Scheduling of client-server applications

Jacek Błażewicz*, Paolo Dell'Olmo†, Maciej Drozdowski*‡

Abstract

In this paper we analyze the problem of deterministic scheduling of applications (programs) in a client-server environment. We assume that the client reads data from the server, processes it, and stores the results on the server. This paradigm can also model a wider class of parallel applications. The goal is to find the shortest schedule. It is shown that the general problem is computationally hard. However, any list scheduling algorithm delivers solutions not worse than twice the optimum when tasks are preallocated, and three times the optimum when tasks are not preallocated. A polynomially solvable case is also presented.

Keywords: Deterministic scheduling, multiprocessor tasks, communication delays, distributed processing.

Introduction

Parallel processing has received a lot of attention for many years. Efficiency is a crucial part of a parallel computer system success. Hence, there is a necessity for the development of efficient scheduling algorithms.

In this paper we consider scheduling in a client-server environment (cf. e.g. (Adler, 1995), (Deux *et al.*,1991)), (Tanenbaum, 1989). We assume that the process of cooperation between the server and the client has the following form. The client reads some amount of data from the server, processes it and stores the results on the server. It can be an example of processing a simple transaction in a distributed database. For example, the client reads a record from a database, updates it and stores the result. This model

*Instytut Informatyki, Politechnika Poznańska, ul. Piotrowo 3A, 60-965 Poznań, Poland. The research has been partially supported by a KBN grant and project CRIT2.

†Istituto di Analisi dei Sistemi ed Informatica del C.N.R., Viale Manzoni 30, 00185 Roma, Italy, and Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Via della Ricerca Scientifica, 00133 Roma, Italy.

‡Corresponding author.

can be also applied to a wider class of distributed applications. Though the roles are different, an analogous situation appears in the master-slave parallel processing (Jean-Marie *et al.*, 1998), (Aref and Tayyib, 1998). The master processor sends data to a slave for remote processing and the results are sent back to the master from the slave. As an example consider a distributed branch-and-bound algorithm in which the master manages a queue of nodes to expand, the slaves expand the nodes from the queue and return only the offspring leading to potentially better solutions (Gendron and Crainic, 1994), (Mitra *et al.*, 1997). Similarly, in parallel implementations of local search methods such as tabu search, simulated annealing, genetic algorithms etc. (Garcia *et al.*, 1994), (Porto and Ribeiro, 1995), the master delegates search in the neighborhood of the current solution(s) to the slaves. The slaves return the most promising results for selection of the candidates for further search. Analogous situation appears in the case of a file server shared by a set of applications distributed in the network (Fleischmann, 1994). Application roll-in (i.e. unfolding the application code and data) and roll-out (storing data or results) also can be represented by our client-server model. The proposed model is applicable also in production environment. The role of the server can be performed e.g. by a robot delivering parts to other machines. In the fabrication of integrated circuits certain workstations are revisited a number of times during the production of a wafer. Such a hub machine is an equivalent of our server. The production context has been considered e.g. by Kubiak *et al.* (1996), Lane and Sidney (1993), and Wang *et al.* (1997). For the sake of simplicity in the future discussion we will refer to the client-server example terminology. Note, that in all the above examples the communication operation involves simultaneously both the server and the client.

Now, we define our problem more formally. Server S is connected to set $\mathcal{P}$ of m processors $P_1, P_2, \ldots, P_m$ where client applications are executed. Thus, by a processor we mean *only* the machine running applications (tasks). The interconnection network is a bus or a star where the server can communicate with only one processor at a time. Processors can be characterized by two features. Firstly, the processors are either dedicated or identical. In the former case we assume that each task is *preallocated* and the processing operation can be performed by a particular processor only. In the latter case tasks are not preallocated and any processor can execute any task. Secondly, processors may have limited or unlimited number of *buffers*. If a processor

has only one buffer, data must be sequentially read from the server, processed by the processor, and written back to the server. No other sequence of the operations is possible. For example, two (or more) consecutive read operations are not allowed. Therefore, the processor cannot execute any other task during the communication period. When each processor has more than one buffer, it is possible to perform simultaneously computations and communications with the server.

Client applications constitute a task system. Set $\mathcal{T}$ of tasks consists of n elements. As was mentioned a task consists of three sequential operations: reading, processing, writing. This can be represented as a chain of three operations constituting the task. The times required to execute the above three operations are given, and will be denoted r_j, p_j, w_j, $(j = 1, \ldots, n)$ for reading from the server, processing, and writing to the server, respectively. When processors are identical, the three execution times are independent of the processors. When the processors are dedicated and tasks are preallocated we will denote by $\phi_j = P_i$ that processor P_i is required by task T_j. We assume, that the activities performed by the server alone in connection with executing the tasks are included in the communication delays (i.e. in r_j or w_j). When the communication times for all tasks are equal we will say that this is *unit communication time* (UCT) case. Finally, we assume that processing operations are *nonpreemptable*, i.e. once started, the operation must be processed continuously till the very end. We also assume, in general, that communication operations are nonpreemptable. Schedule length (makespan) is the optimality criterion. The above problem will be called *client-server scheduling problem* (CSS).

Here, we survey previous results in the scheduling area pointing out similarities and differencies with the proposed model. Our problem bears some similarity to the reentrant flow-shop considered by Kubiak *et al.* (1996), Lane and Sidney (1993), Wang *et al.* (1997), flexible flow-shop (Hoogeveen *et al.*, 1996) or the job-shop model in general. Yet, the results known from the literature do not exactly coincide with the problem we analyze, and only few results can be applied in our case. Observe, that here the role of the buffers is different than in the above models. In the job-shop model, while processing one operation at the first machine (assume it is our 'server') the other machine (our processor) can process some operation of another task. However, the first operation of a task can be always performed independently of the status of the next required machine. It is not always the case here.

For example, for the one buffer case communication operations require the server and the processor simultaneously. Hence, the buffer is required to perform the first (communication) operation, and the very same buffer is necessary for the second (processing) operation. Note, that thus the communication requires simultaneously two machines (the server and one of the processors from $\mathcal{P}$) and this is a special case of the multiprocessor task system (cf. (Błażewicz *et al.*, 1992), (Błażewicz *et al.*, 1986), (Drozdowski, 1996), (Du and Leung, 1989), (Veltman *et al.*, 1990)). In particular, scheduling file transfers was examined by Coffman *et al.* (1985) and Kubale (1987), but preallocation of tasks was essential and processing operations were not taken into account. When the number of buffers is limited (but greater than 1) the server cannot start sending until a buffer of the processor is released.

