Implementation of Deterministic Finite Automata on Parallel Computers*

Jan Holub Stanislav Štekr
Department of Computer Science and Engineering,
Faculty of Electrical Engineering,
Czech Technical University in Prague,
Karlovo náměstí 13, 121 35, Prague 2, Czech Republic,
e-mail: holub@fel.cvut.cz

Abstract

We present implementations of parallel DFA run methods. We compare the parallel methods with well known sequential versions of these methods and find whether and under what conditions is worthy to use the parallel methods of simulation of run of finite automata.

We introduce the paralell DFA run methods for general DFA, which are universal, but due to the dependency of simulation time on the number of states $|Q|$ of automaton being run, they are suitable only for run of automata with the smaller number of states.

On the other hand, if we apply some restrictions to properties of automata being run, we can reach the linear speedup compared to the sequential simulation method. First, we show methods benefiting from k-locality that allows optimum parallel run of exact and approximate pattern matching automata.

Finally, we show the results of experiments conducted on two types of parallel computers (Cluster of workstations and Symmetric shared-memory multiprocessors).

1 Introduction

Finite automata (also known as *finite state machines*) are the formal system for solving many tasks in Computer Science. The finite automata run very fast and there exists many efficient implementations (e.g., [Tho68, NKW05, NKW06]).

The increase of computation power of available computers is based not only on the increase of CPU frequency but also on other modern technologies. The finite automata implementations have to consider these technologies. A recent paper [Hol08] provides a survey of various finite automata implementations considering CPU (like [NKW05, NKW06]). One of the latest technologies used is a usage of multiple CPU core. The speed of sequential run of the finite automata cannot follow the increase of computation power of computers that is mostly based on dual-core and quad-core processors. Therefore a demand on parallel run of the finite automata strenghtens.

The parallel run of deterministic finite transducer was first described in [LF80]. We implement deterministic finite automata run on two parallel computer architectures. Since finite automata are very often used in the approximate and exact pattern matching, we also describe methods of parallel simulation on these automata exploiting their special properties—they are synchronizing automata.

2 Finite Automata

Nondeterministic finite automaton (NFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a set of input symbols, δ is a mapping $Q \times (\Sigma \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. *Deterministic finite automaton* (DFA) is a special case of NFA, where δ is a mapping $Q \times \Sigma \mapsto Q$. We define $\hat{\delta}$ as an extended transition function: $\hat{\delta}(q, \varepsilon) = q$, $\hat{\delta}(q, ua) = p \iff \hat{\delta}(q, u) = q', \delta(q', a) = p, a \in \Sigma, u \in \Sigma^*$.

A *configuration of DFA* is a pair $(q, w) \in Q \times \Sigma^*$. The *initial configuration of DFA* is a pair (q_0, w) and a *final (accepting) configuration of DFA* is a pair (q_f, ε), where $q_f \in F$. A *move of DFA* $M = (Q, \Sigma, \delta, q_0, F)$ is a relation $\vdash_M \subseteq (Q \times (\Sigma^* \setminus \{\varepsilon\})) \times (Q \times \Sigma^*)$ defined as $(q_1, aw) \vdash_M (q_2, w)$, where $\delta(q_1, a) = q_2$, $a \in \Sigma$, $w \in \Sigma^*$, $q_1, q_2 \in Q$. The symbol $\vdash_M^*$ denotes a transitive and reflexive closure of relation $\vdash_M$.

In the previous definition we talk about *completely defined DFA*, where there is for each source state and each input symbol *exactly one* destination state defined. How-

*This research has been partially supported by the Ministry of Education, Youth and Sports under research program MSM 6840770014 and the Czech Science Foundation as project No. 201/06/1039.

ever, there is also *partially defined DFA*, where there is for each source state and each input symbol *at most one* destination state defined. The partially defined DFA can be transformed to completely defined DFA introducing a new state (so called sink state) which has a self loop for each symbol of Σ and into which all non-defined transitions of all states lead.

3 DFA run

3.1 Sequential DFA run

The sequential run of DFA uses tha fact that there is always just one state[1] active during the computation.

Algorithm 3.1 (Basic sequential run of DFA)
Input: A transition table δ and a set F of final states of DFA, input text $T = t_1 t_2 \dots t_n$, q is active state, q_0 is initial state
Output: Information whether DFA accepts whole input text or not
Method:

```
j ← 0
q ← q0
while j ≤ n do
   q ← δ[q, tj]
   j ← j + 1
endwhile

if q ∈ F then
   write('The automaton accepts the text.')
else
   write('The automaton does not accept the text.')
endif
```

Note that the Algorithm 3.1 (run of accepting automaton) can be used only for purposes where we need to know if whole input text is accepted by automaton M or not. Sometimes an information about all reached final states is necessary, so Algorithm 3.2 (run of pattern matching automaton) is a modification solving this situation. This algorithm differs in position of *if* statement, so everytime the automaton reaches a final state, an information about this state and the position of the last read symbol is written out. This kind of finite automaton is widely used in pattern matching [Mel95, Mel96, Hol96, HIMM01].

Theorem 3.1
Let DFA $M = (Q, \Sigma, \delta, q_0, F)$ is run by Algorithm 3.1 or 3.2. Then it runs in time $\mathcal{O}(n)$, where n is the length of the input text. Space complexity is $\mathcal{O}(|Q||\Sigma|)$.

[1] In this text, we suppose run of completely defined automaton. Any partially defined automaton can be converted to the equivalent complete one as mentioned in Section 2.

Algorithm 3.2 (Basic sequential run of DFA—pattern matching version)
Input: A transition table δ and a set F of final states of DFA, input text $T = t_1 t_2 \dots t_n$, q is active state, q_0 is initial state
Output: Information about all reached final states
Method:

```
j ← 0
q ← q0
while j ≤ n do
   q ← δ[q, tj]
   j ← j + 1
   if q ∈ F then
      write('Final state q reached at position j.')
   endif
endwhile

endif
```

Proof
Algorithm 3.1 or 3.2 has one *while* cycle having the number of iterations equal to the length of the input text. Inside this loop, there is only retrieving of corresponding transition δ in the transition table. We expect transition table implemented as a *completely defined transition table* (for completely defined DFA), which means it is implemented as two-dimensional array indexed by states of DFA in first dimension and by characters of input alphabet in the second one. Finding the corresponding state in this transition table has complexity $\mathcal{O}(1)$, so the total complexity of algorithms defined above is $\mathcal{O}(n)$.
Space complexity is given by size of transition table which contains $|Q| \times |\Sigma|$ values, where $|\Sigma|$ is the size of the input alphabet. □

3.2 DFA run on a COW

In this section we describe a method of parallel run of DFA on a cluster of workstations (COW). Let us remind that on COW-based parallel computers *message passing* is used for exchanging data among processors.

