Scheduling multiprocessor tasks on hypercubes

Maciej Drozdowski*

Abstract

The problem to be addressed here is one of scheduling multiprocessor tasks, i.e. tasks which may require more than one processor at a time. This model is quite natural for many parallel applications. It is, however, different than standard approach in the scheduling theory. In tis paper, a low order polynomial-time preemptive scheduling algorithm is proposed for tasks scheduled on uniform processors, when schedule length is the performance measure. The results of computational experiments over the algorithm are reported.

Keywords: Deterministic scheduling, multiprocessor task systems, uniform processors, preemptive schedule, scheduling on a hypercube, complexity analysis.

1 Introduction

Classical scheduling models assume that each task requires for its processing one processor at a time [3, 6, 8]. It turns out, however, that in systems of microprocessors one often has tasks requiring several processors simultaneously (e.g. [4, 19, 20] and lots of others). It is for instance the case of self-testing multi-microprocessor systems in which one processor is used to test others or in fault detection-systems in which test signals stimulate the elements to be tested; then outputs are analyzed simultaneously [2, 9]. New parallel algorithms and corresponding future task systems create another domain of application for this kind of scheduling [13, 17]. It is not difficult to give examples of the computational problems from mathematics, physics, electronics and computer graphics (e.g. computations on matrices) which can be easily divided into subproblems solvable "almost" independently in parallel. This means that copies of the program solving the problem must communicate

*Instytut Informatyki Politechniki Poznańskiej, Poznań, Poland. The research has been partially supported by Grant of the State committee for Scientific Research (KBN 3P40600106).

from time to time. No matter what kind of communication medium is used, the full advantage of parallelism can only be taken if the copies are running in parallel in the real time [12, 1]. Otherwise, one running module of the program may wait for communication with a module which is temporarily swapped out from the processor. In such a situation the speed of execution depends mainly on the work of the scheduling algorithm swapping tasks on processors [13, 12]. Hence, it seems desirable to run in the real time copies of the program requiring more than one processor simultaneously.

In order to model task sets for the above applications, one can divide the set of tasks $\mathcal{T}$ into subsets $T^1, T^2, \ldots, T^k$ with $\mid T^i \mid = n_i, (i = 1, \ldots, k)$ and $n_1 + n_2 +, \ldots, + n_k = n$. Each task T_i^j requires exactly j arbitrary processors simultaneously during a prespecified period t_i^j of its processing. We assume that tasks are independent and each processor can be assigned only one task at a time. The objective is to find the shortest feasible schedule. This model is useful both from theoretical and from practical point of view. For one thing, it is often the case that it is difficult to change the number of processors used by a task. This reflects, for example, the level of a scheduler in the operating system. What is more, it enables deriving complexity results valid also for more sophisticated models.

Our definition of task system follows [4]. A different definition of the task system including the dependence of a processing time on a number of processors executing some particular task was given in [11] (the so-called Parallel Task System - PTS). These two models are closely related and dependent on each other. The model we consider here (so-called Multiprocessor Task System - MTS) is a special case of PTS. NP-hardness results obtained for MTS are automatically valid for PTS. In particular, in [11] it has been shown that for nonpreemptive scheduling and for precedence constraints consisting of chains MTS problem is strongly **NP**-hard. For independent tasks and nonpreemptive scheduling the problem is strongly **NP**-hard for five processors ([11]). Polynomial-time algorithms for some special cases of MTS are also known ([4]).

For the preemptive MTS and identical processors already some results have been obtained, too. For independent multiprocessor task systems with m fixed (m is the number of processors) the problem can be solved in polynomial time using a linear programming formulation [4, 5]. While in general, this problem is **NP**-hard [10]. If there are T^1 -tasks and T^k -tasks in the system only, the optimal schedule can be constructed in O(n) time [4]. A special case of the preemptive MTS is scheduling on a hypercube of processors

[7, 18]. An $O(m^2n^2)$ algorithm was proposed in [18] to schedule preemptively tasks requiring a number of processors which is a power of two.

In this paper, we extend the MTS model by considering a uniform k-processor system. A system of uniform processors can be a model for a computer system consisting of heterogenic processors or a system in which some processors have to do additional work "in the background" (e.g. passing a massage in the node-to-node communication network). A k-processor system consists of disjoint k-tuples of processors. All k processors in the same k-tuple have the same speed $s_{ik+1} =, \dots, = s_{(i+1)k}$ $i = 0, 1, \dots, m/k - 1$. This assumption is justified in practice, because of necessary synchronization of parts of the same task running simultaneously. Thus, the slowest processor speed is the speed of the whole tuple. We give a low order polynomial algorithm for preemptive scheduling of the task sets $T^1, \dots, T^k$, where for every pair $T^j, T^i, j/i \in Z^+$. It is worth mentioning that this covers also the problem of scheduling on the hypercube described in [7, 18], but the methods used here are different.

Throughout this paper we will be denoting processing requirements of T^1-tasks by a vector of standard processing times $\bar{t}^1 = [t_1^1, t_2^1, \dots, t_{n_1}^1]$. Thus, the time needed to process T_j^1 on a processor of speed s_l is $t_j^1/s_l, j = 1, 2, \dots, n$. Similarly, tasks from sets $T^2, \dots, T^k$ are characterized by vectors of standard processing times $\bar{t}^2, \dots, \bar{t}^k$, and by requirements of $2, \dots, k$ processors, respectively, at the same time by any task from the respective set. A real processing time of task T_j^i depends on a processing speed s_l of the i-tuple of processors assigned to the task, and it is equal to t_j^i/s_l. All tasks are assumed to be preemptable, i.e. their processing may be interrupted at any moment and restarted later (perhaps on a different processor) without additional costs. The objective is to find the shortest possible schedule, i.e. one for which $C_{max} = \max_{T_j^i}\{C_j^i\}$ is minimal, where C_j^i is a completion time of task $T_j^i, j = 1, \dots, n_i; i = 1, \dots, k$.

The rest of the paper is organized as follows. Section 2 describes the scheduling algorithm. In Section 3 results of computational experiments are reported.

2 The Algorithm

In this section, the algorithm solving the problem of preemptive scheduling independent multiprocessor tasks on uniform processors will be presented.

This algorithm solves a special case of the problem which has been proven to be **NP**-hard in general ([10]). Restriction that we impose on the general formulation requires that for every pair of task types $T^j, T^i, j/i \in Z^+$. This let us formulate a low-order polynomial time algorithm. Such a restriction is justified from the practical point of view because it covers also the case of scheduling on hypercubes of processors. While scheduling on a hypercube, which has a power of two processors, each task requires a sub-cube consisting of a number of processors which is a power of two, too.

