RAIRO-Inf. Theor. Appl. **39** (2005) 403-419
DOI: 10.1051/ita:2005025

EFFICIENT SIMULATION OF SYNCHRONOUS SYSTEMS BY MULTI-SPEED SYSTEMS *

Tomasz Jurdziński[1, 2], Mirosław Kutyłowski[3] and Jan Zatopiański[3]

Abstract. We consider systems consisting of finite automata communicating by exchanging messages and working on the same read-only data. We investigate the situation in which the automata work with constant but different speeds. We assume furthermore that the automata are not aware of the speeds and they cannot measure them directly. Nevertheless, the automata have to compute a correct output. We call this model multi-speed systems of finite automata. Complexity measure that we consider here is the number of messages sent by the automata. The main result of this paper is that multi-speed systems are as powerful as synchronous systems, in which all automata work with the same speed.

Mathematics Subject Classification. 03D15, 68Q45.

1. Introduction

It is a challenging question to understand capabilities of asynchronous computing. It turned out to be one of the hardest problems in distributed computing.

Keywords and phrases. Multi-head automata, systems of finite automata, communication complexity, message complexity.

* *Partially supported by KBN, grant 8T11C 03221 and grant 3 T11C 011 26 in year 2004. This work has been done while the first author was at Institute of Computer Science of Chemnitz University of Technology, and the third author at Institute of Computer Science, Wrocław University.*

[1] Institute of Computer Science, Wrocław University, Przesmyckiego 20, PL-51-151 Wrocław, Poland.

[2] Institute of Mathematics and Computer Science, Kassel University, Heinrich-Plett-Str. 40, D-34132 Kassel, Germany.

[3] Institute of Mathematics, Wrocław University of Technology, Wybrzeże Wyspiańskiego 27, PL-50-370 Wrocław, Poland.

In this paper we study this problem for low level processing units – finite state machines. Once they operate on the same read-only input, it is still unclear what is the influence of asynchrony on the computing power.

In this paper, we investigate one case, namely we assume that the speeds of processing units need not to be equal, but remain constant during the computation. The question that we address here is whether such a model is weaker than in the case, when all processing units work with the same speed. Quite unexpectedly, there is a communication efficient simulation of systems with all units working with the same speed on systems where the speeds may differ by a bounded factor. On the other hand, recall that fully asynchronous systems of finite automata recognize only regular languages as long as $o(n)$ messages are sent [6].

2. Computational model

Below we describe a system S consisting of finite automata $A_1, \ldots, A_k$. Let Q_i, for $i \leq k$, denote the (finite) set of states of A_i, with a distinguished state q_i^{start}, which is the initial state of A_i. There is a number of *accepting* and *rejecting* states. An automaton reaching such a state sends a message that halts the whole system S in an accepting or rejecting state, respectively. We assume that the system works in a way that prevents sending a rejecting and an accepting message during the same computation.

Since the number of automata is constant, we assume that the messages are delivered to all automata – this does not change the number of messages by more than a constant multiplicative factor. Each automaton has a number of buffers of a constant size, each buffer corresponding to messages coming from a different automaton. If the incoming messages fill the whole buffer and the next message arrives, we may assume that either the message that does not fit into the buffer is lost or that the buffer works as a shift register with the oldest messages removed. For each of these settings, we get the same results.

The system S uses a read-only input tape. The input tape contains an input word w embraced by special symbols $\$_L$ and $\$_R$ called, respectively, the *left* and the *right* end-marker. The symbols from the tape may be read by the automata. Each automaton has exactly one reading head and determines its moves on the tape. The role of the end-markers is to prevent the heads from leaving $\$_L w \$_R$. Each automaton starts the computation on the left end-marker. During a step, an automaton may write messages into buffers through which it communicates with the other automata, and reads from its buffers storing messages from the other automata. A transition function of an automaton depends on the current internal state, the symbol seen on the tape by the head of the automaton, and the oldest message in each buffer. Within a step, the automaton enters a new internal state, moves the head by at most one position on the input tape, removes the oldest message from each of the buffers (through which it receives messages from the

other automata), and, depending on its new state, may send a message. More formally, a single step of automaton A_i is described by a transition function δ_i. If Σ is a finite input alphabet, Δ is a finite alphabet of messages, $\perp$ is a special symbol meaning "no message" or "no move", then

$$\delta_i : \; Q_i \times (\Delta \cup \{\perp\})^{k-1} \times \Sigma \;\longrightarrow\; Q_i \times (\Delta \cup \{\perp\})^{k-1} \times \{L, R, \perp\}\,.$$

Let the buffers of A_i be denoted by $B_{i,1}, \ldots, B_{i,i-1}, B_{i,i+1} \ldots B_{i,k}$, where $B_{i,j}$ is devoted for messages coming from A_j to A_i.

A step of A_i is executed as follows. Assume that A_i reads a symbol u from the input tape and the (oldest) messages from the buffers $B_{i,1}, \ldots, B_{i,i-1}, B_{i,i+1} \ldots B_{i,k}$ are respectively $\mu_1, \ldots, \mu_{i-1}, \mu_{i+1}, \ldots, \mu_k$ (if there is no message in a buffer $B_{i,j}$, then we put $\mu_j = \perp$). If q is the current internal state of A_i and

$$\delta_i(q, \mu_1, \ldots, \mu_k, u) = (q', \psi_1, \ldots, \psi_k, r),$$

then:

- A_i changes its internal state to q';
- the messages $\mu_1, \ldots, \mu_k$ read from the buffers are removed;
- for $j \neq i$, if $\psi_j \neq \perp$, then A_i sends ψ_j to the buffer $B_{j,i}$;
- the head of A_i moves on the input tape one position to the left (if $r = L$), or to the right (if $r = R$), or stands idle (if $r = \perp$).

