Bull. Nov. Comp. Center, Comp. Science, 30 (2010), 15–25
© 2010 NCC Publisher

Parallel simulation of asynchronous cellular automata on different computer architectures*

K. Kalgin

Abstract. An overview and experimental comparative study of parallel algorithms of asynchronous cellular automata simulation are presented. The algorithms are tested for physicochemical process model of the surface reaction $CO + O_2$ over the supported Pd nanoparticles on different parallel computers. For the testing, we use shared memory computers, distributed memory computers, i.e., clusters, and a graphical processing unit. Characterization of these algorithms in terms of the methods of parallelism maintenance is given.

1. Introduction

Asynchronous cellular automata (ACA) are used for simulation of physical and chemical processes on molecular level, for example, for studying oscillatory chemical surface reactions [1, 2], absorption, sublimation and diffusion of atoms in the epitaxial growth processes [3]. Simulation of natural processes requires a huge cellular space and millions of iterative steps for obtaining a real scene of the process. Therefore, it requires high computing costs. The fact is, ACA cannot be parallelized so easily as synchronous cellular automata (SCA). As distinct from SCA, the ACA functioning is a sequential application of a transition rule to randomly selected cells. The cells are selected with equal probabilities and irrespective of the process history.

Parallelization of ACA is performed by the domain decomposition method: each process hosts its own domain of cells and stores copies of boundary cells of neighboring processes. A parallel algorithm simulation should preserve the behavioral properties of ACA: *Independence*, *Fairness*, *Correctness*, and *Efficiency*. Independence means an independent selection of cells in the course of simulation. Fairness means that different cells are selected with equal probabilities. Correctness means deadlock-absence and coherence of boundary cell states and corresponding copies in different processes. Efficiency implies that t_k is less than t_1 for a certain k. Here t_k is the total time of parallel algorithm execution on k processors, and t_1 is the total time of sequential algorithm execution on one processor.

*Supported by Presidium of Russian Academy of Sciences, Basic Research Program No. 2 (2009), and Siberian Branch of Russian Academy of Sciences, SBRAS Interdisciplinary Project 32 (2009).

There are several parallel algorithms of the ACA simulation on computers with different architectures. In [4], an algorithm suitable only for shared memory computers only is proposed. Parallel algorithms for distributed memory computers are presented in [5, 6]. In addition, [7] and [8] describe a practical approach to parallel simulation of ACA, where a given ACA is transformed to a synchronous one, called block-synchronous cellular automaton (BSCA) that approximates its evolution and also provides easy parallelization.

This paper presents a comparative study of the above-mentioned parallel algorithms and their efficiency on computers with different architectures. Section 2 gives a formal definition of ACA. Section 3 outlines the main ideas of these algorithms and briefly characterizes them with respect to the above four properties. In Section 4, the following parallel computer systems are overviewed: shared memory computers, distributed memory computers (i.e., clusters), and a graphical processing unit (GPU) supporting CUDA [9]. These systems are used for testing implementations of the algorithms in question. Section 5 describes an ACA model of the physicochemical process of the surface reaction $CO + O_2$ over the supported Pd nanoparticles [2]. The typical sizes of cellular array for a model are from 100×100 to $10,000 \times 10,000$. A usual simulation process with a 1000×1000 cellular array takes more than twenty four hours on one core of Intel Core i7. Also, this section presents the results of testing the above-mentioned parallel algorithms implemented for this model on different parallel architectures. All the tested combinations of computer architectures and parallel algorithm implementations are presented in Table 1. We do not present results obtained for the algorithm given in [5] as far as additional costs of the parallel computations maintaining are too large for the model under consideration.

Table 1. The tested combinations of computer architectures and parallel algorithm implementations

Reference	Shared memory	Distributed memory	Graphical processing unit
[4]	+	−	−
[6]	+	+	−
[7, 8]	+	+	+

2. Asynchronous cellular automata

An asynchronous cellular automaton is specified by the tuple $\langle Z^d, A, \Theta \rangle$, where Z^d is a finite set of *cell coordinates*, A is an *alphabet*, i.e. a finite set of cell states, and Θ is a transition rule.

A pair $(\boldsymbol{x}, a) \in Z^d \times A$ is called a *cell*, where $a \in A$ is a state of a cell and $\boldsymbol{x} \in Z^d$ are its coordinates.