Now, we will describe the algorithm in the rough outline. First, a lower bound on the schedule length is calculated. Then we schedule tasks starting from tasks sets requiring the biggest number of processors and the longest processing time within each task-type, and finishing with uni-processor tasks (T^1) which are the shortest. In the course of scheduling we use a set of rules guaranteeing preserving the biggest possible processing capacity for the remaining tasks. It is possible, however, that a feasible schedule may not exist within a lower bound. In such a case the schedule is lengthened by some calculated time. Each step of the algorithm will be described in the sequel.

We start by calculating a lower bound on the schedule length. Let all tasks in each set be ordered according to the nonincreasing processing times (i.e. $T^i_{j+1} \le T^i_j$ for $i = 1, \ldots, k$ $j = 1, \ldots, n_i - 1$) and the processor set according to nonincreasing processor speeds, respectively. Consider k relaxed versions of the problem:

k) T^k-tasks only and k-processor system consisting of m/k processors.

$\vdots$

l) $T^l-, \ldots, T^j-, \ldots, T^k-$tasks ($l < \ldots < j < \ldots < k$) each of which is treated as one T^l-type task,$\ldots, j/l$ T^l-type tasks, $\ldots, k/l$ T^l-type tasks, respectively, while the processor system consists of l-processors the total number of which is m/l.

$\vdots$

1) $T^1-, \ldots, T^k$-tasks treated as $1, \ldots, k$ uni-processor tasks while the processor set is $P_1, \ldots, P_m$.

The lower bound on the length of the schedule can be calculated for the above relaxed versions of the problem according to the following formulae:

$$C(k) = \max\{\max_{1\le g<m/k} \frac{\sum_{j=1}^{g} t_j^k}{\sum_{j=1}^{g} s_{kj}}, \frac{\sum_{j=1}^{n_k} t_j^k}{\sum_{j=1}^{m/k} s_{kj}}\};$$

$$\vdots$$

$$C(l) = \max\{\max_{1\le g<m/l} \frac{\sum_{j=1}^{g} t_j^l}{\sum_{j=1}^{g} s_{lj}}, \frac{\sum_{j=1}^{n_l+\ldots+(k/l)n_k} t_j^k}{\sum_{j=1}^{m/l} s_{lj}}\};$$

$$\vdots$$

$$C(1) = \max\{\max_{1\le g<m} \frac{\sum_{j=1}^{g} t_j^1}{\sum_{j=1}^{g} s_j}, \frac{\sum_{j=1}^{n_1+\ldots+kn_k} t_j^k}{\sum_{j=1}^{m} s_j}\}.$$

Note, that each of the above formulae denotes that the longest task must fit in the fastest processor, the two longest tasks must fit in the two fastest processors etc. This follows standard approach for tasks requiring only one processor at the time [14, 16]. Clearly, $C = \max\{C(1), \ldots, C(k)\}$ is a lower bound on the schedule length. Let us denote by $PC_i = s_i C$ a *processing capacity* of processor P_i.

The algorithm will use three rules (cf. [16]) which will be given below. It will be applied first to schedule T^k-tasks on m/k k-processors, then to schedule T^p-tasks on m/p p-processors (where $p = \max\{l : l < k \text{ and } T^l \in \mathcal{T}\}$), and so forth until the uni-processor tasks. Within a certain task set, rules 1,2, and 3 will be applied to the tasks scheduled according to the order of nonincreasing standard processing times. At each stage of the algorithm, the processors will be ordered according to nonincreasing values of PC_i (note, that initially this order coincides with the order of nonincreasing processing speeds).

For the sake of simplicity of defining the three rules we will ignore actual number of required processors and denote the task by T_j and its processing time by t_j. Suppose we have to schedule T_j and we are considering the last P_l for which $PC_l \ge t_j$.

If $t_j = PC_l$ then apply Rule 1:

Rule 1: Schedule task T_j on the processor P_l in such a way that the interval [0,C] is completely filled with T_j. Set PC:=0 and renumber the processor set according to nonincreasing processing capacities.

If $PC_l > t_j > PC_{l+1}$ then apply Rule 2:

Rule 2: Calculate the time u such that T_j is completely processed in the intervals [0,u] on processor P_l and $[u, C]$ on processor P_{l+1}, respectively. Combine processors P_l and P_{l+1} to a *composite processor* P_l with $PC_l := PC_l + PC_{l+1} - t_j$. Set PC_{l+1}:=0 and renumber the processor set according to nonincreasing processing capacities.

When rules 1 and 2 can no longer be applied, then we are necessarily in one of the following cases:

a) $t_j < PC_l$ with either $l = m$ or $PC_{l+1} = \ldots = PC_m = 0$ (i.e., the processing requirements of T_j will not entirely fill the smallest positive remaining capacity of a single processor);

b) $PC_l > t_j > PC_{l+1}$ and no u (as in Rule 2) can be found. This case can occur only if rule 3 has already been applied: processors are then loaded in some time intervals in [0,C].

Then we apply the following:

Rule 3: Schedule task T_j and the remaining tasks in any order in the remaining free processing intervals from left to right starting with processor P_l and use processor P_i, $i < l$, only if P_{i+1} is completely filled.

As $C \geq C(k)$ we know that a feasible schedule for the set of T^k-tasks must exist. It remains to show that also $T^p, \ldots, T^1$-tasks can be scheduled in the remaining processing intervals and if not, that no feasible schedule for the given problem instance with schedule length C will exist.

From the calculation of C we know that there is enough processing capacity in the interval [0,C] to schedule all the tasks on the given set of processors. In case of infeasibility it might happen that the length of some task will prevent the construction of a feasible schedule. To check this, we calculate the processing capacities in the interval [0,C] for the processor system remaining after scheduling the set of $T^k, \ldots, T^f$-tasks. Let PC_i^f be the remaining processing capacity of an original or composite e-processor P_i in the interval [0,C] after the scheduling of all tasks from sets $T^k, \ldots, T^f$. Remember that these processors are ordered according to nonincreasing *remaining* processing capacities. Let T^e be the next task type to be scheduled (after T^f).

