

# Sequential and Parallel Cellular Automata-Based Scheduling Algorithms

Franciszek Seredynski and Albert Y. Zomaya, *Senior Member, IEEE*

**Abstract**—In this paper, we present a novel approach to designing cellular automata-based multiprocessor scheduling algorithms in which extracting knowledge about the scheduling process occurs. This knowledge can potentially be used while solving new instances of the scheduling problem. We consider the simplest case when a multiprocessor system is limited to two-processors, but we do not imply any limitations on the size and parameters of parallel programs. To design cellular automata corresponding to a given program graph, we propose a generic definition of program graph neighborhood, transparent to the various kinds, sizes, and shapes of program graphs. The cellular automata-based scheduler works in two modes. In learning mode we use a genetic algorithm to discover rules of cellular automata suitable for solving instances of a scheduling problem. In operation mode, discovered rules of cellular automata are able to automatically find an optimal or suboptimal solution of the scheduling problem for any initial allocation of a program graph in two-processor system graph. Discovered rules are typically suitable for sequential cellular automata working as a scheduler, while the most interesting and promising feature of cellular automata are their massive parallelism. To overcome difficulties in evolving parallel cellular automata rules, we propose using coevolutionary genetic algorithm. Discovered this way, rules enable us to design effective parallel schedulers. We present a number of experimental results for both sequential and parallel scheduling algorithms discovered in the context of a cellular automata-based scheduling system.

**Index Terms**—Cellular automata, coevolution, genetic algorithms, multiprocessor scheduling, two-processor systems.

## 1 INTRODUCTION

AN increasing number of research problems and real-life applications need massively parallel computing. It is still not clear how future computing devices offering needed enormous computational power will look. A great hope today are naturally and bio-inspired nonstandard computational techniques, such as *neural networks, genetic algorithms or simulated annealing, and new emerging computational paradigms, such as immune systems, molecular computation, computation in cellular automata, and quantum computing*. One can notice an increasing number of publications [7], [14], [19], [37], workshops, and conferences [4], [8], [11], [24] devoted to these methods.

To the most interesting or promising results obtained with use of bio-inspired techniques and recently reported belongs discovery [3] of rules for the majority classification problem, which are better than currently known human rules, successfully solving a number of communication problems like, fraud detection [5] in mobile-phone systems or solving problems related to financial economics like, an optimization portfolio problem [16].

Multiprocessor scheduling belongs to a special category of computational problems. On one hand, it is closely related to the issue of practical performance of current and future computers. On the other hand is the problem, even

limited to the simplest case considered in this paper, when we have to work with the two-processor system, but any parallel program is an example of computationally difficult an unsolved research problem, known as an NP-complete problem [10].

Current works concerning the scheduling problem are oriented to either derived exact solutions [2], [9] for selected problems for which such solutions can be found or designing heuristic algorithms to find usually near optimal solutions. In the latter case, effective heuristics like, list scheduling [35], clustering scheduling algorithms [12], or critical path-based heuristics [17] were developed. The prevailing majority of these scheduling algorithms are sequential ones and a new perspective direction in this area is developing parallel scheduling algorithms [1].

Commonly recognized weaknesses of the above mentioned heuristic scheduling algorithms is their sensitivity to scheduling parameters, a lack of scalability and determinism which is unable, in general, to reach optimal solutions. Developing stochastic global search techniques based on natural and bio-inspired methods opened new possibilities to deriving good quality solutions. *Heuristics based on genetic algorithms (GA) [18], [39], [40], neural networks, and simulated annealing [20], [23], are used effectively today to solve scheduling problems.*

*Today, heuristic scheduling algorithms have passed a long way in their evolution to be able to produce high quality solutions, but they are still the subject of intensive study to improve their performance.* While scheduling quality has been significantly improved, one of the main problems remains the minimization of scheduling overhead represented by cost of running the scheduler. One of the main sources of scheduling overhead is neglecting potential

- F. Seredynski is with the Polish-Japanese Institute of Information Technologies, Koszykowa 86, 02-008 Warsaw, Poland, and the Institute of Computer Science, Polish Academy of Sciences, Ordona 21, 01-237 Warsaw, Poland. E-mail: sered@ipipan.waw.pl.
- A.Y. Zomaya is with the School of Information Technologies, The University of Sydney, Sydney, NSW 2006 Australia. E-mail: zomaya@it.usyd.edu.au.

Manuscript received 20 June 2001; accepted 14 Feb. 2002.

For information on obtaining reprints of this article, please send e-mail to: [tpds@computer.org](mailto:tpds@computer.org), and reference IEEECS Log Number 114384.

knowledge about the scheduling problem which could be gained during solving instances of the scheduling problem.

Note that the prevailing number of scheduling algorithms do not extract, conserve, and reuse any knowledge about the problem while solving instances of the scheduling problem. Applying GA for scheduling may serve as a typical illustration of this situation. To run a GA-based scheduler, a random initial population of potential solution is created and the population is evolved with the use of genetic operators until a solution is found. To find a solution of a new instance of the scheduling problem, the instance which is only a modification of the previous instance, or a composition of some instances solved earlier, general knowledge about previous solutions cannot be used. A new searching process must be started from the beginning by creating an initial random population of potential solutions.

The motivation of our work is to develop a framework for designing scheduling algorithms where knowledge about scheduling process can be extracted and potentially used for solving new instances of scheduling problem. We focus our attention in the paper on the first issue, extracting knowledge about the scheduling process available during solving instances of the scheduling problem. For this purpose, we propose using a recently emerged and very promising hybrid technique combining evolutionary computation and computation with cellular automata (CA).

The CA presents a highly parallel and distributed system of single, locally interacting units which are able to produce a global behavior [22], [32], [36]. The CA can be considered as a model of naturally existing systems produced by natural evolution. Such systems are capable of producing globally coordinated information processing, unguided by any global criterion or central control. Information processing capabilities of such systems are not explicitly represented in their components but rather in their interconnections. These capabilities are more powerful than ones done by elementary components or their combinations. For these reasons, CA has been used to model different physical and biological phenomena such as fluid flow, galaxy formation, avalanches, earthquakes, growth of stony corals, and other biological and physical pattern formations.

Despite the known future of CA as machines which are capable of universal computation, in the sense of a Turing machine, these capabilities were not explored well enough due to huge spaces of local CA rules representing possible solutions. Most applications of CA were, therefore, a result of clever, but time-consuming, hand-designing rather than an oriented search. Only recent works [6], [3] on applying evolutionary computation and, in particular, GA to design CA opened new possibilities for doing it automatically. Recent results show that such CA systems, combined with evolutionary techniques for discovering local rules, can be effectively used to solve complex problems such as classification and synchronization [6], [3], [31] or cryptography [34]. We follow this line of research and, in this paper, we review and extend the recently proposed technique for scheduling, based on applying GA and CA [26], [28].

The remainder of the paper is organized as follows: Section 2 presents the scheduling problem. Section 3 gives an overview of CA. Section 4 presents the concept of multiprocessor scheduling with the use of CA. Section 5 contains experimental results concerning sequential CA applied to

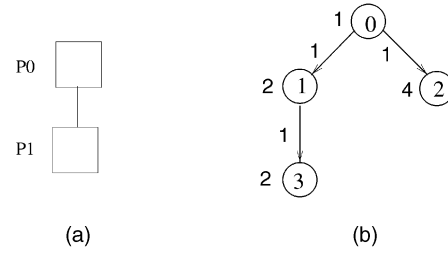


Fig. 1. Examples of (a) a system graph and (b) a precedence task graph.

scheduling. Section 6 describes the coevolutionary GA-based engine for discovering parallel CA scheduler and presents experimental results concerning the discovery with coevolutionary GA scheduling rules. Finally, Section 7 contains conclusions and discusses future works.

## 2 MULTIPROCESSOR SCHEDULING PROBLEM

A multiprocessor system is represented by an undirected unweighted graph  $G_s = (V_s, E_s)$ , called a *system graph*.  $V_s$  is the set of  $N_s$  nodes of the system graph representing processors with their local memories of a parallel computer of MIMD architecture.  $E_s$  is the set of edges representing bidirectional channels between processors and defines a topology of the multiprocessor system. Fig. 1a shows an example of a system graph representing a multiprocessor system consisting of two-processors  $P0$  and  $P1$ . This topology will be used in all experiments presented in this work. It is assumed that all processors have the same computational power and communication via links does not consume any processor time.

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, which are indivisible computational units. There exists a precedence constraint relation between the tasks  $k$  and  $l$  in the precedence task graph if the output produced by task  $k$  has to be communicated to the task  $l$ .

A program graph has two attributes: weights  $b_k$  and weights  $a_{kl}$ . Weights  $b_k$  of the nodes describe the processing time (computational cost) needed to execute a given task on any processor of a given multiprocessor system.  $E_p$  is the set of edges of the precedence task graph describing the communication pattern between the tasks. Weights  $a_{kl}$  of the edges describe communication time (communication cost) between pairs of tasks  $k$  and  $l$  when they are located in the neighboring processors. If the tasks  $k$  and  $l$  are located in the same processor, then the communication delay between them will be equal to 0.

Fig. 1b shows an example of the program graph consisting of four tasks with their order numbers from 0 to 3. All communication costs of the program graph are equal to 1 (see marked edges). Computational costs of tasks (marked on their left side) are 1, 2, 4, and 2, respectively.

