Bull. Nov. Comp. Center, Comp. Science, 46 (2022), 13–25
© 2022 NCC Publisher

Software implementation of asynchronous and synchronous cellular automata with maximum domino tiles coverage*

S. Kireev, Yu. Trubitsyna

Abstract. A two-dimensional asynchronous cellular automaton covering the cellular array with a maximum number of domino tiles is considered. For the purpose of parallel implementation, a transition to the synchronous operation mode was made. The paper presents asynchronous and synchronous cellular automata implementations. Their evolution and performance characteristics are analyzed. It is shown that the synchronous automaton creates the desired pattern. In the implementation of the transition rule, matching of multiple templates was optimized by combining the templates into a multi-template and applying a single bitwise matching operation.

Keywords: cellular automata, pattern formation, domino pattern.

Introduction

Cellular automata is a powerful tool for studying the behavior of complex systems. When working with such models, large data arrays can be used, which is why the software implementation of cellular automata requires a lot of time to execute. In such cases, the problem of parallelization arises for running on parallel computing systems.

One of the actual problems for cellular automata application is solving the task of pattern formation [1–4]. In papers [5–7], Domino cellular automata were proposed that form a domino pattern with different target properties. A Domino cellular automaton [5] is an asynchronous cellular automaton represented by a two-dimensional square lattice of cells with two basic states "0" and "1". As a result of the cellular automaton operation, domino tiles are formed that cover the cellular array. A domino tile (Figure 1) is a two-dimensional array of size 3×4 (horizontal domino) or 4×3 cells (vertical domino). In the center of the array, there are two cells with the state "1", along the edges there are cells with the different state (for example, "0").

The current work considers the problem of parallel implementation of the Domino cellular automaton. Creating efficient parallel implementations

*This work was carried out under the state contract with ICMMG SB RAS (251-2021-0005).

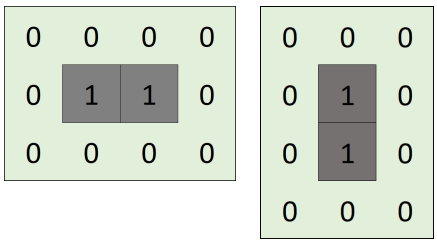

Figure 1. Domino tiles

for asynchronous cellular automata is hard and requires the use of complex algorithms [8, 9]. At the same time, there is an easy solution: to change the operation mode from asynchronous to synchronous. Synchronous cellular automata are easily implemented in parallel using the domain decomposition technique [10, 11]. However, we face a problem of admissibility for this change: whether the modified cellular automaton is able to solve the same problem (for example, see [12]). The purpose of this paper is to implement a synchronous version of Domino cellular automaton and compare its characteristics with these for the asynchronous one, as well as with the results given in [6].

The paper is organized as follows. Section 1 describes the Domino cellular automaton. Section 2 presents asynchronous and synchronous cellular automata implementations: algorithms and performance results. Section 3 describes the optimization technique used in the cellular automata implementation. Section 4 presents and discusses the results of asynchronous and synchronous cellular automata operation and their comparison with the results of the original paper [6].

1. Domino Cellular Automaton

The Domino cellular automaton is represented by a two-dimensional cellular array on a Cartesian grid. Evolution of the cellular automaton is a sequence of iterations where a transition rule is applied to the cells for a certain number of times. The array boundaries can be considered periodic or not; in the framework of this work this is insignificant. The attributes of each cell are coordinate indices i and j, state s, and hit count h. The coordinates i and j determine the position of a cell in the cellular array. A cell can have one of three states: $s \in \{\text{“0”}, \text{“1”}, \text{“\#”}\}$, where “#” denotes the cells that do not change their states during evolution, and is treated as “0” in the transition rule. We consider such an initial state of the array, in which the states of the boundary cells are set to “#”, and the inner ones are set equiprobably to “0” or “1”.

The transition rule is a function that takes the states of the selected cell (and its neighbors) as input and calculates the new state of the cell. In this paper, a transition rule defined in [6] is used, which tries to maximize the number of dominoes in a cellular array. The cell neighborhood is 24 nearest cells forming a 5×5 square.