3.2.1 DFA run method

When running DFA sequentially on input text T of size n, we start in one initial state and after n steps we reach a state from set Q.

The basic idea of run of DFA on a COW is to divide the input text among all processors, run the automaton on each of them, and then join all subresults.

Let us have an DFA $M = (Q, \Sigma, \delta, q_0, F)$ and cluster of workstations with $|P|$ processors. The input text is sliced into $|P|$ parts $T_1, T_2, \ldots T_{|P|}$ using block data decomposition. On every part T_i of input text, DFA is run and after reading all symbols some state is reached.

A problem comes with joining of subresults. We always need to connect the last active state of processor P_i to the first active state of processor P_{i+1} (so that the initial state of processor P_{i+1} is the last active state of P_i). The problem is that the last active state of processor P_i is known after reading whole part T_i of input text. If each P_{i+1} would need to wait for the last active state P_i and then process part T_{i+1}, we reach sequential complexity (or even worse because sending results between processors is very expensive operation).

The way, how to solve this problem, is to consider all states (one after other) as initial states and for each of them to find corresponding last active state. After doing this, we have got mappings of one *initial state* to one *last active* state. This mapping can be simply reduced by parallel binary reduction [LF80].

Algorithm 3.3 shows us a way, how to run DFA on a COW using basic DFA run. We suppose, that:

- each processor has built the transition table δ,
- each processor has the set of final states F,
- processors are ranked by *continuous linear sequence of IDs starting with zero* and each of them knows its own as a value of variable[2] P_i,
- processor P_0 knows which state is the initial state (q_0),
- each processor has access to its part of the input text (see below),
- at least two processors execute this algorithm,
- the number $|P|$ of processors executing the algorithm does not change during algorithm execution and all processors know the value.

We implement a mapping of *possible initial states* to *possible last active states* as vector $\mathcal{L}$ (of size $|Q|$):

$$\mathcal{L}_{P_i} = \begin{bmatrix} l_0 \\ l_1 \\ \vdots \\ l_{|Q|-1} \end{bmatrix}, \tag{1}$$

[2]This variable is often named 'my_rank' in MPI programs.

where l_j, $0 \leq j < |Q|$, is the last active state assuming that processor P_i starts in state j and processes part T_i of the input text (i.e., $\hat{\delta}(q_j, T_i) = l_j$).

The set F of final states is implemented as bit vector $\mathcal{F}$:

$$\mathcal{F} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{|Q|-1} \end{bmatrix}, \tag{2}$$

where bit $f_j = 1$, if $q_j \in F$, or $f_j = 0$, otherwise.

We also need to implement a vector $\mathcal{R}$ in which we store information about the automaton run. It depends on our requirements what kind of information we want to store. We can store a complete sequence of configurations $(q_0, w) \vdash_M^* (q_f, \varepsilon)$, but for our purposes (without loss of generality) we store only a count of final states reached. Each element r_i of this vector contains a number of reached final states assuming initial state q_i:

$$\mathcal{R} = \begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_{|Q|-1} \end{bmatrix} \tag{3}$$

Finally, transition function δ is implemented as a transition table $\mathcal{T}$ of size ($|Q| \times |\Sigma|$), where $a \in \Sigma$ and $q_j \in Q$ such that $q_j = \delta(q_i, a)$:

$$\mathcal{T}[i, a] = q_j, q_j \in Q \tag{4}$$

3.2.2 Distributing and finishing partial results

After running DFA in parallel, each processor P_i has built mapping $\mathcal{L}_{P_i}$ (*possible initial state* to *possible last active state*) in local memory. In order to finish parallel DFA run, we need to join these mappings (reduce results from processors).

There are two possible methods, how to reduce data from processors. The first method is based on trivial reduction.

This reduction is based on fact, that only the processor P_0 knows the initial state of automaton M. Hence, only the processor P_0 can send the last active state l_0 and the number of reached final states r_0 to the next processor P_1. This processor uses incoming value l_0 to determine which of possible last active states is correct and sends it to the next processor as an active start state. Incoming value r_0 is added to a corresponding value and also sent to the next processor.

If we want to use the binary reduction, we do not start from the first processor, because more reductions are made

Algorithm 3.3 (Basic run of DFA on a COW)
Input: A transition table $\mathcal{T}$, set of final states $\mathcal{F}$, mapping from possible initial state to possible last visited state $\mathcal{L}$ and a set $\mathcal{R}$ of possibly reached final states of DFA, input text $T = t_1 t_2 \dots t_n$ and initial state q_0
Output: Output of run of DFA
Method: Set S of active states is used, each processor has its unique number P_i, number of processors is $|P|$.

for all $P_0, P_1 \dots P_{|P|-1}$ **do in parallel**
  $j \leftarrow \lfloor P_i \frac{n}{|P|} \rfloor$
  $end\_position \leftarrow \lfloor (P_i + 1)\frac{n}{|P|} \rfloor - 1$

  for $k \leftarrow 0, 1 \dots |Q| - 1$ **do**
    $\mathcal{L}[k] \leftarrow k$ /* intitialize vector $\mathcal{L}$ */
    $\mathcal{R}[k] \leftarrow 0$ /* intitialize vector $\mathcal{R}$ */
  endfor

  while $j \leq end\_position$ **do**
    for $i \leftarrow 0 \dots |Q| - 1$ **do**
      $\mathcal{L}[i] \leftarrow \mathcal{T}[\mathcal{L}[i], t_j]$ /* evaluate transition */
      if $\mathcal{L}[i] \in \mathcal{F}$ **then**
        $\mathcal{R}[i] \leftarrow \mathcal{R}[i] + 1$
      endif
    endfor
    $j \leftarrow j + 1$

  endwhile

endfor

MPI_Barrier() /* wait for the slowest processor */

$result \leftarrow$ **perform_parallel_reduction**()
/* see Binary 3.6 or Trivial 3.4 reductions */

Algorithm 3.4 (Parallel trivial reduction for Algorithm 3.3)
Input: All variables and results of Algorithm 3.3 and temporary variables $\mathcal{L}_{temp}$, $\mathcal{R}_{temp}$
Output: Reduced results stored in memory of P_0
Method: All processors perform this reduction, communication is performed sequentially in order to reduce the size of messages being sent.