The purpose of the *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. Found optimal schedule is

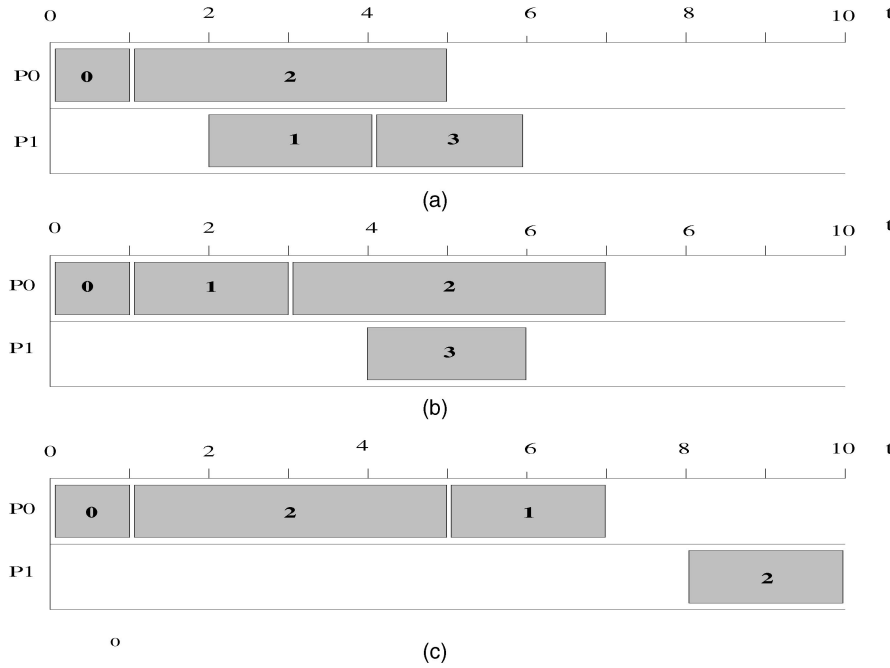


Fig. 2. A schedule represented by a Gantt chart for a problem from Fig. 1. (a) Optimal schedule. (b) and (c) Schedules for different scheduling policies.

usually represented by a Gantt chart (see Fig. 2a) showing allocation of tasks to processors and time when a given task starts and finishes execution.

For the purpose of the next sections, we will introduce some additional notions. We assume that, for each node  $k$  of a precedence task graph, there are defined sets of *predecessors*( $k$ ), *brothers*( $k$ ) (i.e., nodes having at least one common predecessor), and *successors*( $k$ ). For example, node 1 from Fig. 1b has the set of *predecessors*(1) = {0}, the set of *brothers*(1) = {2}, and the set of *successors*(1) = {3}. **Nodes without predecessors will be called starting nodes and nodes without successors will be called exit nodes.**

We also assume that two additional attributes can be defined for each node  $k$  of a precedence task graph: the level and the colevel. **The level  $h_k$**  of a node  $k$  is defined as

$$h_k = \begin{cases} b_k, & \text{for an exit node} \\ \max_{l \in \text{successors}(k)} (h_l + a_{kl}) + b_k, & \text{for other nodes,} \end{cases} \quad (1)$$

i.e., it is the maximal length of the longest path from a node  $k$  to an exit node. **The colevel  $d_k$**  of a node  $k$  is defined as

$$d_k = \begin{cases} b_k, & \text{for a starting node} \\ \max_{l \in \text{predecessors}(k)} (d_l + a_{lk}) + b_k, & \text{for other nodes,} \end{cases} \quad (2)$$

i.e., it is the length of the longest path from the starting node to node  $k$ . Values of the level and the colevel of a given task are *static* and do not depend on the allocation of a program graph in the processors of a parallel system. However, if we calculate them for a task of a program graph allocated in the system graph, these values will depend on the allocation. We will call values of the level or colevel calculated for tasks of a program graph allocated in the system graph the *dynamic level* or *colevel*, respectively.

The response time  $T$  for a given precedence task graph depends on *allocation* of tasks in multiprocessor topology and *scheduling policy* applied in individual processors:

$$T = f(\text{allocation}, \text{scheduling\_policy}). \quad (3)$$

Let us assume that tasks 0, 1, and 2 from Fig. 1b are allocated in processor  $P0$  and the task 3 in the  $P1$ . After execution of task 0, processor  $P0$  may choose for execution either task 1 or task 2, depending on a scheduling policy. Figs. 2b and 2c show two different response times  $T$  corresponding to the same allocation of tasks, but different scheduling policies. We will assume that a scheduling policy is fixed for a given run of a scheduling algorithm and is the same for all processors.

### 3 CELLULAR AUTOMATA

One-dimensional CA [33], [36] is a collection of two-state (binary) elementary automata (cells) arranged in a lattice of length  $N$  and interacted locally in a discrete time  $t$ , usually in a parallel and synchronous way. Fig. 3a shows the example of such a CA. White or black color of a cell denotes its actual state 0 or 1, respectively. For each cell  $i$ , called the central cell, a neighborhood of a radius  $r$  is defined. Figs. 3b and 3c show examples of the neighborhood of cell  $i$  of radius  $r = 1$  and  $r = 2$ , respectively, consisting of  $n_i = 2r + 1$  cells, including cell  $i$ .

It is assumed that, a state  $q_i^{t+1}$  of the cell  $i$  at the time  $t + 1$  depends only on states of its neighborhood at the time  $t$ , i.e.,

$$q_i^{t+1} = f(q_i^t, q_{i1}^t, q_{i2}^t, \dots, q_{ni}^t). \quad (4)$$

A transition function  $f$  defines a rule of updating cell  $i$ . It is usually assumed that a CA is uniform (homogeneous), i.e., the neighborhood relation and the transition function

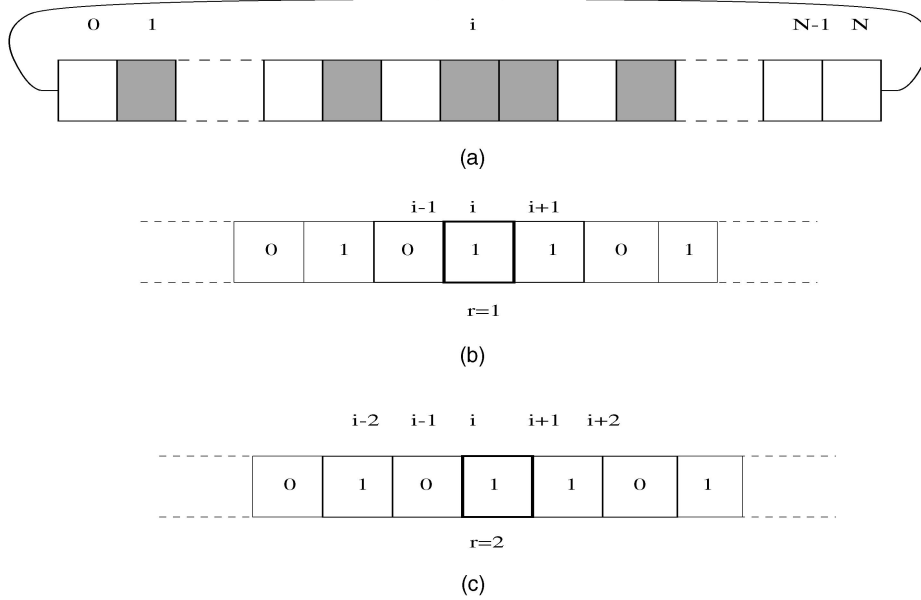


Fig. 3. (a) One-dimensional cellular automata of the length  $N$ , (b) a neighborhood of a radius  $r = 1$ , and (c) a neighborhood of a radius  $r = 2$ .

are the same for every cell, including cells 0 and  $N - 1$ . For such a CA, we can rewrite the dependence (4) as

$$q_i^{t+1} = f(q_{i-r}^t, \dots, q_{i-1}^t, q_i^t, q_{i+1}^t, \dots, q_{i+r}^t). \quad (5)$$

If we consider a neighborhood of a radius  $r = 1$ , then a set of possible neighborhood states is  $\{000, 001, \dots, 111\}$ . Table 1 lists all these states and shows the example of the transition function  $f_g$ .

A rule represented by this function and called the *general rule* says that if, at time  $t$  a neighborhood of the cell  $i$  is  $\{011\}$ , then the state of the cell at time  $t + 1$  should be equal to 0.

A length  $L_g$  of the general rule and a number of neighborhood states for a binary uniform CA is  $L_g = 2^n$ , where  $n = n_i$  is the number of cells of a given neighborhood and the number of such rules can be expressed as  $2^{L_g}$ . For a CA with  $r = 2$ , the length of the rule is equal to  $L_g = 32$  and the number of such rules is  $2^{32}$  and grows very fast with  $L_g$ . For this reason, some other types of rules are used to make them shorter and decrease their total number.

## 4 MULTIPROCESSOR SCHEDULING WITH CELLULAR AUTOMATA

### 4.1 A Concept of Cellular Automata-Based Scheduler

We will assume that an elementary automaton (cell) is associated with each task of the program graph. 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  $P_0$  ( $P_1$ ). The concept of the CA-based scheduler is illustrated in Fig. 4.

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  $P_0$ , and tasks 1 and 2 to processor  $P_1$  (see Fig. 4 (upper)). An initial state of CA corresponding to the program graph is set according to the initial allocation of tasks (see Fig. 4 (lower left)). 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  $T$  (see (3)). The final state of the CA corresponds to the final allocation of tasks in the system (Fig. 4 (lower right)).

