Designing Cellular Automata-based Scheduling Algorithms

Franciszek Seredyński*[*] and Cezary Z. Janikow
Department of Mathematics and Computer Science
University of Missouri - St. Louis
St. Louis, MO 63121
email: sered@arch.umsl.edu, janikow@umsl.edu

Abstract

In this paper, we present a systematic approach to designing cellular automata - based algorithms for multiprocessor scheduling. We consider a simple case of two processors. However, we do not restrict parallel programs. We show how to design local neighborhoods and corresponding cellular automata (CA) for a given program graph. We also show how to discover, by genetic algorithm (GA), rules of CA – suitable for solving a given scheduling problem. We evaluate the discovered rules in terms of applicability to cope with different instances of the scheduling problem.

1 INTRODUCTION

Multiprocessor scheduling, even when limited to a two processor system but any parallel program, is known to be an NP-complete problem (El-Rewini et al. 1994). While the prevailing majority of known scheduling algorithms are sequential, a new promising direction involves parallel scheduling algorithms (Ahmad & Kwok 1995).

Recent results (Andre et al. 1996; Das et al. 1996; Capcarrère et al. 1998; Hordijk et al. 1998; Menge & Tomassini 1997; Sipper 1997), showing that CA combined with GA can be effectively used to evolve highly parallel and distributed algorithms to solve complex problems, encourage similar approaches to solving the scheduling problems. In this paper, we review and extend the recently proposed technique for scheduling, based on applying CA (Seredynski 1998).

[*]on leave from Institute of Computer Science, Polish Academy of Sciences, Ordona 21, 01-237 Warsaw, Poland.

The remainder of the paper is organized as follows. The next section introduces CA and the scheduling problem, and then it outlines the idea of CA-based scheduling algorithm. Section 3 provides design details of the CA. Section 4 describes the two-phase architecture the CA-based scheduler: the phase of discovering local rules, and the normal operation phase. Section 5 presents empirical evaluation of the discovered rules.

2 CELLULAR AUTOMATA AND MULTIPROCESSOR SCHEDULING

2.1 SCHEDULING PROBLEM

The two processor system is represented by an undirected unweighted graph, called a *system graph*, consisting of two nodes representing processors and a single edge representing bidirectional channels between processors. A parallel program is represented by a weighted directed acyclic graph $G_p = < V_p, E_p >$, called a *precedence task graph*, or a *program graph*. V_p is the set of N_p nodes of the graph, representing elementary tasks. Figure 1 (upper) shows examples of the program graph consisting of four tasks ordered from 0 to 3, and the system graph representing a system consisting of two processors *P0* and *P1*.

The program graph has some number of parameters or attributes. The number b_k, associated with node k, denotes the processing time needed to execute task k on any (homogeneous) processor of the system. For example, task 0 in Figure 1 (upper) has $b_0 = 1$. Edges of the task graph describe the communication patterns between the tasks. Each edge (k,l) has weight a_{kl}, describing communication time needed to send intermediate results of computation from task k to task l in a given multiprocessor architecture. All edges of the program graph from Figure 1 (upper) have the same communication time equal to 1.

We assume that, for each node k of the precedence task graph, sets of *predecessor s(k)* , *brothers(k)* (i.e. nodes having at least one common predecessor), and *successors(k)* are defined. For example, task 0 has neither predecessors nor brothers, but it has two successors - tasks 1 and 2. In turn, task 1 has one predecessor (task 0), one brother (task 2) and one successor (task 3). We also assume that, for each node k of the precedence task graph, some parameters such as *static* and *dynamic level* and *co-level* can be defined (El-Rewini et al. 1994).

The purpose of *scheduling* is to distribute the tasks among the processors in such a way that the precedence constraints are preserved, and the *response time* T (the total execution time) is minimized. The response time for a given allocation of tasks in a multiprocessor topology depends on the *scheduling policy* applied in a processor. We assume that the scheduling policy is fixed for a given run and the same for all processors.

We intend to use CA to solve the scheduling problem. The following section outlines the basic definition and the necessary notions concerning CA.

2.2 CELLULAR AUTOMATA

A one dimensional CA (Wolfram 1984) is a collection of two-state cells, arranged in a lattice and locally interacting in parallel, in discrete time steps t. For each cell i, called the central cell, a neighborhood of radius r is defined, consisting of $n_i = 2r + 1$ cells (including cell i).