for all P_0 **do in parallel**
  MPI_Send(*to* P_1 *data* $\mathcal{R}[q_0]$)
  MPI_Send(*to* P_1 *data* $\mathcal{L}[q_0]$)
endfor

for all $P_1, P_2, \dots P_{|P|-2}$ **do in parallel**
  $\mathcal{R}_{temp} \leftarrow$ **MPI_Recv**(*from* P_{P_i-1})
  $\mathcal{L}_{temp} \leftarrow$ **MPI_Recv**(*from* P_{P_i-1})
  $\mathcal{R}[\mathcal{L}_{temp}] \leftarrow \mathcal{R}[\mathcal{L}_{temp}] + \mathcal{R}_{temp}$
  MPI_Send(*to* P_{P_i+1} *data* $\mathcal{R}[\mathcal{L}_{temp}]$)
  MPI_Send(*to* P_{P_i+1} *data* $\mathcal{L}[\mathcal{L}_{temp}]$)
endfor

for all $P_{|P|-1}$ **do in parallel**
  $\mathcal{R}_{temp} \leftarrow$ **MPI_Recv**(*from* $P_{|P|-2}$)
  $\mathcal{L}_{temp} \leftarrow$ **MPI_Recv**(*from* $P_{|P|-2}$)
  MPI_Send(*to* P_{P_0} *data* $\mathcal{R}[\mathcal{L}_{temp}]$)
  MPI_Send(*to* P_{P_0} *data* $\mathcal{L}[\mathcal{L}_{temp}]$)
endfor

for all P_0 **do in parallel**
  $\mathcal{R}_{temp} \leftarrow$ **MPI_Recv**(*from* $P_{|P|-1}$)
  $\mathcal{L}_{temp} \leftarrow$ **MPI_Recv**(*from* $P_{|P|-1}$)
  $\mathbf{return}(\mathcal{R}_{temp}, \mathcal{L}_{temp})$
endfor

in one parallel step. This makes reduction more complicated, because not only one value needs to be sent between processors, but complete vectors $\mathcal{L}$ and $\mathcal{R}$ must be reduced. We define binary operator $\oplus_{DFA}$, which makes one mapping $\mathcal{L}_{P_iP_j}$ from mappings $\mathcal{L}_{P_i}$ and $\mathcal{L}_{P_j}$, where P_i and P_j are processors performing actual step of binary reduction. This newly created mapping $\mathcal{L}_{P_iP_j}$ is built this way:

$$\mathcal{L}_{P_iP_j} = \begin{bmatrix} \mathcal{L}_{P_j}[\mathcal{L}_{P_i}[0]] \\ \mathcal{L}_{P_j}[\mathcal{L}_{P_i}[1]] \\ \vdots \\ \mathcal{L}_{P_j}[\mathcal{L}_{P_i}[|Q|-1]] \end{bmatrix}. \quad (5)$$

This vector $\mathcal{L}_{P_iP_j}$ is either complete result of binary reduction or will be used in next reduction operation. At the end of the binary reduction we have got mapping $\mathcal{L}_{P_iP_j}$, where $i = 0$ and $j = |P| - 1$, hence value of $\mathcal{L}_{P_iP_j}[q_0]$ is the last active state of run of automaton M.

Vector $\mathcal{R}$ is reduced similarly:

$$\mathcal{R}_{P_iP_j} = \begin{bmatrix} \mathcal{R}_{P_i}[0] + \mathcal{R}_{P_j}[\mathcal{L}_{P_i}[0]] \\ \mathcal{R}_{P_i}[1] + \mathcal{R}_{P_j}[\mathcal{L}_{P_i}[1]] \\ \vdots \\ \mathcal{R}_{P_i}[|Q|-1] + \mathcal{R}_{P_j}[\mathcal{L}_{P_i}[|Q|-1] \end{bmatrix}. \quad (6)$$

We can see, that we need to have also vector $\mathcal{L}$ to reduce vector $\mathcal{R}$. This is the reason why it is necessary to reduce both vectors within one reduction operation.

Algorithm 3.5 shows possible implementation of this operator as a function.

Theorem 3.2
The DFA run method shown in Algorithm 3.3 using trivial

Algorithm 3.5 (Binary reduction operator for Algorithm 3.6)
Input: vectors $\mathcal{L}_{P_i}$ and $\mathcal{R}_{P_i}$ from processor P_i and vectors $\mathcal{L}_{P_j}$ and $\mathcal{R}_{P_j}$ from processor P_j
Output: Vectors $\mathcal{L}_{P_0 P_{|P|-1}}$ and $\mathcal{R}_{P_0 P_{|P|-1}}$ stored in memory of P_i
Method:

```
function Reduction_operator_⊕DFA((L_i,R_i),(L_j,R_j))
{
   for x ← 0 ... |Q| − 1 do
      L_{P_i P_j} ← L_{P_j}[L_{P_i}[x]]
      R_{P_i P_j} ← R_{P_i}[x] + R_{P_j}[L_{P_i}[x]]
   endfor
   return(L_{P_i P_j},R_{P_i P_j})
}
```

Algorithm 3.6 (Parallel binary reduction for Algorithm 3.3)
Input: All variables and results of Algorithm 3.3
Output: Reduced results stored in memory of P_0
Method: All processors perform this reduction, communication si performed in pairs in order to reduce number of parallel steps. After performing this reduction, complete result of DFA run is stored in $\mathcal{L}_{P_0 P_{|P|-1}}[q_0]$ and number of reached final states in $\mathcal{R}_{P_0 P_{|P|-1}}[q_0]$

```
for all P_0, P_1 ... P_{|P|-1} do in parallel
   MPI_Reduce(data R_{P_Pi},L_{P_Pi}
   using Reduction_operator_⊕DFA store results on P_0)
endfor

for all P_0 do in parallel
   return(R_{P_0 P_{|P|-1}}[q_0], L_{P_0 P_{|P|-1}}[q_0])
endfor
```

reduction 3.4 performs the run of DFA in parallel.

Proof
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA, $T = t_0 t_1 \dots t_n$ be an input text and $P = P_0, P_1, \dots, P_{|P|-1}$ be a set of processors running this DFA run. At the beginning of the algorithm, the input text is divided among processors such as if all parts of it will be concatenated in order of increasing processor numbers, the original input text will be reconstructed.

Let processor P_i has its part of input text $T_i = t_{\lfloor i\frac{n}{|P|}\rfloor} t_{\lfloor i\frac{n}{|P|}\rfloor+1} \dots t_{\lfloor (i+1)\frac{n}{|P|}\rfloor-1}$. In the sequential run the configuration of automaton M after reading the previous symbol $t_{\lfloor i\frac{n}{|P|}\rfloor-1}$ will be $(q_j, t_{\lfloor i\frac{n}{|P|}\rfloor} t_{\lfloor i\frac{n}{|P|}\rfloor+1} \dots t_n)$, where q_j is some state of set Q. Since processor P_i runs the DFA starting in all possible states $q_j \in Q$ and computes configurations according to these states, no configuration can be missed.