Multi-speed and asynchronous systems. The automaton A_i needs time s_i for executing each step. The value $v_i = \frac{1}{s_i}$ is called the *speed* of A_i. In particular, if $s_i = 1$ for each i, then we call the system **synchronous**. If s_i can be changed by an adversary at each step to an arbitrary value, we call the system *asynchronous*. For a **multi-speed** system, the numbers $s_1, \ldots, s_k$ might be arbitrary, but they do not change during a computation and are not known in advance. Additionally, we assume that they differ at most by a constant factor, that is, $\frac{\max(s_1,\ldots,s_k)}{\min(s_1,\ldots,s_k)} \leq z$ where the parameter z is known to all automata.

Each computation must terminate in an accepting or in a rejecting state. Since for the same input many communication patterns are possible, we say that the system accepts language L if and only if for $x \in L$ every computation on x halts in an accepting state, and for $x \notin L$ every computation on x halts in a rejecting state. That is, a computation should yield a correct answer regardless of the current speeds. Without loss of generality we can assume that every computation begins and terminates by sending a message.

Message complexity. We consider the number of messages sent by a system as a complexity measure of a computation. Since there are finitely many configurations on which a message may depend, we may assume that each message has a constant size. In this way, the number of messages corresponds roughly to the total volume of communication.

We consider the worst case complexity, that is, we say that a problem requires $f(n)$ messages, if for infinitely many n, there is at least one input of size n for which $f(n)$ messages are sent, and for each input of size n at most $f(n)$ messages are sent.

3. Synchronous *versus* multi-speed systems

Classical results on communication complexity are focused on the model in which communication is the main measure of efficiency (ignoring time and space complexity). There are relatively few results concerning the situation that other resources are limited. Perhaps, the most prominent direction here was initiated in [1,2], where tradeoffs between communication and space were found.

For systems of finite automata, the first results on communication complexity can be found in [4,7,8]. However, these results concern only synchronous systems. Asynchronous systems were investigated and compared with synchronous systems in [6].

Message complexity for synchronous systems. For synchronous systems even $O(1)$ messages suffice to recognize some non-trivial languages. A simple example is the language $L_{\text{twice}} = \{a^k 1 b^k | k \in \mathbb{N}\}$: two synchronous automata can recognize it with one message. It has been shown [4] that there is a dense hierarchy of languages for message complexity between $\Theta(1)$ and $\Theta(n)$. Even for languages requiring a constant number of messages, each additional message makes the class of languages larger [4]. An interesting phenomenon is that there is no language with message complexity $\omega(1)$ and $o(\log\log\log n)$ [3]. It has been also shown that there is a tight hierarchy of functions which require $\omega(n)$ messages for inputs of size n (see [4]).

Message complexity for asynchronous systems. "Side-channel" information such as the moment of sending a message is very useful for constructing algorithms in synchronous systems – it allows to keep track about relative head positions on the input tape. For a synchronous systems the sender of a message cannot move its head until the new synchronous step, whereas in an asynchronous system the sender may move its head arbitrarily far. So, several algorithmic tricks known for synchronous systems are useless for asynchronous computations.

An asynchronous system can emulate a synchronous one by sending an extra message from each automaton whenever any head moves. No automaton proceeds until it receives such auxiliary messages from all other automata. This technique, called *step-by-step synchronization*, requires $\Omega(t)$ messages for simulating t steps of a synchronous system. So, even if a synchronous system is message efficient, the asynchronous system constructed in this way may generate a lot of messages.

Inefficiency of step-by-step synchronization is not a coincidence. It was shown that if an asynchronous system uses $o(n)$ messages, then the language recognized by the system is regular, so no communication is necessary at all [6,9]. Moreover, weakness of asynchronous systems may be shown also for languages requiring $\Omega(n)$ messages. A good example is the language L_{trans} consisting of the words of the

form $U\#U^T$, where U^T denotes transposition of binary matrix U, and the matrices are written row by row. It is easy to show that L_{trans} has message complexity $\Theta(n)$ on synchronous systems. By applying step-by-step synchronization to this algorithm, we see that one can recognize L_{trans} with $O(n^{3/2})$ messages on an asynchronous system. On the other hand, it was shown [6,9] that any asynchronous system requires $\Omega(n^{3/2}/\log n)$ messages to recognize L_{trans}.

Message complexity of multi-speed systems. Fixed velocities of automata in multi-speed systems, even though they can be different in each run of the system, allow to store some information by the distances between the end-markers and the heads of the automata. This technique was used [5] to show that certain non-regular languages can be efficiently recognized by multi-speed systems with a sublinear number of messages, whereas every fully asynchronous system needs at least a linear number of messages. It was also proved [5] that the language $L_{\text{pow}} = \{0^n : \sqrt{n} \in \mathbb{N}\}$ can be recognized by a multi-speed system of finite automata with $O(\sqrt{n})$ messages, just as in the case of synchronous automata.

New results. The main result of this paper is the equivalence of synchronous and multi-speed systems: with a linear overhead in the number of messages any synchronous system can be simulated by a multi-speed system:

Theorem 3.1. *Let L be a language and M be a synchronous system consisting of m automata recognizing L using at most $f(n)$ messages for inputs of size n. Then, there exists a multi-speed system M' consisting of $3m + 2$ automata that recognizes L using at most $O(m \cdot f(n))$ messages regardless of the speeds of the automata as long as they differ at most by a multiplicative factor z, where z is a constant.*

4. General techniques for multi-speed systems

In this section we present a couple of tricks that are used in the simulation of synchronous systems. Some of these techniques were introduced in [5].