It is assumed that state q_i^{t+1} of cell i, at the time $t+1$, depends only on states of its neighborhood at time t. That is, $q_i^{t+1} = f_g(q_i^t, q_{i1}^t, q_{i2}^t, ..., q_{ni}^t)$, where f_g, called a *general rule*, defines rules for updating cell i. The length L_g of the general rule, which is the number of neighborhood states for a binary uniform CA, is $L_g = 2^{n_i}$, where n_i is a number of cells of the neighborhood. The number of such rules is 2^{L_g}, which has fast growth with L_g. For this reason, it may be beneficial to use other rules of shorter length. One possible choice is to define the transition function based not on individual cells of the neighborhood but on their average behavior. For example, let us define function f_t based of the sum of cells: $q_i^{t+1} = f_t(q_{i-r}^t + ... + q_{i-1}^t + q_i^t + q_{i+1}^t + ... + q_{i+r}^t)$. This will dramatically reduce rule length (we will call this a totalistic rule). Of course, one of the question we need to investigate is whether this trade-off is indeed beneficial.

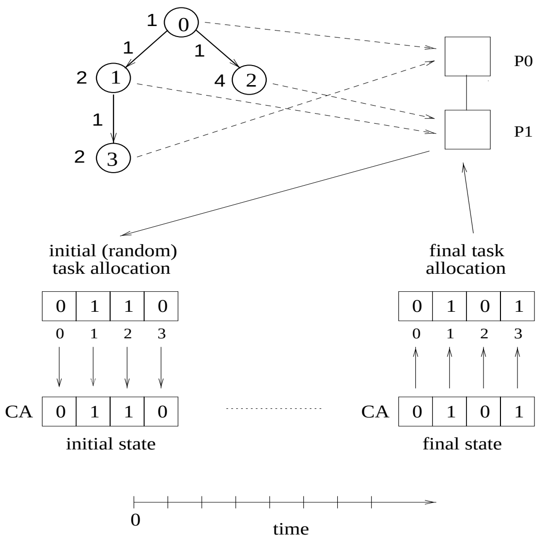


Figure 1: An idea of CA-based scheduler: an example of a program graph and a system graph (upper), a concept of corresponding CA-based scheduler (lower)

2.3 CA-BASED SCHEDULER

We outline the concept of a CA-based scheduler in the following way. An elementary automaton is associated with each task of the program graph. The topology of the program graph defines the structure of the CA. Therefore, the structure of the CA is not regular. However, each elementary automaton is binary – since we consider two-processor architectures, we use state 0 (1) of a cell to indicate that the corresponding task is allocated to processor $P0$ ($P1$).

The CA-based scheduler is illustrated in Figure 1. Initially, the program tasks are randomly allocated to the processors. For example, task allocation $(0, 1, 1, 0)$ indicates allocation of tasks 0 and 3 to processor $P0$, and tasks 1 and 2 to processor $P1$. Also, an elementary automaton is associated with each node of the program graph. Next, the CA starts to evolve, according to some predefined rule. Changing states of the evolving CA corresponds to changing the allocation of tasks in the system graph, what results in changing the response time T. The final state of the CA corresponds to the final allocation of tasks in the system (Figure 1 (lower-right)).

To design the architecture of a CA-based scheduler some essential questions must be considered. For example: how to define a local neighborhood, and what transition function to use. We would like to use a generic definition of neighborhood, transparent to the various kinds, sizes, and shapes of potential program graphs. We use the following strategy: neighborhood for an elementary automaton, associated with a task of the program graph, is based only on the sets sets of the task's predecessors, brothers and successors. Even then, potential neighborhood may vary in size – we will use additional means to deal with this problem.

An related question is that of potential irregularities in program graphs. For example, some tasks have no predecessors. For that, we extend the program graph by adding dummy nodes and taking this into account when coding neighborhoods.

The ideas presented above only outline architectural details for the CA-based scheduler. The actual architecture will be more complex.