At the end of the run the reduction of results is made in a way that the correct initial state is chosen and the result corresponding to this initial state is sent to the next processor. The last processor performing the reduction has the result of the whole DFA run.

□

Theorem 3.3
The run of general DFA shown in Algorithm 3.3 using Parallel trivial reduction shown in Algorithm 3.4 runs in time $\mathcal{O}(\frac{|Q|n}{|P|} + \log|P| + |P|)$, where $|Q|$ is the number of states of automaton M, n is the length of input text and $|P|$ is number of processors in cluster of workstations running this algorithm.

Proof
See the Algorithm 3.3. First, we focus on the first part of complexity formula —$\mathcal{O}(\frac{|Q|n}{|P|})$. The input text is divided among processors using block data decomposition, so each processor takes $n/|P|$ symbols. These symbols are read using *while* cycle, therefor number of iterations of this cycle is $n/|P|$. Inside the *while* cycle there is one *for* cycle, providing computing of possible last active states. Number of iterations of this cycle is $|Q|$. This gives us first part of complexity formula.

The second part—$\mathcal{O}(\log|P|)$ is complexity of *Barrier*, which is necessary to make correct reduction.

The third part of formula—$\mathcal{O}(|P|)$ is complexity of trivial reduction shown in Algorithm 3.4. Here, each processor sends partial result to next processor (last processor $P_{|P|-1}$ sends partial result to P_0 in order to finish computation in memory of the first processor), so $|P|$ send operations must be performed. □

Theorem 3.4
The run of general DFA shown in Algorithm 3.3 using Parallel binary reduction shown in Algorithm 3.6 runs in time $\mathcal{O}(\frac{|Q|n}{|P|} + \log|P| + |Q|\log|P|)$, where $|Q|$ is number of states of automaton M, n is the length of input text and $|P|$ is number of processors in cluster of workstations running this algorithm.

Proof
First two parts of complexity formula are same as in proof of Theorem 3.3.

The third part of formula—$\mathcal{O}(|Q|\log|P|)$ is the complexity of Parallel binary reduction 3.6 using Binary reduction operator 3.5. This reduction consists of $\mathcal{O}(\log|P|)$ calls of reduction operator. This operator has one *for* cycle having $|Q|$ iterations, hence complete complexity of binary reduction is $\mathcal{O}(|Q|\log|P|)$. □

Example 3.5
Let M be a DFA for exact string matching for pattern $S = banana$,
$T = abananabananabaabananabananaababanan$ be an input text and $|P| = 4$ is number of processors. Automaton M is shown in Figure 1.

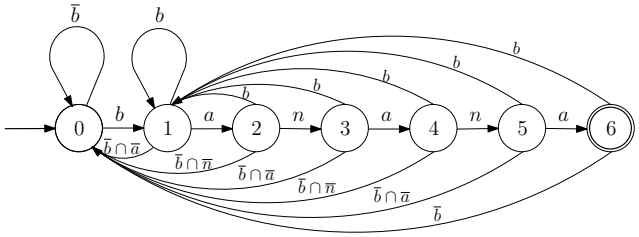

Figure 1. DFA for exact string matching for pattern $S = banana$

δ_M	a	b	n
0	0	1	0
1	2	1	0
2	0	1	3
3	4	1	0
4	0	1	5
5	6	1	0
6	0	1	0

The input text has length 36 symbols, so it can be divided into 4 blocks having 9 symbols each. The following table shows the first phase of Algorithm 3.3 before reduction. Each processor computes possible last active state supposing all states one by one as initial and counts number of visited final states. After computing this table the results are reduced as is shown on Figure 2. The initial state of automaton M is q_0, which has index 0 in the table, so result of the DFA run is:

- last active state $LAS = 5$
- number of reached final states is 4

We can see that results of parallel DFA run are same as results of the sequential DFA run.

	-	a	b	a	n	a	n	a	b	a	*# of reached final states*
P_0	0	0	1	2	3	4	5	6	1	2	1
	1	2	1	2	3	4	5	6	1	2	1
	2	0	1	2	3	4	5	6	1	2	1
	3	4	1	2	3	4	5	6	1	2	1
	4	0	1	2	3	4	5	6	1	2	1
	5	6	1	2	3	4	5	6	1	2	2
	6	0	1	2	3	4	5	6	1	2	1

	-	n	a	n	a	b	a	a	b	a	*# of reached final states*
P_1	0	0	0	0	0	1	2	0	1	1	0
	1	0	0	0	0	1	2	0	1	2	0
	2	3	4	5	6	1	2	0	1	2	1
	3	0	0	0	0	1	2	0	1	2	0
	4	4	5	0	0	1	2	0	1	2	1
	5	0	0	0	0	1	2	0	1	2	0
	6	0	0	0	0	1	2	0	1	2	0

	-	n	a	n	a	b	a	b	a	b	*# of reached final states*
P_2	0	0	0	0	0	1	2	3	4	5	0
	1	0	0	0	0	1	2	3	4	5	0
	2	3	4	5	6	1	2	3	4	5	1
	3	0	0	0	0	1	2	3	4	5	0
	4	5	6	0	0	1	2	3	4	5	1
	5	0	0	0	0	1	2	3	4	5	0
	6	0	0	0	0	1	2	3	4	5	0

	-	a	a	b	a	b	a	n	a	n	*# of reached final states*
P_3	0	0	0	1	2	1	2	3	4	5	0
	1	2	0	1	2	1	2	3	4	5	0
	2	0	0	1	2	1	2	3	4	5	0
	3	4	0	1	2	1	2	3	4	5	0
	4	0	0	1	2	1	2	3	4	5	0
	5	6	0	1	2	1	2	3	4	5	1
	6	0	0	1	2	1	2	3	4	5	0

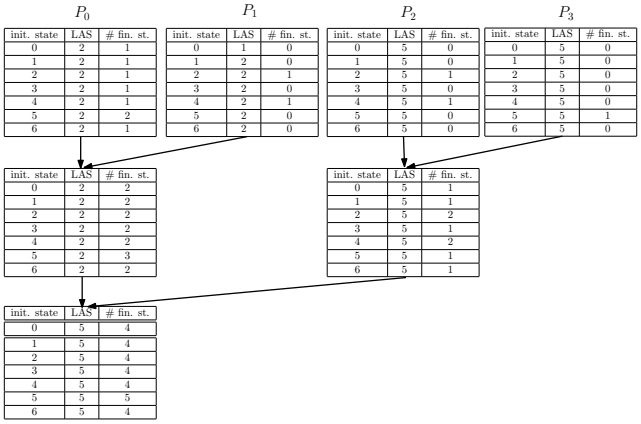

Figure 2. Binary reduction of partial results for Example 3.5

3.2.3 Analysis of DFA run method

Sequential method of run of general DFA has time complexity $\mathrm{SU}(n) = \mathcal{O}(n)$, so it depends only on length of input text. We can see in Theorems 3.3 and 3.4 that parallel DFA run method depends in addition on the number of processors $|P|$ and on the number of states of automaton M being run.