One of the main tasks in a simulation of a synchronous system by a multi-speed system is to keep track of the head positions. Our approach is to store information about head positions of a synchronous system by head positions of the automata of the multi-speed system. However, "storing" and "retrieving" must be performed in a special way, so that it works in a multi-speed system (and does not require many messages). The storing procedure, quite obvious for synchronous systems, requires some technicalities in the multi-speed case.

Storing a head position. Let X be an automaton of a multi-speed system which head position is to be stored by the head position of an auxiliary automaton B. The following procedure, called `dump`, achieves this goal. For clearness of exposition we assume that X is faster than B. This can be checked as follows: X sends a special message to B in its two consecutive steps, B responds with a message confirming arrival of the first message from X; if X gets the response before sending the second message then it is slower. Note that if B is too fast, we may slow it down

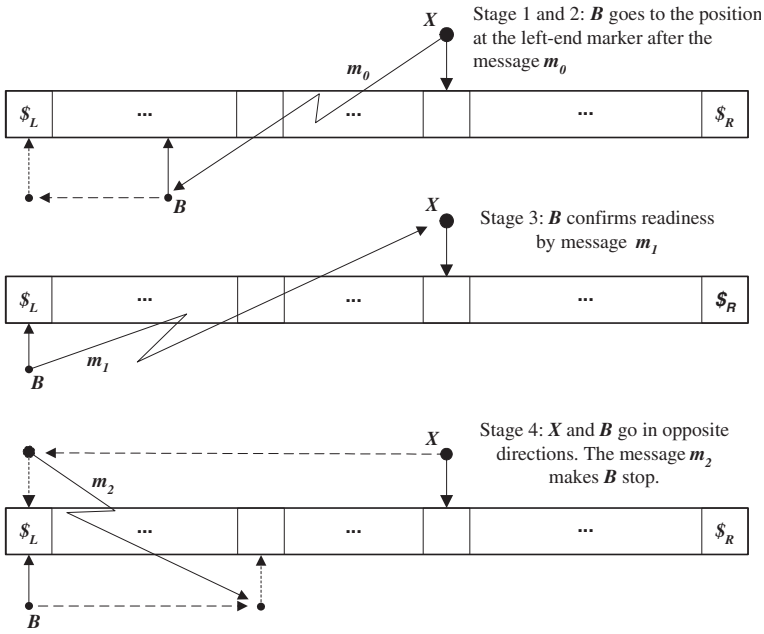

FIGURE 1. Stages of storing a head position.

artificially $k = \lceil z \rceil$ times: one step of the procedure is executed in every k steps of B.

`dump` $X \to B$`:` (see Fig. 1)

1. automaton X sends a message m_0 and waits for a response from automaton B;
2. after receiving m_0 from X, automaton B goes to the left end-marker and confirms its arrival there by sending a message m_1;
3. after receiving the confirmation m_1, X starts moving to the left end-marker; simultaneously B goes to the right; one move is made per step;
4. as soon as X reaches the left end-marker, it sends a message that makes the automaton B stop, the message sent by X encodes the number of symbols encountered by X on the way to the left end-marker modulo some constant c (an appropriate choice of c will be discussed later). B stores this message in its internal state.

For short input words the above protocol may not work properly, but then a single automaton can be used to accept or reject.

Note that after executing the store operation, the distance between the left end-marker and the head position of B need not to be equal to the original distance between the head of X and the left end-marker. However it is almost `proportional`

to this distance. Counting modulo c is necessary, since there are some small inaccuracies due, among other reasons, to the fact that the distances are integers and the speed ratio need not to be an integer.

The procedure `dump` can be generalized so that the position of the head of X is stored by many automata. The procedure below describes storing the head position of X so that each of the automata $A_{i_1}, \ldots, A_{i_m}$ encodes this position for a later retrieval (for clearness of exposition we assume that X is the fastest automaton):

`dump` $X \to A_{i_1}, \ldots, A_{i_m}$:

1. the automaton X sends the message m_0 and waits for the responses from automata $A_{i_1}, \ldots, A_{i_m}$;
2. after receiving m_0 from X, the automata $A_{i_1}, \ldots, A_{i_m}$ go to the left end-marker and confirm the arrival there by sending a message m_1 to X and all of $A_{i_1}, \ldots, A_{i_m}$;
3. after receiving the last message m_1 confirming the arrival, $A_{i_1}, \ldots, A_{i_m}$ start moving to the right, with X going simultaneously to the left end-marker, one move is made per step by each automaton;
4. as soon as X reaches the left end-marker, it sends a message m_2 that makes the automata $A_{i_1}, \ldots, A_{i_m}$ stop, the message m_2, called *correcting information*, encodes the number of symbols encountered on the way to the left end-marker modulo some constant c; the automata $A_{i_1}, \ldots, A_{i_m}$ store this message in their states.

Restoring a head position. To restore the original head position of X we use almost the same protocol, but the roles of automata are reversed. Only one automaton is necessary for retrieving the original position of X, the other automata involved in the `dump` operation can be retained for a later use:

`restore` $X \leftarrow B$:

1. automaton B sends a message m'_0 and waits for a response from automaton X;
2. after receiving m'_0 from B, automaton X goes to the left end-marker and informs about its arrival there by sending a message m'_1;
3. after receiving message m'_1 from X, automaton B starts moving to the left end-marker, simultaneously X goes to the right, one move is made per step by each automaton;
4. as soon as B reaches the left end-marker, it sends a message that makes the automaton X stop, the message sent by B encodes the correcting information obtained by B during the dump operation; X uses this information to adjust its head position to the correct one: less than c moves are made.