From [16] it is known that a feasible schedule for the set of T^e-tasks exists if and only if

$$\sum_{i=1}^{g} PC_i^f \geq \sum_{i=1}^{g} t_i^e \text{ for } g = 1, \ldots, m/e - 1 \text{ and } \sum_{i=1}^{m/e} PC_i^f \geq \sum_{i=1}^{n_e} t_i^e \qquad (1)$$

and that we can construct it by applying rules 1-3 to the set of T^e-tasks using the processor system resulting from the assignment of the $T^k, \ldots, T^f$-tasks.

Now, assume that no feasible schedule can be found in this way. First, we will show that no other assignment of the set of $T^k, \ldots, T^f$-tasks than the one generated by rules 1-3 can achieve feasibility for the set of T^e-tasks. Let pc_i^f be the remaining processing capacity of e-processor P_i after any feasible assignment of tasks from sets $T^k, \ldots, T^f$.

Theorem 1 *Using rules 1-3 we can always guarantee that* $\sum_{i=1}^{q} PC_i^f \geq \sum_{i=1}^{q} pc_i^f$ for $q = 1, \ldots, m/e$ and $f = k, \ldots, 1$.

Proof. Using the above rules we schedule one by one the tasks from set T^k. Having selected the first task T_j^k, assume we are using rules 1 or 2. Let l be the index such that $PC_l > t_j^k \geq PC_{l+1}$. The composite processor has a remaining processing capacity which satisfies $PC_{l+2} \leq PC_l + PC_{l+1} - t_j^k \leq PC_{l-1}$ and no reordering of the processors is necessary.

On the other hand, if we combine P_i and P_q $(i < q)$ we will have $PC_i + PC_q - t_j^k < PC_i$ since $PC_q < t_j^k$. Let r be the new index of the composite processor after reordering the processor set. We will have $\sum_{h=1}^{z} pc_h^k < \sum_{h=1}^{z} PC_h^k$ for $z = i, \ldots, r-1$ (and $r > i$). In general, P_i or P_q could be any feasible composition of processors other than P_l and P_{l+1}. Important is that some PC_i has been used unnecessarily. For rule 3, the conclusion is immediate.

After scheduling T_j^k we have the problem to schedule $n_k - 1$ T^k-tasks on $m-1$ processors (T_j^k was scheduled by applying rules 1 or 2) or on m processors (T_j^k was scheduled by applying Rule 3). For the next T^k-task to be scheduled the same argument applies. Induction over the number of tasks proves that the theorem holds while scheduling tasks of one type (i.e. T^k). This means that after scheduling T^k-tasks the remaining processor system has the biggest possible processing capacity.

The same arguments can be applied to the following types of tasks. Hence, induction (the number of required processors is decreasing in the subsequent task types) over the task types proves correctness of the theorem. $\square$

From Theorem 1 we know that if the schedule is not feasible, then infeasibility could not have been avoided and the extending of the schedule is inevitable. Infeasibility is a result of the fact that even though the processor system contains sufficient total processing capacity but it is spread over several processors rather than accumulated on one processor. In this case inequalities (1) are not satisfied for l tasks $T_j^e, \ldots, T_{j+l-1}^e$ $(l > 1)$. For the sake of notation we denote these tasks $T_1^{e*}, \ldots, T_l^{e*}$. Now, we introduce a new term - *dead processing capacity* which is just the amount of processing time by which inequalities (1) are violated. Dead processing capacities for the tasks which cannot be scheduled feasibly can be calculated from the equations:

$$DP_i^e = \sum_{h=1}^{j+i-1} t_h^e - \sum_{h=1}^{j+i-1} PC_h^e + \sum_{h=1}^{i-1} DP_h^e \tag{2}$$

Note, that it is not necessary the case that $T_1^{e*}, \ldots, T_l^{e*}$ are consecutive tasks of T^e, i.e. there can be additional tasks between T_1^{e*} and T_l^{e*} which could be scheduled feasibly (for which (1) holds). In this case the following discussion is also valid, but (2) should be appropriately modified (by changing range of summation) to reflect this situation.

Now, a key question is by what amount of time the schedule should be extended. A feasible schedule will exist if and only if inequalities (1) are satisfied. In such an extended schedule dead processing capacities will be reduced to zero. After lengthening of the schedule only some processors will create additional processing capacity usable for tasks $T_1^{e*}, \ldots, T_l^{e*}$. Thus, in order to calculate proper length of extension we have to know which processors will take part in creating processing capacity for $T_1^{e*}, \ldots, T_l^{e*}$ after lengthening the schedule. This information can be collected in the course of building a partial schedule (i.e. the schedule for tasks preceding the infeasible ones).

First, let us describe the process of accumulating processing capacity during scheduling of the tasks according to the above three rules. When a task is scheduled according to Rule 2, the remaining capacity of a composite processor is passed to subsequent tasks. Thus, the lengthening of a schedule causes that the increase of processing capacity of all the processors forming a composite processor will be passed to the subsequent tasks. Only if after lengthening of the schedule the task occupies the whole f-processor (not composite, but using original numbering) will not the above scheme work. In this case, Rule 1 is applied and the processing capacity of a particular f-processor disappears from the set of processing capacities. Hence, the increase of the processing capacity of this f-processor is not passed to the subsequent tasks.

Here, we will describe operations on a special data structure devised for storing information which processors will take part in creating processing capacity after lengthening the schedule. We start with a vector $[1, \ldots, m]$ of indices of processors divided into m/k sub-vectors (sub-lists) containing numbers of real processors forming k-processors.

Any time we schedule task T_j^e according to Rule 1 on e-processor P_l we have to join sub-vectors (lists) corresponding to P_l and P_{l+1}. We do it in such

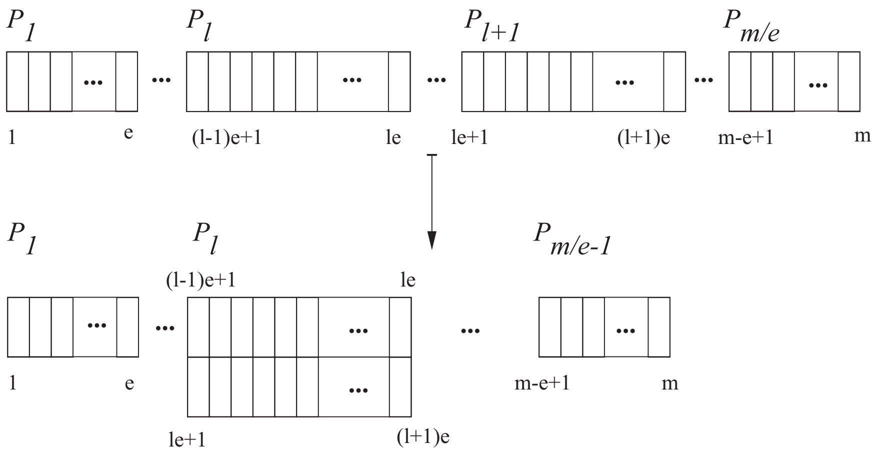

Figure 1: Operation on the processor indices' data structure for Rule 2.