If we suppose that the length of the input text is far greater than the number of processors ($n \ll |P|$), we can ig-

nore the barrier part $\mathcal{O}(\log |P|)$ in the complexity formula. This overhead is common in parallel algorithms and barrier is made only once per run of the algorithm. Contrary to the sequential run of DFA, the parallel run depends also on the number of states $|Q|$. This dependency is present because of precomputing possible terminal states (there is a lot of subresults computed by each processor, but only on of them is used).

If we use the trivial reduction shown in Algorithm 3.3, we need time $\mathrm{T}(n, |P|) = \mathcal{O}(\frac{|Q|n}{|P|} + \log |P| + |P|)$ to run it. If we suppose that number of processors $|P|$ is much smaller than the length of the input text n and the number of states $|Q|$, we can omit the barrier and reduction parts of complexity formula, so we get complexity $\mathrm{T}(n, |P|) = \mathcal{O}(\frac{|Q|n}{|P|})$. Speedup is then

$$\mathrm{S}(n, |P|) = \mathcal{O}\left(\frac{n}{\frac{|Q|n}{|P|}}\right) = \mathcal{O}\left(\frac{|P|}{|Q|}\right) \tag{7}$$

We can see that parallel speedup depends on the number of processors $|P|$ and the number of states $|Q|$. If we increase $|P|$, we speed up the run of the algorithm. It is obvious that if we run a DFA with more states than the number of processors we can use, we do not reach the optimum time of computation. On the other hand the run of DFAs with less states is faster than the sequential algorithm.

If we use the parallel binary reduction shown in Algorithm 3.4, we run DFA in time $\mathcal{O}(\frac{|Q|n}{|P|} + \log |P| + |Q| \log |P|)$. At this formula, we can not simply omit the reduction part of formula $\mathcal{O}(|Q| \log |P|)$ because it depends not only on the number of processors $|P|$, but also on the number of states $|Q|$. As mentioned above, this method of DFA run is not suitable DFAs with more states than the number of processors, so if we accept this, we can get rough approximation of speedup which is the same as in Formula 7.

3.2.4 Method improvements

If we look at method of subresult reduction, we see that the first processor P_0 never uses items of vectors $\mathcal{R}$ and $\mathcal{L}$ not corresponding to *initial state* q_0. It means that when this processor knows its initial state (and it is always), it does not need to compute all of the possible values. This improvement can be easily implemented by modifying *for* cycle, which is nested in input text reading *while* cycle[3].

As described above, this DFA run negatively depends on the number of states of automaton being run. This is the reason why the automaton should be minimized before the run.

[3]This can be done easily by breaking the loop after state for q_0 is computed, or by modifying of iteration condition.

3.2.5 Load balancing

Load balancing of this method can be achieved by nonuniform size of blocks of block data decompostion. We almost assume computation power of individual processors as equal, but it is not always truth and even if it is, we can find out that during computation some processors can be slower. It can happen e.g. because we do not often have parallel computer running only our application and scheduler of operating system does not give all of the computation resources to our application.

We can run some heuristics on each processor, supposed to be used for running our algorithm, and figure out approximate computation power. This power can be used to find out, how large block of input data can be given to each processor. This method is easy and working but, as mentioned above, the computation power given to our algorithm can change during the run of the algorithm. We can then face to a situation when our heuristic tells us to give very large block of input data to some processor, which will get overloaded by some other application somewhere between doing our heuristic and running our algorithm, so we will have to wait until this slowly working processor processes large block of data meanwhile other processors have nothing to work on.

Using an improvement, where we do not precompute vectors $\mathcal{R}$ and $\mathcal{L}$ on first processor P_0, we can use saved computation power by giving larger block of data to this processor. Size of this block should be calculated with respect to saved iterations of *for* cycle, time needed to access next symbol of input data and time needed to compute transition.

3.2.6 Summary of DFA run method on COW

We have introduced the method of run of general DFA on the Cluster of workstations with two possible methods of reduction of partial results. We have designed algorithm, which does not need any communication operation during reading of input text, but the penalty for this is necessity to precompute possible initial states, which has increased complexity $|Q|$ times.

This method is not suitable for run of DFAs with large number of states, but may fit for parallel run of small DFAs with a large input text.

3.3 Run of DFA on a SMP

In this section we describe a method of parallel run of DFA on a Symmetric shared-memory multiprocessors. Contrary to processors of *COW*, *SMP* processors have shared address space, so that each processor can access memory of another one.

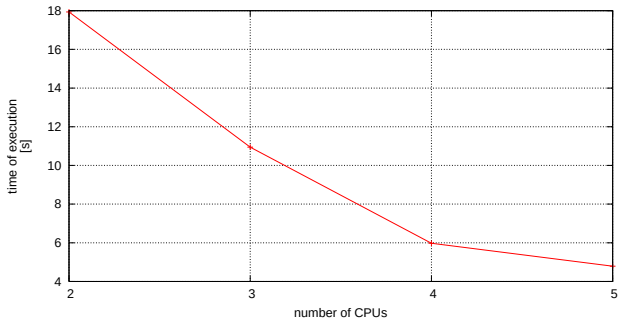

Figure 3. Example of block data decomposition using second method, where 15 data elements is divided among 6 processors

3.3.1 Basic DFA run

The idea of a basic DFA run is the same as in DFA run on a *COW*[4]—we divide the input text among processors, run the automaton on each of them, supposing each state of automaton as initial state, and join partial results into the result of the run. Since we have a share memory at disposal, we do not need to send messages in order to join subresults. At the beginning of the DFA run we can allocate shared memory for all processors, let each processor to work on its part of memory and compute final result of DFA run using this memory at the end of the DFA run.