To construct the CA-based scheduler, one must find answer on several questions:

1. What is the topological structure of proposed CA: linear, as shown in Fig. 4 (lower left), or nonlinear, related in some way to the topological structure of a program graph;
2. What kind of a local neighborhood of a program graph is the most appropriate to design corresponding CA; and
3. How to find in a huge space of CA rules, the rule capable of solving the scheduling problem.

In the approach we adopt, the structure of the CA is nonlinear and corresponds to the topology of the program graph. We will use a generic definition of neighborhood, transparent to the various kinds, sizes, and shapes of potential program graphs. We use the following strategy:

TABLE 1  
Example of a General Rule for One-Dimensional CA with a Radius  $r = 1$

|                       |     |     |     |     |     |     |     |     |
|-----------------------|-----|-----|-----|-----|-----|-----|-----|-----|
| state number          | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| state of neighborhood | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| $f_g$                 | 0   | 1   | 1   | 0   | 1   | 1   | 0   | 1   |



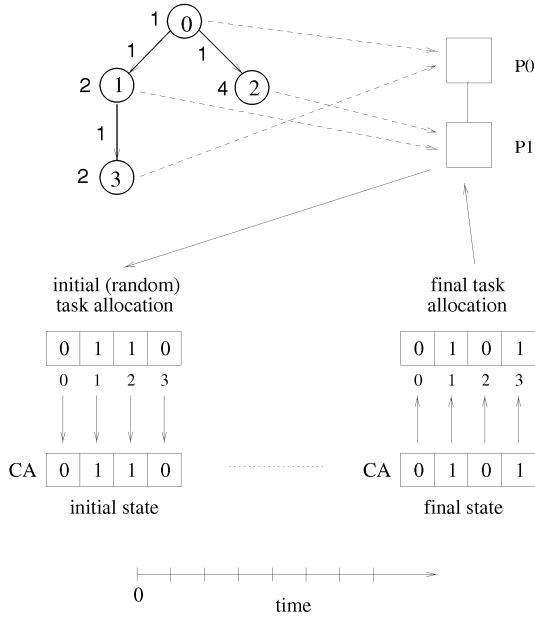


Fig. 4. An idea of a CA-based scheduler: an example of a program graph and a system graph (upper), corresponding CA performing scheduling (lower).

The neighborhood for an elementary automaton, associated with a task of the program graph, is based only on the sets of the task's predecessors, brothers and successors. Even then, the potential neighborhood may vary in size—we will use additional means to deal with this problem. A related question is that of potential irregularities in program graphs. For example, some tasks have no predecessors (brothers, successors). For that, we extend the program graph by adding dummy nodes and taking this into account when coding the state of neighborhoods.

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

## 4.2 Selected Neighborhood

A neighborhood of a central task consists of three subneighborhoods and includes this task. Each subneighborhood of a cell associated with a task  $k$  is created only by two selected representatives of a set of predecessors, brothers, and successors, respectively. 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 earlier, the following attributes are assigned to task  $k$  of the program graph:  $a_{kl}$ ,  $b_k$ , static level, dynamic level, and static and dynamic coveles.

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. If corresponding tasks of a subneighborhood are missing in a program graph, dummy tasks are introduced. **So, the selected neighborhood of a given cell associated with a central task always consists of seven cells and includes this cell. This type of neighborhood we call a selected neighborhood.**

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

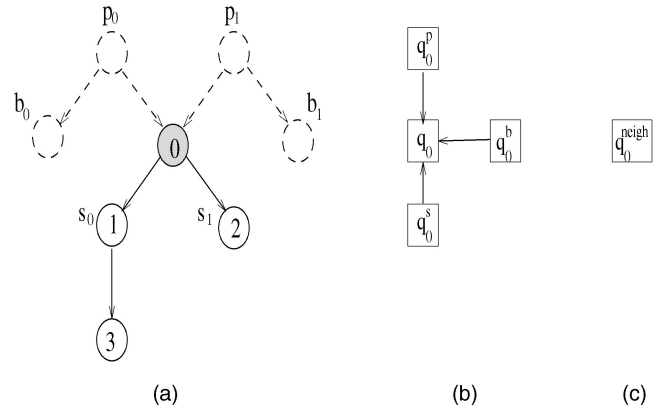


Fig. 5. (a) Selected neighborhood: creating a neighborhood for task 0 from Fig. 1, (b) a state of cell 0 depends on the states of subneighborhoods created by predecessors, brothers, and successors, and (c) the state of a neighborhood of the cell 0 is evaluated.

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 subneighborhood corresponding to such a situation is created by adding a pair of dummy tasks and associating with them a pair of cells; the states of these cells (denoting processors where the tasks are allocated) are undefined and the state of such a subneighborhood takes a special value.
- If only one predecessor (brother or successor) exists for a given task, the subneighborhood 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 the existing cell (i.e., it is assumed that a dummy task is allocated to the same processor as the real task in the subneighborhood).
- 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.

Fig. 5 illustrates neighborhoods created for task 0 of the program graph from Fig. 1. Task 0 does not have any predecessors, so two dummy task-predecessors  $p_0$  and  $p_1$  are created (Fig. 5a). 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 task 0. After constructing neighborhoods for all cells associated with tasks, it is necessary to define states of the subneighborhoods  $q_k^p$ ,  $q_k^b$ , and  $q_k^s$  (Fig. 5b) and the state  $q_k^{neigh}$  of the selected neighborhood (Fig. 5c).

The central cell associated with task  $k$  takes the value 0 or 1. Values of each pair of cells corresponding to subneighborhoods 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  $P0$ ).

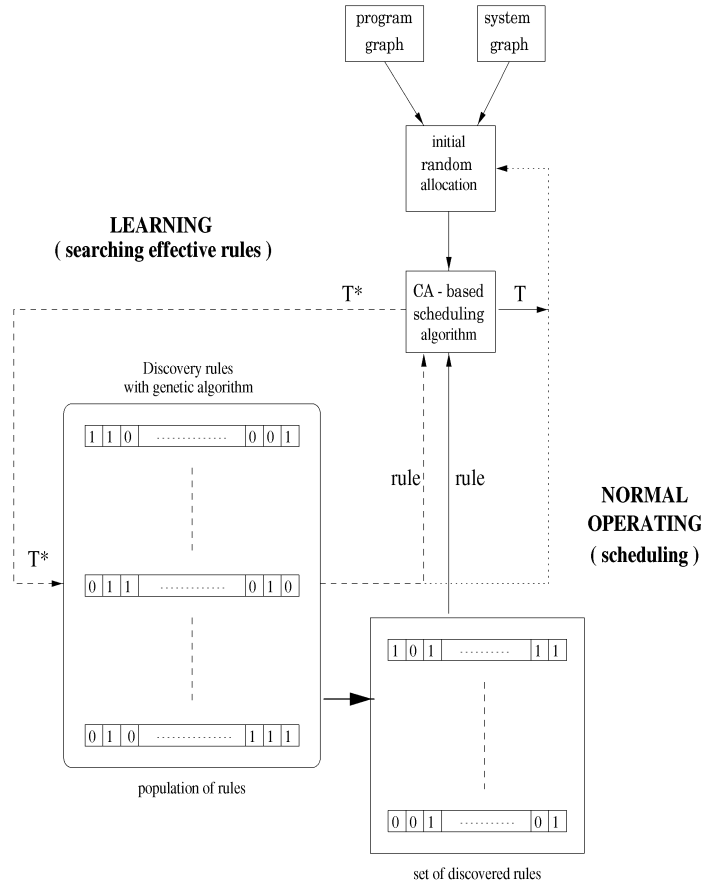


Fig. 6. An architecture of a CA-based scheduler.

- 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 state  $q_k$  of the central cell may take two values ( $\{0, 1\}$ ) and states  $q_k^p$ ,  $q_k^b$ , and  $q_k^s$  of respective subneighborhoods 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.

State  $q_k$  of the central cell is updated according to such a function. GA will be used to search the space for the best rule, i.e., a rule of CA providing a solution of scheduling problem.

### 4.3 Discovery of CA Rules for Scheduling

Fig. 6 presents the architecture of the CA-based scheduler. The scheduler operates in two modes: learning mode (Fig. 6 (left)) and operation mode (Fig. 6 (right)).

In the learning mode, CA rules are discovered by the GA. It is expected that discovered 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 CA rules is created and a set of test problems—initial random allocations of tasks of the program graph is generated. States of the CA are initialized according to the first test problem and the CA is equipped with the rule from the population of rules. CA starts to evolve, changing its states during predefined number of steps, which 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 a set of test problems represented by different initial allocations. This results in evaluation of some fitness value  $T^*$  for the rule, which is the sum of values  $T$  corresponding to the individual runs.

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 (Fig. 6 (right)).

In the 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, the CA will be able to find, in a finite number of steps, allocation of tasks providing the minimal value of  $T$ .