A set of cells $\Omega \subset Z^d \times A$ is called a *cellular array* if there does not exist a pair of cells with equal coordinates and $\{\boldsymbol{x} \mid (\boldsymbol{x}, a) \in \Omega\} = Z^d$. Since between the cells in a cellular array and their coordinates there exists a one-to-one correspondence, we will further identify each cell with its coordinates.

The *transition rule* Θ is a probabilistic function:

$$\Theta : A^{|T|} \to A^{|T|},$$

where the *template* T is a set of *naming functions* $\phi_i : Z^d \to Z^d$, $T = \{\phi_1, \phi_2, \ldots, \phi_{|T|}\}$. The template determines the *neighborhood* of a cell $\boldsymbol{x}$:

$$T(\boldsymbol{x}) = \{\phi_1(\boldsymbol{x}), \phi_2(\boldsymbol{x}), \ldots, \phi_{|T|}(\boldsymbol{x})\}.$$

Further we use a 2D rectangular space Z^2:

$$Z^2 = \{(i, j) \mid 1 \le i \le N_x,\ 1 \le j \le N_y\}.$$

We use the templates T_1, T_5, and T_{13}, where

$$\begin{aligned} T_k(\boldsymbol{x}) &= \{\boldsymbol{x} + \boldsymbol{v}_0, \boldsymbol{x} + \boldsymbol{v}_1, \ldots, \boldsymbol{x} + \boldsymbol{v}_{k-1}\}, \\ V &= \{\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_{12}\} = \{(0,0), (0,1), (1,0), (0,-1), (-1,0), \\ &\quad (1,1), (1,-1), (-1,1), (-1,-1), (0,2), (2,0), (0,-2), (-2,0)\}. \end{aligned}$$

An *application of the transition rule to the cell* $\boldsymbol{x}$ results in updating the neighboring cells $T(\boldsymbol{x})$ with the new states $\Theta(T(\boldsymbol{x}))$.

As usual, the transition rule can be expressed as *substitution* or as *composition* of several transition rules. The most widespread rules of composition are *random execution* (R), *sequential execution* (S), and *randomly ordered sequential execution* (RS). These rules can be given by formulas:

$$\Theta_R = R(\Theta_1, p_1; \Theta_2, p_2; \ldots; \Theta_n, p_n), \tag{1}$$

$$\Theta'_R = R(\Theta_1, \Theta_2, \ldots, \Theta_n), \tag{2}$$

$$\Theta_S = S(\Theta_1, \Theta_2, \ldots, \Theta_n), \tag{3}$$

$$\Theta_{RS} = RS(\Theta_1, \Theta_2, \ldots, \Theta_n), \tag{4}$$

$$T_{\Theta_R} = T_{\Theta'_R} = T_{\Theta_S} = T_{\Theta_{RS}} = \bigcup_{i=0}^{n} T_{\Theta_i}. \tag{5}$$

The result of application of Θ_R to $\boldsymbol{x}$ coincides with that of application of Θ_i to $\boldsymbol{x}$ with probability p_i. If probabilities p_i are omitted (2), then they are equal to $1/n$. The result of application of Θ_S to $\boldsymbol{x}$ coincides with sequential applications of $\Theta_1, \Theta_2, \ldots, \Theta_n$ to $\boldsymbol{x}$. The result of application of Θ_{RS} to $\boldsymbol{x}$

coincides with sequential applications of randomly ordered $\Theta_1, \Theta_2, \dots, \Theta_n$ to $\boldsymbol{x}$.

An elementary transition rule can be written down as a substitution in the following form:

$$\Theta_{\text{sub}} : \{a_1, a_2, \dots, a_n\} \xrightarrow{p} \{a'_1, a'_2, \dots, a'_n\}. \tag{6}$$

Application of Θ_{sub} to a cell $\boldsymbol{x}$ results in replacing the states of the cells $T_{\Theta_{\text{sub}}}(\boldsymbol{x})$ with probability p. Here the probability p and the states a'_i can be functions of current states, $p = p(a_1, a_2, \dots, a_n)$, $a'_i = f_i(a_1, a_2, \dots, a_n)$.