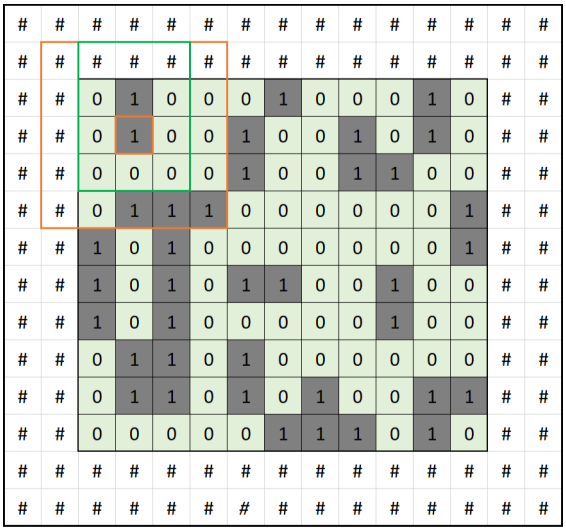

Figure 2. Cellular array with a window (marked in orange)

The transition rule depends on the value of the hit count h. Auxiliary structures "window" and "template" are used for its calculation. The window is a 5×5 part of the cellular array that contains the selected cell (in the center) and its neighborhood (Figure 2). The template is a 2D array of 3×4 cells (horizontal template) or 4×3 cells (vertical template) representing a domino tile. There are 24 templates in total, 12 horizontal (Figure 3) and 12 vertical. The vertical templates are obtained by rotating each of the 12 horizontal templates 90 degrees clockwise. Templates are applied to the window, and the match of the corresponding cells is determined. Each template has a selected cell (yellow color in Figure 3) arranged in the center of window (Figure 4). The selected cell does not participate in comparison. Some templates in their original form may not fully fit in the window (see Figure 3, bottom line), so their cells marked with "$*$" also do not participate in comparison. A template matches if all the participating cells do match.

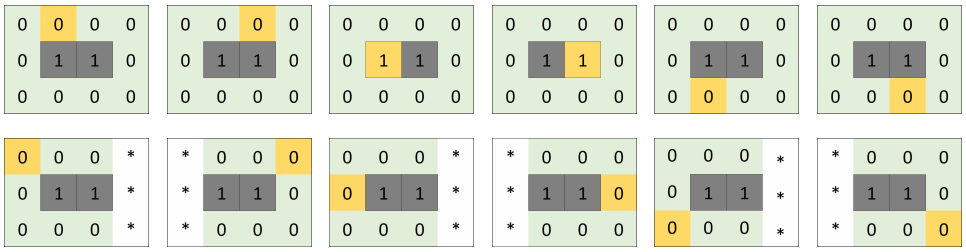

Figure 3. Horizontal templates. With the selected cell (yellow), the template is applied to the center of the window

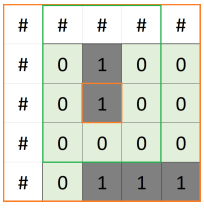

Figure 4. Template (with green border) applied to the window (with outer orange border)

Next, consider calculating the value of the hit counter h for a given cell of the cellular array. All 24 templates are applied to the window with a given cell in the center, and the number of matched templates is calculated. If there is a matching template with the selected cell marked "1", then the hit counter h is set to 100. Otherwise, the hit counter h is set to the number of matched templates. The templates are designed in such a way that the number of hits never exceeds four.

The states of the cells change as a result of evolution. Here we use the transition rules described in subsections 3.1 and 4.2 of [6]. It is stated there that the combination of these two rules seeks to maximize the number of dominoes in the cellular array, so that the maximum coverage of the array with dominoes can be obtained. The combined rule is as follows:

$$s'(i,j) = \begin{cases} random \in \{0,1\}, & \text{if } h = 0, \text{ applied with probability } \pi_0, \\ random \in \{0,1\}, & \text{if } h = 1, \text{ applied with probability } \pi_1, \\ 0, & \text{if } h = 2,3,4, \\ 1, & \text{if } h = 100, \\ s(i,j), & \text{otherwise,} \end{cases} \tag{1}$$

where $s(i,j)$ is the previous state of the cell, $s'(i,j)$ is the new state of the cell, π_0 is the probability that ensures the introduction of noise into the model so that the evolution of the cellular automaton can continue in the case when all cells of the array have a hit counter equal to zero, π_1 is the probability that introduces noise to minimize the number of cells with a hit counter value of one and contribute to the formation of new dominoes.

