgment

The authors thank to CECAR - FCEN - UBA and III-LIDI- Facultad de Informtica - UNLP for the opportunity to have access to a 32-Core IntelXeon and a 8-Core Xeon in CECAR and a BLADE multicore Xeon(TM) 5030 and a GPU Geforce GTX 560TI in III-LIDI.

References

1. Beraha N., Barreto R., Soba A., Carusela M.F. in preparation.
2. Dongarra J., Sterling T., Simon H., Strohmaier E., High-performance computing: clusters, constellations, MPPs, and future directions. Journal of Computing in Science & Engineering, **7** (2005) 51–59
3. Honeycutt, Rebecca L., Physical Review A (Atomic, Molecular, and Optical Physics), Volume 45, Issue 2, January 15, (1992), pp.600-603
4. Januyszewski M., Kostur M., Accelerating numerical solution of stochastic differential equations with CUDA. Computer Physics Communictions. Elsevier. **181** (2010) 183–188
5. Jin H., Jespersen D., Mehrotra P., Biswas R., Chapman B. High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Computing. Elsevier. **37** (2011) 562-575
6. Lobachev O., Guthe M., Loogen R. Estimating parallel performance. Journal of Parallel and Distributed Computing. Elsevier. **73** (2013) 876-887
7. Nickolls J., Dally W.: The GPU Computing Era. IEEE Micro. (2010) 56–69
8. Muresano R., Rexachs D., Luque E. How SPMD applications could be efficiently executed on multicore environments?. IEEE International Conference on Cluster Computing and Workshops. (2009)
9. Schenk O., Christen M., Burkhart H.: Algorithmic performance studies on graphics processing units. Parallel Distributed Computing. Elsevier. **68** (2008) 1360–1369