Now, let us discuss in detail the role of correcting information. Let the value stored (*i.e.* the original distance between the head of X and the left end-marker) be d_0. If we assume that automata X and B start moving at the same moment, then after performing the dump operation the distance between the left end-marker and the position of the head of B equals $\lceil d_0 \cdot \frac{v_B}{v_X} \rceil$, where v_B and v_X denote the speeds

of B and X. This distance is not the same as d_0 and we shall be able to use it only with connection to X. If we take into account the fact that there might be a small delay $\tau_B < \frac{1}{v_B}$ of starting automaton B, the correct expression for the distance mentioned is $\lceil (\frac{d_0}{v_X} - \tau_B) \cdot v_B \rceil$.

As we have seen, the operation of storing a distance through `dump` operation is not precise ($\tau_B = 0$ only for synchronous systems). However, we cope with this problem by a careful use of the dump and restore operations. The key point is that they have the following important property: the value restored by the automaton X (without the final adjustment) differs from the original value at most by a constant additive factor. It allows the system to store and restore the original value without loosing precision, since a constant can be kept in the internal memory of the automata.

Lemma 4.1. *Let the speeds of automata X and B be equal, respectively, v_X and v_B, with $v_X \leq v_B$. Suppose that the original distance between the head of X and the left end-marker is d_0. Then after the operations* `dump` $X \to B$ *and* `restore` $X \leftarrow B$ *without the final adjustment, the distance between the head of X and the left end-marker is a number d_2 such that $|d_0 - d_2| \leq z + 1$.*

Proof. Let d_1 be the distance between the head of automaton B and the left end-marker after the operation `dump` $X \to B$. We have:

$$d_1 = \left\lceil \left(\frac{d_0}{v_X} - \tau \right) \cdot v_B \right\rceil$$

with $\tau < \frac{1}{v_B}$, and

$$d_2 = \left\lceil \left(\frac{d_1}{v_B} - \tau' \right) \cdot v_X \right\rceil$$

with $\tau' < \frac{1}{v_X}$. So

$$d_2 < \left(\frac{d_1}{v_B} \right) \cdot v_X + 1 < \left(\frac{d_0}{v_X} \cdot v_B + 1 \right) \cdot \frac{v_X}{v_B} + 1 = d_0 + \frac{v_X}{v_B} + 1 \leq d_0 + z + 1.$$

The last inequality follows from the fact that $\frac{v_X}{v_B} \leq z$, because v_X is faster than v_B. Similarly,

$$\begin{aligned} d_2 &\geq \left(\frac{d_1}{v_B} - \tau' \right) \cdot v_X > d_1 \cdot \frac{v_X}{v_B} - 1 \\ &> \left(\frac{d_0}{v_X} \cdot v_B - 1 \right) \cdot \frac{v_X}{v_B} - 1 = d_0 - \frac{v_X}{v_B} - 1 \\ &\geq d_0 - z - 1. \end{aligned}$$

□

Lemma 4.1 suggests the following method of "adjustment" which will guarantee that the positions of the head of X before `dump` $X \to B$ and after `restore` $X \leftarrow B$

are equal. During the execution of the 4th step of `dump` $X \to B$ (after reaching the endmarker), in message m'_1 automaton X encodes a value d, which is its position at the beginning of `dump` $X \to B$ modulo $2(z+1)+1$. Further, B encodes this value d in its state. Then, B sends back d to X at step 4 of `restore` $X \leftarrow B$. Finally, X changes its position at the end of `restore` $X \leftarrow B$, to the nearest position congruent with d modulo $2(z+1)+1$. As we know by Lemma 4.1, the position of X at the end of `restore` $X \leftarrow B$ differs from its original position by at most $z+1$, so we ensure the correct location of X at the end of the procedure restore.

The command `restore` $X \leftarrow B$, introduced above, has a disadvantage that B does not retain the value that is "loaded" into X. It is a problem, if we would like to use the value encoded by position of the head of B many times. The following command solves this problem:

`restore` $X \leftarrow B(B')$:

1. do in parallel:
   { `restore` $X \leftarrow B$ | `dump` $B \to B'$ }
2. `restore` $B \leftarrow B'$.

The first line of the procedure `restore` $X \leftarrow B(B')$ describes the same execution as for `restore` $X \leftarrow B$, but additionally an auxiliary automaton B' keeps track of the value encoded by B. In fact, the execution of `restore` $X \leftarrow B$ and `dump` $B \to B'$ is the same from the point of view of B, so we can do it in parallel. Then, in line 2 automaton B' is used to recover the original position of B.

Comparing values

Now we consider the second important task. Having stored many values in head positions of auxiliary automata, we would like to compute the minimum of them. In a synchronous system this is not a problem: the heads of automata encoding the values mentioned start moving to the left. The first head which reaches the left end-marker corresponds to the minimum value. In the case of a multi-speed system the situation is a little bit more complicated.