a way that real processors form pairs: $(P_{(l-1)e+1}, P_{le+1}), (P_{(l-1)e+2}, P_{le+2}), \ldots, (P_{le}, P_{(l+1)e})$ (cf.Fig. 1) This can be done in a constant time if we operate on lists (and/or each pair can be represented as a set). After such an operation data structure with indices of real processors corresponding to e-processor P_l have two lists in parallel. This action is done due to the fact that after eventual lengthening of the schedule T_j^e will be scheduled according to Rule 2. If P_{l+1} does not exist (P_l is the last e-processor) then we do not change anything in the list of indices.

If Rule 2 is used to schedule any task we perform the same action as for Rule 1 because it is equivalent to forming composite e-processor P_l.

When Rule 3 is applied then there are two cases in which we have two different operations to perform:

a) $t_j^e < PC_l^f$ with either $l = m/e$ or $PC_{l+1}^f = \ldots = PC_m^f = 0$.

If P_{l+1} does not exist (P_l is the last e-processor and $l = m/e$) we change nothing in the processor indices data structure because no new composite processor is formed. Otherwise, we have to join into "parallel" list, lists representing $PC_{l+1}^f = \ldots = PC_m^f$ due to the fact that after eventual lengthening task T_j^e will benefit from additional processing capacity of all slower processors.

b) $PC_{l-1}^f > t_j^e > PC_l^f$ and no u (as in Rule 2) can be found.

T_j^e will be scheduled in the remaining free processing intervals from left

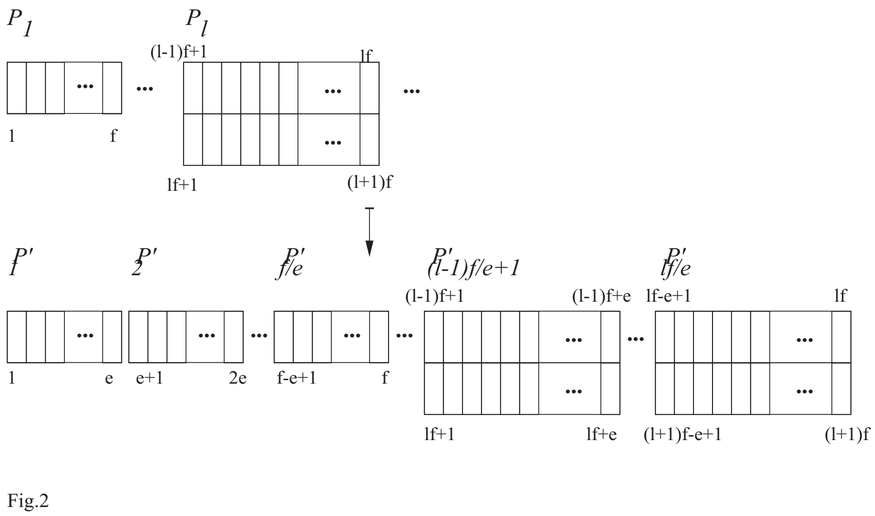

Figure 2: Operation on the processor indices' data structure after T^f is scheduled and before scheduling T^e.

to right starting with e-processor P_l and ending on e-processor P_{l-1}. In this case we join vectors representing P_l and P_{l-1} as for Rule 1 and Rule 2 because eventual lengthening of the schedule will give some gain of processing capacity from both P_l and P_{l-1} to the task following T_j^e.

When one task set, say T^f, is scheduled and we turn to the next task type, say T^e, we have to modify processor indices data structure (cf. Fig. 2). We split f processor long vectors (lists) into e processor long lists (which number is f/e). We do the same with "parallel" lists created during performing Rule 1 and Rule 2. In a word, we create f/e "parallel" lists of length e. Each e processor long list represents real processors that have contribution in the capacity of the particular e-processor. "Parallel" list means more than one real processor has contributed in processing capacity of e-processor.

Let us denote by A_i^e the set of processors indicated on the first position (positions) of the list (possibly "parallel") associated with e-processor P_i, right before any T^e-task is scheduled. In this moment we check (1) and calculate (2) (if necessary).

Theorem 2 *If there is no feasible schedule for the set of T^e-tasks, i.e.*

$DP_i^e > 0$, *then we have to lengthen our schedule by at least*

$$\delta^e = \max_{1 \le g \le l} \{ \frac{\sum_{i=1}^{g} DP_i^e}{\sum_{P_i \in \cup_{h=1}^{j+g-1} A_h^e} s_i} \} \tag{3}$$

Proof. Consider any schedule of $T^k, \ldots, T^f$ tasks which gives certain dead processing capacities DP_i^e. According to Theorem 1 there exists a schedule constructed by the rules 1,2,3 for which dead processing capacity is not larger than DP_i^e. Let ε be the minimum amount of time by which we have to lengthen the schedule to make it feasible. After lengthening the schedule processing capacity of each original processor P_i is $PC_i' = PC_i + s_i\varepsilon$. Thus, processing capacity of e-processor P_j assigned as before lengthening to process T_1^{e*} is now

$$(PC_j^f)' = PC_j^f + \varepsilon \sum_{P_i \in \cup_{h=1}^{j} A_h^e} s_i \tag{4}$$

where A_h^e is a set of processors contributing in processing capacity of e-processor P_h. In this equation we assume that all e-processors preceding j will participate in creating processing capacity for infeasible tasks as it is in Rule 2. We have to guarantee that $\sum_{h=1}^{j+g-1} (PC_h^f)' \ge \sum_{h=1}^{j+g-1} t_h^e$ for $g = 1, \ldots, l$. After substitution of the sum of $(PC_h^f)'$ we have (from 4)

$$\sum_{h=1}^{j+g-1} PC_h^f - \sum_{h=1}^{j+g-1} t_h^e + \varepsilon \sum_{P_i \in \cup_{h=1}^{j+g-1} A_h^e} s_i \ge 0 \text{ for } g = 1, \ldots, l \tag{5}$$

and after substitution of the sums of PC_h^f and t_h^e with DP_h^e we have (from 2)

$$\varepsilon \sum_{P_i \in \cup_{h=1}^{j+g-1} A_h^e} s_i - \sum_{h=1}^{g} DP_h^e \ge 0 \text{ for } g = 1, \ldots, l \tag{6}$$