An ACA simulation process is split to *iterations*. An iteration comprises $|Z^2| = N_x N_y$ transition rule applications to randomly chosen cells.

3. Parallel algorithms

In papers [4, 5], an ACA is defined as a discrete event model that evolves in continuous time. Transition rule applications to different cells asynchronously occur at random times. These applications form a Poisson process for each cell. For different cells, these Poisson processes are independent and the application rate is the same for each cell. Parallelization of the ACA model is performed by the domain decomposition: each process hosts its own domain of cells and copies of the boundary cells of the neighboring processes. For the correct simulation of the Poisson process for each cell, every computing process controls its own local time. The *process local time* is the next time instant of a transition rule application to a newly selected cell from its domain. A process increments its own local time by an exponentially distributed pseudorandom number after each transition rule application.

In [4], an algorithm suitable for shared memory computers is proposed. Each process repeats the following steps while its own local time is less than the predefined $T_{\max}$: (1) selects a random cell from its domain, (2) waits for the situation when a minimal local time of the neighboring processes is greater than its own local time for the cells belonging to the domain boundary, (3) applies a transition rule to the cell, and (4) increments its own local time according to the Poisson distribution. Independence and Fairness of the algorithm are provided by the independence and fairness of random cell selection from the domain and the way of incrementing the local time. Correctness is provided by the synchronization based on domain's local time briefly described at step (2). Efficiency is provided by the property that boundary cells are seldom selected in large domains.

In [5], a modified Time Warp algorithm is presented. Time Warp [10] is an optimistic parallel algorithm for the simulation of any discrete event model on distributed memory computers. The main idea of the parallel Time Warp algorithm is as follows. In contrast to the previous algorithm, application of the transition rule and computing a new local time are performed without waiting for the neighboring processes (each process "hopes" that the neighboring processes will not change the boundary cell states). If a process changes the boundary cell state, then it sends a message to its neighbors called a *positive message.* When a process receives a positive message "from the future" (i.e., its own local time is less than that of a sender), it saves the message for the future processing. A situation when a process receives a positive message "from the past" is called *causality error.* In this situation, a recovery mechanism should be initiated. Recovery from the prematurely executed steps results in two things to be rolled back: the cellular array and the messages sent to other processes. The rolling back of the cellular array is accomplished by periodically saved updated cell states and restoring the boundary cell states, which are valid for the rolled back local time. The rolling back of previously sent positive messages is accomplished by sending *anti-messages.* If a process receives an anti-message that corresponds to an unprocessed positive message, then these two messages annihilate each other and the process proceeds. If there arrives an anti-message that corresponds to a positive message, which has been already processed, then the process has made an error and is also to be rolled back.

A consequence of the recovery mechanism is that more anti-messages can recursively be sent to other processes. Independence and Fairness of the algorithm are provided in the same way as in the previous algorithm. Correctness is provided by the recovery mechanism from causality errors (see [5] for detail). In a few words, efficiency is provided by optimistic behavior of processes. However, there are some costs for saving the history of sending messages and updating the boundary cell states. If the costs are sufficiently large, then the efficiency of the algorithm decreases. Also, the efficiency significantly depends on the following two parameters of the transition rule [5]: the amount of work to be performed for the rule computing and an average number of actually changed cell states after application of the rule.

In [6], an algorithm suitable for distributed memory computers is presented. The ACA is defined as described in Section 2. Let us consider a sequence of randomly selected coordinates $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ for an iteration. According to the decomposition of the cellular array to the domains $d_1, d_2, \ldots, d_p$, we can divide the sequence X into $2p$ parts: for each domain d_k we take its internal I_k and boundary B_k subsequences. The subsequences I_k and B_k can be formed with the use of uniform, exponential, and binomial pseudorandom number generators. The algorithm is based on the stochastic properties of I_k and B_k, and on planning the order of interactions between processes. The main idea of the parallel algorithm is as follows. Firstly, a process k (hosting a domain d_k) forms the pair (I_k, B_k) independent of other processes. Secondly, each process avoids unnecessary synchronizations because it is informed about the neighbors subsequences $B_{k'}$. This means that the process waits only for those new boundary cell states that will be actually used by it. Note that in the first algorithm [4] the process has to wait in any case. Independence and Fairness of the algorithm are provided by the method, in which I_k and B_k are formed (for detail see [6]). Correctness is provided by means of the synchronization based on B_k. Efficiency is provided by avoiding unnecessary synchronization.