On the other hand, when buffers are not a scarce resource (denoted by $buffers = \infty$), our problem becomes equivalent to some earlier models. When tasks are not preallocated and one of the two communication operations (either all reading or all writing operations) have negligible duration, our problem becomes equivalent with two-stage flexible flow-shop (e.g. (Hoogeveen *et al.*, 1996)). The complexity results of Hoogeveen *et al.* (1996) are applied in this work. When $m = 1$, $buffers = \infty$ and $w_j = 0$ for $j = 1, \ldots, n$ our problem is equivalent to two machine flowshop, and hence, the Johnson's rule can be applied (cf. e.g. Błażewicz *et al.*,1996). The case of preallocated tasks ($buffers = \infty$) is very much alike reentrant shops considered in (Kubiak *et al.*, 1996), (Lane and Sidney, 1993), (Wang *et al.*, 1997). In particular, in (Kubiak *et al.* 1996) it was assumed that the odd operations in a chain return to one 'server' machine. While using the 'server' by one task a processor could process another operation of some other task (which is not always the case here). Preallocation was assumed and mean flow time was the optimality criterion. In (Wang *et al.*, 1997), where schedule length was the optimality criterion, tasks visited the 'server' twice: at the first and at the last $(m + 2)$th operation. Processors $P_1, \ldots, P_m$ were required to perform, respectively, operations 2nd,$\ldots, m$th of a task. The problem is **NP**-hard already for $m = 1$. A branch-and-bound algorithm and a set of dominance properties for the general problem have been presented. Our problem with $m = 1$ and unlimited number of buffers is equivalent to the problem considered in (Wang *et al.*, 1997). Lane and Sidney (1993) also investigated a problem equivalent to ours when $m = 1$ and $buffers = \infty$. Two heuristics were proposed and analyzed for the mean flow time and schedule

length criteria. The authors considered baking of semiconductor wafers as one of the applications for their model. Many wafers can be baked together in a single batch. Thus, total processing time can be reduced. This is not possible in our problem. The case of preallocated tasks and unlimited number of buffers is a special case of a job-shop with tasks comprising at most three operations. It is well known that job-shop problem with three processors and tasks comprising at most two operations is strongly **NP**-hard (e.g. (Gonzales and Sahni, 1978)). However, the proofs of complexity for these problems cannot be applied in our case because the routing of operations is more restricted here. Furthermore, the existence of the third operation in a task prevents from applying in our problem approximate algorithms and special polynomial-time cases from two-stage flexible flowshop (Chen, 1993), (Hall *et al.*, 1996). Thus, some earlier results can be applied in our case and some other cannot.

The organization of the work is as follows. In Section 2 complexity results are presented. In Section 3 heuristic algorithms are described and their worst case performance ratios are analyzed. Finally, Section 4 contains special cases solvable in polynomial time.

NP-hardness results

In this section we analyze the complexity of the problem under different assumptions. We start with the case of a limited number of buffers.

Observation 1 *For $m = 1$ and one buffer any schedule introducing no idle time on the processor is optimal.*

Proof. A lower bound on the schedule length is the total duration of operations involving the processor, i.e. $\sum_{j=1}^{n}(r_j + p_j + w_j)$. Since the processor has one buffer the sequence of the operations on the processor must be: reading, processing, writing. Any order of executing tasks is feasible. The lower bound is attained provided that there is no unnecessary idle time between the operations. □

Theorem 1 *The CSS problem with preallocation is strongly* **NP***-hard even for $m = 2$ and one buffer.*

Proof. We prove strong **NP**-hardness by reduction of 3-PARTITION to a decision version of our problem. 3-PARTITION is defined as follows.

3-PARTITION
INSTANCE: Set A of $3q$ integers a_j $(j = 1, \ldots, 3q)$, such that $\sum_{j=1}^{3q} a_j = Bq$ and $B/4 < a_j < B/2$ for $j = 1, \ldots, 3q$. Without loss of generality we assume that $B > 2q$.
QUESTION: Can A be partitioned into q disjoint subsets $A_1, \ldots, A_q$ such that $\sum_{a_j \in A_i} a_j = B$ for $i = 1, \ldots, q$?
The above problem can be reduced to the our one by the following transformation:

$n = 4q$
$\mathcal{T} = X \cup Y$
$X: \ r_j = w_j = 1, p_j = 6B^4 + B^3, \phi_j = P_2$ for $j = 1, \ldots, q$
$Y: \ r_j = w_j = B^4, p_j = B^2 a_{j-q}, \phi_j = P_1$ for $j = q+1, \ldots, 4q$
$y = q(6B^4 + B^3 + 2)$.

We ask whether for the above task set a schedule of length at most y exists. Please note that this transformation will be also used in the proof of the next theorem. Now, we have to prove that a feasible schedule of length at most y exists if and only if the answer to 3-PARTITION is positive. Suppose the answer to 3-PARTITION is positive, then a feasible schedule of length y looks like the one in Fig.1.

Insert Fig.1 here

Suppose the answer to our scheduling problem is positive. Since each processor has only one buffer the communication from the viewpoint of the processor must always follow the pattern: reading, writing, reading, writing, ..., writing. The sum of communication and processing times of tasks in set X preallocated to P_2 is exactly equal to y. Hence, tasks from X must be executed one by one on P_2 during the whole length of the schedule as depicted in Fig.1. While tasks in X are processed on P_2, the server is free in q intervals of length $6B^4 + B^3$ each. Only in these intervals can tasks from set Y use the server. In each of these intervals exactly 6 communications with the server must be done by tasks in set Y, otherwise no schedule of length y exists. Since the pattern reading/writing must be observed, exactly 3 tasks must be fully executed in each of the considered intervals. We see that no processing can be executed on P_1 in parallel with set X task communications. Thus, in each of q intervals there remain B units of free time on P_1 where processing must be performed. Since the amount of free time on P_1 is exactly

what is required by tasks in Y the answer for 3-PARTITION must be positive. □

Theorem 2 *The CSS problem without preallocation is strongly* **NP**- *hard even for* $m = 2$ *and one buffer.*

Proof. The transformation in this case remains the same as in the proof of Theorem 1 except for the fact that the processing can be done on an arbitrary processor. When the answer to 3-PARTITION is positive a feasible schedule of length y is the same as in Fig.1.

Observe, that in the feasible schedule the server can be idle only B^3q units of time, while both processors can be idle at most $2q$ units of time. Thus, the server cannot start with sending operation to a task from set Y because this would result in at least $B^4 > 2q$ units of idle time on the other processor. Hence, the server must start sending to a task from set X. Furthermore, communications with tasks from set Y must be executed in parallel with processing tasks from set X.

Suppose the server sends data to task T_h from set X first, and immediately after this, again to task T_i from set X. Since there is only one buffer on each processor and the pattern of reading, writing must be observed on both processors, the server must idle until the writing operation of the first task. The idle period lasts as long as processing operation of the first task minus duration of the reading operation of the second task. This would result in $6B^4 + B^3 - 1 > B^3q$ units of the server being idle. Thus, consecutive sending operations to two tasks from set X are not allowed. Suppose that between reading operations of the two tasks from set X exactly one task of set Y is processed. This would result in the server being idle at least $6B^4 + B^3 - 2B^4 - 1 > B^3q$ units of time between the reading and the writing operation of T_h. Thus, more than one task from Y must be executed while processing a task from X. Suppose two tasks from Y are processed between two consecutive reading operations of tasks from X. Again, this would result in the server being idle at least $6B^4 + B^3 - 4B^4 - 1 > B^3q$ units of time between the reading and writing operation of T_h. Hence, at least three tasks from Y must follow a sending to a task from X. Suppose four tasks from Y follow the sending to a task from X. In this case the processing operation of the task from X ends before or coincides with performing reading operation of the fourth task from Y. Then, the processor which processed the task

from X would have to wait for the end of the communication to the fourth task of Y. Thus, at least $7B^4 + B^2(a_j + a_k + a_l) - (6B^4 + B^3) > 2q$ idle time would be created on the processor. We conclude that exactly three tasks from Y must be executed while processing a task from X.