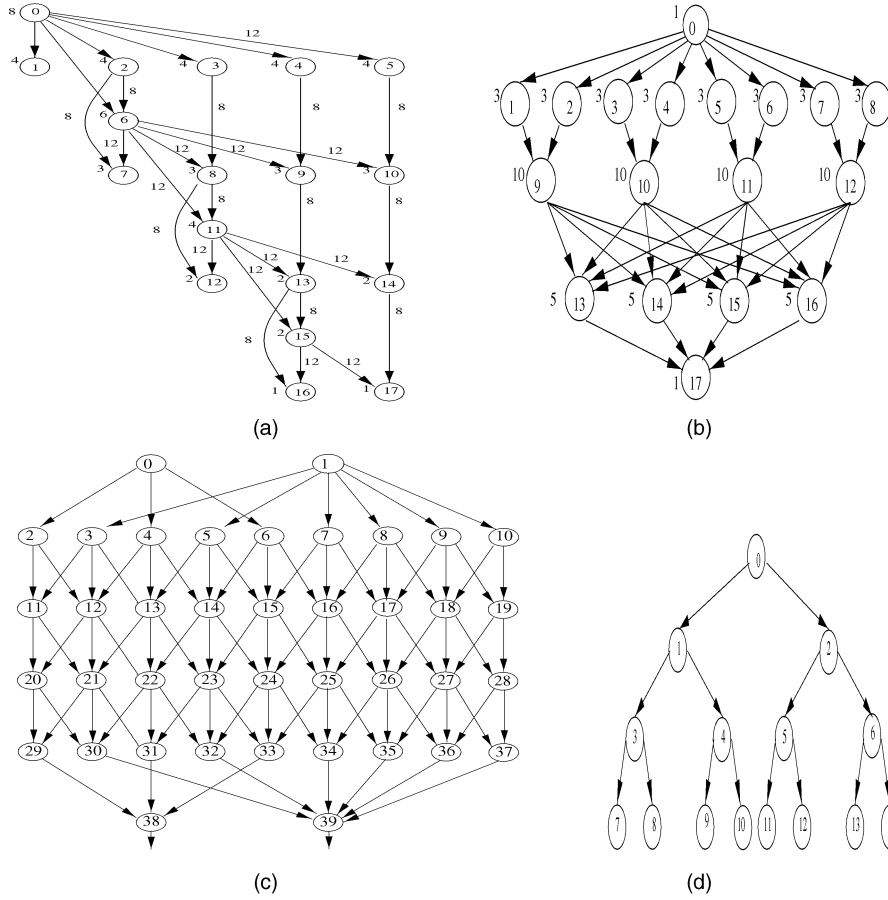


Fig. 7. Program graphs: (a) *gauss18*, (b) *g18*, (c) *g40*, and (d) *tree15*.

The GA [6], [13], [21] 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

**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  $P()$  according to  $T_i^*$

move  $E$  of the best individuals from  $P(t)$  to  $P(t+1)$

**FOR**  $k = 1$  **TO**  $n_{pop} - E$

**REPEAT**

$rule_1^{parent} = \text{select}()$

$rule_2^{parent} = \text{select}() \neq rule_1^{parent}$

$(rule_1^{child}, rule_2^{child}) = \text{crossover}(rule_1^{parent}, rule_2^{parent})$

$\text{mutation}(rule_1^{child}, rule_2^{child})$

**UNTIL**  $\text{Hamming}(rule_1^{child}, rules) \geq H$

**AND**

$\text{Hamming}(rule_2^{child}, rules) \geq H$

$t = t + 1$

**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. The best  $E$  rules (with minimal  $T_i^*$ ) are moved to the 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 to or greater than a predefined number  $H$ . A new set of test problems is created for rules in a new generation. The evolutionary process is continued a predefined number of generations. When it is completed, discovered rules are stored.

After the run of the GA, its population contains rules suitable for CA-based scheduling. We can find the quality of these rules in the operation mode. We can generate a number of test problems and use them to test each of the rules we found.

## 5 SEQUENTIAL CA FOR SCHEDULING

In the experiments reported in this section, it is assumed that the CA works asynchronously, i.e., at a given moment

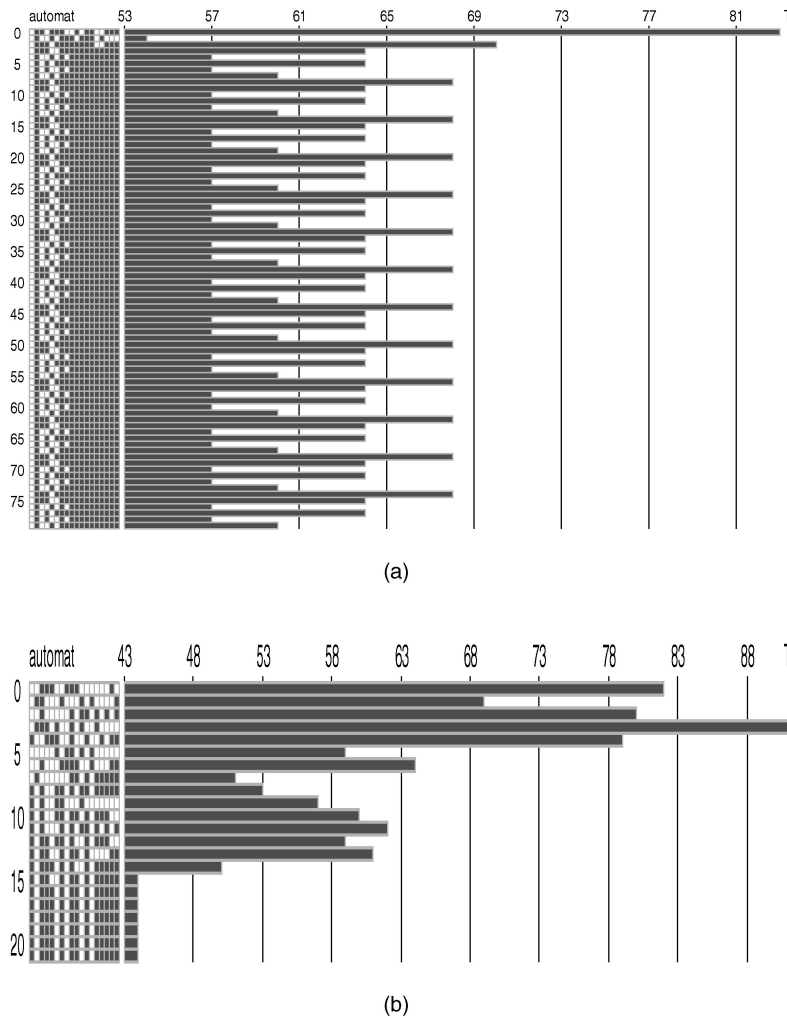


Fig. 8. Space-time diagrams of sequential CA-based scheduler with the best rule found for *gauss18* (a) in generation 5 and (b) generation 100.

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).

A number of experiments with program graphs available in the literature have been conducted. The first program graph referred as *gauss18* [17] is shown in Fig. 7a. It represents the parallel Gaussian elimination algorithm consisting of 18 tasks. The next program graph *g18* is shown in Fig. 7b [9]. Computational costs of tasks are shown in the figure. Communication costs are all the same and equal to 1. Fig. 7c presents a program graph *g40* with computational and communication costs equal to 4 and 1, respectively. Fig. 7d shows a binary out-tree program graph. We refer to it as *tree15*. Also, we use binary out-trees *tree63* and *tree127*. Computation and communication weights of out-trees are equal to 1.

**Experiment #1: Program Graph *gauss18*.** In the learning mode of this experiment, a population of rules of GA was equal to 100. Figs. 8a and 8b present runs of the CA-based scheduler for the rules found by the GA in the fifth and the 100th generation. The left part of the figures 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 performance by the CA of step 0 (see Fig. 8a), all cells are in some states corresponding to allocation of tasks (white cell—a corresponding task is allocated in  $P_0$ , black cell—a task is allocated in  $P_1$ ) and the value of  $T$  corresponding to this allocation is greater than 81. After a few steps, the CA starts to oscillate, repeating a sequence of six states with resulting patterns of task allocation and corresponding changing values of  $T$ . Fig. 9a shows a response time  $T$  corresponding to the initial tasks' allocations (*init*  $T$ ) and final tasks' allocations (*final*  $T$ ) corresponding to the best rule in a given generation.

For each generation of the GA, a new set of four test-problems is created. The CA with a given rule and an initial state corresponding to a given initial allocation is allowed to run 100 steps. An efficiency of a given rule is evaluated as the average value of response times found for each test-problem. To calculate  $T$  for a given final allocation of tasks, a scheduling policy of the type of a task with the highest value of a dynamic level-first, was applied. After evaluation of all rules from a population, GA operators are applied: Elitist strategy is applied to



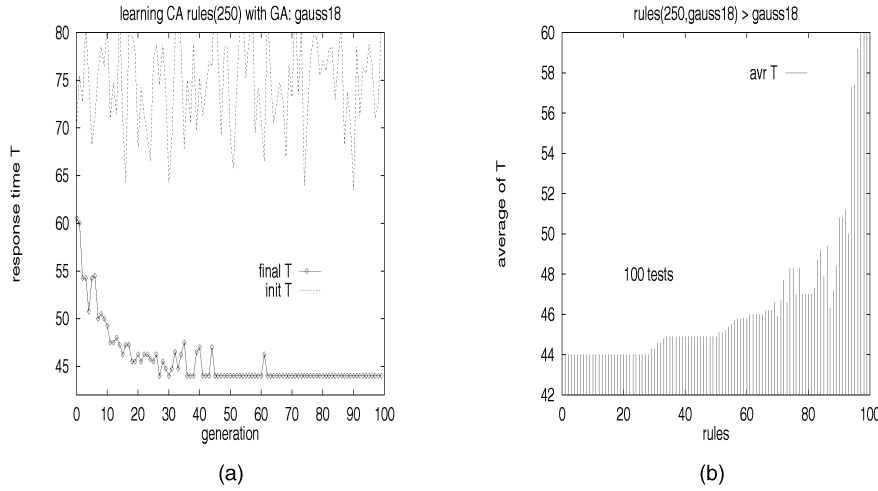


Fig. 9. Sequential CA-based scheduler for *gauss18*: (a) learning mode and (b) operation mode.

the set of the best rules, crossover with a probability  $p_c = 0.95$ , and a bit-flip mutation with  $p_m = 0.001$ .