Thus, $\varepsilon \ge \delta^e = \max_{1 \le g \le l} \{ \sum_{i=1}^{g} DP_i^e / \sum_{P_i \in \cup_{h=1}^{j+g-1} A_h^e} s_i \}$ is the minimal amount of time by which we have to lengthen the schedule. $\square$

After lengthening the schedule we again apply the same procedure to the set of tasks. If the schedule is still infeasible we have to lengthen it once more. This may only happen when the layout of tasks from $T^k, \ldots, T^f$ changes on the original processors preceding the ones on which $T_1^{e*}, \ldots, T_l^{e*}$ fall [5]. This can be a result of the fact that slower processors would have processing capacity big enough to accommodate (as in Rule 1) whole tasks which previously had to be executed partially on faster processors. What is more, this situation may happen at most $O(m^2)$ times during the whole execution of the algorithm because there can be at most m tasks scheduled according to Rule 1 and (1) can be violated at most $2m-1$ times.

Theorem 3 *After lengthening of the schedule by δ^e calculated according to Theorem 2, the new schedule is feasible for $T_1^{e*}, \ldots, T_l^{e*}$ when the real sets of original processors participating in processing capacity of the e-processors on which $T_1^{e*}, \ldots, T_l^{*e}$ fall are not proper subsets of $\cup_{h=1}^{j+g} A_h^e$ for $g = 1, \ldots, l$.*

Proof. Let us denote by B_h^e the set of the indices of the original processors which participate in processing capacity of e-processor P_h. Note, that the $\cup_{h=1}^{m/e} A_h^e$ can differ from $\cup_{h=1}^{m/e} B_h^e$ because due to applying Rule 1, some processors may disappear from $\cup_{h=1}^{m/e} B_h^e$. This means that it is possible that excess of processing capacity of some processor(s) will not be passed for use by subsequent tasks. The result is that the increase of processing capacity caused on such a processor after lengthening the schedule is not usable for the following tasks. This situation is different than the one we were expecting after the extension and rescheduling. What is more, it is difficult to foresee such cases before rescheduling. Hence, while modifying the processor indices data structure this situation is never expected to happen.

Now, we have to distinguish between four cases:
1) $\cup_{h=1}^{j} A_h^e = \cup_{h=1}^{j} B_h^e$
2) $\cup_{h=1}^{j} A_h^e \supset \cup_{h=1}^{j} B_h^e$
3) $\cup_{h=1}^{j} A_h^e \subset \cup_{h=1}^{j} B_h^e$
4) $\cup_{h=1}^{j} A_h^e \cap \cup_{h=1}^{j} B_h^e = \emptyset$ or $(\cup_{h=1}^{j} A_h^e \not\supset \cup_{h=1}^{j} B_h^e$ and $\cup_{h=1}^{j} A_h^e \not\subset \cup_{h=1}^{j} B_h^e)$.

The first case is exactly what we were expecting in Theorem 2 and the schedule is feasible.

The second case takes place in the situations described in the first paragraph of this proof. It is easy to verify that at least one task will not be scheduled feasibly.

The third case takes place when some task has been expected to fall on processors P_a and P_{a+1}, but was scheduled on P_b and (perhaps) on $P_b + 1$. The schedule is obviously feasible for T_1^{e*} because the schedule is longer and we can schedule all the tasks preceding T_1^{e*}, while for this particular task more processors participate in creating processing capacity. Now, a question arises whether the schedule is still optimal (the shortest possible).

Let us consider some task which after rescheduling is using different processors than expected. Since the schedule is longer the task cannot be scheduled (according to rules 1,2,3) on faster processors, so $b \geq a+1$. The task scheduled according to Rule 1 before the lengthening, is expected in handling the processor indices data structure to be executed according to Rule 2 after the lengthening. When a task is scheduled before lengthening according to Rule 3 the set of indices of original processors is joined until the last e-processor and no change in A_i^e is possible. Thus, we can analyze only the case when a task is scheduled before lengthening according to Rule 2. After lengthening, such a task will leave free for the subsequent tasks e-processor P_a. So its processing capacity will be $PC_a^{f"} = PC_a^f + \delta^e s_a$. On the other hand, the expected remaining capacity on this e-processor was $PC_a^{f'} = PC_a^f + PC_{a+1}^f + \delta^e(s_a + s_{a+1}) - t$, where t is the processing time of the considered task. We can say that $PC_a^{f"} \geq PC_a^f$ because after rescheduling e-processor P_a is completely free and has faster original processors than expected. What is more, $PC_{a+1}^f + \delta^e(s_{a+1}) \geq t$ because the considered task has been feasibly scheduled on slower processors. Thus,

$$0 \leq PC_a^{f"} - PC_a^{f'} = -PC_{a+1}^f - \delta^e s_{a+1} + t \leq 0$$

this is possible only for 0, i.e. processing capacities of the completely free processor and the expected processing capacity are equal. Hence, the schedule is not longer than necessary. Induction over the tasks finishes this part of the proof.

Finally, the last case can be analyzed in the similar way as the third one. The difference is that all the tasks are scheduled on slower processors than expected. $\square$

A new problem to deal with is the infeasibility for consecutive task types. We can handle this in two ways. First: increase schedule length iteratively every time it appears. This method can be formulated in the following algorithm.

ALGORITHM 1

STEP 1: Calculate C, and schedule all T^k-tasks in $[0, C]$ using rules 1-3;
STEP 2: FOR each $T^j \in [T^p, \ldots, T^1]$ DO (* $p = \max\{i : i < k, T^i \in \mathcal{T}$ *)
BEGIN
STEP 3: WHILE no feasible schedule for T^j -tasks exist DO
BEGIN

Calculate δ^j; $C := C + \delta^j$;
STEP 4: FOR each T^i IN $[T^k, \ldots, T^i]$ DO schedule T^i-tasks;
(* rescheduling, $i = \min\{q : q > j, T^q \in \mathcal{T}\}$ *)
END;
STEP 5: schedule T^j;
END;
END.