Consider the amount of processing the three tasks $T_j, T_k, T_l \in Y$ receive. Assume that the sum of processing times is greater than B^3, i.e. $B^2(a_i + a_j + a_k) \geq B^3 + B^2$. This would result in the processor which processed a task from X being idle at least $B^2 > 2q$ units of time. Such a situation is not allowed because $2q$ is the maximum allowed idle time on both processors and $B > 2q$. This reasoning can be applied to each triplet of tasks out of set Y executed in parallel with some processing operation of an X set task. Since $\sum_{i=1}^{3q} a_i = qB$ and for each of q triplets $B^2(a_i + a_j + a_k) \leq B^3$, we conclude that the three tasks of each triplet must be processed in B^3 time. Hence, a positive answer to 3-PARTITION. □

Let us consider now the case of the unlimited number of buffers.

Theorem 3 *The CSS problem is* **NP***-hard even for* $m = 1$ *and unbounded number of buffers.*

Proof. The proof is similar to the one given by Rinnoy Kan (1976). We prove this theorem by reduction from PARTITION:

PARTITION
INSTANCE: Set A of q numbers a_j $(j = 1, \ldots, q)$, such that $\sum_{j=1}^{q} a_j = 2B$.
QUESTION: Can A be partitioned into two disjoint subsets A_1, A_2 such that $\sum_{a_j \in A_1} a_j = \sum_{a_j \in A_2} a_j = B$?
The instance of our problem can be built as follows:

$n = q + 1$
$r_j = w_j = 0; p_j = a_j$ for $j = 1, \ldots, q$
$r_n = p_n = w_n = B$
$y = 3B$.

A feasible schedule of length y is depicted in Fig.2. Since operations of T_n must be executed continuously, two time slots B units long remain available on P_1. Since no idle time is allowed and operations are nonpreemptable, no feasible schedule of length y can exist without positive answer to the PARTITION problem. □

Insert Fig.2 here

Theorem 4 *The CSS problem with preallocation and unbounded number of buffers is strongly* **NP***-hard for* $m=2$.

Proof. We prove the theorem by reduction from 3-PARTITION. We assume (w.l.o.g.) that $B > 2q, B > 4$. The instance of CSS can be constructed as follows:

$n = 4q + 1$
$r_j = a_j, p_j = B^3 a_j, w_j = 1, \phi_j = P_1$ for $j = 1, \ldots, 3q$
$r_{3q+1} = 1, p_{3q+1} = B^4 + B, w_{3q+1} = 1, \phi_{3q+1} = P_2$
$r_j = B^4, p_j = B^4 + B, w_j = 1, \phi_j = P_2$ for $j = 3q + 2, \ldots, 4q$
$r_{4q+1} = B^4, p_{4q+1} = 4q, w_{4q+1} = 1, \phi_{4q+1} = P_2$
$y = q(B^4 + B + 4) + 2$.

We ask whether a schedule not longer than y exists. If a 3-PARTITION exists a feasible schedule of length y may look like the one in Fig.3. Note, that P_1 can start processing tasks immediately after the first reading operation and some idle time may appear on P_1 after three consecutive processing operations. However, for the sake of presentation clarity we assumed in Fig.3 that P_1 is idle during reading operations. Mind that it is not prerequisite for the existence of a feasible schedule when a 3-PARTITION exists. There can be also other schedules not longer than y when a 3-PARTITION exists. Insert Fig.3 here

Suppose, a feasible schedule not longer than y exists. The server must communicate all the time because the sum of communication times is equal to y. The total processing on P_2 is $q(B^4 + B + 4)$ and the shortest reading and writing operations last two units of time. Thus, P_2 must compute all the time with the exception of two units of time: the first and the last unit of time in the schedule. Moreover, T_{3q+1} must start the schedule and T_{4q+1} must finish the schedule, otherwise P_2 is idle longer than two units of time. The total processing requirement on P_1 is qB^4. Hence, after including the first reading operation and the last writing operation of the server, P_1 can be idle at most $q(B + 4)$ time units. To avoid idling on P_2 the second task executed on P_2 must start reading its data at time $B + 1$ at the latest. Therefore, no more than three reading operations for the tasks executed on P_1 can be done. Also, no less than three communications to P_1 can be done. Suppose it is otherwise and (w.l.o.g.) exactly two tasks T_i, T_j are started on P_1, T_i is started first. Then, there would be excessive idle time on P_1 since the end of T_j processing operation till the end of the reading

operation of the task processed by P_2. $B^4 + a_j$ is the span of the interval from the completion of T_i reading operation till the end of reading operation of the task processed by P_2. $B^3(a_i + a_j)$ is the time of processing operations which can be executed on P_1 in this interval. The idle time on P_1 would be at least $B^4 + a_j - B^3(a_i + a_j)$. Since $B > 2q$ and $B > 4$ we have $B^4 + a_j - B^3(a_i + a_j) > B^4 - B^3(B-1) = B^3 > B^2 + B^2 > qB + 4q$, while the idle time on P_1 cannot be greater than $q(B+4)$.

The sum of processing times of the three tasks allocated to P_1 must be equal to B^4. If it is less, then it is at most $B^4 - B^3$ which results in $B^3 > q(B+4)$ idle time on P_1 while reading operation of the second task allocated to P_2. Suppose it is more, then their reading operations last longer than B and the reading operation of the second task allocated to P_2 cannot start in time, which results in additional idle time on P_2. Consequently, schedule of length y cannot exist. We conclude that the three tasks must be processed in exactly B^4 time units.

The same reasoning can be applied to the following tasks assigned to P_2. The reading operations of these tasks cannot be started later than by $1 + iB^4 + (i+1)B$ for $i = 1, \ldots, q-1$. This creates free time interval for at most three reading operations for tasks assigned to P_1. Also no less than three tasks can read from server otherwise there will be excessive idle time on P_1 during the next reading operation of a task assigned to P_2. The processing times of the three tasks must be equal exactly B^4, otherwise either P_1 or the server must be idle. We conclude that for each triplet of tasks T_i, T_j, T_k assigned to P_1 the processing time satisfies $B^3(a_i + a_j + a_k) = B^4$. Hence, a positive answer to 3-PARTITION problem. □

Theorem 5 *The CSS problem without preallocation, and with unbounded number of buffers is strongly* **NP***-hard even for* $m = 2$.

Proof. When $\forall_{j \in \mathcal{T}}\ r_j = 0$ our problem boils down to the two-machine flexible flowshop shown to be strongly **NP**-hard in (Hoogeveen *et al.*, 1996). □

Theorem 6 *The CSS problem without preallocation, and with unbounded number of buffers, unit communication times is* **NP***-hard for* $m = 2$ *and* **NP***-hard in the strong sense for arbitrary* $m > 2$.