In [7, 8], a practical analog of the ACA is described. This model is called Block-Synchronous Cellular Automata due to the ways of asynchronism reduction. An iteration of the BSCA is a sequence of m stages. At each stage, the transition rule is synchronously applied to all cells from a randomly selected set $S_i \subset \Omega$, where $\{S_1, S_2, \ldots, S_m\}$ is partitioning of the cellular array Ω with the following properties:

$$\bigcup_{i=1}^{m} S_i = \Omega, \tag{7}$$

$$\forall i\ \forall j : \ S_i \cap S_j = \emptyset, \quad i \neq j, \tag{8}$$

$$\forall i\ \forall j : \ |S_i| = |S_j|, \tag{9}$$

$$\forall i\ \forall a \in S_i\ \forall b \in S_i : T(a) \cap T(b) = \emptyset, \quad a \neq b. \tag{10}$$

Such a reduction of asynchronism (at the expense of **I**ndependence) results in a very simple parallel algorithm. In this algorithm, processes have to synchronize with each other (to send and to receive new boundary cell states) only between stages. A high efficiency of the algorithm is provided by rare process synchronizations. Correctness and fairness are provided by properties (10) and (7)–(9), respectively.

4. Computer architectures overview

For testing the algorithms given above, we use computers with three types of architecture: multicores and multiprocessors with shared memory, a cluster with distributed memory and graphical processing unit (GPU). Parameters of computers with shared memory *Core-i7* and *SMP-8* are given in Table 2.

The cluster *MVS-100k* consists of SMP-8 nodes connected through Infiniband.

Table 2. Multicores and multiprocessors parameters

Computer	Processor	GHz	Total cores	Memory controller
Core-i7	1×Intel Core i7	2.6	4	integrated
SMP-8	2×Intel Xeon 5140	2.3	8	separate

The GPU *GTX-280* consists of a 240-core processor and 2Gb off-chip *global memory.* The cores are grouped in 8-core *multiprocessors.* Each multiprocessor has its own 16Kb on-chip *shared memory* and 32Kb *register file.* Note, that the multiprocessor belongs to SIMD (Single Instruction Multiple Data) class in Flynn's taxonomy: at each clock all eight cores perform the same instruction but using different arguments.

A parallel program intended for running on GTX-280 consists of thousands of threads grouped in *blocks.* Each block consists of no more than 512 threads. One block of threads can be run on one multiprocessor, only. But one multiprocessor can manage up to 8 blocks. Threads of the same block can communicate all to all through the shared memory of a multiprocessor and synchronize. But threads from different blocks cannot communicate and can not synchronize in the same way.

The point is, one cannot directly implement parallel algorithms of the above-discussed ACA simulation with exception of the BSCA. The reason is in the SIMD architecture of multiprocessor and the impossibility of synchronization of threads belonging to different blocks.

5. Simulation of surface reactions on palladium

A model of oscillatory dynamics of the reaction $CO + O_2$ over the supported Pd nanoparticles is described in [2]. This model is a combination of the model for the $CO + O_2$ reaction over the Pd(110) single crystal [11] and the stochastic model for imitating the supported nanoparticle with a dynamically changing shape and surface morphology [12]. The model consists of the following processes: CO adsorption $(\Theta_1, \Theta_2, \Theta_6)$, CO desorption $(\Theta_3, \Theta_4, \Theta_7)$, O_2 adsorption (Θ_9^i), CO diffusion $(\Theta_{11}^i, \Theta_{12}^i, \Theta_{13}^i)$, Pd atoms diffusion (Θ_{14}^i), subsurface oxygen O_{ss} formation (Θ_5), CO+O and CO+O_{ss} reactions $(\Theta_8, \Theta_{10}^i, \Theta_{15}^i)$. In terms of the ACA this model can be described as follows.