3 DEFINING LOCAL NEIGHBORHOOD

3.1 SELECTED NEIGHBORHOOD

Selected neighborhood uses only two representatives of each of the sets of predecessors, brothers, and successors, in forming the neighborhood. The representatives are selected on the basis of respectively maximal and minimal values of some attributes of tasks in the given set. As we stated in Section 2, the following attributes are assigned to task k of the program graph: a_{kl}, b_k, static level, dynamic level, and static and dynamic co-levels. In a given run of the scheduling algorithm, one attribute for each set of predecessors, brothers and successor is selected. The attributes selected for each set may be different.

Selected neighborhood uses 7 cells (including cell k). The part of the neighborhood corresponding to the two associated representatives will be referred as a *sub-neighborhood* of the cell k.

Because the structure of a program graph and corresponding CA is irregular, the number of predecessors, brothers or successors may be less than two or they may have the same values of attributes, the following solutions for special cases have been accepted:

- if predecessors (brothers or successors) do not exist for a given task, a sub-neighborhood corresponding to such a situation is created by adding a pair of dummy tasks and associating with them a pair of cells; states of these cells (denoting processors where the tasks are allocated) are undefined and the state of such a sub-neighborhood takes a special value
- if only one predecessor (brother or successor) exists for a given task, the sub-neighborhood corresponding to this situation is created by adding a single dummy task/cell; the state of this cell will be the same as the state of existing cell (i.e., it is assumed that a dummy task is allocated to the same processor as the real task in the sub-neighborhood)
- if the number of predecessors (brothers or successors) is greater than two and all of them have the same value of an attribute, then we select two different tasks with the smallest and largest order number (as seen in Figure 1 (upper))

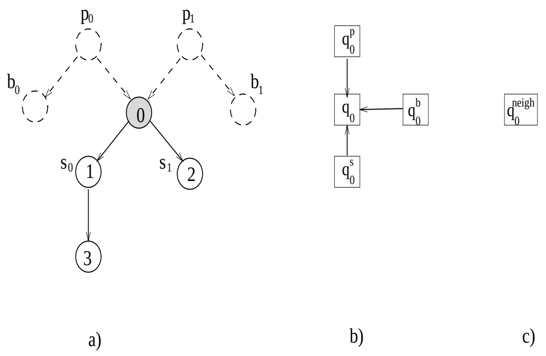


Figure 2: Selected neighborhood: creating a neighborhood for the task 0 from Figure 1 (a), a state of the cell 0 depends on states of sub-neighborhoods created by predecessors, brothers and successors (b), a state of a neighborhood of the cell 0 is evaluated (c)

Figure 2 illustrates neighborhoods created for tasks of the program graph from Figure 1. Task 0 does not have predecessors, so two dummy task-predecessors p_0 and p_1 are created (Figure 2a). For the same reason, two dummy task-brothers b_0 and b_1 are created. Real tasks 1 and 2 are considered as task-successors s_0 and s_1 of the task 0.

After constructing the neighborhood, it is necessary to define states of the sub-neighborhoods q_k^p, q_k^b, and q_k^s (Figure 2b), and the state q_k^{neigh} of the selected neighborhood (Figure 2c). The central cell takes the value 0 or 1. Values of each pair of cells are mapped into one of five values describing the state of the pair in the following way:

- state 0: values of both cells of the pair are the same and equal to 0 (both tasks corresponding to cells are in the processor *P0*)
- state 1: the first cell takes value 0, and the second one takes value 1 (corresponding tasks are in *P0* and *P1* respectively)

- state 2: the first cell takes value 1, and the second one takes value 0 (corresponding tasks are in *P1* and *P0* respectively)
- state 3: values of both cells of the pair are the same and equal to 1 (both tasks corresponding to cells are in the processor *P1*)
- state 4: values of cells are undefined (there are no tasks corresponding to these cells).

Because the state q_k of the central cell may take two values ({0,1}), and states q_k^p, q_k^b, q_k^s of respective sub-neighborhoods may take five values ({0,1,2,3,4}), the total number of states of the neighborhood is $2 * 5 * 5 * 5 = 250$. The length of a rule is 250 bits, and thus there are 2^{250} possible transition functions.

A GA will be used to search the space for the best CA rule.