The complexity of this algorithm has two components: calculating a lower bound in $O(n \log n)$ time, and scheduling and extending of the schedule. The optimal length of the schedule can be found in $O(m^2)$ trials and each rescheduling can take at most $O(n)$ time. Hence, a total time complexity is $O(nm^2 + n \log n)$.

The second method of dealing with infeasibility is more sophisticated and complex. It simulates an assignment of necessary processing capacity to the processor set (in other words, checks (1) in advance). This can be done in the following way. Add $\varepsilon \sum_{P_i \in A_h^e} s_i$ processing capacity to each e-processors P_h. This will set to 0 the processing capacity of the e-processor for which δ^e was chosen as maximum in (3) after scheduling $T_1^{e*}, \ldots, T_l^{e*}$. If again the schedule is infeasible, calculate new δ^e, add it to the previously calculated one and repeat the above procedure. Now we will formulate our scheduling algorithm for the second method of handling infeasibility.

ALGORITHM 2

PROCEDURE SIMULATE_ASSIGNMENT(T^j:task set);
BEGIN
IF inequality (1) holds for T^j THEN
SIMULATE_ASSIGNMENT(T^i)
(* $i = \max\{q : q < j, T^q \in \mathcal{T}$ *)
ELSE
BEGIN
Calculate δ^j; $C := C + \delta^j$;
Increase processing capacity to the processor set;
SIMULATE_ASSIGNMENT(T^i)
END
END; (* of SIMULATE_ASSIGNMENT *)

(* main body of the algorithm *)
STEP 1: Calculate C, and schedule all T^k-tasks in $[0, C]$ using rules 1-3;

STEP 2: FOR each $T^j \in [T^p, \ldots, T^1]$ DO (* $p = \max\{i : i < k, T^i \in \mathcal{T}$ *)
BEGIN
STEP 3: WHILE no feasible schedule for T^j -tasks exist DO
BEGIN
SIMULATE_ASSIGNMENT(T^j); (* lengthen the schedule *)
STEP 4: FOR each T^i IN $[T^k, \ldots, T^i]$ DO schedule T^i-tasks;
(* where $i = \min\{q : q > j, T^q \in \mathcal{T}$ *)
(* rescheduling and updating processor indices data structure *)
END;
STEP 5: schedule T^j;
END;
END.

Calculating the lower bound needs $O(nlogn)$ time. The application of the rules 1-3 has time complexity $O(n)$. The loop of the algorithm (STEP 2) will be carried out $O(m)$ times (at most $O(m)$ task types). Since the situation that the schedule have to be extended can take place at most $O(m^2)$ times, STEP 3 will be repeated at most $O(m^2)$ times over all task sets (all $T^p, \ldots, T^1$ from STEP 2). Procedure SIMULATE_ASSIGNMENT will be executed recursively into the depth at most $O(m)$ and calculation of δ^j takes $O(m)$ time at most. So, we have a total time complexity of $O(nm^4 + nlogn)$ to solve our problem and generate an optimal schedule. As it can be seen the complexity of ALGORITHM 2 is bigger than the complexity of ALGORITHM 1, so one may ask what is the advantage of using a more complex algorithm. In practice it appears, that the worst case complexity is difficult to achieve. What is more, it is possible to calculate the optimal length of the extension in one recursive sequence of SIMULATE_ASSIGNMENT calls without cumbersome rescheduling. Let us give now an example of such situation.

EXAMPLE.
$T^1 = \{T_1^1\}, T^2 = \{T_1^2\}, T^4 = \{T_1^4\}, t_1^1 = 17, t_1^2 = 16, t_1^4 = 15, m = 8, s_1 = s_2 = s_3 = s_4 = 2, s_5 = s_6 = s_7 = s_8 = 1.$

We calculate $C(4) = 7\frac{1}{2}, C(2) = 9\frac{1}{5}, C(1) = 9\frac{10}{11}, C = 9\frac{10}{11}$. Partial schedule for T^4-tasks is given in Fig. 3.

We go to step 2. Data structure with processor indices is $\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}, \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix}$. There remains (on duoprocessors) $PC_1^4 = PC_2^4 = 14\frac{8}{11}$ processing capacity from scheduling T^4. It is not sufficient to schedule T^2, (1) does not

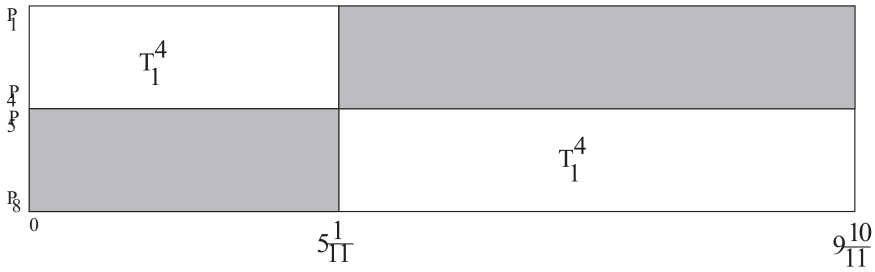

Fig. 3

Figure 3: EXAMPLE - a partial schedule.

hold - go to SIMULATE_ASSIGNMENT(T^2). $DP_1^2 = 1\frac{3}{11}$, $A_1^2 = \begin{bmatrix} 1 \\ 5 \end{bmatrix}$, $\delta = \frac{14}{33}$, $C = 10\frac{1}{3}$. New (increased) processing capacity is $PC_1 = PC_2 = 0, PC_3 = PC_4 = 16$. T_1^2 can be scheduled according to Rule 1, indices data structure is not changed because T_1^2 was executed on the last duo-processor. Go to SIMULATE_ASSIGNMENT(T^1). (1) does not hold, $DP_1^1 = 1, A_1^1 = \begin{bmatrix} 1 \\ 5 \end{bmatrix}, \delta = \frac{1}{3}, C = 10\frac{2}{3}$. STEP 4 and STEP 5 - T^4 and T^2 are rescheduled. Again STEP 2, (1) holds for T^1, then execute STEP 5.

Schedule is feasible and presented in Fig. 4.

3 Results of experiments

The algorithm presented in Section 2 (ALGORITHM 1) has been implemented and tested in the series of simulations. The goal of simulations was to verify the behavior of the algorithm for a wide range of data instances. Hence, the measurement of the processing time and the number of iterations was the main objective. Experimental software has been written in Turbo Pascal 6.0 and executed on IBM AT clone under the control of MS-DOS 3.30 operating system.