Let $A_1, \ldots, A_k$ store head positions of the same automaton A (may be, from different moments) in the sense that operations `dump` $X \to A_1, \ldots,$ `dump` $X \to A_k$ have been executed. Our aim is to "implement" a following simple procedure that determines and stores by X the minimum of numbers $A_1, \ldots, A_k$:

1. $X \leftarrow A_1$, $Y \leftarrow X$ (Y is a current minimum);
2. `for` $i \leftarrow 2$ `to` k;
3. $X \leftarrow X - A_i$;
4. `if` $(X < 0)$ `then` $X \leftarrow Y$;
5. `else` $X \leftarrow A_i$ `and` $Y \leftarrow X$.

To implement the above procedure we have to perform subtracting. For computing $X - A$ we can use a modification of operation `restore` $X \leftarrow A$, such that the automaton X will not start from the left end-marker, but from its original position; and it is going to the left instead of right. Automaton X cannot cross the end-marker in the case that the value subtracted is bigger than the value of A. We deal with negative values by storing an appropriate information in the internal state.

We use the notation $< \texttt{restore}_{\text{mode}}\ X \leftarrow A >$ for modified operation restore with the following options:

mode	behavior
$\perp R$	X makes no move at the beginning of the protocol and then goes to the right (for implementing addition)
$\perp L$	X makes no move at the beginning of the protocol and then goes to the left (for implementing subtraction)

The following procedure $X \leftarrow \texttt{minimum}(A_1, \ldots, A_k)$ finds the minimum of the values stored by $A_1, \ldots, A_k$:

```
1.  restore X ← A_1(A'_1)
2.  dump X → Y', Y''
3.  restore X ← Y''
    /* the current minimum is A_1 */
4.  for i ← 2 to k do
    /* subtracting value from the current minimum*/
5.    restore_⊥L X ← A_i(A'_i)
6.    if (X is positive) /* a new minimum */
7.      { restore X ← A_i(A'_i)
8.        dump X → Y', Y''
9.        restore X ← Y''}
10.   else /* restoring the previous minimum */
11.     { restore X ← Y'(Y'') }
12. endfor
```

Observe that the number of messages required by each of procedures `dump`, `restore` and `minimum` is independent of the size of the input, *i.e.* bounded by a constant (dependent only on the size of the original synchronous system).

4.1. Formal description of the algorithm

We have described basic procedures like `dump` and `restore`. Such procedures can be combined in branches, loops or sequences by using standard techniques for automata systems provided that a constant number of states is used.

Formal descriptions of systems of finite automata (by transition functions) are very hard to read (similarly as programs for Turing machines, for example). That is why we started with informal descriptions. Although it is intuitively clear that one can run them using multi-speed systems of finite automata, we present as an example definitions of the transition for the procedure $\texttt{dump}\ X \rightarrow A_{i_1}, \ldots, A_{i_m}$.

Example 4.2 (Transition function for $\texttt{dump}\ X \rightarrow A_{i_1}, \ldots, A_{i_m}$)**.** For the sake of simplicity of notation we describe $\texttt{dump}\ A_{m+1} \rightarrow A_1, \ldots, A_m$. Let $c = 2z + 3$, where z is a maximal ratio between speeds of the fastest and the slowest automaton

of the system. Further, let

$$\begin{array}{lcll} q_{start} & \in & Q_i & \text{for} \quad i \in \{1,\ldots,m+1\}, \\ q_{wait2,V} & \in & Q_{m+1} & \text{for} \quad V \in \{0,1\}^m \\ q_{st2,V}, q'_{st2,V} & \in & Q_i & \text{for} \quad i \in \{1,\ldots,m\}, V \in \{0,1\}^m \\ q_{st3} & \in & Q_i & \text{for} \quad i \in \{1,\ldots,m\}, \\ q_{st3,l} & \in & Q_{m+1} & \text{for} \quad l \in \{0,\ldots,c-1\} \\ q_{C_l} & \in & Q_i & \text{for} \quad i \in \{1,\ldots,m\}, l \in \{0,\ldots,c-1\}, \\ q_{end}, q_{ready} & \in & Q_{m+1} & \end{array}$$

and $m_0, m_1, \mathit{finished}, C_0, \ldots, C_{c-1} \in \Delta$.

Assume that each of automata $A_1, \ldots, A_{m+1}$ starts to execute `dump` $X \rightarrow A_{i_1}, \ldots, A_{i_m}$ in the state q_{start}. Further, this state does not belong to the set of states of automata A_i for $i > m+1$, these automata do not change their states and positions until the message *finished* is sent by A_{m+1} (*i.e.* they wait).

We assume also that all states of automata used in the description presented below do not occur during any other (than `dump` $X \rightarrow A_{i_1}, \ldots, A_{i_m}$) part of computation of the system. Next, we assume that each of the automata $A_1, \ldots, A_m$ is not faster than A_{m+1} (we have presented a method of ensuring this condition in the discussion before the description of `dump` $X \rightarrow B$).

In order to simplify the description assume that each automaton has k buffers, the ith buffer for the messages sent by A_i (so, each automaton may also send messages to itself, what naturally does not change the computational power of the model). Further, each time the transition function is undefined for states of automata used below, we assume that the automaton does not change the state nor the head position in this step of computation.

A vector $V \in \{0,1\}^m$ stored in states is used to remember which automata of $A_1, \ldots, A_m$ has already reached the left endmarker.

Transitions describing step 1 of the procedure
The automaton $X = A_{m+1}$ sends the message to other automata:

- $\delta_{m+1}(q_{start}, \perp^i, a) = (q_{wait2,V}, (m_0)^m \perp^{k-m}, \perp)$ for each $a \in \Sigma$, $V = 0^m$.