Proof. The proofs we present are analogous to the ones for nonpreemptive multiprocessor scheduling (Karp, 1972), (Garey and Johnson, 1978). We give our versions for the completeness of the presentation. First, we tackle **NP**-hardness for two machines. The transformation is done from PARTITION problem defined in Theorem 3. The instance of CSS can be defined as follows:

$n = q$
$m = 2$
$r_j = w_j = 1$ for $j = 1, \ldots, q$
$p_j = B^2 a_j$ for $j = 1, \ldots, q$
$y = B^3 + 2q$.

We assume additionally that $B > 2q$. Suppose, the answer to PARTITION is positive, then by scheduling processing operations of the tasks corresponding to A_1 on P_1, processing operations of the remaining tasks on P_2, and performing communications in any order (but without idle time) we get a schedule of length y. Suppose, the answer is positive to CSS problem, then processing operations on P_1 and P_2 last (both) at most B^3. Hence, a positive answer to the PARTITION problem.

Next we prove, by transformation from 3-PARTITION, that the problem is **NP**-hard in the strong sense when the number of processors is not fixed. The 3-PARTITION problem is defined as in the proof of Theorem 1. Additionally we assume (w.l.o.g.) that $B > 6q$. The corresponding instance of our scheduling problem is defined as follows:

$n = 3q$
$m = q$
$r_j = w_j = 1$ for $j = 1, \ldots, 3q$
$p_j = B^2 a_j$ for $j = 1, \ldots, 3q$
$y = B^3 + 6q$.

When the answer to the 3-PARTITION is positive a feasible schedule may look like the one in the Fig.4. Assume, the answer to our scheduling problem is positive. Let us analyze the amount of processing the tasks receive on one processor. Suppose, it is greater than B^3. Then it is at least $B^3+B^2 > B^3+6q$ and a feasible schedule may not exist. Assume it is less than B^3. Then it is at most $B^3 - B^2$ and on some other processor the tasks must be processed longer than B^3 and a feasible schedule does not exist. Thus, the tasks on each processor are executed exactly B^3 units of time. By choice of values

Insert Fig.4 here

$B/4 < a_j < B/2$ $(j = 1, \ldots, q)$ in the 3-PARTITON exactly three tasks must be executed by each processor in B^3 units of time. Hence, the answer to 3-PARTITION is positive. □

Now, let us comment on the case with preemptable communications (which seem to be closer to computer reality). The problems with unlimited buffers and identical processors are **NP**-hard for $m = 2$ and strongly **NP**-hard for arbitrary $m > 2$ as the underlying parallel processor scheduling problems have such complexity (cf. Theorem 6). Moreover, Theorem 3 through Theorem 6 are valid even if communications are preemptable. The case with one buffer, preemptable communications, and dedicated processors remains open.

Approximation algorithms

In this section we analyze the performance of list scheduling algorithms. We show that any list scheduling algorithm performs relatively well. By a list scheduling algorithm we mean here a greedy heuristic which leaves no processor idle if it is possible to process some task on it. The algorithms can be defined as follows.

SERVER:
while there is something to process **do**
begin
1: if it is possible to receive anything – receive it;
2: if it is possible to send anything – send it;
end

PROCESSOR:
while there is something to process **do**
begin
1: read data;
2: process a task;
3: write results back;
end

Note, that this description matches a wide range of list scheduling methods. The above algorithm can be implemented to run in $O(n)$ time. Now, we will analyze the worst case performance of list scheduling algorithms. Let us

denote by E the length of the schedule built by our heuristic, and by OPT the length of the optimal schedule.

Theorem 7 *The worst case performance ratio of any list scheduling algorithm for CSS problem and preallocated tasks satisfies*

$$\frac{E}{OPT} \leq 2$$

and this bound is tight.

Proof. First, let us observe that in the schedule built by a greedy heuristic we can distinguish two types of time intervals. In the intervals of the first type the server is communicating. In the intervals of the second type the server is idle because it is waiting for completion of processing operations on at least one processor or for the completion of the last processing operation on the processors. The total length of the first type intervals is equal to the total requirement for communication. The server may not wait longer for the completion of the last processing operation on any processor than the processing on the most loaded processor minus the amount of processing performed in the intervals where all processors work and the server is idle. Thus, the total length of the second type intervals may not be greater than the processing time on the most loaded processor P_*. Thus, the length E of the schedule built by a greedy heuristic is bounded from above by

$$E \leq \sum_{j=1}^{n}(r_j + w_j) + \sum_{j \in \mathcal{T}_*} p_j$$

where $\mathcal{T}_*$ is the set of tasks assigned to P_*. Since both $\sum_{j=1}^{n}(r_j + w_j) \leq OPT$ and $\sum_{j \in \mathcal{T}_*} p_j \leq OPT$ we have $E \leq 2OPT$ and $\frac{E}{OPT} \leq 2$.

Now, we will demonstrate that this bound is tight. Consider the following example: $m = 3$, $n = x + 1$, where $x \in Z^+$ and x is even, $r_j = w_j = 1$ for $j = 1, \ldots, n$, $p_j = 1, \phi_j = P_{2+(j-1) \bmod 2}$ for $j = 1, \ldots, n-1$, $p_n = 2x, \phi_n = P_1$. The optimal schedule is presented in Fig.5a and has length equal to the total communication time, i.e. $OPT = 2x + 2$. The schedule built by a greedy heuristic may look like the one in Fig.5b which has length $E = 4x + 2$. Thus, $\lim_{x \to \infty} \frac{E}{OPT} = 2$. Note, that only one buffer is used on each processor. The bound is attained despite the fact that processors may have more buffers. □

Insert Fig.5 here

Theorem 8 *The worst case performance ratio of any greedy heuristic for CSS problem and non-preallocated tasks satisfies*

$$\frac{E}{OPT} \leq 3$$

and this bound is tight.

Proof. As in the previous proof, we can distinguish two types of time intervals in the schedule built by a greedy heuristic. In the intervals of the first type the server is communicating, in the intervals of the second type the server is idle because it is waiting for a writing operation from any processor. The total length of the first type intervals is equal to $\sum_{j=1}^{n}(r_j + w_j)$. Among the intervals of the second type two sub-types can be distinguished. In the intervals of the first sub-type the server is idle and all processors are working. In the intervals of the second sub-type the server is idle but not all processors are working because there are no more tasks to be started. Total length of the first sub-type intervals may not be greater than $\sum_{j=1}^{n} \frac{p_j}{m}$ which is the case in which the whole load is evenly distributed among the processors. The length of the second sub-type intervals may not be greater than the longest processing operation. Thus, length E of the schedule built by our heuristic is bounded from above by

$$E \leq \sum_{j=1}^{n}(r_j + w_j) + \sum_{j=1}^{n} \frac{p_j}{m} + \max_{1 \leq j \leq m}\{p_j\}$$

Since both $\sum_{j=1}^{n}(r_j + w_j) \leq OPT$ and $\sum_{j=1}^{n} \frac{p_j}{m} + \max_{1 \leq j \leq m}\{p_j\} \leq 2OPT$ we have $E \leq 3OPT$ and $\frac{E}{OPT} \leq 3$.