3.1.1 FULL NEIGHBORHOOD WITH TOTALISTIC RULES

To reduce the number of states of CA another definition of a neighborhood called *a full neighborhood* was considered. It is assumed under this approach that a neighborhood of a given cell k is constituted of single cells, each representing the corresponding set of predecessors, brothers and successors. If some sets are empty then we add dummy tasks to the program graph, and we also introduce a special state for the corresponding cells. The state of a cell, representing a given set, will be defined in a way similar to that how totalistic rules are defined. The state of the central cell will be defined as previously by an order number of a processor where the task is allocated. Thus, the neighborhood will consists of 4 cells.

Figure 3 shows how such a totalistic neighborhood is created for tasks for the program graph from Figure 1. A dummy task-predecessor is added to create the set P_0 of predecessors of task 0. In the same way, the set B_0 of its brothers is created. Tasks 1 and 2 create the set S_0 of its successors.

As previously, we will assume that some attributes for tasks from corresponding sets of neighbors of a given task were selected. Let $\mathcal{P}_{P0}^k$ and $\mathcal{P}_{P1}^k$ be the sets of predecessors of a task k, allocated in processors $P0$ and $P1$, respectively; $\mathcal{B}_{P0}^k$ and $\mathcal{B}_{P1}^k$ be the sets of brothers allocated in $P0$ and $P1$, respectively; $\mathcal{S}_{P0}^k$ and $\mathcal{S}_{P1}^k$ be the sets of successors allocated in $P0$ and $P1$, respectively.

Let us calculate, for the sets $\mathcal{S}_{P0}^k$ and $\mathcal{S}_{P1}^k$, values of some totalistic measure concerning the accepted attribute. This totalistic measure can be $\sum_{i \in \mathcal{P}_{P_0}^k} d_i$, and $\sum_{j \in \mathcal{P}_{P1}^k} d_j$, respectively, where d is the value of the selected attribute. In a similar way, values for $\mathcal{B}_{P0}^k$, $\mathcal{B}_{P1}^k$, $\mathcal{S}_{P0}^k$ and $\mathcal{S}_{P1}^k$ are also defined.

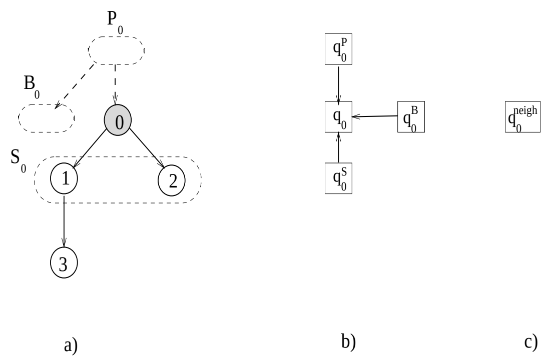


Figure 3: Full totalistic neighborhood: creating a neighborhood for the task 0 from Figure 1 (a), a state of the cell 0 depends on states of sub-neighborhoods created by predecessors, brothers and successors (b), a state of a neighborhood of the cell 0 is evaluated (c)

The state of a given sub-neighborhood cell (e.g., for predecessors) is defined as follow:

- state 0: if $\sum_{i \in \mathcal{P}_{P_0}^k} d_i > \sum_{j \in \mathcal{P}_{P1}^k} d_j$
- state 1: if $\sum_{i \in \mathcal{P}_{P_0}^k} d_i < \sum_{j \in \mathcal{P}_{P1}^k} d_j$
- state 2: if $\sum_{i \in \mathcal{P}_{P_0}^k} d_i = \sum_{j \in \mathcal{P}_{P1}^k} d_j$
- state 3: a set of a given sub-neighborhood contains only a dummy task.

States of remaining cells are defined in a similar way. After defining states of sub-neighborhoods, the state of the neighborhood of the central cell is defined (Figure 3b, c). The total number of neighborhood states, taking into account states of defined above sub-neighborhoods, is $2 * 4 * 4 * 4 = 128$. The rule length is now 128 bits, and the total number of rules for the problem is 2^{128}.

4 CA-BASED SCHEDULER

4.1 ARCHITECTURE OF THE SCHEDULER

Figure 4 presents the architecture of the CA-based scheduler. The scheduler operates in two modes: the learning mode (Figure 4, left) and the operating mode (Figure 4, right).

In the learning mode, CA rules are discovered by the GA. It is expected that the rules will be suitable to solve the scheduling problem for any initial allocation of tasks for a given instance of the problem. Tasks of the program graph representing a given instance of the problem are initially randomly allocated to processors of the parallel system. The CA is built for the program graph and a predefined type of a local neighborhood.