Remark 3.6
Here, we suppose usage of *OpenMP* library, its pragmas and functions, so all variables, memory allocations, and memory writes in the algorithm, executed before entering a parallel section (we use pseudoalgorithm notation, but in the source code pragma **#pragma omp parallel** is used) are made over the shared memory. It means that in the parallel section these values can be accessed by processors and even if they are at the beginning of parallel section marked as *private*, they will contain original values.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Algorithm 3.7 shows us a way, how to run DFA on a SMP using basic DFA run. We suppose, that:

- each processor has built the transition table δ,
- each processor has the set of final states F,
- processors are ranked by *continuous linear sequence of IDs starting with zero* and each of them knows its own as a value of variable P_i,
- processor P_0 knows which state is the initial state (q_0),
- each processor has access to its part of the input text (see below),
- the number $|P|$ of processors executing the algorithm does not change during algorithm execution and all processors know the value.

As in DFA run on a *COW*, we need to implement vectors $\mathcal{L}$ and $\mathcal{R}$. These vectors have similar purpose. Since in *SMP* the memory is shared, vectors $\mathcal{L}$ and $\mathcal{R}$ for all processors compose matrices (vectors of vectors). Vectors in the matrices are indexed by the processor number. Vector $\mathcal{F}$ and matrix $\mathcal{T}$ are the same (See Formulae 2 and 4 respectively). Since they are shared, they are set up only at beginning of run of algorithm and then the are used by all processors only for reading.

Mapping of *possible initial states* to *possible last active states* is implemented as matrix $\mathcal{L}$ (of size $|Q| \times |P|$):

$$\mathcal{L}[p] = \begin{bmatrix} l_0 \\ l_1 \\ \vdots \\ l_{|Q|-1} \end{bmatrix}, \tag{8}$$

where $p \in P$ is a number of processor l_j, $0 \leq j < |Q|$, is the last active state assuming that processor p starts in state j and processes part T_p of the input text (i.e., $\hat{\delta}(q_j, T_p) = l_j$).

Matrix $\mathcal{R}$ in which we store count of reached final states[5] is implemented as below. The matrix has size $|Q| \times |P|$ and each processor P_i has $\mathcal{R}[P_i]$ part of it. Each item r_i of this vector contains number of reached final states assuming initial state q_i:

$$\mathcal{R}[p] = \begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_{|Q|-1} \end{bmatrix} \tag{9}$$

The reduction of partial results is made either sequentially (see Algorithm 3.8) by one processor, which accesses shared memory and computes the final result, or by all processors using binary reduction (see Algorithm 3.9), where more processors access different memory cells and join them into the final result.

Theorem 3.7
The run of general DFA shown in Algorithm 3.7 using the sequential reduction shown in Algorithm 3.8 runs in time $\mathcal{O}(\frac{|Q|n}{|P|} + \log|P| + |P|)$, where $|Q|$ is the number of states of automaton M, n is the length of the input text and $|P|$ is number of processors running this algorithm.

[4] Described in Section 3.2.

[5] As we have mentioned in Section 3.2, we can store more complex informations than is the count of reached final states.

Algorithm 3.7 (Basic run of DFA on a SMP)
Input: A transition table $\mathcal{T}$, set of final states $\mathcal{F}$, mapping from possible initial state to possible last visited state $\mathcal{L}$ and a set $\mathcal{R}$ of possibly reached final states of DFA, input text $T = t_1 t_2 \dots t_n$ and initial state q_0
Output: Output of run of DFA
Method: Set S of active states is used, each processor has its unique number P_i, number of processors is $|P|$.

```
for all P_0, P_1 ... P_{|P|-1} do in parallel
   j ← ⌊P_i n/|P|⌋
   end_position ← ⌊(P_i + 1) n/|P|⌋ - 1

   for k ← 0 ... |Q| - 1 do
      L[P_i][k] ← k /* intitialize vector L */
      R[P_i][k] ← 0 /* intitialize vector R */
   endfor

   while j ≤ end_position do
      for k ← 0 ... |Q| - 1 do
         L[P_i][k] ← T[L[P_i][k], t_j] /* evaluate transition */
         if L[P_i][k] ∈ F then
            R[P_i][k] ← R[P_i][k] + 1
         endif
      endfor
      j ← j + 1
   endwhile
endfor

#pragma omp barrier /* wait for the slowest processor */

result ← perform_parallel_reduction()
/* see Binary 3.9 or Trivial 3.8 reductions */
```

Proof
See the Algorithm 3.7. All $|P|$ processors in parallel read $n/|P|$ input symbols (this is achieved by *while* loop). After reading each of this symbol, one *for* cycle with $|Q|$ iterations is performed, so complexity of computation of partial results is $\mathcal{O}(\frac{|Q|n}{|P|})$.

When we have computed partial results, we need to perform the barrier synchronization in order to wait for the slowest cpu. The barrier has complexity $\mathcal{O}(\log |P|)$.

The reduction of results using sequential reduction shown in Algorithm 3.8 takes $|P|$ steps, because it consists of one *for* cycle with $|P|$ iterations. □

Theorem 3.8
The run of general DFA shown in Algorithm 3.7 using the parallel binary reduction shown in Algorithm 3.9 runs in time $\mathcal{O}(\frac{|Q|n}{|P|} + \log |P| + |Q| \lceil \log |P| \rceil)$, where $|Q|$ is number of states of automaton M, n is the length of input text and $|P|$ is number of processors running this algorithm.

Algorithm 3.8 (Sequential reduction for Algorithm 3.7)
Input: All variables and results of Algorithm 3.7 stored in shared memory and temporary variables $\mathcal{L}_{temp}$, $\mathcal{R}_{temp}$
Output: Reduced results stored in shared memory
Method: Only one processor performs this reduction, reads data from shared memory and stores result in variables $\mathcal{R}[0]$ and $\mathcal{L}[0]$

```
L_temp ← q_0
R_temp ← 0

for k ← 0, 1, ..., |P| - 1 do
   R_temp ← R_temp + R[k][L_temp]
   L_temp ← L[k][L_temp]
endfor

R[0][q_0] ← R_temp
L[0][q_0] ← L_temp
```

Algorithm 3.9 (Binary reduction for Algorithm 3.7)
Input: All variables and results of Algorithm 3.7 stored in shared memory and temporary variables $\mathcal{L}_{temp}$, $\mathcal{R}_{temp}$
Output: Reduced results stored in shared memory
Method: All processors perform this reduction, read data from shared memory, write partial results and store final result in variables $\mathcal{R}[0]$ and $\mathcal{L}[0]$

```
for all P_0, P_1, ..., P_{|P|-1} do in parallel
   for m ← 1, 2, ..., ⌈log |P|⌉ do
      if (P_i mod 2^m) = 0 and (P_i + 2^{m-1}) < |P| then
         for x ← 0 ... |Q| - 1 do
            R[P_i][x] ← R[P_i][x] + R[P_i + 2^{m-1}][L_{P_i}[x]]
            L[P_i][x] ← L[P_i + 2^{m-1}][L[P_i][x]]

         endfor
      endif

   endfor

endfor
```

join vectors $\mathcal{R}$ and $\mathcal{L}$. This join is executed inside a *for* cycle, which has $|Q|$ iterations. Complexity of the reduction is then $|Q|\lceil \log |P| \rceil$.