To demonstrate that the bound is tight consider the following instance. We assume that processors have at least two buffers. Let $k, l \in Z^+$ be two integers such that l is even, $l > m > 2$, $k(m-2)$ is a multiple of the number of processors m. Furthermore let:

$n = k(m-2) + 1 + (klm + 2(m-2)(1-k))/2$
$\mathcal{T} = X \cup Y \cup Z$
$X : r_1 = w_1 = 1, p_1 = klm + 2(m-2) = a$
$Y : r_j = w_j = 1, p_j = lm$ for $j = 2, \ldots, k(m-2) + 1$
$Z : r_j = w_j = 1, p_j = 1$
for $j = k(m-2) + 2, \ldots, k(m-2) + 1 + (klm + 2(m-2)(1-k))/2$.

The optimal schedule is depicted in Fig.6a. This schedule has a particular structure. Task T_1 is started first, processed on processor (w.l.o.g.) P_m, and its writing operation completes the schedule. Tasks from set Y are performed in k blocks of $m-2$ tasks occupying processors $P_1, \ldots, P_{m-2}$. Let us assume that tasks from set Y are started in the order of their indices, and their processing operations are assigned consecutively to processors $P_1, P_2, \ldots, P_{m-2}, P_1, P_2, \ldots$ etc. The reading operations in the first block of Y tasks are performed with a unit delay. In the last time unit of task T_2 processing operation a new reading operation is started for task T_m. Both T_2 and T_m are processed by P_1. After this reading operation the writing operation of T_2 is initiated. Analogously on the following processors and in the following blocks of Y tasks, first reading operations are executed for the tasks to be started, then the writing operations of the completed tasks are done. The final writing operations of the tasks from set Y are also executed with unit delays. The free time slots remaining on the server after performing communication operations for tasks from set X and Y are filled by communication operations of the tasks from set Z. Note that the number of such free time slots is even and equal to $a-2 \mid Y \mid = klm + 2(m-2) - 2k(m-2) = klm + 2(m-2)(1-k) = 2 \mid Z \mid$. The communications in the free time slots are done according to the pattern: reading, reading, writing, writing. When the number of tasks in set Z is not even then the last two communication operations for the task from set Z are reading, writing. Processing operations for tasks from set Z are executed on P_{m-1}. The total length of the optimal schedule is equal to the required amount of communications, i.e. $OPT = a + 2$.

A schedule built by a list scheduling algorithm may be as the one in Fig.6b. First, the tasks from set Z are processed which saturates the server. Then, tasks from set Y are executed in blocks of m tasks which saturates the processors. Task T_1 is started as the last one immediately after the processing operation of task T_l – the first task from set Y in the last block of set Y tasks (cf. Fig.6b). Tasks from set Z continuously occupy the server during time interval of length $b = 2 \mid Z \mid = klm + 2(m-2)(1-k)$. The time from starting the first reading operation of a task from set Y till the end of task T_l processing operation is equal to $c = lm\frac{|Y|}{m} + 1 = lm\frac{k(m-2)}{m} + 1 = klm\frac{m-2}{m} + 1$. The length of the period from starting the processing operation of T_1 till the completion of its writing operation is $d = a + 1 = klm + 2(m-2) + 1$. In what follows we are going to demonstrate that in the limit values of b, c, and

d can be made very close to each other and to OPT. The total length of the schedule in this case is:
$E = b + c + d = klm + 2(m-2)(1-k) + klm(1-\frac{2}{m}) + 2 + klm + 2(m-2)$.
Thus $\lim_{k,l,m\to\infty} \frac{E}{OPT} = \lim_{k,l,m\to\infty} \frac{3klm+2(m-2)(1-k)-2kl+2+2(m-2)}{klm+2m-2} = 3$. □

Observation 2 *When tasks are non-preallocated and the number of buffers is unbounded the* Longest Processing Time (LPT) *heuristic applied to processing operations builds schedules with the worst case performance ratio not greater than* $\frac{7}{3} - \frac{1}{3m}$.

Proof. Explanation of the above observation is the following: Build a schedule for processing operations only using LPT heuristic. Before processing operations, processors read all data in an arbitrary order. After processing operations, processors write results in an arbitrary order. Reading and writing operations are always executed sequentially, and therefore may not last longer than OPT. Processing operations may not last longer than $(\frac{4}{3} - \frac{1}{3m})Q$, where Q is the length of the shortest schedule for the processing operations considered alone (Graham, 1969), (Błażewicz *et al.*, 1996). Since $Q \leq OPT$, a schedule built in the above way is not longer than $OPT(\frac{7}{3} - \frac{1}{3m})$. □

Special case solvable in polynomial time

In this section we consider UCT non-preallocated tasks. Unfortunately, the case of computational complexity for UCT preallocated tasks remains open.

Theorem 9 *The case of UCT non-preallocated tasks with identical processing operations can be solved in polynomial time.*

Proof. Let k denote the duration of the identical processing operations, i.e. $p_j = k$ for $j = 1, \ldots, n$. We will examine four different cases separately.

Case $k \leq m - 1$. An example of the optimal schedule is depicted in Fig.7. The server performs repetitively $\lceil k \rceil + 1$ sending, and $\lceil k \rceil + 1$ receiving operations. This pattern should be changed when $n \bmod (\lceil k \rceil + 1) \neq 0$. In such a case, the server should receive and send $n \bmod (\lceil k \rceil + 1)$ times, and then receive $\lceil k \rceil + 1$ times at the end of the schedule. Note, that the

Insert Fig.7 here

server is saturated and the schedule cannot be made shorter. One buffer is enough to implement the schedule, and only $\lceil k \rceil + 1$ processors are required to process the tasks. Thus, all processors take part in computations when $m - 2 < k \leq m - 1$.

Case $m - 1 < k$, $buffers = 1$

In this case repetitive pattern of reading from the server and writing results back is obligatory for the processors. Therefore, the server should first send to all m processors. Then, for each processor with ready results the server should perform a pair of operations: receiving from the processor and sending to the processor (cf. Fig.8). The schedule ends with a series of writing operations. To verify that such schedules are optimal consider schedule length from the viewpoint of the server. The schedule must start with the first reading operation and must finish with the last writing operation. Between these two other communication operation are executed. In all other time intervals the server is idle. The idle time on the server is unavoidable in three cases:

Insert Fig.8 here

1. Until the first writing operation server can send to m processors, in the rest of time it is unavoidably idle.
2. Analogously, after the last reading operation the server is idle with the exception of at most m writing operations.
3. When all processors have been already activated the idle time between reading by some processor and writing by the same processor can only be filled by at most $2(m - 1)$ communication operations of other processors.

Note, that our method introduces idle times of the above three types only. For $k \leq 2(m-1)$ the server is continuously busy with the exception of at most $2(k - m + 1)$ units of time at the beginning and at the end of the schedule. For $k > 2(m - 1)$ the server is idle also between other reading and writing operations of the same processor. Yet, the processors are continuously busy.