The results are presented in Fig. 5, 6, and 7. Each point in these diagrams is an average of at least hundred trials. Execution times of tasks were generated randomly from the interval [0,10] with uniform probability

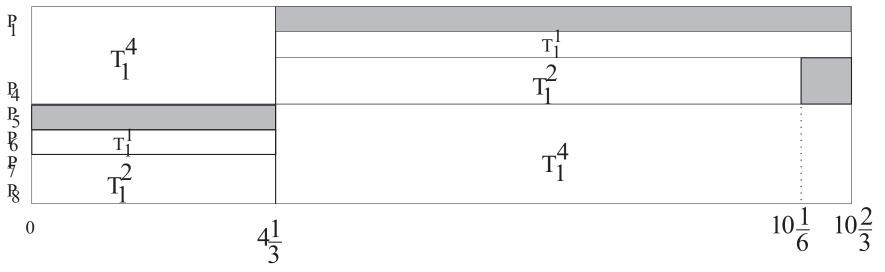

Fig. 4

Figure 4: EXAMPLE - the optimal schedule.

distribution. Similarly, the number of processors required by tasks was taken from a uniform distribution.

In Fig.5 one can find that the execution time of the algorithm is growing almost linearly with the growth of the processor number.

When the number of processors is growing, then the execution time of the algorithm is growing a bit faster than linearly, which is in accordance with theoretical expectations. This can be found in Fig.6.

The last parameter analyzed was the number of iterations of the algorithm. The number of iterations is a number of necessary extensions of the schedule length, in other words. The diagram in the Fig.7 presents dependence of the number of iterations (vertical axis - in %) on the value of $\frac{n}{m}$ ratio (horizontal axis). It turned out that the biggest number of iterations is achieved for $m \sim n$.

This result can be explained by the influence of two factors. When the number of tasks is smaller than the number of processors then it is highly probable that the length of the schedule will be determined by one long task. Then the rest of the tasks will have their own processors and will be scheduled feasibly.

On the other hand, when the number of tasks is big, most of the times, processing capacity of processors is shared 'fairly'. This is a result of the fact that the number of long tasks requiring a lot of processors is small. Hence, after scheduling tasks requiring many processors, processing capacity can be consumed on all processors of k-processor and fairly passed to the succeeding tasks. In other words, the remaining processing capacity after scheduling T^k

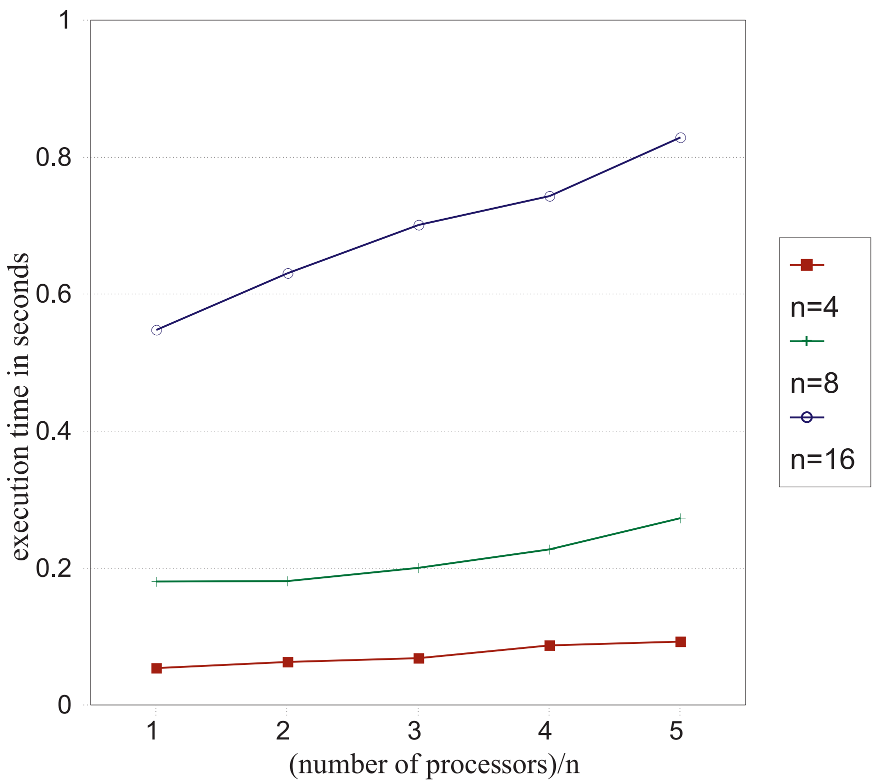

Figure 5: Execution time of the algorithm vs. the number of the processors.

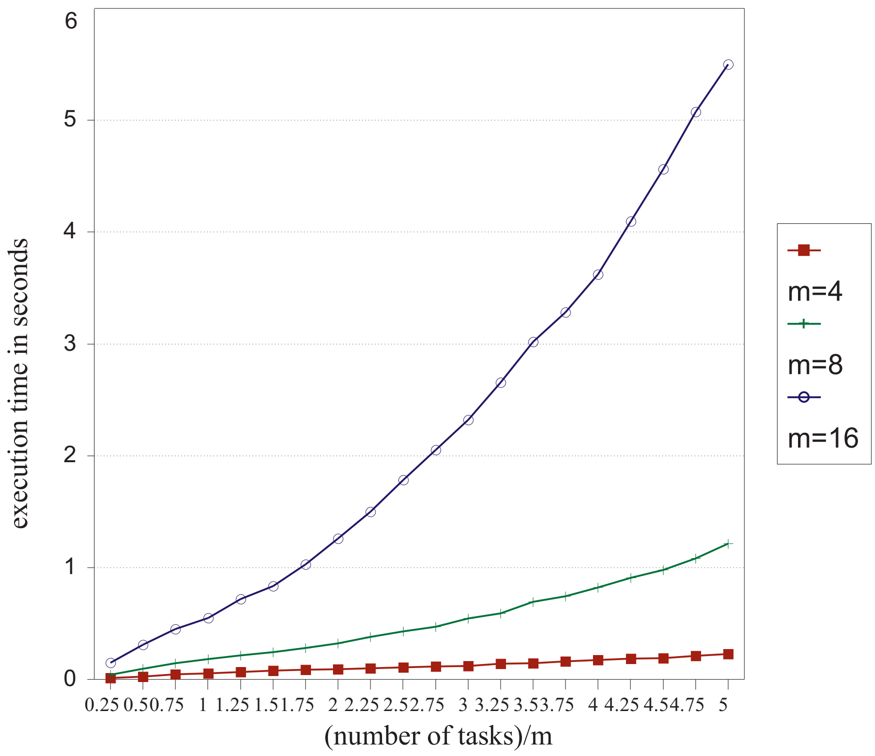

Figure 6: Execution time of the algorithm vs. the number of the tasks.