The evolution of the asynchronous Domino cellular automaton occurs by applying the transition rule (1) to arbitrarily chosen cells of the array. Thus, the chosen cell state is updated right after the rule application. The number of rule applications per one iteration of evolution is equal to the number of cells in the array. Within one iteration, the rule can be applied to some cells several times, and not to others at all.

In synchronous mode, the rule is applied once to all cells of the array within single iteration, and their states are updated simultaneously. In software implementation, simultaneity of changes is simulated by introducing a copy of the original cellular array, where the new cell states are written during the iteration.

Ultimately (both for synchronous and asynchronous modes), when applying the rule (1) with proper π_0 and π_1 parameters values, the array is

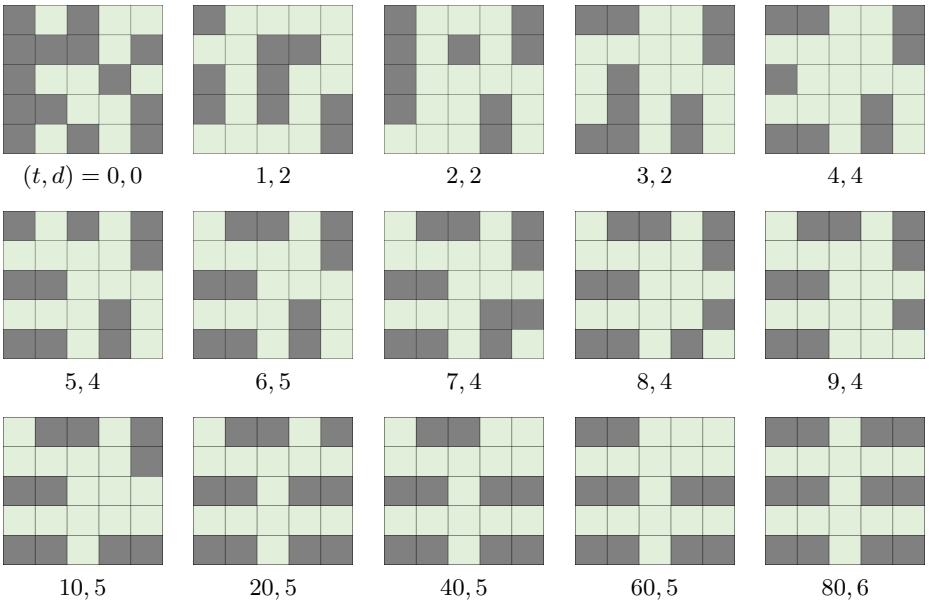

Figure 5. Evolution process of the Domino cellular automaton, t is an iteration number, d is a number of domino tiles formed. Dark cells have state "1", light cells have state "0"

covered with dominoes. The example of the evolution process is shown in Figure 5.

2. Implementation

Algorithm 1 presents one iteration of the asynchronous cellular automaton Domino. The inputs of the algorithm are a cellular array (*array*) that changes its state during the execution, the probabilities π_0 and π_1 and a list of domino templates (*listOfDominoTemplates*). In lines 3–5, cell coordinates are randomly selected, and the current state of the corresponding cell is obtained. If it is not a border cell ("#"), the new state is calculated. In line 7, the window with the selected cell in the center is taken. Then all the templates are applied to the window, and the "hits" value is calculated (lines 8–16). In lines 17–30, a transition rule (1) is used to calculate the new state (*newState*), which is further assigned to the selected cell.

```
1 procedure AsyncIter(array, π0, π1, listOfDominoTemplates)
2   for step = 1 to size(array) do
3     i ← randomRowNumber(array)
4     j ← randomColumnNumber(array)
5     oldState ← array[i][j]
6     if oldState <> "#"
```

```
7       window ← getWindowByCenter(i, j)
8       hits ← 0
9       for all dominoTemplate in listOfDominoTemplates do
10        if match(window, dominoTemplate) = true then
11          state ← getSelectedCellState(dominoTemplate)
12          if state = "1" then
13            hits ← 100
14            break
15          else
16            hits ← hits + 1
17      if hits = 0 then
18        if randomProbability() < π0 then
19          newState ← selectRandomOf("0", "1")
20        else
21          newState ← oldState
22      else if hits = 1 then
23        if randomProbability() < π1 then
24          newState ← selectRandomOf("0", "1")
25        else
26          newState ← oldState
27      else if hits = 100 then
28        newState ← "1"
29      else
30        newState ← "0"
31      array[i][j] ← newState
```