One can notice that GA discovers (see Fig. 9b), in generation 46, 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 Fig. 9a). The GA quickly modifies this rule and it successfully passes all subsequent tests.

Fig. 8b shows a space-time diagram of such a rule existing in the generation 100. One can see that the CA-based scheduler working with the found rule needs about 15 time steps to find the tasks' allocation corresponding to the minimal value of  $T$ .

After the run of the GA, its population contains rules suitable for CA-based scheduling. The rules are denoted as *rules(250)*, which means that they were found for the selected neighborhood and the length of each rule is  $L = 250$ . The quality of these rules can be found in the operation mode. We generate a number of test problems and use them to test each of the found rules. Fig. 9b shows the results of the test conducted with an initial allocation of 100 random *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.

**Experiment #2: Program Graph.** A population of rules of size 200 was used in the learning mode in this experiment. For each of the GA generations, a set of five test problems was created. Fig. 10a shows that rules of CA providing optimal scheduling with a response time  $T = 80$  were found after 160 generations. Fig. 10b shows the performance of found rules evaluated in the operation mode. One can see that the best rules found in the learning mode provide near optimal solutions in the operation mode.

Fig. 11 shows a run of CA-based scheduler with the best found rule. One can see that the CA finds a steady-state corresponding to an allocation providing an optimal response time  $T = 80$  in step 14.

**Experiment #3: Program Graph *g18*.** The scheduling policy of the type, the highest value of a static level-first, was used in this experiment. GA needs about 20 generations (see Fig. 12a) to discover, in a learning process, a CA rule providing an optimal solution with  $T = 46$ . Fig. 12b

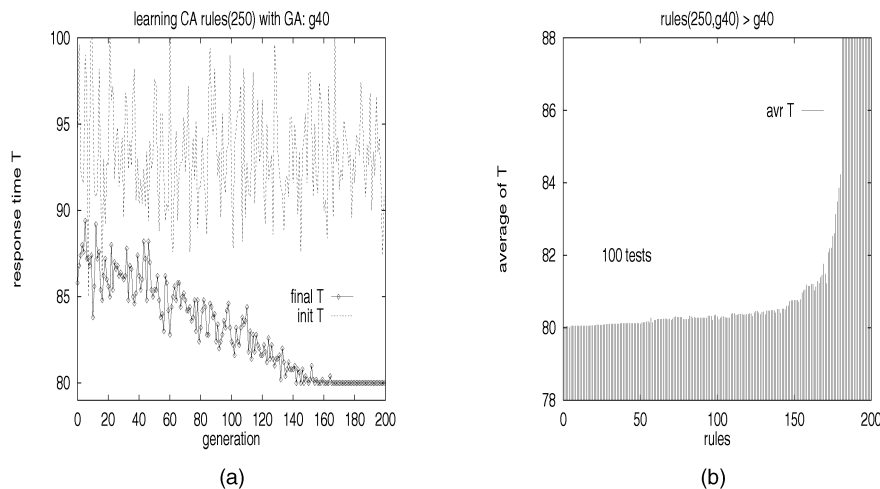


Fig. 10. Sequential CA-based scheduler for *g40*: (a) learning mode and (b) operation mode.

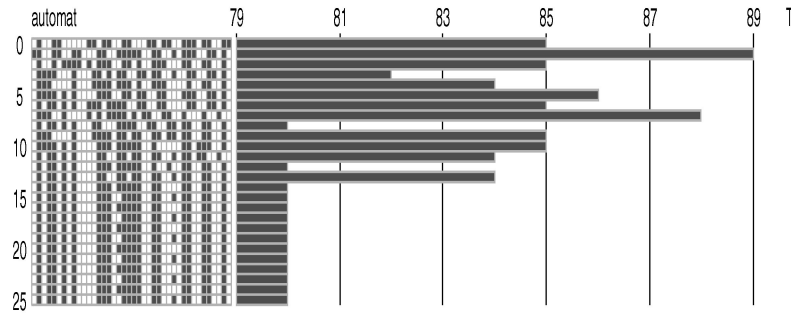


Fig. 11. Space-time diagram of a CA-based scheduler for *g40*.

shows the performance of found rules evaluated in the operation mode. Near 20 generations, the best rules were found in the learning mode, providing the optimal solution in the operation mode. Fig. 13 shows a space-time diagram of the CA-based scheduler for the best found rule for this instance problem.

**Experiment #4: Program Graphs *tree15*, *tree63*, and *tree127*.** The GA needs less than five generations (not shown) to discover a CA rule for the program graph *tree15* providing an optimal response time  $T = 9$ . The experiment was conducted with scheduling policy of the type: the lowest order number of a task-first. CA rules discovered the small size of a binary out-tree can be effectively used in the operation mode to schedule binary out-trees of much greatest sizes. Found CA rules are able to solve the scheduling problem with the binary out-tree consisting with much more tasks. Fig. 14 shows a space-time diagram of CA working with the same rule and solving the problem with the binary out-tree *tree127* consisting of 127 tasks.

## 6 PARALLEL CA FOR SCHEDULING

Results of experimental study presented in the previous section, obtained with the version of the CA-based scheduler running under DOS have shown that GA was

able to discover effective rules for scheduling for a number of program graphs from the known literature. However, discovered rules were working in a deterministic sequential mode of CA, i.e., only one cell could update its state in time. The order of updating was predefined by numbering the tasks in a program graph. This means that one of the most interesting features of CA—their massive parallelism—was not explored. For this reason, an attempt to develop a new enhanced Windows'98 version of the scheduler was undertaken. The main feature of the scheduler is a new, much more powerful coevolutionary GA-based engine for discovery CA rules and some visualization tools enabling tracing the work of the scheduler.

### 6.1 Coevolutionary Genetic Algorithm for Discovery CA Rules

One of the most promising lines of research in the area of parallel evolutionary computing is a development of *coevolutionary algorithms* [27]. The idea of coevolutionary algorithms comes from the biological observation of natural selection, which shows that coevolving a number of *species* defined as collections of phenotypically similar individuals, is more realistic than simply evolving a population containing representatives of one species. So, instead of evolving a population of similar individuals representing a global solution, it is more appropriate to coevolve subpopulations of individuals representing specific parts of the global solution.

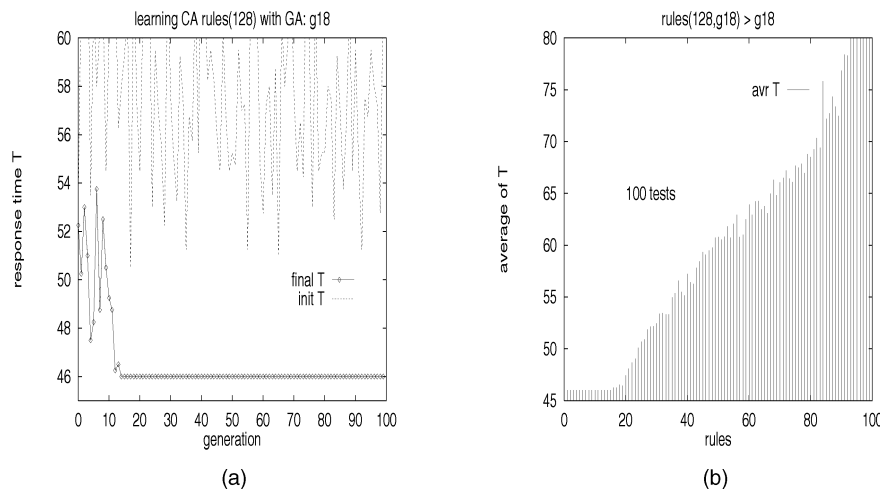
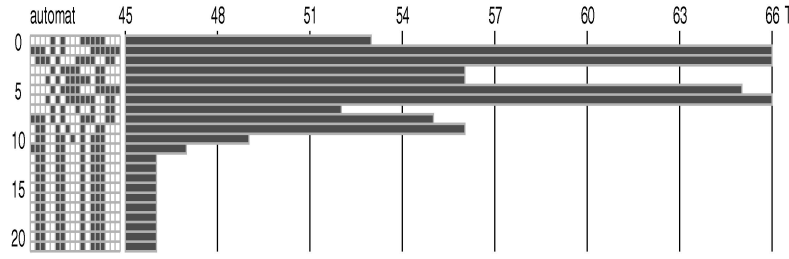


Fig. 12. Sequential CA-based scheduler for *g18*: (a) learning mode and (b) operation mode.

Fig. 13. Space-time diagrams of CA-based scheduler for  $g_{18}$ .

Among recently proposed coevolutionary algorithms is the *coevolutionary GA* [25], based on a *predator-prey* paradigm [15]. The algorithm is described below, with the use of a parallel processing language OCCAM-like notation. In particular, sequential and parallel processes are specified by SEQ and PAR constructors, respectively. Output process send!  $x$  and input process receive?  $y$  are used to denote sending a value  $x$ , receiving a value  $y$ , respectively. Comments concerning the algorithm follow the symbols — — .