An initial population of GA containing rules is created. For a given random allocation of the program graph into the system graph, the CA is initialized and equipped with the rule from the population of rules. CA evolves its states during predefined number of steps, what results in changing the allocation of tasks of the program graph.

The response time T for the final allocation is evaluated. For a given rule this evaluation procedure is repeated predefined number of times for test problems represented by different initial allocations. This results in evaluation of some fitness value T^* for the rule, which is the sum of T values corresponding to the individual runs, and modified to convert a problem of minimization (scheduling problem) to a problem of maximization requested by GA.

After evaluation of the entire population, genetic operators are involved. The evolutionary process continues a predefined number of generations, after which the discovered rules are stored (Figure 4, right).

In the normal operation mode, the program graph used in the learning mode is randomly allocated, CA is initiated and equipped with a rule taken from the set of discovered rules. We expect that in this mode, for any initial allocation of tasks of the given program graph, CA will be able to find, in a finite number of steps, allocation of tasks providing the minimal value of T.

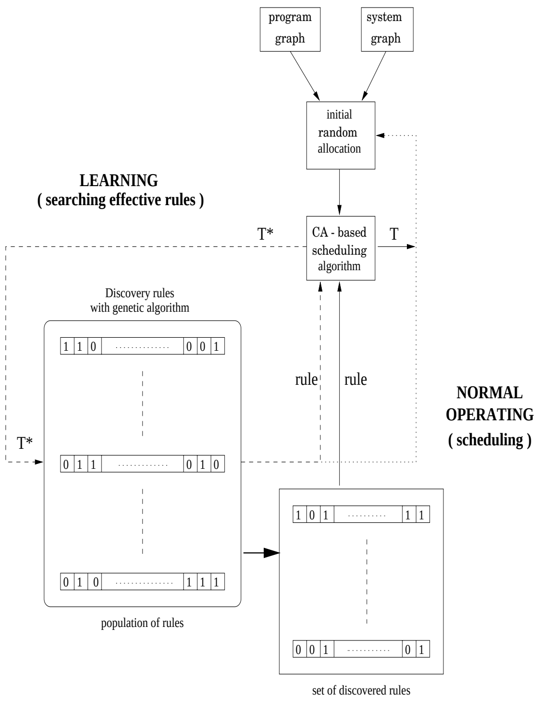


Figure 4: An architecture of CA-based scheduler

4.2 DISCOVERY CA RULES BY GA

The GA (Das et al. 1996, Goldberg 1989) used to discover CA rules for the CA-based scheduler is described below.

GA to discover CA rules:

```
t=0
create an initial population $P()$ of size n_pop of rules
WHILE termination_condition NOT TRUE
BEGIN
  create a set of a size n_test of test problems
  IF hill-climbing_condition TRUE
    THEN hill-climbing
  FOR $i = 1$ TO n_pop
    BEGIN
      $T^*_{ij} = 0$
      FOR $j = 1$ TO n_test
        $T^*_{ij} = T^*_{ij} +$ CA($rule_i$, $test_j$, $seq/par$,
            $CA\_steps$)
    END
  sort $P()$ in decreasing order
  move $E$ of the best individuals from $P(t)$ to $P(t+1)$
  FOR $k = 1$ TO $n\_pop - E$
    REPEAT
      $rule_1^{parent}$=select()
      $rule_2^{parent}$=select() $\neq rule_1^{parent}$
      ($rule_1^{child}$, $rule_2^{child}$)= crossover($rule_1^{parent}$,
                                                  $rule_2^{parent}$)
      mutation($rule_1^{child}$, $rule_2^{child}$)
    UNTIL Hamming ($rule_1^{child}$, $rules$)>=$H$
      AND
            Hamming ($rule_2^{child}$, $rules$)>=$H$
  $t = t + 1$
END
problem_solution = the best rules from $P()$
```

After creating an initial population $P()$ of random rules of the CA and a set of test problems, each rule is tested by running the CA. The CA can run in one of two modes: sequential (seq) and in parallel (par). Each run of the CA lasts a predefined number $CA - steps$ of time steps. After evaluation of the fitness function T^*_i of each rule, the rules are sorted in decreasing order. The best E rules are moved to the