Algorithm 2 presents one iteration of the synchronous cellular automaton Domino. In this algorithm, the transition rule is applied to all cells of the array, and the new state of the cell is written not to the original array, as it was when implementing the asynchronous mode, but to the intermediate one (*newArray*) (lines 31, 33). After the iteration is completed, the new array replaces the original one (line 34). This provides an imitation of the simultaneous update of the states of all cells.

```
1 procedure SyncIter(array, π0, π1, listOfDominoTemplates)
2   for all cell in array do
3     i ← getRow(cell)
4     j ← getColumn(cell)
5     oldState ← array[i][j]
6     if oldState <> "#" then
        /* Lines 7–30 from AsyncIter procedure */
31      newArray[i][j] ← newState
32    else
33      newArray[i][j] ← oldState
```

34 $array \leftarrow newArray$

Both algorithms were implemented in C++. Figure 6 shows their performance comparison (with the optimization technique applied, which is described below is Section 3). The performance is presented in MCell/s units (millions of cells processed per second) for different cellular array sizes (including 1-width borders with state "#"). The tests were performed on Intel Core i9-9900K 3.6 GHz.

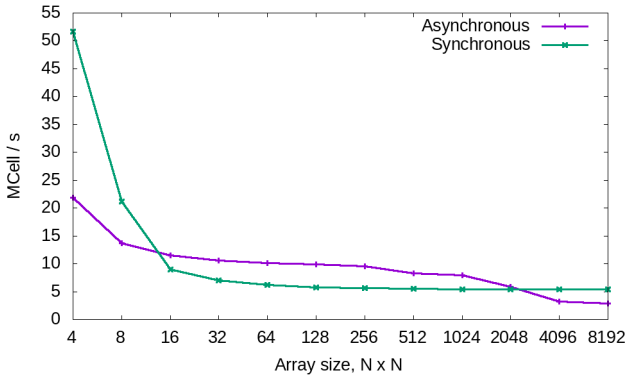

Figure 6. Performance comparison of asynchronous and synchronous Domino cellular automata implementations

For very small array sizes, which are practically not interesting, the synchronous implementation works faster. For medium and larger array sizes, the performance of synchronous implementation decreases to a certain level and remains constant, while the asynchronous implementation outperforms the synchronous one by about 60 %, until the amount of its data does not fit in the cache (with array size larger than 1024×1024). Then its performance falls to about half of the synchronous implementation performance.

3. Optimization of the Hit Count Calculation

The transition rule is based on the results of matching the neighborhood of a cell with 24 templates. Applying templates to the window sequentially one by one takes a lot of time. Therefore, the matching procedure was optimized, which gave a speedup of approximately 25 %.

The idea of optimization is to combine all 24 templates into a single bitmap structure, which is dubbed "multi-template" therein. The multi-template is implemented by two 5×5 two-dimensional arrays comprising the elements which are Boolean vectors of length 24. Each original template is mapped to a specific bit for each Boolean vector (Figure 7). In the first array ("location"), bit values encode the location of the template in the

template 1	template 2	template 3	template 4	...	template 24
0	1	0	1	...	0

Figure 7. Example of bit representation of one cell of multi-template arrays

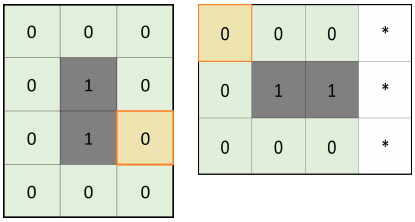

Figure 8. Templates used to construct an example of multi-template

window. In the second array ("values"), the values of the corresponding bits encode the values of the template cells.

Consider the formation of a multi-template using the templates shown in Figure 8 as an example. Let the template on the left corresponds to the first bit of the Boolean vectors, and the template on the right to the second one.

Figure 9 shows the encoding of the first template in a multi-template. It can be seen that in the "location" array (left), in those cells where the template is located in the window, the first bits of the Boolean vectors take the value 1, the rest are 0. The first bits of the Boolean vectors of the "values" array (right) completely repeat the values of the cells of the corresponding template.

Figure 10 highlights the encoding of the second template in a multi-template. In this case, the second bits were filled according to the same principle that was described above.