The state a of a cell $\boldsymbol{x}$ is written down as $[n, \alpha]$, where n is the number of Pd atoms, $n \in \{0, 1, 2, \ldots\}$, and α is the state of the surface Pd, $\alpha \in \{\emptyset, \mathrm{CO}, \mathrm{O}, \mathrm{O}_{ss}, \mathrm{CO.O}_{ss}\}$;

$$A = \{0, 1, 2, \ldots\} \times \{\emptyset, \mathrm{CO}, \mathrm{O}, \mathrm{O}_{ss}, \mathrm{CO.O}_{ss}\},$$

$$\Theta = R(\Theta^1, \Theta^2, \Theta^3, \Theta^4),$$

$$\Theta^i = S(R(\Theta_1, p_1; \ldots, \Theta_8, p_8; \Theta_9^i, p_9; \Theta_{10}^i, p_{10}; S(\Theta_{11}^i, \ldots \Theta_{14}^i), p_{11}), \Theta_{15}^i),$$

$$\Theta_{15}^i = RS(\Theta_{15}^{1,0}, \ldots, \Theta_{15}^{5,0}, \Theta_{15}^{0,1}, \ldots, \Theta_{15}^{0,5}, \Theta_{15}^{1,i}, \ldots, \Theta_{15}^{5,i}, \Theta_{15}^{i,1}, \ldots, \Theta_{15}^{i,5}),$$

$$T_\Theta(\boldsymbol{x}) = T_{\Theta^i}(\boldsymbol{x}) = T_{13}(\boldsymbol{x}),$$

$$T_{\Theta_j}(\boldsymbol{x}) = T_1(\boldsymbol{x}),$$

$$T_{\Theta_j^i}(\boldsymbol{x}) = T_1(\boldsymbol{x}) \cup T_1(\boldsymbol{x} + \boldsymbol{v}_i), \quad j = 9, \ldots, 13,$$

$$T_{\Theta^i_j}(\boldsymbol{x}) = T_5(\boldsymbol{x}) \cup T_5(\boldsymbol{x} + \boldsymbol{v}_i), \quad j = 14, 15,$$

$$T_{\Theta^{k,m}_{15}}(\boldsymbol{x}) = T_1(\boldsymbol{x} + \boldsymbol{v}_k) \cup T_1(\boldsymbol{x} + \boldsymbol{v}_m);$$

$$\begin{aligned}
\Theta_1 &: \ \{[n, \emptyset]\} \to \{[n, \mathrm{CO}]\},\\
\Theta_2 &: \ \{[0, \emptyset]\} \to \{[0, \mathrm{CO}]\},\\
\Theta_3 &: \ \{[n, \mathrm{CO}]\} \to \{[n, \emptyset]\},\\
\Theta_4 &: \ \{[0, \mathrm{CO}]\} \to \{[0, \emptyset]\},\\
\Theta_5 &: \ \{[n, \mathrm{O}]\} \to \{[n, \mathrm{O}_{ss}]\},\\
\Theta_6 &: \ \{[n, \mathrm{O}_{ss}]\} \to \{[n, \mathrm{CO.O}_{ss}]\},\\
\Theta_7 &: \ \{[n, \mathrm{CO.O}_{ss}]\} \to \{[n, \mathrm{O}_{ss}]\},\\
\Theta_8 &: \ \{[n, \mathrm{CO.O}_{ss}]\} \to \{[n, \emptyset]\},\\
\Theta^i_9 &: \ \{[n, \emptyset], [n, \emptyset]\} \to \{[n, \mathrm{O}], [n, \mathrm{O}]\},\\
\Theta^i_{10} &: \ \{[n, \mathrm{CO}], [n, \mathrm{O}_{ss}]\} \to \{[n, \emptyset], [n, \emptyset]\},\\
\Theta^i_{11} &: \ \{[n, \mathrm{CO}], [m, \emptyset]\} \to \{[n, \emptyset], [m, \mathrm{CO}]\},\\
\Theta^i_{12} &: \ \{[n, \mathrm{CO}], [m, \mathrm{O}_{ss}]\} \to \{[n, \emptyset], [m, \mathrm{CO.O}_{ss}]\},\\
\Theta^i_{13} &: \ \{[n, \mathrm{CO.O}_{ss}], [m, \mathrm{O}_{ss}]\} \to \{[n, \mathrm{O}_{ss}], [m, \mathrm{CO.O}_{ss}]\}\\
\Theta^i_{14} &: \ \{[n+1, \emptyset], a_1, \ldots, a_4, [m, \emptyset], a_6, \ldots, a_9\} \xrightarrow{p'}\\
&\quad\ \{[n, \emptyset], a_1, \ldots, a_4, [m+1, \emptyset], a_6, \ldots, a_9\},\\
\Theta^{k,j}_{15} &: \ \{[n, \mathrm{CO}], [n, \mathrm{O}]\} \to \{[n, \emptyset], [n, \emptyset]\}.
\end{aligned}$$