#### Coevolutionary GA:

*chromosome 1*: global structure representing a solution of a problem

*chromosome 2*: additional structure representing constraints  $\bar{y}$  of a problem

*optimization criterion*: global function  $f(\bar{x}, \bar{y})$

*population 1*: main subpopulation  $P^1()$

*population 2*: additional subpopulation  $P^2()$

*population structure*: two interacting subpopulations

$t = 0$

SEQ

initialize  $P^1(t)$  and  $P^2(t)$

WHILE *termination\_condition* NOT TRUE

SEQ

$t = t + 1$

SEQ  $i = 1$  FOR  $n\_encounters$

SEQ

PAR  $j = 1$  FOR 2 — —

running coevolving subpopulations

SEQ

select individuals  $I_k^j(t)$  from  $P^j(t)$

confront selected individuals

evaluate result (fitness of individuals)  
of confrontation

select a pair of parents in both  
 $P^1(t)$  and  $P^2(t)$

crossover over pairs of parents

mutate in *children*

replace parents in  $P^1(t)$  and  $P^2(t)$

*problem\_solution* = the best individual  $\bar{x}$

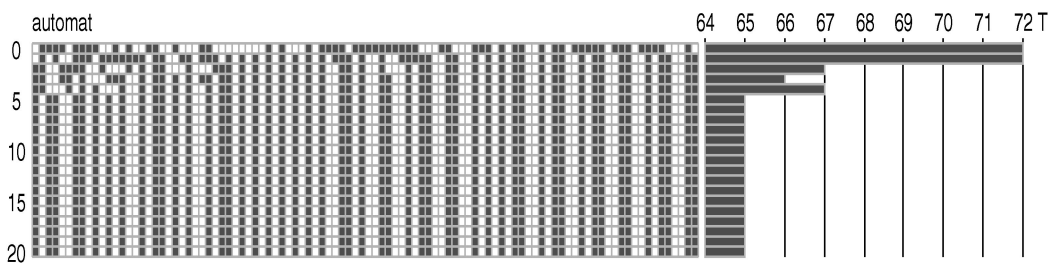
from the subpopulation  $P^1(t)$ .

The algorithm operates on two subpopulations: the main subpopulation  $P^1()$  containing individuals  $\bar{x}$  and an additional subpopulation  $P^2()$  containing individuals  $\bar{y}$  coding some constraints, conditions, or simply test points concerning a solution  $\bar{x}$ . Both, or only one, subpopulation, evolve to optimize a global function  $f(\bar{x}, \bar{y})$ .

A single act of coevolution is based on independent selection of individuals  $\bar{x}$  and  $\bar{y}$  from subpopulations, to encounter them and evaluate their  $f(\bar{x}, \bar{y})$ . The manner of assigning a fitness to the individuals stems from the predator-prey relation. The success of one individual should be a failure of the second one. During one generation, individuals are confronted a predefined number  $n\_encounters$  times. At the end of the evolution process, the best individual from  $P^1()$  is considered as a solution of a problem.

In the case of the CA-based scheduler, the main population of the coevolutionary GA contains the  $N^{main}$  CA rules and the additional population contains the  $N^{test}$  tests—the initial allocations of a program graph. During a given generation, each individual from the main population is tested, as previously stated, on each individual of the additional population. The same genetic operators as described earlier are applied to the main population. The additional population is initially randomly created, but, opposite to the previous version of the system, the set of tests in the next generations will be controlled by its own GA.

As a fitness function of an individual-test of the additional population, we choose the value of  $T_{test}^*$ , which is the average of final values of  $T$  obtained by all rules of the main population on this test. Genetic operators of *tournament* selection with elitism, crossover, and mutation are

Fig. 14. Space-time diagram of CA-based scheduler using rules(128,  $tree_{15}$ ) for  $tree_{127}$ .

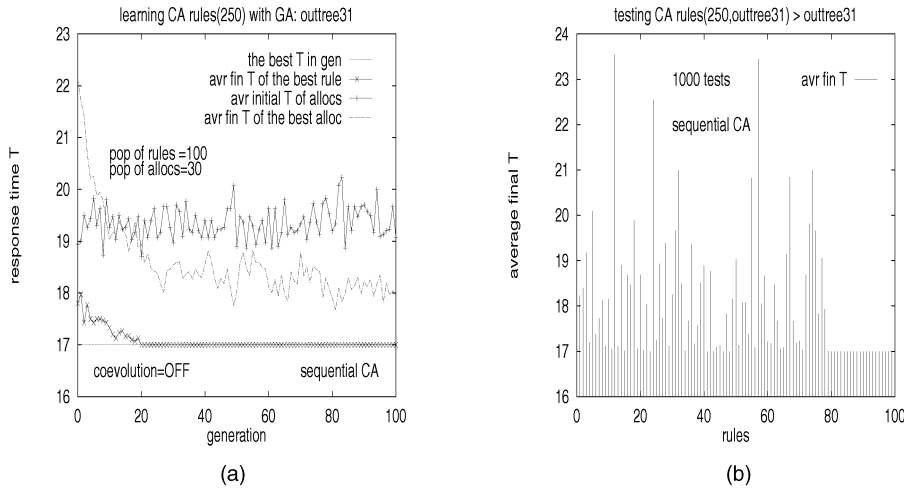


Fig. 15. Evolving scheduling rules for deterministic sequential CA: (a) learning mode of the scheduler and (b) operation mode.

applied to individuals of the population. We expect that this coevolutionary mechanism will develop more valuable tests for the population of rules, better than random population of tests, which should significantly improve the quality of discovered rules.

## 6.2 Experimental Results: *out - tree31* Case Study

The main purpose of the performed experiments was to study the influence of the coevolutionary GA on discovery scheduling rules for parallel CA. For this purpose, a program graph *tree31* was selected. The program graph consists of 31 tasks with computational and communication costs equal to 1 and is an enlarged version of *tree15*.

In all of the conducted experiments, the following parameters were used: The size of a population of rules  $N^{main}$  was equal to 100, with the size of elite equal to 20. Not only elite but all individuals from the population could take part in mating with a probability of crossover  $p_c^{main} = 0.9$  and a probability of mutation  $p_m^{main} = 0.1$ . A selected neighborhood was created using *level* as attributes of task-predecessors and task-brothers and *dynamic level* as an attribute of task-successors. To calculate  $T$  for a given allocation of tasks, a *scheduling\_policy* of the type a task with the highest value of a dynamic level-first was applied.

CA was allowed to run 25 time steps and the value of  $T$  corresponding to a final allocation of tasks was calculated as the average on the base of the three last steps of CA. The size of a population of tests  $N^{test}$  was equal to 30. When coevolution was turned on, the following genetic operators were applied to the population of tests: tournament selection and elite with size equal to 1, crossover with  $p_c^{test} = 0.9$ , and mutation with  $p_m^{test} = 0.005$ . The evolutionary process was observed during 500 generations.

**Experiment #5: Discovery of Rules for Deterministic Sequential CA.** In the experiment reported in this section, it is assumed that CA works sequentially and deterministically (as in previous experiments). A single step of 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 a predefined number  $G = 25$  of steps, with time steps equal to  $G * N_p = 25 * 31 = 775$ .

Fig. 15 shows the results of a typical experiment with evolving scheduling rules for deterministic sequential CA. Figs. 15a and 15b present learning and operation modes of the scheduler, respectively. The experiment is conducted without coevolution and one can see (Fig. 15a) that evolving scheduling rules for deterministic sequential CA is an easy problem for this type of a program graph.

GA discovers a rule providing an optimal scheduling with  $T = 17$  after about 20 generations (see *avr fin T of the best rule* in Fig. 15a). The average value of initial allocations *avr initial T of allocs* generated randomly in each generation of GA oscillates around a value of  $T_0 = 19.2$ . It means that rules exposed to test problems are, on the average, of the same degree of difficulty during the whole evolutionary process.

To see how difficult generated allocations are, we define, for each of them, the average final  $T$  over all rules from a population which was tested on this allocation. Observing the average final  $T$  of the most difficult allocation *avr fin T of best alloc* one can see that, generated randomly, tests become easier for rules in each generation. After generation 20, when the best rule was discovered, each statistical rule finds an allocation with final  $T$  better than a statistical initial allocation with corresponding  $T_0$ .

Fig. 15b shows the operation mode of the scheduler. In this mode, each rule in the final population is exposed to 1,000 random initial allocations of the program graph. The figure shows that the near 20 the best rules in the sequential CA find the optimal scheduling with  $T = 17$  in all tests.

**Experiment #6: Discovery of Rules for Parallel CA, without Coevolution.** We now assume a parallel work of CA, which means that, at a given moment of time all cells update their states. In this experiment, GA without coevolution is applied to discover rules for CA. Fig. 16 shows results of a typical experiment with evolving scheduling rules for parallel CA, without coevolution.