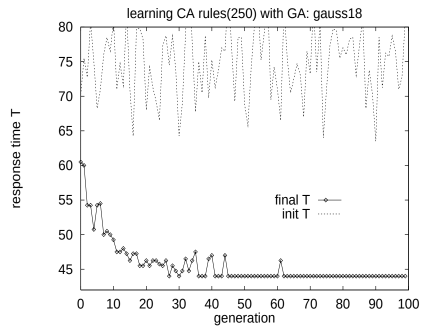


Figure 5: Learning mode: rules for CA-based scheduler are discovered by GA

currently created population $P(t+1)$. To the remaining rules of the $P(t)$ genetic operators of selection (select), crossover and mutation are applied. New rules are accepted to the $P(t+1)$ if their Hamming distance to the rules from the $P(t)$ is equal or greater than a predefined number H. If the fitness function of rules is not improved during a predefined number of generations then an operator of hill-climbing is applied. The evolutionary process is continued a predefined number of generations, and when is completed discovered rules are stored.

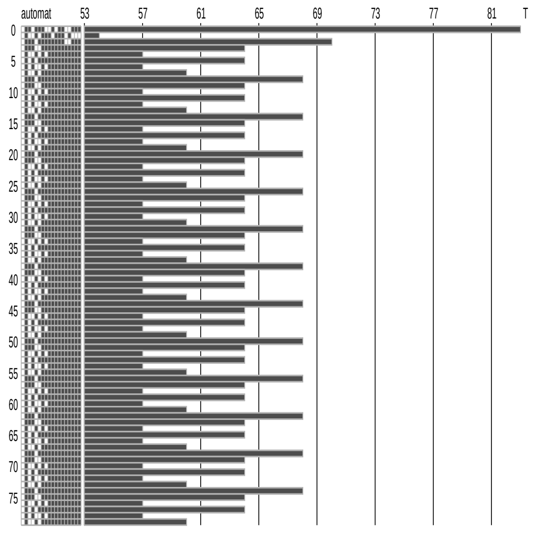
In experiments reported in the paper it is assumed that the CA works sequentially, i.e. at a given moment of time only one cell updates its state. An order of updating states by cells is defined by their order number corresponding to tasks in the precedence task graph. A single step of running the CA is completed in N_p (N_p - a number of tasks of a program graph) moments of time. A run of CA consists of the CA_steps.

A program graph used in the experiment presented in the paper represents the parallel Gaussian elimination algorithm consisting of 18 tasks. We refer to this program graph as $gauss18$ (Seredynski 1998).

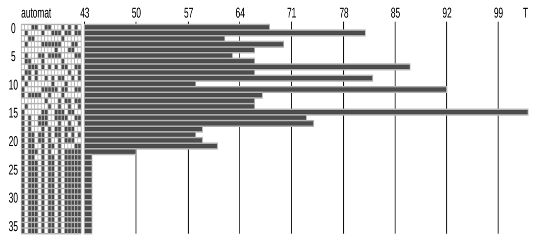
The CA neighborhood was created using the idea of the selected neighborhood. To calculate T for a given final allocation of tasks a scheduling policy of the type: a task with the highest value of a dynamic level-first, was applied.

In the learning mode of the scheduling algorithm a population of rules of a size 100 was used, and the learning process was observed during 100 generations. A proportional selection with elitist strategy was used. A crossover with a probability $p_c = 0.95$, a bit-flip mutations with $p_m = 0.001$ were applied.

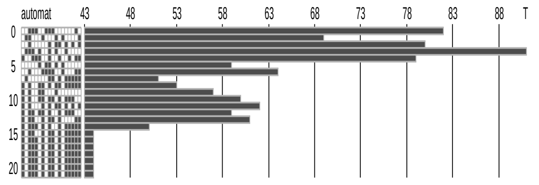
Figure 5 shows a response time T changing during the evolutionary process. In the conducted experiments, for each generations of GA the set of four test-problems



a)



b)



c)

Figure 6: Space-time diagrams of CA-based scheduler with the best rule found for $gauss18$ in generation 5 (a), generation 50 (b), and generation 100 (c)

was created.