Here for Θ^i_{14}, the states a_i are $a_i = [n_i, \alpha_i]$, and the probability p' is that of the actual movement of Pd atom, which depends on changing the total energy of the atomic connections ΔE, $p' = exp(-\Delta E/kT)$. The probabilities p_i depend on the rates of the processes k_i, $p_i = k_i / \sum_{j=1}^{11} k_j$. For concrete values $k_1, k_2, \ldots, k_{11}$ and concrete energies of atoms connections see [2].

The algorithm [4] is implemented as a multithreaded program using POSIX Threads. Each thread controls its own local time and hosts a part of the cellular array (domain). The algorithm [6] is implemented using MPI (Message Passing Interface). For sending short messages, containing the new cell states, MPI_Bsend (buffered mode) is used. The algorithm [7, 8] is implemented using MPI and OpenMP. In each node SMP-8, one process with eight threads is executed. Using OpenMP reduces the number of actually executed processes and therefore, also reduces communication costs. For GTX-280, the algorithm [7, 8] is implemented using CUDA [9]. Each thread deals with the neighborhood of a particular cell. First, it loads the states of the neighboring cells to the shared memory. Then the thread computes new states for the neighborhood. After that the thread returns the new states to the global memory.

Results of testing on the shared memory computers, the cluster and the GPU are presented in Figures 1, 2, and Table 3, respectively. All the tests are performed with several cellular array sizes (1000×1000, 2000×2000, 4000×4000, and 8000×8000). As usual, the efficiency of parallelization is $E_p = T_1/(pT_p)$.

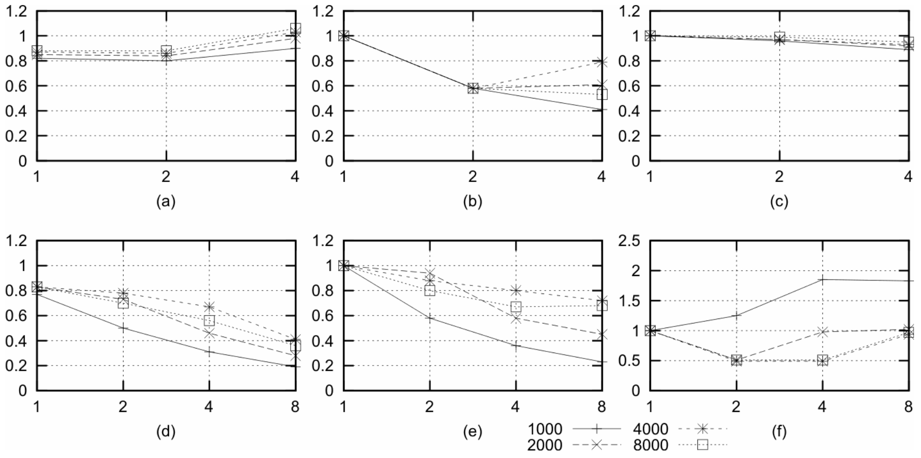

Figure 1. Efficiency of the parallel algorithms ((a, d) for [4], (b, e) for [6], and (c, f) for [7, 8]) for Core-i7 (a, b, c) and SMP-8 (d, e, f). Along x-axis are the numbers of used cores, along y-axis, efficiency is shown

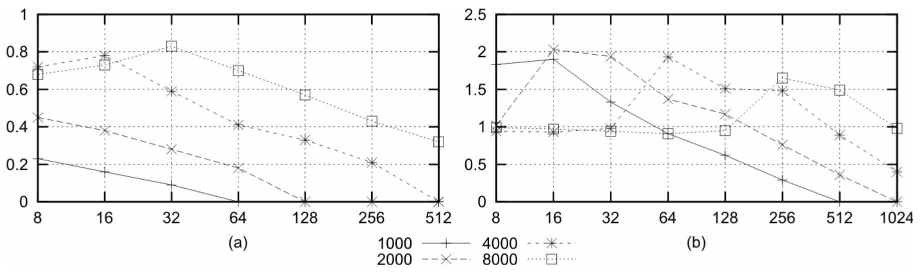

Figure 2. Efficiency of the parallel algorithms ((a) for [6], and (b) for [7, 8]) for MVS-100k. Along x-axis are the numbers of used cores, along y-axis, efficiency is shown