Case $m-1<k\leq 2m-1$, $buffers \geq 2$.

Processing operations are too short to fill two buffers on all processors before the end of any processing operation. However, it is possible to read to two buffers despite that one of them may become ready for writing operation (cf. Fig.9). Thus, $2m$ series of reading operations fill two buffers on all processors. Before the end of the second series of reading operations all processors have at least one buffer ready for writing. After a series of m writing operations processors have also the second buffer ready for writing. In $4m$ units of time the server works without any idle time. A schedule built in the

Insert Fig.9 here

above way saturates the server. Hence, it is optimal if $n \bmod 2m = 0$. When $n \bmod 2m \neq 0$ one should change the final series of reading/writing operations analogously to case $k \leq m-1$. Namely, after repeating $\lfloor \frac{n}{2m} \rfloor - 1$ times $2m$ series of reading and then $2m$ series of writing operations, the last cycles should consist of $2m$ read operations, $n \bmod 2m$ write operations, $n \bmod 2m$ read operations, and finally $2m$ write operations (cf. Fig.9). Without losing optimality, all processors may start processing operations in discrete moments of time.

Case $k > 2m-1$, $buffers \geq 2$.

When $k > 2m-1$, it is possible to fill two buffers on each processor before the first writing operation. After completion of processing operations storing the results on the server and refilling the buffers with new data is possible without idle time on the processors. Thus, two series of m reading operations of consecutive processors start the communication. Then series of m writing and m reading operations follow. Finally, two series of m writing operations complete the communications. Note, that all processors work without breaks between processing operations and the schedule cannot be made shorter. Two buffers are sufficient in this case.

The schedules can be executed in $O(n)$ time. However, the pattern of the schedule can be identified in shorter time. Observe that all the schedules in this case are determined by two factors: Relation of k to m, and the number of full repetitions activating tasks on all processors with some additional "odd" tasks activating some processors only (e.g. in the case of $k \leq m-1$ the number of "odd" tasks is $n \bmod (\lceil k \rceil + 1)$). The relation of k and m can be identified in $O(\max\{\log m, \log k\})$ time. The existence of the "odd" tasks can be verified in $O(\log n + \log m)$ time. Thus, the optimal schedule can be identified and executed in time polynomial in the size of the input. □

Corollary 1 *The case of UCT non-preallocated tasks with processing operations not longer than* $m-1$ *where* $m>1$ *can be solved in polynomial time.*

Proof. From the discussion of case $k \leq m-1$ in Theorem 9 we conclude that when processing times are not longer than $m-1$, a schedule can be built in which the server is saturated. Therefore, the same method can be applied. □

Corollary 2 *The optimal schedule on one processor can be found in polynomial time for UCT when either* $p_j \leq 1$ *for* $j = 1, \ldots, n$ *or* $p_j > 1$ *for*

$j = 1, \ldots, n$.

Proof. It is a consequence of cases $m - 1 < k \leq 2m - 1, buffers \geq 2$ and $k > 2m - 1, buffers \geq 2$ in Theorem 9 and of Observation 1 which, in particular, deals with $buffers = 1$ case. □

Conclusions

In this work we considered the problem of deterministic scheduling applications running on separate processors but requiring access to the server for initial reading the data and final storing the results. We proved that the problem is **NP**-hard in the strong sense when there is only one buffer at the processors. It is also strongly **NP**-hard when the number of buffers is unlimited. We have shown that any list scheduling algorithm builds schedules with the length not worse than twice the optimal length in the preallocated case and three times the optimal length in the non-preallocated case. Special cases solvable in polynomial time have been identified. Yet, some challenging problems remain open: Do polynomial algorithms exist for UCT tasks with arbitrary processing times, $m = 1$ with finite or infinite number of buffers? Is the UCT case computationally hard for preallocated tasks? Further research may include, considering other optimality criteria, other server and interconnection topology characteristics.

References

Adler, R.M. (1995) Distributed Coordination Models for Client/Server Computing. *IEEE Computer* **28**, 14-22.

Aref, M.M. and Tayyib M.A. (1998) Lana-Match algorithm: a parallel version of the Rete-Match algorithm. *Parallel Computing* **24**, 763-775.

Błażewicz,J., Dell'Olmo, P., Drozdowski, M. and Speranza, M.G. (1992) Scheduling multiprocessor tasks on three dedicated processors. *Information Processing Letters* **41**, 275-280. Corrigendum: (1994) *Information Processing Letters* **49**, 269-270.

Błażewicz, J., Drabowski, M. and Węglarz, J. (1986) Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers* **C-35**, 389-393.

Błażewicz, J., Ecker, K., Pesch, E., Schmidt, G. and Węglarz, J. (1996) *Scheduling Computer and Manufacturing Processes*, Heidelberg: Springer.

Chen B. (1993) Analysis of a heuristic for scheduling two-stage flow shop with parallel machines, Econometric Institute, Erasmus Univ. Rotterdam, Report 9338/A.

Coffman Jr., E.G., Garey, M.R., Johnson, D.S. and LaPaugh, A.S. (1985) Scheduling file transfers, *SIAM Journal on Computing* **3**, 744-780.

Deux, O. *et al.* (1991) The O_2 system. *Communications of the ACM* **34**, 34-48.

Drozdowski, M. (1996) Scheduling multiprocessor tasks - An overview, *European Journal of Operational Research* **94**, 215-230.

Du, J. and Leung, J.Y-T. (1989) Complexity of scheduling parallel task systems, *SIAM J. Discrete Mathematics* **2**, 473-487.

Fleischmann, A. (1994) *Distributed Systems: Software Design & Implementation*. Springer-Verlag, Heidelberg.

Garcia, B.-L., Potvin, J.-Y. and Rosseau, J.-M. (1994) A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints. *Computers and Operations Research* **21**, 1025-1033.

Garey, M.R., and Johnson, D.S. (1978) Strong NP-completeness results: motivation, examples and implications, *J. Assoc. Comput. Mach.* **25**, 499-508.

Gendron, B. and Crainic, T.G. (1994) Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research* **42**, 1042-1066.

Graham,R.L. (1969) Bounds on multiprocessing anomalies. *SIAM J. Appl. Math.* **17**, 263-269.

Gonzales, T. and Sahni, S., (1978) Flowshop and jobshop schedules: Complexity and approximation, *Operations Research* **26**, 36-52.

Hall, N.G., Potts, C.N. and Sriskandarajah, C. (1996) Parallel machine scheduling with a common server, *PMS'96 The Fifth International Workshop on Project Management and Scheduling, Abstracts*, April 11-13, Poznań, Poland, pp. 102-106.

Hoogeveen, J.A., Lenstra, J.K. and Veltman, B. (1996) Preemptive scheduling in a two-stage multiprocessor flow shop is NP-hard. *European Journal of Operational Research* **89**, 172-175.

Jean-Marie, A., Lefebvre-Barbaroux, S. and Liu, Z. (1998) An analytical approach to the performance evaluation of master-slave computational models. *Parallel Computing* **24**, 841-862.