100...0	100...0	100...0	000....0	000....0
100...0	100...0	110...0	010....0	010....0
100...0	100...0	110...0	010....0	010....0
100...0	100...0	110...0	010....0	010....0
000....0	000....0	000....0	000....0	000....0

000...0	000...0	000...0	000....0	000....0
000...0	100...0	000...0	000....0	000....0
000...0	100...0	000...0	010....0	010....0
000...0	000...0	000...0	000....0	000....0
000....0	000....0	000....0	000....0	000....0

Figure 9. Encoding of the first template in a multi-template, "location" array (left) and "values" array (right)

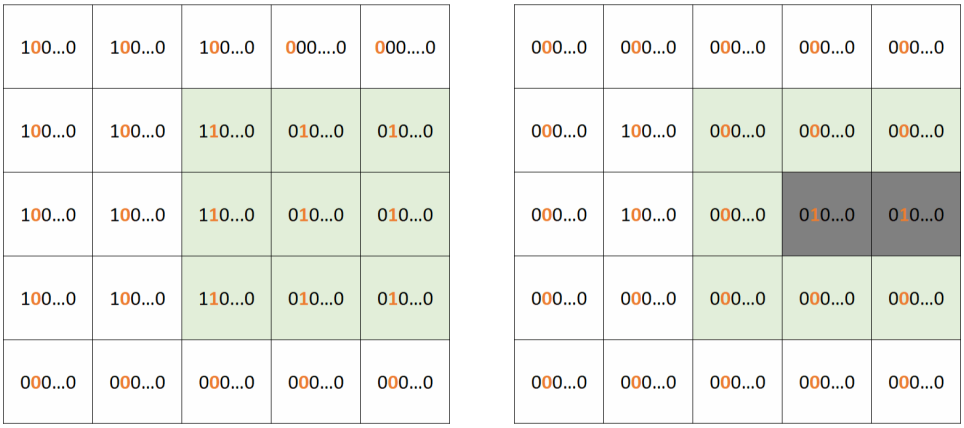

Figure 10. Encoding of the second template in a multi-template, "location" array (left) and "values" array (right)

The remaining 22 components of the Boolean vector are filled in the same way. Thus, all 24 patterns can be encoded into one multi-template. The result of applying a multi-template to a 5×5 window using bitwise operations is a Boolean vector of length 24, where bit value 1 marks the cell hits in a specific template.

4. Cellular Automata Operation Results and Discussion

First of all, the evolution of the synchronous and asynchronous Domino cellular automata are compared. The number of dominoes depending on the iteration number for both modes is shown in Figure 11. We use the array size 100×100 (excluding border cells with state "#"), and the basic values of the parameters $\pi_0 = 0.5$, $\pi_1 = 0.005$. To understand the influence of the transition rule parameters on the evolution of cellular automata, we change one of the probabilities, while the other was fixed.

The graphs in Figure 11 show that the evolution of both cellular automata can be divided into two stages. At the first stage, there is a rapid increase in the number of dominoes to a certain level. The synchronous cellular automaton reaches this level in about twice as many iterations as the asynchronous one. It can be related to the fact that in the asynchronous cellular automaton the maximum rate of propagation of changes is greater than in the synchronous one, since each change in the state of the cell is taken into account in subsequent changes in the states of neighboring cells within one iteration. This is contrast to the synchronous one, where changes in the state of the cell are taken into account only when the iteration ends.

At the second stage of evolution, both in asynchronous and synchronous modes, the number of dominoes fluctuates, slowly increasing on average.