Transitions describing step 2 of the procedure
Transitions of the automaton A_i for $i \in \{1,\ldots,m\}$:

- $\delta_i(q_{start}, M_1, \ldots, M_m, m_0, \perp^{k-m-1}, a) = (q_{st2,V}, \perp^k, L)$ for $a \neq \$_L$, $V_j = 1$ iff $M_j \neq \perp$ for $j = 1, \ldots, m$ (the automaton A_i starts moving left and collecting information which of $A_1, \ldots, A_m$ have already reached $\$_L$);
- $\delta_i(q_{st2,V}, M_1, \ldots, M_m, \perp^{k-m}, a) = (q_{st2,V'}, \perp^k, L)$ for $a \neq \$_L$, $V, V' \in \{0,1\}^m$, $V'_j = V_j$ if $M_j = \perp$ and $V'_j = 1$ if $M_j \neq \perp$, (A_i is moving left and collects information which of $A_1, \ldots, A_m$ have already reached $\$_L$);
- $\delta_i(q_{st2,V}, M_1, \ldots, M_m, \perp^{k-m}, \$_L) = (q'_{st2,V'}, m_1^{m+1}, \perp^{k-m-1}, \perp)$ for $V, V' \in \{0,1\}^m$, $V'_j = V_j$ if $M_j = \perp$ and $V'_j = 1$ if $M_j \neq \perp$, (A_i reaches $\$_L$ and informs about it by sending the message m_1);

- $\delta_i(q'_{st2,V}, M_1, \ldots, M_m, \perp^{k-m}, \$_L) = (q'_{st2,V'}, \perp^k, \perp)$ for $V, V' \in \{0,1\}^m$, $V \neq 1^m$, $V'_j = V_j$ if $M_j = \perp$ and $V'_j = 1$ if $M_j \neq \perp$, (A_i stays at $\$_L$ and waits until all of $A_1, \ldots, A_m$ reach $\$_L$);
- $\delta_i(q'_{st2,V}, \perp^k, \$_L) = (q_{st3}, \perp^k, R)$ for $V = 1^m$ (A_i starts step 3, because all automata $A_1, \ldots, A_m$ have already finished step 2).

Transitions of $X = A_{m+1}$:

- $\delta_{m+1}(q_{wait2,V}, M_1, \ldots, M_m, \perp^{k-m}, a) = (q_{wait2,V'}, \perp^k, \perp)$ for $a \in \Sigma$, $V, V' \in \{0,1\}^m$, $V \neq 1^m$, $V'_j = V_j$ if $M_j = \perp$ and $V'_j = 1$ if $M_j \neq \perp$ (the automaton $X = A_{m+1}$ does not move its head until all automata $A_1, \ldots, A_m$ reach $\$_L$, it collects information which of $A_1, \ldots, A_m$ have already reached $\$_L$);
- $\delta_{m+1}(q_{wait2,V}, \perp^k, a) = (q_{st3,0}, \perp^k, L)$ for $V = 1^m$ (A_{m+1} starts step 3, because each automaton has reached $\$_L$ already).

Transitions describing step 3 of the procedure

- $\delta_i(q_{st3}, \perp^k, a) = (q_{st3}, \perp^k, R)$ for $a \in \Sigma$, $i \in \{1, \ldots, m\}$ (automaton A_i moves right during step 3);
- $\delta_{m+1}(q_{st3,l}, \perp^k, a) = (q_{st3,(l+1) \bmod c}, \perp^k, L)$ for $a \neq \$_L$, $l \in \{0, \ldots, c-1\}$ (A_{m+1} moves left during step 3 and counts its position modulo c).

Transitions describing step 4 of the procedure

- $\delta_{m+1}(q_{st3,l}, \perp^k, \$_L) = (q_{end}, m_2^m, \perp^{k-m}, \perp)$, where $m_2 = C_l$, $l \in \{0, \ldots, c-1\}$, (A_{m+1} has reached the left endmarker, it sends a message m_2 containing l, its original position modulo c to $A_1, \ldots, A_m$;
- $\delta_i(q_{st3}, \perp^m, C_l, \perp^{k-m-1}, a) = (q_{C_l}, \perp^k, \perp)$ for $i \in \{1, \ldots, m\}$, $l \in \{0, \ldots, c-1\}$ (after reaching the message from A_{m+1}, automaton A_i finishes step 3 and stores in a state the correcting information C_l sent by A_{m+1});
- $\delta_{m+1}(q_{end}, \perp^k, \$_L) = (q_{ready}, \mathit{finished}^k, \perp)$ (A_{m+1} sends a unique message $finished$ to all automata, which encodes the fact that `dump` $X \to A_{i_1}, \ldots, A_{i_m}$ is finished).

5. Simulating synchronous systems

In this section we prove Theorem 3.1.

Outline of the simulation. Let M be a synchronous system consisting of automata $A_1, ..., A_m$. Between receiving consecutive messages each automaton A_j behaves like a finite automaton. So we divide the computation of M into *silent stages*, *i.e.* periods during which all automata work and no message is sent, and *communication stages*, *i.e.* single steps during which at least one automaton sends a message. Both stages interlace. Since automata in a silent stage do not interact, a silent stage can be simulated independently for each automaton.