Karp, R.M. (1972) Reducibility among combinatorial problems. In *Complexity of computer computation*, eds. R.E. Miller and J.W. Thatcher (pp.85-104) New York:Plenum Press.

Kubiak, W., Lou, S.X.C. and Wang,Y. (1996) Mean flow time minimization in reentrant job-shops with a hub, *Operations Research* **44**, 764-776.

Kubale, M. (1987) The complexity of scheduling independent two-processor tasks on dedicated processors, *Information Processing Letters* **24**, 141-147.

Lane, E.L., and Sidney J.B. (1993) Batching and scheduling in FMS hubs: flow time considerations, *Operations Research* **41**, 1091-1103.

Mitra, G., Hai, I. and Hajian, M.T. (1997) A distributed processing algorithm for solving integer programs using a cluster of workstations. *Parallel Computing* **23**, 733-753.

Porto, S. and Ribeiro, C. (1995) Parallel Tabu Search Message-Passing Synchronous Startegies for Task Scheduling Under Precedence Constraints. *Journal of Heuristics* **1**, 207-223.

Rinnoy Kan, A.H.G. (1976) *Machine scheduling problems: Classification, complexity and computations*, The Hague:Martinus Nijhoff.

Tanenbaum, A.S. (1989) *Computer Networks.* Prentice-Hall International, Englewood Cliffs.

Veltman, B., Lageweg, B.J. and Lenstra, J.K. (1990) Multiprocessor scheduling with communications delays, *Parallel Computing* **16**, 173-182.

Wang, Y., Sethi, S.P., van de Velde, S.L. (1997) Minimizing makespan in a class of reentrant shops, *Operations Research* **45**, 702-712.

List of figures

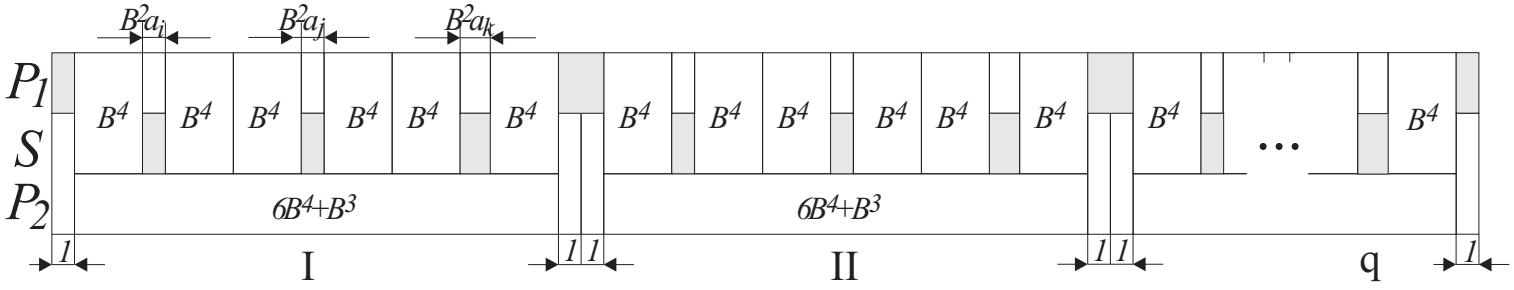

Fig.1

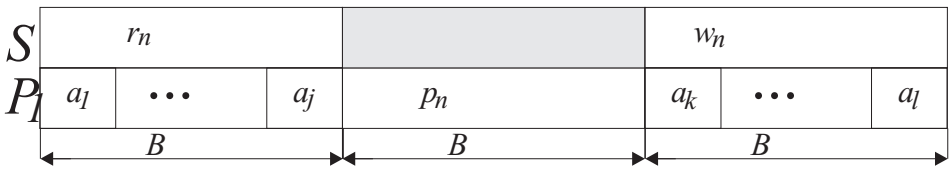

Fig.2

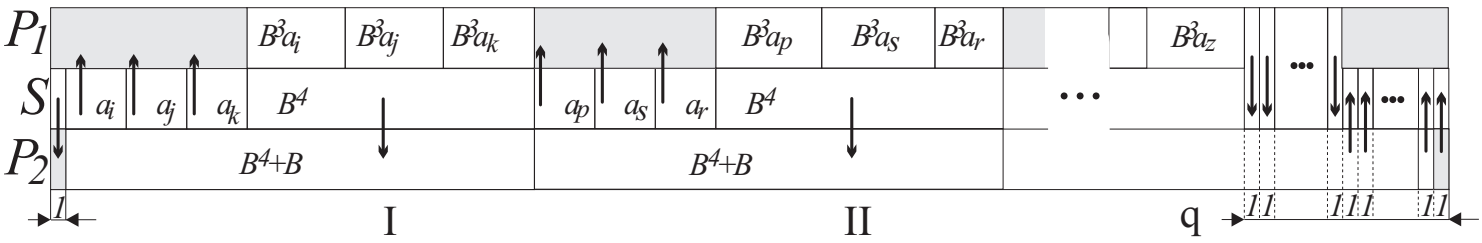
P1
S
P2
B3ai
B3aj
B3ak
B3ap
B3aS
B3ar
B3az
ai
aj
ak
ap
aS
ar
B4
B4+B
1
I
II
q