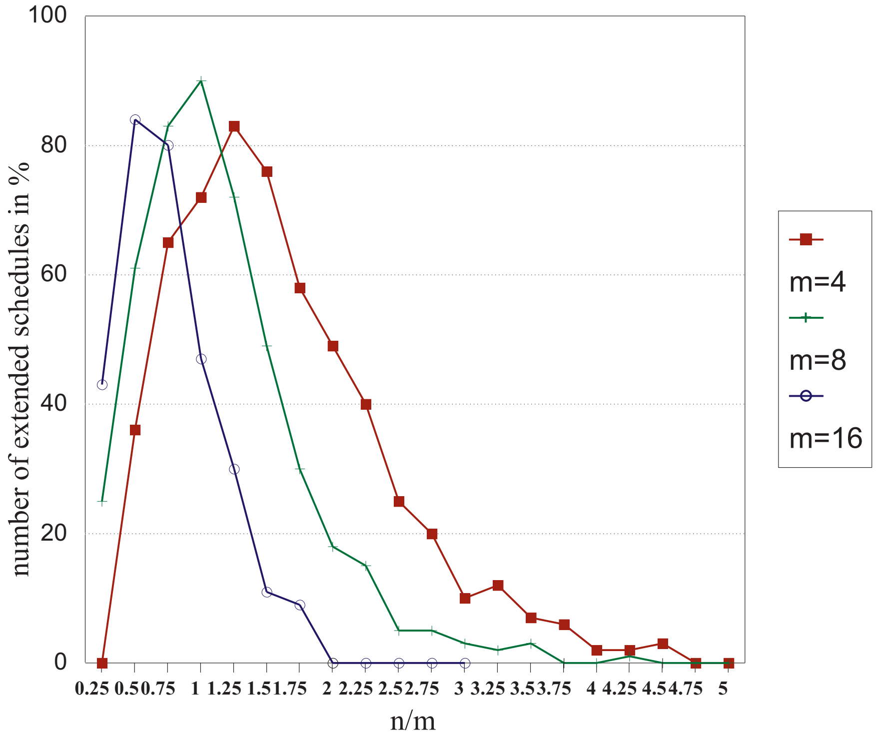

Figure 7: Number of iterations vs. $\frac{n}{m}$.

can be passed to subsequent tasks as during scheduling of k uni-processor tasks. This situation is more like the situation assumed while calculating $C(1)$. Thus, the lower bound is more precisely estimating C^*_{max}.

4 Conclusions

In the paper, a new model of deterministic scheduling, applicable in multi-microprocessor systems such as shared memory multiprocessors or hypercubes of processors, has been considered. It has been assumed that any task may require more than one processor at a time. The presented $O(nm^2 + nlogn)$ time algorithm find a minimum length schedule on uniform processors under the assumption that tasks require certain numbers from one to k processors. A more general problem where tasks may require any fixed number of processors from the set $\{1, \ldots, k\}$ may be solved via linear programming approach ([5]). Further generalizations include, among others, deadline scheduling problems which are very important from the practical point of view. These problems are now being studied.

References

[1] M.J.Atallah, C.Lock Black, D.C.Marinescu, Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations, Journal of Parallel and Distributed Computing 16, 1992, pp. 319-327.

[2] A. Avizienis, Fault tolerance: the survival attribute of digital systems, Proceedings of the IEEE 66, 1978, pp. 1109-1125.

[3] K. Baker, Introduction to sequencing and Scheduling, New York, John Wiley & Sons 1974.

[4] J. Błażewicz, M. Drabowski and J. Węglarz, Scheduling multiprocessor tasks to minimize schedule length, IEEE Transactions on Computers C35, 1986, pp. 389-393.

[5] J. Błażewicz, M. Drozdowski, G. Schmidt, D.deWerra, Scheduling independent multiprocessor tasks on a uniform k-processor system, Report R-92/030, Institute of Computing Science, Technical University of Poznań, Poland.

[6] J.Błażewicz, K.Ecker, G.Schmidt, J.Węglarz, Scheduling in Computer and Manufacturing Systems, Springer , New York, 1992.

[7] G.I.Chen, T.H.Lai, Preemptive scheduling of independent jobs on a hypercube, IPL 28, 1988, 201-206

[8] E.G. Coffman Jr., Computer and Job-Shop Scheduling Theory, New York, John Wiley & Sons, 1976.

[9] M. Dal Cin and E. Dilger, On diagnosibility of self-testing multimicroprocessor systems, Microprocessing and Microprogramming 7, 1981, pp. 177-184.

[10] M.Drozdowski, Problemy i algorytmy szeregowania zadań wieloprocesorowych, Ph.D. thesis, Institute of Computing Science, Technical University of Poznań, Poland, 1992.

[11] Jinzhong Du and Joseph Y-T. Leung, Complexity of scheduling parallel task systems, SIAM J.Disc. Math. 12, 1989, pp. 473-487.

[12] D.G.Feitelson, L.Rudolph, Gang Scheduling Performance Benefits for Fine-Grain Synchronization, Journal of Parallel and Distributed Computing 16, 1992, pp.306-318.

[13] E. Gehringer, D. Siewiorek, Z. Segall, Parallel processing. The Cm* experience, Digital Press, 1987.

[14] T. Gonzalez and S. Sahni, Preemptive scheduling of uniform processor systems, J. ACM 25, 1978, pp. 92-101.

[15] E.L. Lawler, Recent results in theory of machine scheduling, in Bachem et al. (eds.): Mathematical Programming. The State of Art, Springer Verlag 1983, pp. 200-234.

[16] G. Schmidt, Scheduling on semi-identical processors, Zeitschrift für Oper. Res. Theory 28, 1984, pp. 153-162.

[17] C.Seitz, The Cosmic Cube, Comm. of the ACM, vol. 28, No. 1, 1985.

[18] X.Shen, E.M.Reingold, Scheduling on a hypercube, Information Processing Letters, vol. 40, no.6, (1991), 323-328.

[19] J.A. Stankovic and K. Ramamritham, Hard Real-Time Systems, Computer society of the IEEE, Washington, 1988.

[20] R.Williams, Finite Element Algorithms and Parallel Computers, CCSF-8-91, 1991.