3.3.2 Analysis of DFA run method

We can see, that we get the same complexity as in the run of DFA on a *COW*. In comparsion with *COW*-based algorithm, we do not need to explicitly send messages, but we benefit from the shared memory.

We can also speed up run of the algorithm by computing of transition table in parallel and access it in shared memory. Against this can be fact that some computers have the access to shared memory as an expensive operation (e.g. due to cache coherency overhead, limitations of bus and so on...).

4 Experiments

Algorithms for the parallel run of DFAs and DFAs were implemented in **C** programming language using *MPI* environment and *OpenMP* environment. We measured time of execution on *SMP* computer and *COW* computer.

4.1 Used parallel computers

Star Star is a cluster of workstations with 16 nodes (INTEL Pentium III 733 MHz, 256 MB RAM, HD 30 GB) interconnected by Myrinet and Ethernet network. Application for this cluster was written using *MPI*. Because of problems with using standard *MPI* functions handling reading of file, we had to used standard *C* functions *fopen*, *fread*, ...etc. A problem we faced to was that we could not upload files with experimental input data to individual nodes, which lead to the fact that all read input data was shared by one node. This disadvantage decreased performance, because we were limited by bandwith and latency of network and futhermore loss of performance was caused by overloading of node having input data and its harddrive, which could not read data effectively sequentially, but had to jump in a file as was arriving request from individual nodes.

Altix Altix is a symmetric shared-memory multiprocessor with 32 processors (16x 1,3 GHz 3MB L3 cache, 16x 1,5 GHz 6MB L3 cache) interconnected by *NUMAlink* network. Each processor has its own local memory, which is fast and can access to shared memory (but accessing of shared memory is much slower). As in Star, we can reach to a performance bottleneck while accesing one data file by multiple processors. Also e.g. having of transition table in shared memory lead to higher execution time. The measuring applications were written in C using OpenMP.

Figure 4 shows a comparison of execution time of general DFA of sequential and parallel run on Altix, number of states $|Q| = 7$. We can see, that time of parallel algorithm is worse than for sequential. This is caused by collisions on a bus, overloading the harddrive by multiple file accesses and higher time complexity of parallel algorithm (precomputing of possible initial states).

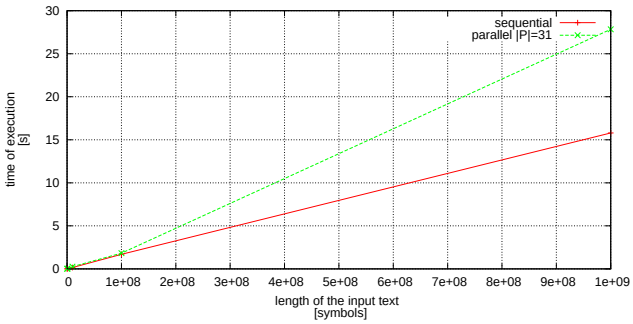

Figure 4. Execution times of general DFA in sequential and parallel run on Altix ($|Q| = 7$)

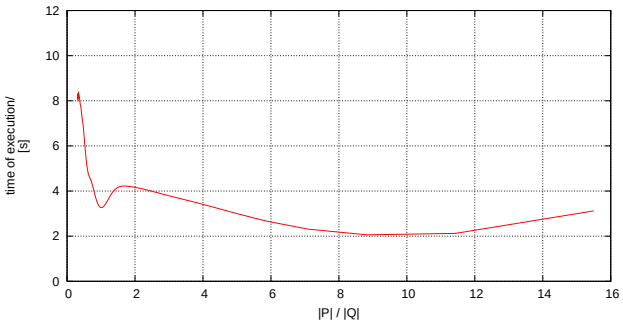

Figure 5. Dependency of execution time of parallel run of DFA on $|P|/|Q|$ (Altix, $|P| = 31$, $n = 10^8$)

Figure 5 shows a dependency of execution time of parallel run of DFA on $|P|/|Q|$ (Altix, $|P| = 31$, $n = 10^8$). This graph shows performance of execution time of parallel run related to number of processors to one state of automaton. We can see, that for $|P|/|Q| < 1$ is performance low, but if we increase the number of processors, we speedup the computation. For $|P|/|Q| > 10$ performance descends due to collisions on bus and higher time needed for reduction of results.

5 Parallel run of pattern searching DFAs

In this section, we show parallel runs of pattern matching finite automata. These non-general automata can be run

in parallel without neccesity to precompute possible initial states, so the complexity of the run does not depend on the number of states of automaton being run. All these runs are based on synchronization of automaton.. We suppose running this run on a *COW*, because it can be simply executed also on *SMP* with only few modifications.

5.1 Synchronization of Finite automata

In the run of general DFA, we had to use precomputing of possible initial states, because we do not have any information about the last active state of automaton which read previous block of input text. We did not know in which state to start the DFA run[6]. If we restrict the DFA run to subset of k-local automata, we do not have to precompute possible initial states, because we can synchronize automatons in each processor and start DFA run from a correct state.

Definition 5.1 (Synchronizing word)
Let us have a DFA $M = (Q, \Sigma, \delta, q_0, F)$. We say that a word $w = a_0 a_1 \dots a_{k-1}$ is synchronizing for M if $\forall p, q \in Q, \hat{\delta}(p, w) = \hat{\delta}(q, w)$.

Definition 5.2 (k-local automaton, synchronizing automaton)
Let us have a DFA $M = (Q, \Sigma, \delta, q_0, F)$. We say that automaton M is k-local if there exists an integer k such that any word of length k is synchronizing. We say, that automaton is synchronizing, if there exists a word $w = \Sigma^*$ of length at least k, which is synchronizing. The number k can be called the synchronization delay of automaton M.

5.2 Parallel run of k-local DFA

5.2.1 Method of the parallel run of k-local DFA

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a k-local DFA, $T = t_1 t_2 \dots t_n$ be an input text and $|P|$ be the number of processors running this DFA. As in methods of run of general DFA, we need to divide the input text among processors using block data decomposition, but in this DFA run method, we need to give to each processor in addition last k symbols of preceding block of input text, so that blocks overlaps in k symbols. This overlapping synchronizes automaton into correct initial state before it reads its part of the input text.