Table 3. Acceleration of the algorithm [7, 8] implemented on GPU in comparison with that on Core-i7

Size	1000	2000	4000	8000
Acceleration	25	31	34	35

6. Conclusion

Results of testing show that the efficiency of the algorithm [4] is high on modern multicore computers (Core-i7) even for relatively small cellular arrays. The algorithm [6] possesses a good efficiency only for large cellular arrays but can be run on a cluster with several nodes. The reason is in additional costs of MPI sendings and receivings. Further improvement of the algorithm should be focused on multithread extensions to reduce the costs of communications. The algorithm [7, 8] shows a high efficiency for all sizes of a cellular array and on all parallel architectures. The reason of such a high performance is in reduction of asynchronism (at the expense of independence). For some models, it is shown [7, 8] that such a reduction does not affect the simulation process. Nevertheless, for each new model one has to make sure that the model allows such a reduction.

Acknowledgements. I would like to thank Prof. O.L. Bandman and Prof. V.I. Elokhin for the fruitful discussions and advise.

References

[1] Danielak R., Perera A., Moreau M., Frankowicz M., Kapral R. Surface Structure and Catalytic CO Oxidation Oscillations. — arXiv:chao-dyn/9602015v1, February 13, 1996.

[2] Elokhin V.I., Latkin E.I., Matveev A.V., Gorodetskii V.V. Application of statistical lattice models to the analysis of oscillatory and autowave processes on the reaction of carbon monoxide oxidation over platinum and Palladium surfaces // Kinetics and Catalysis. — 2003. — Vol. 44, No. 5. — P. 672–700.

[3] Neizvestny I.G., Shwartz N.L., Yanovitskaya Z.Sh., Zverev A.V. 3D-model of epitaxial growth on porous {111} and {100} Si surfaces // Computer Physics Communications. — 2002. — Vol. 147. — P. 272–275.

[4] Lubachevsky B.D. Efficient parallel simulations of asynchronous cellular arrays. — 1987. — Complex Systems. — Vol. 1, No. 6. — P. 1099–1123.

[5] Overeinder Benno J., Sloot Peter M.A. Extensions to Time Warp Parallel Simulation for Spatial Decomposed Applications // Proc. 4th United Kingdom Simulation Society Conference (UKSim 1999), 1999. — P. 67–73.

[6] Kalgin K.V. Parallel simulation of asynchronous Cellular Automata evolution // Bull. Novosibirsk Computing Center. Ser. Comp. Science. — Novosibirsk, 2008. — Iss. 27. — P. 55–63.

[7] Nedea S.V., Lukkien J.J., Hilbers P.A.J., Jansen A.P.J. Methods for parallel simulations of surface reactions // Proc. 17th International Symposium on Parallel and Distributed Processing. — 2003. — arXiv:physics/0209017v1, September 4, 2002.

[8] Bandman O.L. Parallel simulation of asynchronous Cellular Automata evolution // ACRI 2006. — 2006. — P. 41–47. — (LNCS; 4173).

[9] NVIDIA CUDA Programming Guide. — http://www.nvidia.com/object/cuda_get.html.

[10] Jefferson D.R. Virtual time // ACM Trans. on Program. Lang. and Sys. 7,3 (July 1985). — P. 404–425.

[11] Latkin E.I., Elokhin V.I., Matveev A.V., Gorodetskii V.V. The role of subsurface oxygen in oscillatory behaviour of $CO + O_2$ reaction over Pd metal catalysts: Monte Carlo model // J. Molec. Catal. A, Chemical. — 2000. — Vol. 158. — P. 161–166.

[12] Kovalyov E.V., Elokhin V.I., Myshlyavtsev A.V. Stochastic simulation of physicochemical process performance over supported metal nanoparticles // J. Comput. Chem. — 2008. — Vol. 29, No. 1. — P. 79–86.