We will simulate the computation of M stage by stage. The main technical problem of the simulation of M is keeping track of the head positions at the beginning and the end of each stage. We solve this problem by storing and restoring

the head positions of each automaton by the head positions of the auxiliary automata. The next problem is that the end of a silent stage cannot be recognized directly during the simulation of a single automaton, since in many cases we would have to know when a message arrives from another automaton. For this reason, during this phase of the simulation (of a stage), called preprocessing, each automaton is simulated up to the moment, when it sends a message or halts waiting for a message. Let d_i be the number of steps to the first message sent by A_i without receiving any message from other automata (if such an event does not occur, we put $d_i = \infty$). Having this done for all automata of M, we check which simulation took the shortest time and terminated with a message being sent, *i.e.* we look for $t = \min\{d_i | i = 1..k\}$. Obviously, t determines duration of the silent stage. So the simulation can be outlined as follows:

for $j = 0, 1, \ldots$ do
- preprocessing of the jth silent stage;
- determine t_j, the duration of the jth silent stage;
- simulate the jth silent stage for t_j steps; further; simulate the step in which a message (or messages) is sent immediately after the silent stage.

There are some technical problems to be solved. It may happen that an automaton enters a loop without receiving a message, and therefore the simulation (of a computation of such automaton in the silent stage) would last forever. Detecting a loop by checking that an automaton enters the same state of the memory with the same head position would be difficult – it is hard to see that the head is at the same position (using only a finite memory). We use here an easy but effective solution: every simulation is assisted by an automaton which counts steps. If the number of steps is $\lambda \cdot z \cdot n$, where λ bounds from above the number of states of A_i, z is a maximal ratio between speeds of the fastest and the slowest automaton, and n is the length of input, then A_i loops for sure. So we let the counting automaton start from the left end-marker, making one move per $\lambda \cdot z$ steps. If it arrives at the right end-marker, then it sends a special message to terminate the simulation.

Technical details. We define a multi-speed system M' simulating M. The system M' consists of $3m+2$ automata $X, Y, B_1, \ldots, B_m, B'_1, \ldots, B'_m, D_1, \ldots, D_m$. Their roles are the following:

- X simulates behavior of each A_i during a silent stage and stores the internal states of A_i and current messages sent by automata of M. The simulation is sequential, that is, first X simulates A_1, then A_2 and so on;
- $B_1, \ldots, B_m$ store the current positions of the heads of automata $A_1, \ldots, A_m$ during the simulated computation. The second group of automata consisting of $B'_1, \ldots, B'_m$ is used to store the same information as $B_1, \ldots, B_m$;

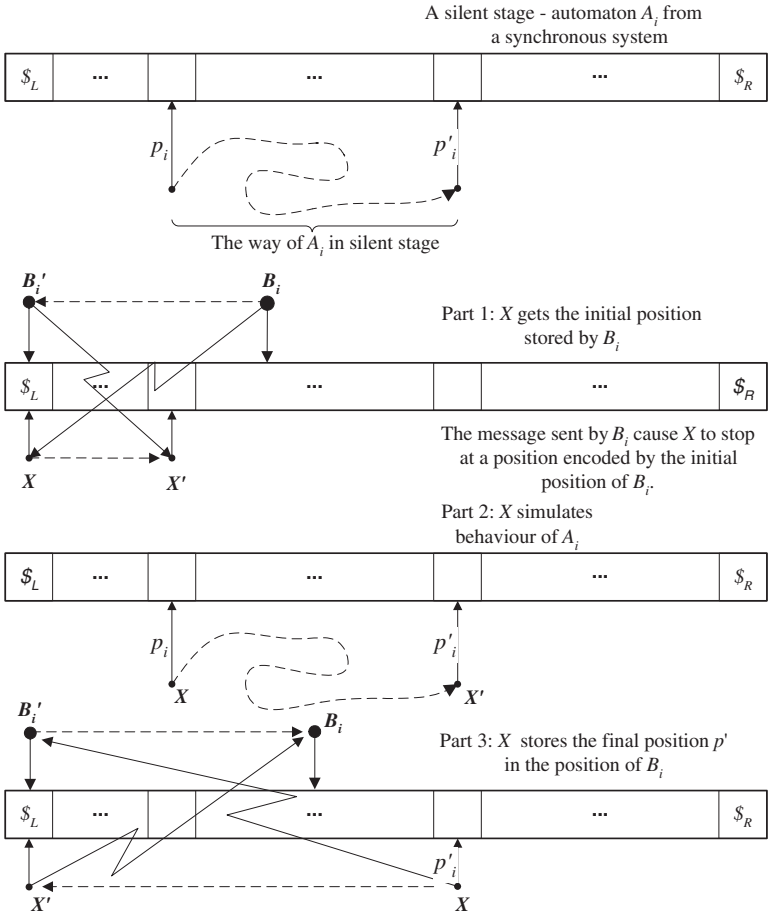

FIGURE 2. Basic steps of a silent stage. The p_i is the position at the beginning of the stage, p'_i is a position at the end of stage. Dashed-line marks the movements.

- $D_1, \ldots, D_m$ are used as auxiliary counters. The value stored in the position of the head of D_i encodes the number of steps performed by automaton A_i during the computation simulated during the preprocessing phase. These values are used to calculate the number of steps of the silent stage.

Now let us describe the simulation in detail. We split the description in two parts: first we describe computations informally, then we give the corresponding pseudo-code (see also Fig. 2).

for $j = 0, 1, \ldots,$ **until the halting message is sent, do**

I *Perform preprocessing for the jth silent stage*:

For each automaton A_i the following steps are performed. The head of X is moved to the position of the head of A_i at the end of the previous

stage. This position is restored from the head position of B_i. The state of A_i at the end of the previous stage, which was stored in the internal state of B_i, gets stored in the internal state of X. The counter D_i is set to zero.