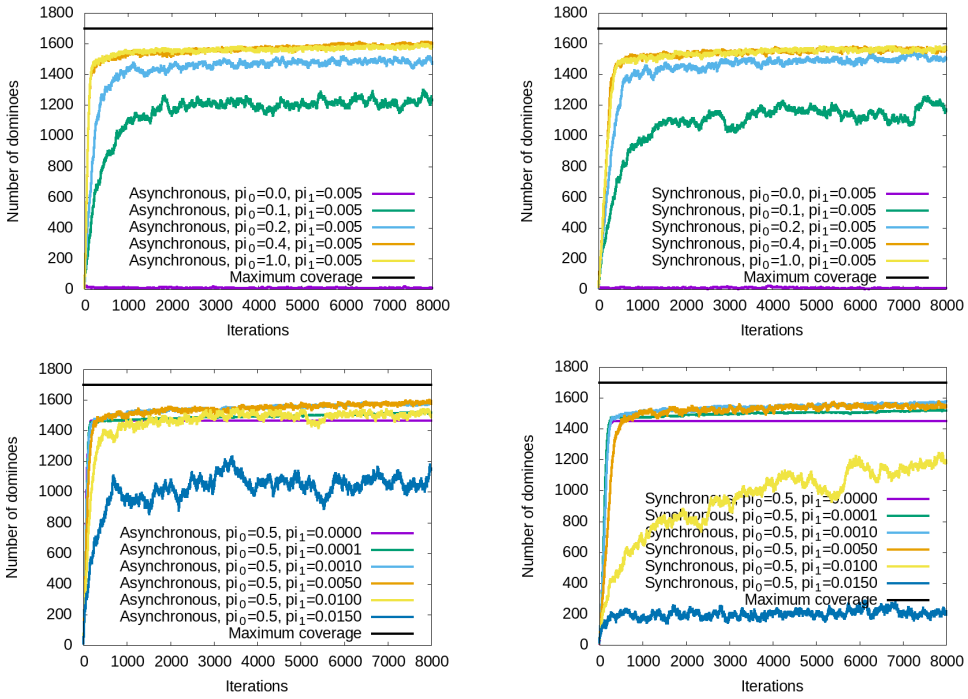

Figure 11. Evolutions of Domino cellular automata depending on the operation mode (left — asynchronous, right — synchronous) and parameter values (upper — π_0, lower — π_1)

Once, the number of dominoes can reach a theoretical maximum (maximum coverage). The growth rate of the graph at this stage determines the time to obtain the maximum coverage. From the results in Figure 11 it can be seen that the optimal values of the parameters (with the highest graphs) are approximately in the ranges $\pi_0 \in [0.4, 1.0]$, $\pi_1 \in [0.001, 0.005]$. When the probabilities are within these limits, the second stage for synchronous and asynchronous cellular automata proceeds similarly. However, the synchronous cellular automaton is more sensitive to probabilities going beyond the given range, in this case its graph goes lower.

In cases with unknown maximum number of dominoes, the following stopping criterion can be proposed: evolution stops when the average number of dominoes does not grow for a given number of iterations (T_1). In this case, averaging is done over a certain number of consecutive iterations (T_2). The greater the T_1 value, the closer we can get to the maximum number of domino tiles, but the longer the calculation will take. T_2 should be large enough to smooth out fluctuations.

Next, the results presented in [6] are compared with those obtained by our implementation in asynchronous and synchronous modes. Figure 12a

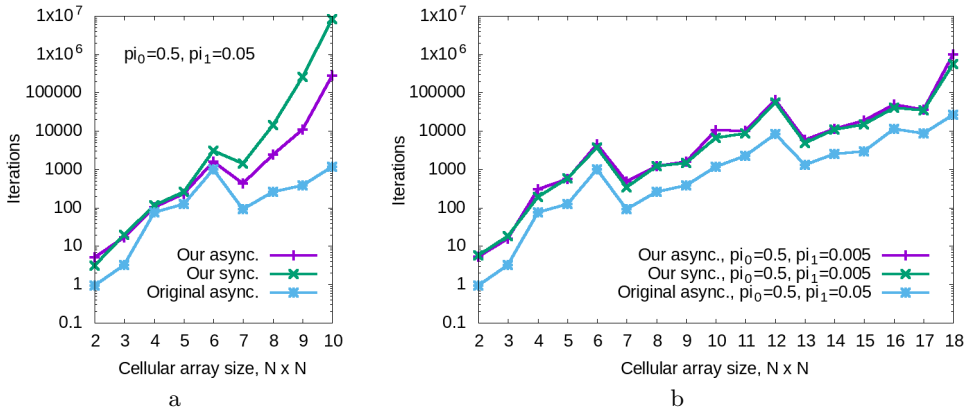

Figure 12. Average number of iterations required to obtain the maximum domino coverage

presents the average number of iterations required to form the maximum coverage, depending on the size of the array. The parameters taken from [6] were the following: $\pi_0 = 0.5$, $\pi_1 = 0.05$ are probabilities presented in the transition rule (1), $n_{\text{runs}} = 100$ is the number of program runs to get the average.