Figure 6a shows a run of the CA-based scheduler with the best rule found in the 5-th generation. Left part of the figure presents a space-time diagram of the CA consisting of 18 cells, and the right part shows graphically a value of T corresponding to the allocation found in a given step. One can see that after the step 0, cells of the CA are in some states corresponding to allocation of tasks (white cell - a corresponding task is allocated in $P0$, black cell - a task is allocated in

$P1$), and the value of T corresponding to this allocation is greater than 81. After few steps, the CA starts to oscillate, repeating a sequence of six states with resulting patterns of task allocation, and corresponding changing values of T. In generation 46, the GA discovers (see Figure 5) a rule providing an allocation with an optimal value $T = 44$.

The found rule is, however, not absolutely the best. The rule does not pass a test on a test problem created in generation 62 (see, Figure 5). The GA quickly modifies this rule and it passes successfully all subsequent tests. Figure 6c shows a space-time diagram of such a rule existing in the generation 100.

4.3 NORMAL OPERATING

After run of the GA its population contains rules suitable for CA-based scheduling. Quality of these rules we can find out in the operating mode. We generate some number of test problems, and use them to test each of found rules. Figure 7 shows results of the test conducted with 100 random initial allocation of the $gauss18$. For each found rule the average value of T (avr T) found by CA in the test problem is shown. One can see that 29 rules are able to find an optimal scheduling for each representative of the test.

5 COMPETITION BETWEEN RULES

A number of experiments with program graphs available in the literature and referred as $gauss18$, $g18$, $g40$, $outtree15$, $outtree63$ and $outtree127$ has been conducted (Seredynski 1998). In all these experiments the GA was able to discover rules of the CA to solve a given instance of the scheduling problem. The question which arises is whether discovered rules are suitable only to solve one specific instance problem, or they can be used also to solve the other instances. To find it out, discovered rules were used in operating mode to solve other instances problems.

Figures 8 and 9 show how the best rules discovered for the $gauss18$ solve the scheduling problem for the binary out-tree $outtree15$. Figure 8 shows that some number of the best CA rules found for the $gauss18$ can be effectively applied to solve the scheduling problem for the $outtree15$. These rules are able to find the optimal solution $T = 9$ (Figure 9) or in the average solutions near to the optimal (Figure 8).

Figures 10 and 11 show how the best rules discovered for the $outtree15$ solve the scheduling problem for the $gauss18$. One can see (Figure 11) that these rules

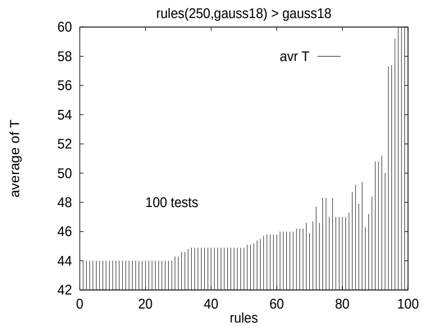


Figure 7: Normal operating: testing discovered rules

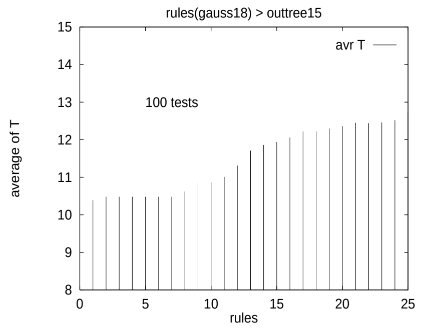


Figure 8: Rules found for $gauss18$ applied to solve $outtree15$ instance problem

have never been able to find the optimal solution which is $T = 44$, and that their average performance (Figure 10) is worse that the performance of the rules discovered for $gauss18$.

Analyzing experiments conducted with other rules we come to a conclusion that some rules found for one instance of the problem are more general and effective than the other rules. The rules can be ordered according their importance and this information can be used to solve new instance problems.