Fig.3

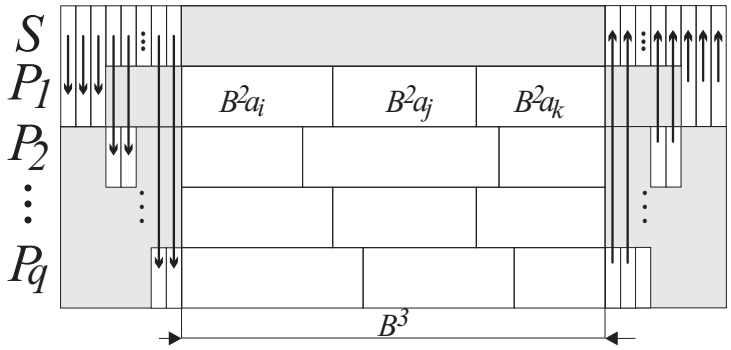

Fig.4

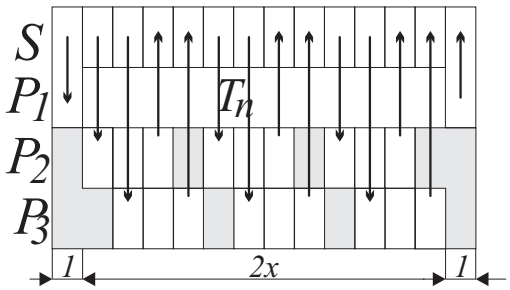

Fig.5a

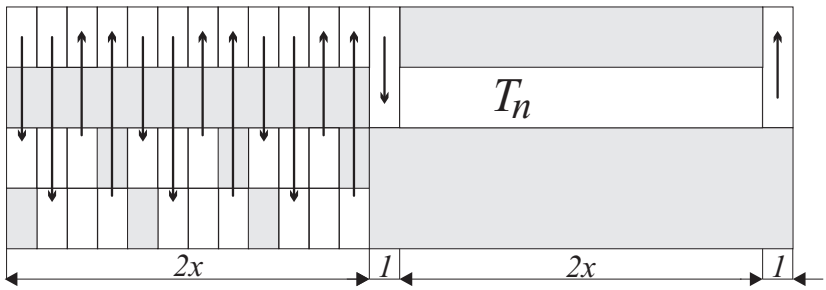

Fig.5b

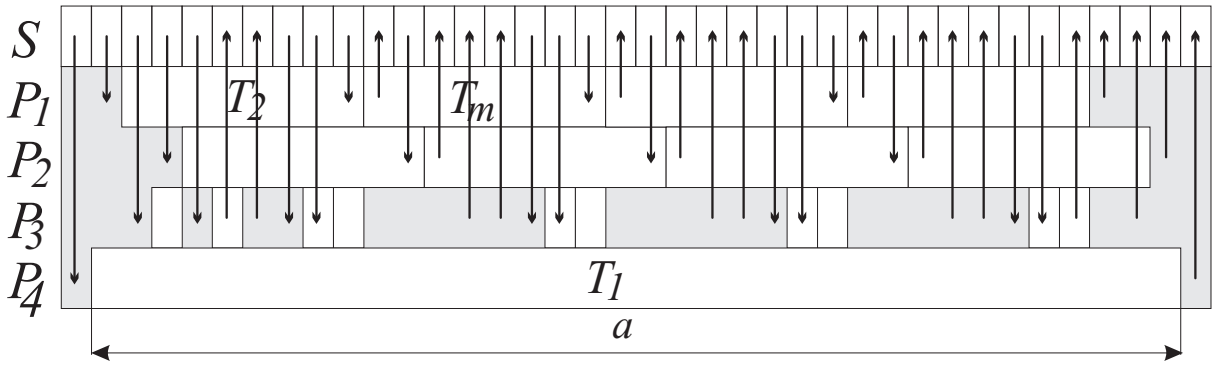

Fig.6a

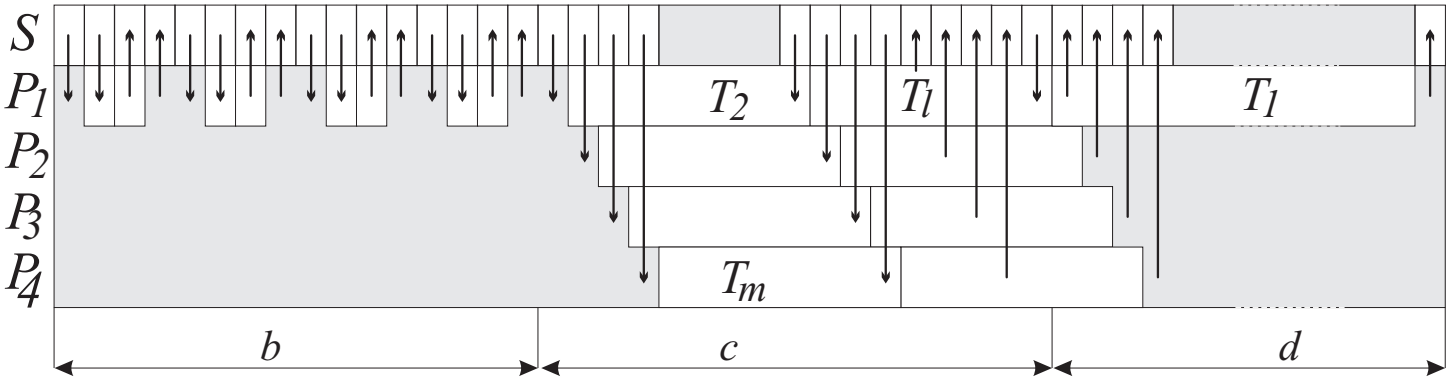

Fig.6b

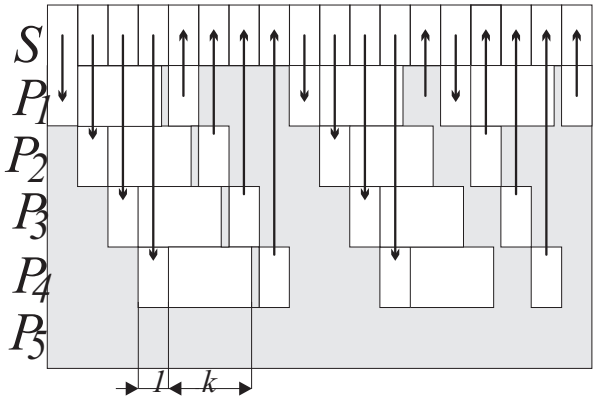

Fig.7

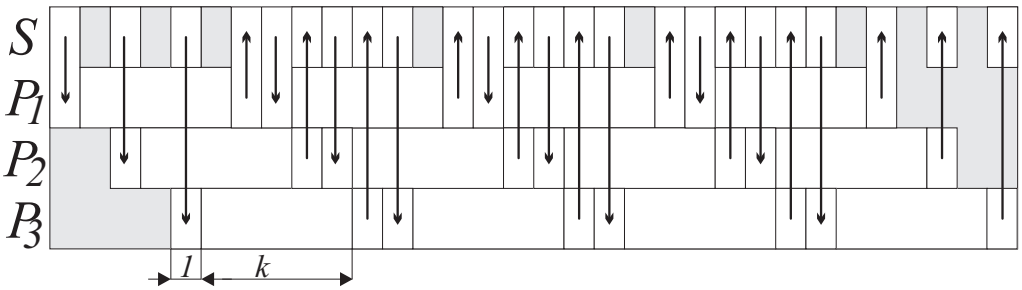

Fig.8

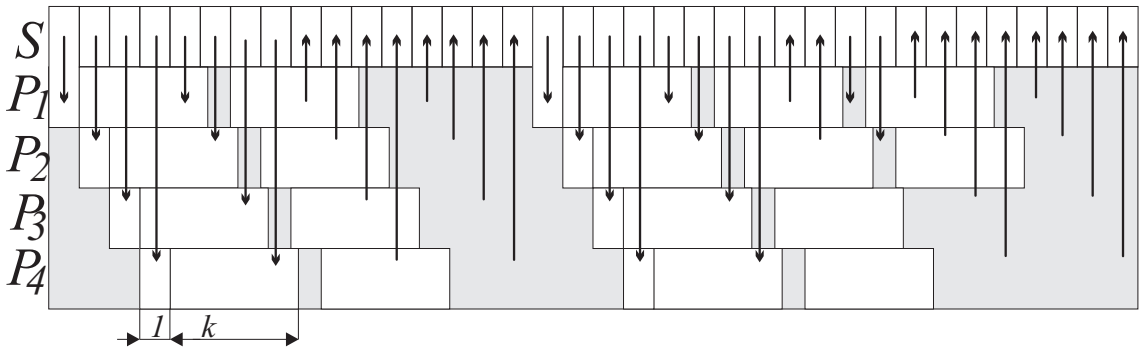

Fig.9