It can be seen that there are certain similarities and differences between the results. The shapes of the graphs remain similar, reflecting the proportion of patterns with maximum coverage out of all possible (for each array size). But the values themselves differ, and this difference grows with the array size. In addition, it can be seen that the synchronous version obtains the solution through more iterations than the asynchronous one.

Note that the probability values used in [6] are not within the range of optimal values determined above. Let's take the probabilities within the range, for example, $\pi_0 = 0.5$, $\pi_1 = 0.005$. The corresponding results are shown in Figure 12b. Now, our results with the new parameter values are more similar to those presented in the original paper. That is, to get the maximum coverage, our implementation with the new parameter values requires about five times as many iterations as the original implementation (with the parameter values taken from the original paper) - regardless of the size of the array. It is also worth mentioning that the synchronous cellular automaton requires, on average, about 15% less iterations to create maximum coverage than the asynchronous one.

Thus, there is a difference between the results from the original paper [6] and our results. The reason for the difference is currently unknown and requires further investigation.

Conclusion

The implementations of asynchronous and synchronous cellular automata covering the array with a maximum number of domino tiles are presented. When implementing the transition function, the matching of multiple templates was optimized using multi-template technique benefiting in a speedup by approximately 25 %. Comparison of the results of cellular automata operation showed that the synchronous cellular automaton is able to solve the same problem as the asynchronous one in a comparable time. Thus, parallel implementation of the Domino cellular automaton is possible. The results obtained, however, differ from the results presented in the paper [6]. The reason for this is to be determined in further research.

References

[1] Chua L.O. Autonomous cellular neural networks: A unified paradigm for pattern formation and active wave propagation // IEEE Transactions on Circuits and Systems / L.O. Chua, M. Hasler, G.S. Moschytz, J. Neirynck, eds. — 1995. — Vol. 42. — P. 559–577.

[2] Suzudo T. Spatial pattern formation in asynchronous cellular automata with mass conservation // Physica A: Statistical Mechanics and its Applications. — 2004. — Vol. 343. — P. 185–200.

[3] Deutsch A., Dormann S. Cellular Automaton Modeling of Biological Pattern Formarion. — Berlin: Birkhäuser, 2004.

[4] Bandman O.L. Metod postroenija kletochno-avtomatnyh modelej processov formirovanija ustojchivyh struktur // Prikladnaya Diskretnaya Matematika. — 2010. — No. 4. — P. 91–99 (In Russian).

[5] Hoffmann R., Deserable D., Seredynski F. A Probabilistic Cellular Automata Rule Forming Domino Patterns // Parallel Computing Technologies. PaCT 2019 / V. Malyshkin ed. — Springer, Cham, 2019. — (LNCS; 11657).

[6] Hoffmann R., Désérable D., Seredyński F. A cellular automata rule placing a maximal number of dominoes in the square and diamond // J. Supercomput. — 2021. — Vol. 77. — P. 9069–9087.

[7] Hoffmann R. Désérable D., Seredyński F. Minimal Covering of the Space by Domino Tiles // Parallel Computing Technologies. PaCT 2021 / V. Malyshkin ed. — Springer, Cham, 2021. — (LNCS; 12942).

[8] Bandman O. Parallel Simulation of Asynchronous Cellular Automata Evolution // Cellular Automata. ACRI 2006 / S. El Yacoubi, B. Chopard, S. Bandini, eds. — Springer, 2006. — Vol. 4173. — (LNCS; 4173).

[9] Kalgin K.V. Parallel simulation of asynchronous cellular automata on different computer architectures // Bull. Novosibirsk Comp. Center. Ser. Computer Science. — Novosoibirsk: NCC Publisher, 2010. — Iss. 30. — P. 15–26.

[10] Medvedev Yu.G. Lattice gas cellular automata for a flow simulation and their parallel implementation // Parallel Programming: Practical Aspects, Models and Current Limitations. Series: Mathematics Research Developments / M.S. Tarkov ed. — Hauppauge, New York: Nova Science Publishers, Inc., 2014. — P. 143–158.

[11] Sabelfeld K., Kireev S., Kireeva A. Parallel implementation of cellular automata model of electron-hole transport in a semiconductor // J. Parallel and Distributed Computing. — 2021. — Vol. 158. — P. 186–195.

[12] Bandman O. Parallelization efficiency versus stochasticity in simulation reaction–diffusion by cellular automata // J. Supercomput. — 2017. — Vol. 73. — P. 687–699.