We show this method in Algorithm 5.1. We can see, that at the beginning of the algorithm the boundaries of the input text are set using the block data decomposition, then for all processors (except P_0) the left boundary is extended by k symbols to the left. Of course we should not count reached final states during synchronizing the automaton. Therefore we add condition $j \geq \lfloor P_i \frac{n}{|P|} \rfloor$ to the last *if* statement.

[6]Except of the first processor, which has this information—the initial state q_0 of the automaton $M = (Q, \Sigma, \delta, q_0, F)$

Algorithm 5.1 (Basic run of k-local DFA)
Input: A transition table $\mathcal{T}$, set of final states $\mathcal{F}$, input text $T = t_1 t_2 \dots t_n$, initial state q_0 and the length of the synchronizing word in variable k
Output: Number of reached final states
Method: Set S of active states is used, each processor has its unique number P_i, number of processors is $|P|$.

```
for all P_0, P_1 ... P_{|P|-1} do in parallel
   j ← ⌊P_i n/|P|⌋
    found ← 0 /* number of reached final states */
   if P_i > 0 then
      /* The first proc. does not need to synchronize */
      j ← j − k
      /* Shift the left boundary. */
   endif
   end_position ← ⌊(P_i + 1) n/|P|⌋ − 1
   q ← q_0

   while j ≤ end_position do
      q ← T[q, t_j] /* evaluate transition */
      if L[i] ∈ F and j ≥ ⌊P_i n/|P|⌋ then
         found ← found +1
      endif j ← j + 1

   endwhile
   MPI_Reduce(data found using operator +
   store results on P_0)
endfor

for all P_0 do in parallel
   return( found )
endfor
```

Theorem 5.3 (Černý's conjecture)
If an n-state automaton is synchronizing, there exists a synchronizing word w of length $|w| \leq (n-1)^2$.

Theorem 5.4
The run of k-local DFA shown in Algorithm 5.1 runs in time $\mathcal{O}(k + \frac{n}{|P|} + \log |P|)$, where $|Q|$ is the number of states of automaton M, n is the length of the input text and $|P|$ is the number of processors running this algorithm.

Proof
See the Algorithm 5.1. All processors execute this algorithm in parallel. It contains one *for* cycle, which has in worst case $k + \frac{n}{|P|}$ iterations. k is the number of steps needed to synchronize automaton (i.e. it is the maximal length of the synchronizing word). At the end of the algorithm, there is one binary reduction, which $\log |P|$ times

uses binary operator $+$. Complexity is then $\mathcal{O}(k + \frac{n}{|P|} + \log|P|)$.

□

Analysis of DFA run method We can see in Theorem 5.4, that complexity of run of k-local DFA depends on the length of synchronizing word k. Contrary to the run of general DFA, here is not complexity multiplied by $|Q|$. It means, that if we omit the time needed to reduce results and expect the k much smaller than the length of the input text (which is usual assumption in pattern matching automata), we get the speedup:

$$\begin{aligned} \mathrm{S}(n,|P|) &= \mathcal{O}\left(\frac{n}{k+\frac{n}{|P|}+\log|P|}\right) \doteq \mathcal{O}\left(\frac{n}{\frac{n}{|P|}}\right) \\ &= \mathcal{O}(|P|) \end{aligned} \quad (10)$$

We can see, that we get the linear speedup for DFAs with $n \gg |Q|$, which is the upper bound of speedup achiveable by parallelization of sequential algorithm.

Algorithm 5.2 (Construction of DFA for the exact string matching)
Input: Pattern $P = p_1 p_2 \dots p_m$.
Output: DFA M accepting language $L(M) = \{wP \mid w \in \Sigma^*\}$.
Method: DFA $M = (\{q_0, q_1, \dots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, where the mapping δ is constructed in the following way:

```
for each a ∈ Σ do
    δ(q_0, a) ← {q_0} /* self-loop of the initial state */
endfor
for i ← 1, 2, ..., m do
    r ← δ(q_{i-1}, p_i)
    δ(q_{i-1}, p_i) ← q_i /* forward transition */
    for each a ∈ Σ do
        δ(q_i, a) ← δ(r, a)
    endfor
endfor
```

6 Conclusion and future work

We have presented implementations of DFA run on two different parallel computer architectures. We first implement parallel run of general DFA. We did some experiments that show it is not so efficient in practice due to bus collisions. Then we select a class of DFA called synchronizing automata. For this class of DFA we design algorithm for parallel run that is simpler and it is expected to be practically efficient. Our next research will focus on parallel simulation of nondeterministic finite automata.

References

[HIMM01] J. Holub, C. S. Iliopoulos, B. Melichar, and L. Mouchard. Distributed pattern matching using finite automata. *J. Autom. Lang. Comb.*, 6(2):191–204, 2001.

[Hol96] J. Holub. Reduced nondeterministic finite automata for approximate string matching. In J. Holub, editor, *Proceedings of the Prague Stringologic Club Workshop '96*, pages 19–27, Czech Technical University in Prague, Czech Republic, 1996. Collaborative Report DC–96–10.

[Hol08] J. Holub. Finite automata implementations considering CPU cache. *Acta Polytechnica*, 47(6):51–55, 2008.

[LF80] R. E. Ladner and M. J. Fisher. Parallel prefix computation. *J. Assoc. Comput. Mach.*, 27(4):831–838, 1980.

[Mel95] B. Melichar. Approximate string matching by finite automata. In V. Hlaváč and R. Šára, editors, *Computer Analysis of Images and Patterns*, number 970 in Lecture Notes in Computer Science, pages 342–349. Springer-Verlag, Berlin, 1995.

[Mel96] B. Melichar. String matching with k differences by finite automata. In *Proceedings of the 13th International Conference on Pattern Recognition*, volume II., pages 256–260, Vienna, Austria, 1996. IEEE Computer Society Press.

[NKW05] E. K. Ngassam, D. G. Kourie, and B. W. Watson. Reordering finite automatata states for fast string recognition. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Conference '05*, pages 69–80, Czech Technical University in Prague, Czech Republic, 2005.

[NKW06] E. K. Ngassam, D. G. Kourie, and B. W. Watson. On implementation and performance of table-driven DFA-based string processors. In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference '06*, pages 108–122, Czech Technical University in Prague, Czech Republic, 2006.

[Tho68] K. Thompson. Regular expression search algorithm. *Commun. ACM*, 11:419–422, 1968.