X simulates behavior of A_i – until A_i sends a message. At the same time automaton D_i moves to the right from the left end-marker, making one move per λz steps. D_i measures time used by X for the current simulation. The message from X stops the movement of D_i: the number of steps executed by X is stored by the head position of D_i and a state of D_i. (The state encodes a *correcting information*, similarly as in procedures dump and restore.) Also, the automaton D_i sends a message when it reaches the right end-marker. This is an evidence that A_i loops.

II *Determine t_j, the duration of the jth silent stage: The minimal time from values stored by $\{D_1, \ldots, D_m\}$ is computed.*

We execute the command `minimum` introduced in the previous section. For the sake of exposition in the third part we assume that A_p is the automaton that sends a message, *i.e.* D_p stores the minimal value.

III *Simulate the jth silent stage for t_j steps and perform the next step when a message is sent:*

The duration of the silent stage, obtained in the previous step, is encoded in the head position of X. Two copies are made: one copy (held by position of automaton Y) is used for a currently simulated automaton, another copy (held by Y') is kept for other automata.

for $i = 1, \ldots, m$ the following simulation is executed in two separate loops:

1. automaton X restores the head position of A_i encoded by B'_i and simulates behavior of A_i for t_j steps; this is possible by using automata Y, Y' storing duration of the silent stage;
2. the communication step after the silent stage is simulated; the position of the head of A_i after the communication step is updated and stored by X again in the head positions of automata B_i, B'_i.

end for stage j

for stage $j = 0, 1, \ldots,$ **until the halting message is sent, do**

I *Perform preprocessing for the jth silent stage*

1. `for` $i \leftarrow 1$ `to` m `do`
2. `begin`
3. `restore` $X \leftarrow B_i$
4. D_i `go to` $\$_L$, then it sends a message
5. `do in parallel till a message is sent by` X `or` D_i`:` $\{D_i$ `counts steps` $\mid X$ `simulates` $A_i\}$
6. `endfor`

II

7. $X \leftarrow \texttt{minimum}(D_1, \ldots, D_m)$

III

8. `dump` $X \rightarrow Y, Y'$
9. `for` $i \leftarrow 1$ `to` m `do`
   `begin`
10. `restore` $X \leftarrow B'_i$
11. `do in parallel till a message from` X `or` Y`:`
    $\{Y$ `go to` $\$_L$, then it sends a message $\mid X$ `simulates` $A_i\}$
12. X adjusts its head position
13. `dump` $X \rightarrow B_i, B'_i$
14. `restore` $Y \leftarrow Y'(Y'')$
    `endfor`
15. `for` $i \leftarrow 1$ `to` m `do`
16. `begin`
    `restore` $X \leftarrow B_i$
    X performs one step of A_i
    `dump` $X \leftarrow B_i, B'_i$
    `endfor`

end for stage j

Complexity of simulation. It follows from the construction that each silent stage of system M is simulated by M' using $O(m)$ messages. Then a step of M in which messages are sent is simulated using $O(m)$ messages. Thus, if the original synchronous system uses $O(f(n))$ messages, the new multi-speed system uses $O(m \cdot f(n))$ messages. Since the number of automata in the system M is constant, the number of messages during the simulation increases only by a constant multiplicative factor m.

6. Conclusions

We have shown that fully synchronous systems of finite automata may be simulated by multi-speed systems with the same asymptotic message complexity, assuming that the ratio between the speed of the fastest and the slowest automaton is bounded by a (known) constant.

The simulation presented in this paper requires that there is a constant bound on the ratio between the speed of the fastest and the slowest automaton and this bound is known in advance. It is an open problem whether analogous result is possible when this ratio is bounded and unknown or unbounded. However we expect that the answer is negative what possibly can be shown using techniques from [6].

References

[1] A. Borodin and S. Cook, A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.* **11** (1982) 287–297.

[2] P. Beame, M. Tompa and P. Yan, Communication-space tradeoffs for unrestricted protocols. *SIAM J. Comput.* **23** (1994) 652–661.

[3] T. Jurdziński and M. Kutyłowski, Communication gap for finite memory devices. *Automata, Languages and Programming, in Proc. ICALP'2001. Lect. Notes Comput. Sci.* **2076** (2001) 1052–1064.

[4] T. Jurdziński, K. Kutyłowski and K. Loryś, Multi-party finite computations. *Computing and Combinatorics, in Proc. COCOON '99. Lect. Notes Comput. Sci.* **1627** (1999) 318–329.

[5] T. Jurdziński, M. Kutyłowski, P. Rzechonek and J. Zatopiański, Communication complexity for multi-speed cooperation automata. *I Konferencja Młodych Matematyków*, Oficyna Wydawnicza Politechniki Wrocławskiej (2001).

[6] T. Jurdziński, M. Kutyłowski and J. Zatopiański, Communication complexity for asynchronous systems of finite devices, in *Proc. 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*, Los Alamitos, CA, April 23–27. *IEEE Comput. Soci.* (2001) 139.

[7] V. Mitrana and C. Martin-Vide, Some undecidable problems for parallel communicating finite automata systems. *Inform. Proc. Lett.* **77** (2001) 239–245.

[8] V. Mitrana, On the degree of communication in parallel communicating finite automata systems. *IWDCAGRS: Proc. International Workshop on Descriptional Complexity of Automata, Grammars and Related Structures*, Otto-von-Guericke Universität (1999) 155–165.

[9] J. Zatopiański, *Computational complexity of limited memory systems*. Ph.D. Thesis, Institute of Electronics, Wrocław University of Technology, Wrocław (2002).

Communicated by C. Choffrut.
Received November, 2002. Accepted September, 2004.