Fig. 16a shows the first 250 generations of the learning mode of the scheduler. One can see that the value of *avr fin T of the best rule* characterizing the best rule in each

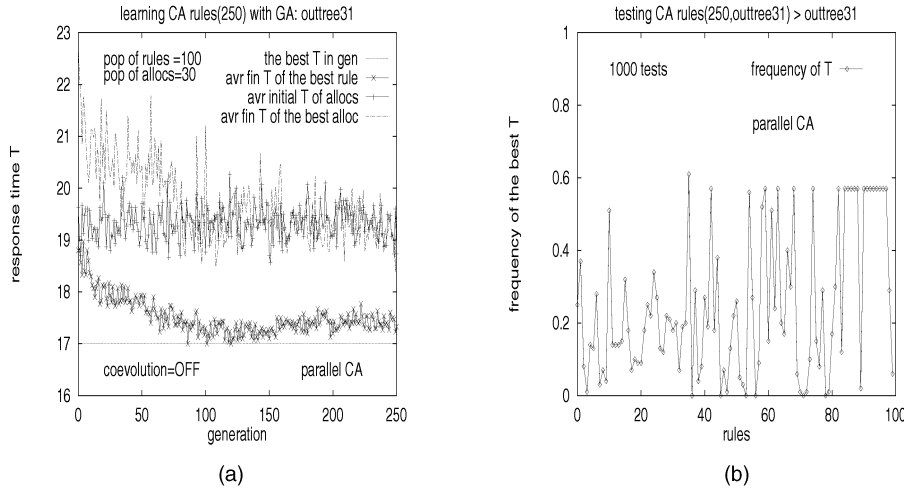


Fig. 16. Evolving without coevolution scheduling rules for parallel CA: (a) learning mode of the scheduler and (b) operation mode.

generation approaches the optimal value but never reaches it. It achieves its local minimum in generation 125 and stabilizes its value around 17.40 in about generation 200.

The average value *avr initial T of allocs* of initial allocations (see Fig. 16b) generated randomly in each generation of GA behaves in the same manner as in the previous experiment. However, observing the average final *T* of the most difficult test-allocation *avr fin T of best alloc*, one can notice that the value decreases only to generation 125, only as long as the best rule improves its quality. After this generation, no new valuable sequence of initial allocations appears in the set of tests to be exposed to the population of rules. Therefore, the learning process in the population of rules is stopped and a better rule for parallel CA will be not discovered. The corresponding value of the *avr initial T of allocs* becomes equal to average value  $T_0$  of initial allocations.

Fig. 16b shows the operation mode of the scheduler, when each rule in the final population is exposed to 1,000 random initial allocations of the program graph. The figure shows the frequency of convergence of CA with a given rule to the allocation corresponding to the optimal

value of  $T = 17$ . One can see that the best rules found for parallel CA converge to the optimal  $T$  in only nearly 60 percent of the cases.

#### Experiment #7: Discovery of Rules for Parallel CA, with Coevolution.

In this experiment, we assume that we have to do the with parallel CA-based scheduler and we apply GA-based engine with coevolution to discover rules for CA. Fig. 17 shows results of the experiment.

One can see (Fig. 17a) that GA with coevolution discovers the best rule providing convergence of parallel CA-based scheduler to the optimal value of  $T = 17$  in about 100 generations. The dynamic of changing the value of *avr fin T of the best rule* in each generation is different than the one in the experiment without coevolution. Also, the behavior of the average value *avr initial T of allocs* of initial allocations (see Fig. 17a) is different. One can notice that improvement of *avr fin T of the best rule* is correlated with changing *avr initial T of allocs*.

The coevolution mechanism which controls changing initial allocations makes the average value *avr initial T of allocs* of initial allocations perform a number of hill climbings, with subsequent falling down, instead of

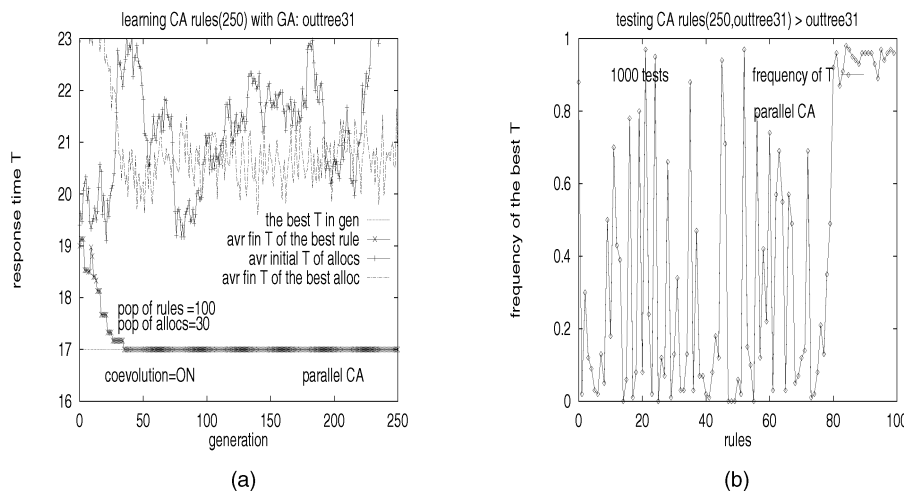


Fig. 17. Evolving with coevolution scheduling rules for parallel CA: (a) learning mode of the scheduler and (b) operation mode.

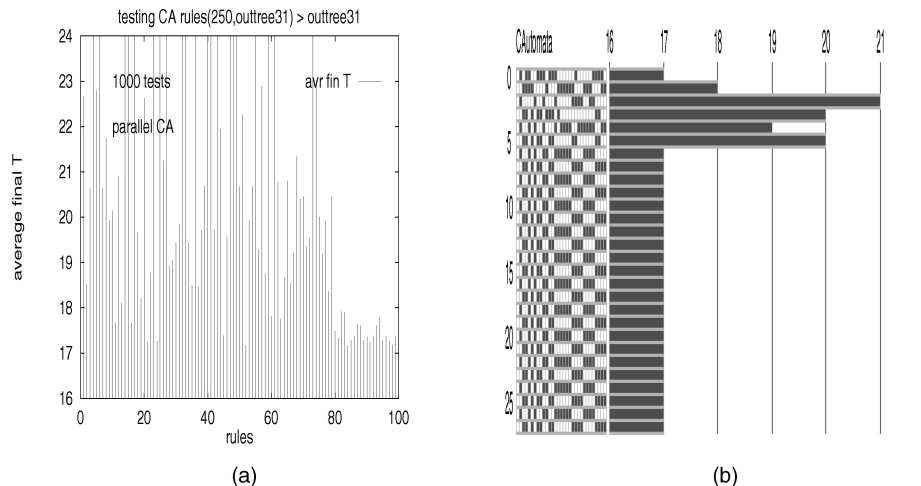


Fig. 18. Parallel CA with coevolution: (a) the average final  $T$  of evolved rules in normal operating mode and (b) example of the behavior of CA.

random oscillation. During hill climbing, a sequence of initial allocations with increasing value of  $T_0$  is generated. These sequences make initial allocations more difficult (see *avr initial T of allocs* in Fig. 17a) and this in turn stimulates GA to improve rules.

Fig. 17b shows the normal operating mode of the scheduler. The figure shows that the frequency of convergence of CA with a given rule to the allocation corresponding to the optimal value of  $T$  is about 80 percent. This value is much higher than in the experiment without coevolution, but smaller than in the sequential CA-based scheduler. However, Fig. 18a shows that the average final  $T$  obtained with the use of these rules is close to the optimal  $T$  and, what is crucial, these near optimal solutions are found in a few time steps of parallel CA (see Fig. 18b) instead of a few hundred steps of sequential CA.

## 7 CONCLUSIONS AND FUTURE WORKS

In this paper, we have presented a novel approach to designing sequential and parallel CA-based scheduling algorithms. We proposed a generic definition of a program graph neighborhood to construct a corresponding CA for any program graph and a coding scheme which maps the essential information about tasks allocation into CA states. We used GA in the learning mode of work of the scheduler to discover scheduling rules of CA. In this mode, knowledge about solving a given instance of the scheduling problem is extracted and coded into CA rules. Rules were discovered which are used in the operation mode by sequential CA for automatic scheduling.

Also, we have shown that coevolutionary algorithms are a very promising technique stimulating the process of discovering effective rules for parallel CA-based algorithms. We compared sequential and parallel CA-based scheduling algorithms and shown advantages of parallel approach. A number of questions in this area are still open. The most important of them is how to use the knowledge extracted during the learning process and coded into CA rules. Some results of study [28], [29] show that rules discovered for different instances of the scheduling problem may be ranked according to their possibility of solving

automatically, in the operation mode, other instances of the scheduling problem. It leads to the concept of an artificial immune system for scheduling problem [30] in which discovered rules are reused to solve new instances of the scheduling problem.

The other related questions are the optimal choice of the CA structure and extending the results on a number of processors greater than two. While we have used a complex nonlinear structure of CA to build a scheduler, one promising direction of research simplifying this structure is using a linear structure. All of these questions are the subject of our current study.

## REFERENCES

- [1] I. Ahmad and Y.-K. Kwok, "On Parallelizing Multiprocessor Scheduling Problem," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 4, pp. 414-432, Apr. 1999.
- [2] J. Blazewicz, K.H. Ecker, G. Schmidt, and J. Weglarz, *Scheduling in Computer and Manufacturing Systems*. Springer, 1994.
- [3] D. Andre, F.H. Bennet III, and J.R. Koza, "Discovery by Genetic Programming of a Cellular Automata Rule that Is Better than Any Known Rule for the Majority Classification Problem," *Proc. First Ann. Conf. Genetic Programming* J.R. Koza, D.E. Goldberg, D.B. Fogel, and R.L. Riolo, eds., pp. 3-11, 1996.
- [4] *Proc. BioSp3: Workshop Bio-Inspired Solutions to Parallel Processing Problems, in conjunction with Int'l Parallel and Distributed Processing Symp. (IPDPS 2002)*, Apr. 2002.
- [5] A. Boukerche and M.S.M. Notare, "Applications of Neural Networks to Mobile Communication Systems" *Solutions to Parallel and Distributed Computing Problems. Lessons from Biological Sciences*, A.Y. Zomaya, F. Ercal, and S. Olariu, eds., pp. 255-268, Wiley & Sons, 2001.
- [6] R. Das, M. Mitchell, and J. Crutchfield, "A Genetic Algorithm Discovers Particle-Based Computation in Cellular Automata," *Parallel Problem Solving from Nature-PPSN III*, pp. 344-353, Y. Davidor, H.-P. Schwefel and R. Männer, eds., Springer, 1994.
- [7] *Artificial Immune Systems and Their Applications*, D. Dasgupta, ed. Springer, 1999.
- [8] CEC 2002: *Congress on Evolutionary Computation*, May 2002.
- [9] H. El-Rewini and T.G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *J. Parallel and Distributed Computing*, vol. 9, pp. 138-153, 1990.
- [10] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York: W.H. Freeman, 1979.
- [11] *Proc. GECCO-2002: Genetic and Evolutionary Computation Conf.*, July 2002.
- [12] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAG's on Multiprocessors," *J. Parallel and Distributed Computing*, vol. 16, no. 4, pp. 276-291, 1992.