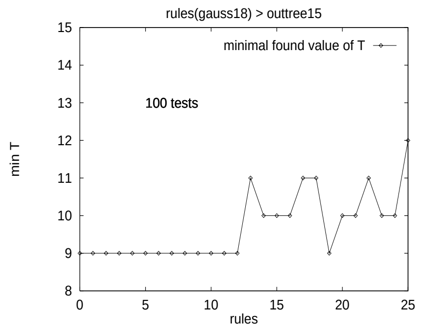


Figure 9: The best solutions of $outtree15$ found by rules of $gauss18$

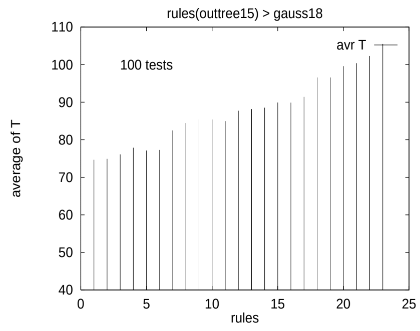


Figure 10: Rules found for *outtree*15 applied to solve *gauss*18 instance problem

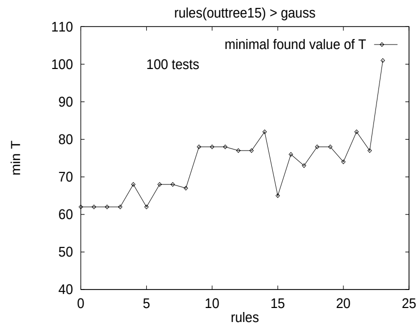


Figure 11: The best solutions of *gauss*18 found by rules of *outtree*15

6 CONCLUSIONS

We have presented in the paper a systematic approach to designing the CA-based scheduling algorithms. We have considered the questions of constructing local neighborhoods for the CA on the base of program graphs. We described the CA-based scheduler working either in the mode of discovering the CA rules by the GA, or in the operating mode. The results of conducted experiments are very promising. They show that the GA is able to discover CA rules suitable to solve the scheduling problem for a given instance of the problem. The preliminary results also show that discovered rules may be used to find optimal or suboptimal solutions of other, not known in advance instances of the problem.

Acknowledgments

The work has been partially supported by the State Committee for Scientific Research (KBN) under Grant 8 T11A 024 11.

References

I. Ahmad, Y. Kwok (1995). A parallel approach for multiprocessor scheduling, In *Proc. of Ninth Int. Parallel Processing Symposium*, Santa Barbara, CA, The IEEE Press

D. Andre, F. H. Bennet III and J. R. Koza (1996). Discovery by Genetic Programming of a Cellular Automata Rule that is Better than any Known Rule for the Majority Classification Problem. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.), *Genetic Programming*, Proceedings of the First Annual Conference 1996, A Bradford Book, The MIT Press

R. Das, M. Mitchell, and J. P. Crutchfield (1994). A genetic algorithm discovers particle-based computation in cellular automata. In Davidor Y., Schwefel H.-P., Männer R. (eds.). *Parallel Problem Solving from Nature – PPSN III*, 344-353. LNCS 866, Springer

H. El-Rewini, T. G. Lewis, and H. H. Ali (1994). *Task Scheduling in Parallel and Distributed Systems*, PTR Prentice Hall

M. Capcarrère, A. Tettamanzi, M. Tomassini and M. Sipper (1998). Studying Parallel Evolutionary Algorithms: The Cellular Programming Case, In A. E. Eiben, T. Back, M. Schoenauer and H.-P . Schwefel (eds.), *Parallel Problem Solving from Nature - PPSN V*, 573-582. LNCS 1498, Springer

D. E. Goldberg (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA

W. Hordijk, J. P. Crutchfield and M. Mitchell (1998). Mechanisms of Emergent Computation in Cellular Automata, In A. E. Eiben, T. Back, M. Schoenauer and H.-P . Schwefel (eds.), *Parallel Problem Solving from Nature - PPSN V*, 613-622. LNCS 1498, Springer

D. Mange & M. Tomassini (eds.) (1997). Bio-Inspired Computing Machines. Towards Novel Computational Architectures. Press Polytechnique et Universitaires Romandes

F. Seredynski (1998). Discovery with Genetic Algorithm Scheduling Strategies for Cellular Automata. In A. E. Eiben, T. Back, M. Schoenauer and H.-P. Schwefel (eds.), *Parallel Problem Solving from Nature - PPSN V*, 643-652. LNCS 1498, Springer

M. Sipper (1997). *Evolution of Parallel Cellular Machines.* The Cellular Programming Approach, LNCS 1194, Springer

S. Wolfram (1984). Universality and Complexity in Cellular Automata, *Physica D* 10, 1-35