- [13] D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, Mass.: Addison-Wesley, 1989.
- [14] T. Gramb, S. Bornholdt, M. Grob, M. Mitchell, and T. Pellizari, *Non-Standard Computation*. Wiley-VCH, 1998.
- [15] W.D. Hillis, "Coevolving Parasites Improve Simulated Evolution as an Optimization Procedure," *Artificial Life II*, C.G. Langton, C. Taylor, J.D. Farmer, and S. Rasmussen eds., Addison-Wesley, 1992.
- [16] J.J. Korczak, P. Lipinski, and P. Roger, "Evolution Strategy in Portfolio Optimization," *Proc. Fifth Int'l Conf. Artificial Evolution*, pp. 225-236, 2001.
- [17] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs onto Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506-521, May 1996.
- [18] Y.-K. Kwok and I. Ahmad, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using a Parallel Genetic Algorithm," *J. Parallel and Distributed Computing*, vol. 47, no. 1, pp. 58-77, 1997.
- [19] *Artificial Life. An Overview*, G. Langton, ed., Bradford Book/MIT Press, 1995.
- [20] J.W. Meyer, "Self-Organizing Processes," *Proc. Int'l Conf. Parallel and Vector Processing (CONPAR94-VAPPVI)*, B. Buchberger and J. Volker, eds., Springer, 1994.
- [21] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1992.
- [22] M. Mitchel, "Computation in Cellular Automata," *Artificial Life II*, pp. 95-140, C.G. Langton, C. Taylor, J.D. Farmer, and S. Rasmussen eds., Addison-Wesley, 1992.
- [23] T.M. Nabhan and A.Y. Zomaya, "A Parallel Simulated Annealing Algorithm with Low Communication Overhead," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 12, pp. 1226-1233, Dec. 1995.
- [24] *Proc. PPSN VII: Parallel Problem Solving from Nature*, Sept. 2002.
- [25] J. Paredis, "Coevolutionary Life-Time Learning," *Proc. Parallel Problem Solving from Nature—PPSN IV*, H.-M. Voigt, W. Ebeling, I. Rechenberg and H.-P. Schwefel, eds., pp. 72-80, 1996.
- [26] F. Seredynski, "Discovery with Genetic Algorithm Scheduling Strategies for Cellular Automata," *Proc. Parallel Problem Solving from Nature—PPSN V*, A.E. Eiben, T. Back, M. Schoenauer, and H.-P. Schwefel, eds. pp. 643-652, 1998.
- [27] F. Seredynski, "New Trends in Parallel and Distributed Evolutionary Computing," *Fundamenta Informaticae*, vol. 35, no. 1-4, pp. 211-230, Aug. 1998.
- [28] F. Seredynski and C.Z. Janikow, "Designing Cellular Automata-based Scheduling Algorithms," *GECCO-99: Proc. Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, eds., pp. 587-594, July 1999.
- [29] F. Seredynski, "Evolving Cellular Automata-Based Algorithms for Multiprocessor Scheduling," *Solutions to Parallel and Distributed Computing Problems. Lessons from Biological Sciences*, A.Y. Zomaya, F. Ercaland, and S. Olariu, eds., pp. 179-207, John Wiley & Sons, 2001.
- [30] F. Seredynski and A. Swiecicka, "Immune-Like System Approach to Multiprocessor Scheduling," *Proc. Fourth Int'l Conf. Parallel Processing and Applied Math.*, Sept. 2001.
- [31] M. Sipper, "Evolution of Parallel Cellular Machines," *The Cellular Programming Approach*, Springer, 1997.
- [32] P.M.A. Slood, J.A. Kaandorp, A.G. Hoekstra, and B.J. Overeinder, "Distributed Cellular Automata: Large-Scale Simulation of Natural Phenomena," *Solutions to Parallel and Distributed Computing Problems. Lessons from Biological Sciences*, A.Y. Zomaya, F. Ercal, and S. Olariu, eds., pp. 1-46, John Wiley & Sons, 2001.
- [33] T. Toffoli and N. Margolus, *Cellular Automata Machines*. MIT Press, 1997.
- [34] M. Tomassini, M. Sipper, and M. Perrenoud, "On the Generation of High-Quality Random Numbers by Two-Dimensional Cellular Automata," *IEEE Trans. Computers*, vol. 49, no. 10, pp. 1140-1151, Oct. 2000.
- [35] Q. Wang and K.H. Cheng, "List Scheduling of Parallel Tasks," *Information Processing Letters*, vol. 37, pp. 291-297, Mar. 1991.
- [36] S. Wolfram, "Universality and Complexity in Cellular Automata," *Physica D.*, vol. 10, pp. 1-35, 1984.
- [37] A.Y. Zomaya, J.A. Andreson, D.B. Fogel, G.J. Milburn, and G. Rozenberg, "Non-Conventional Computing Paradigms in the New Millennium," *Computing in Science and Engineering*, vol. 3, no. 6, pp. 82-99, Nov./Dec. 2001.
- [38] *Solutions to Parallel and Distributed Computing Problems. Lessons from Biological Sciences*. A.Y. Zomaya, F. Ercaland, and S. Olariu, eds., John Wiley & Sons, 2001.
- [39] A.Y. Zomaya and Y.-W. Teh, "Observations on Using Genetic Algorithms for Dynamic Load-Balancing," *IEEE Trans. Parallel and Distributed Systems*, vol. 12, no. 9, pp. 899-911, Sept. 2001.
- [40] A.Y. Zomaya, C. Ward, and B. Macey, "Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 8, pp. 795-812, Aug. 1999.



**Franciszek Seredynski** received the MS and PhD degrees in computer science from the State Electrotechnical University, St. Petersburg, in 1973 and 1978, respectively, and the DSc degree from the Institute of Computer Science, Polish Academy of Sciences in 1998. He is currently a professor of computer science at the Polish-Japanese Institute of Information Technology, Warsaw, and an associate professor at the Institute of Computer Science, Polish Academy of Sciences, Warsaw. He was a visiting researcher at the Institut National Polytechnique de Grenoble, France (1991-1992), International Computer Science Institute, Berkeley, California (1995), Centre for Mathematics and Computer Science, Amsterdam, (1996), and a visiting associate professor at the University of Missouri, St. Louis (1998-1999). His research interests include evolutionary computation techniques, collective behavior of cellular and learning automata, multiagent systems and distributed artificial intelligence, and parallel and distributed processing. He has published more than 100 papers in international journals and conferences. He has served on program committees for a number of international conferences in the areas of evolutionary computing, multiagent systems, and parallel and distributed processing.



**Albert Y. Zomaya** received the PhD degree from the Department of Automatic Control and Systems Engineering, Sheffield University, United Kingdom. He is currently the CISCO Systems Chair Professor of Internetworking in the School of Information Technologies, University of Sydney, Australia. Prior to taking up the current position, he was a full professor in the Department of Electrical and Electronic Engineering, University of Western Australia, where he spent the period spanning 1990-2001. He was also an adjunct professor in the Department of Electrical and Electronic Engineering, University of Western Australia. Dr. Zomaya has to his credit 13 book titles and more than 150 publications in technical journals, collaborative books, and conferences. He is an associate editor for the *International Journal on Parallel and Distributed Systems and Networks*, the *Future Generation Computer Systems Journal*, *Journal of Interconnection Networks*, and *International Journal of Foundations of Computer Science*. He is also the founding editor-in-chief of the Wiley Book Series on Parallel and Distributed Computing. He also served in the past (for two terms) on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Systems, Man, and Cybernetics*. Dr. Zomaya is the editor-in-chief of the *Parallel and Distributed Computing Handbook* (McGraw-Hill, 1996) and serves on the executive board of the International Federation of Automatic Control (IFAC) committee on Algorithms and Architectures for Real-Time Control and the IEEE Task Force on Cluster Computing and he is the chair of the IEEE Technical Committee on Parallel Processing. He was awarded the 1997 Edgeworth David Medal by the Royal Society of New South Wales for outstanding contributions to Australian Science. In September 2000, he was awarded the IEEE Computer Society's Meritorious Service Award. Dr. Zomaya's research interests are in the areas of parallel and distributed computing, computational machine learning, biological and adaptive computing systems, networking, mobile computing and wireless networks, cluster and grid computing, data mining, and scientific computing. He is the founding cochair of the Workshop on Bio-Inspired Solutions to Parallel Processing Problems (BioSP3). He has served in different capacities on the programs of more than a 140 national and international conferences. He is a chartered engineer and a fellow of the Institution of Engineers (UK), a senior member of the IEEE, and a member of the ACM.