Available online at www.sciencedirect.com
SCIENCE DIRECT®

European Journal of Operational Research 146 (2003) 403–416

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

www.elsevier.com/locate/dsw

Discrete Optimization

Algorithms for dynamic scheduling of unit execution time tasks

Antonio Rodríguez Díaz [a], Andrei Tchernykh [b], Klaus H. Ecker [c,*]

[a] University of Baja California, Tijuana, Mexico
[b] CICESE Research Center, Ensenada, Mexico
[c] Institut für Informatik, Technische Universität Clausthal, Julius Albert Str. 4, Clausthal-Zellerfeld D-38678, Germany

Received 11 September 2000; accepted 24 October 2001

Abstract

We analyze performance properties of list scheduling algorithms under various dynamic assumptions and different levels of knowledge available for scheduling, considering the case of unit execution time tasks. We focus on bounds for the ISF (immediate successors first) and MISF (maximum number of immediate successors first) scheduling strategies and show the difference from other bounds obtained for the same problem. Finally, we present case studies and experimental results to assess the average behavior.
© 2003 Elsevier Science B.V. All rights reserved.

Keywords: Scheduling; List scheduling; Worst case analysis; On-line scheduling; UET

1. Introduction

A scheduling problem is called *static* if all information required to develop a feasible (optimal or non-optimal) schedule is available before the first task is actually processed. This means that the number of tasks, their processing times, and precedence constraints are known in advance. When dealing with such problems, we may proceed in two steps: first create a schedule for all tasks, and then execute them according to the schedule. This first phase is also often referred to as *off-line* scheduling. In contrast to this, a scheduling process is called *on-line* if the schedule is constructed on the fly together with the execution of tasks. This is typically the case if the total set of tasks or task parameters is not known in advance, but knowledge about new tasks becomes available during or after the execution of already known tasks. Scheduling is then only possible for the tasks we have already knowledge of, i.e. it can only be done on the basis of a limited planning horizon; if new tasks arrive, the current schedule has to be updated appropriately. In some cases a static aspect would appear, for example, if upon arrival of a new task its exact processing time and dependencies to other already present tasks are available. In the case that only probabilistic information (for example about the processing time) is available, we talk about a *stochastic* scheduling problem; this case is not considered here.

* Corresponding author. Tel.: +49-5323-723553; fax: +49-5323-723573.
E-mail addresses: antonio@faro.ens.uabc.mx (A. Rodríguez Díaz), chernykh@cicese.mx (A. Tchernykh), ecker@informatik.tu-clausthal.de (K.H. Ecker).

0377-2217/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
PII: S0377-2217(02)00236-9

In this paper we consider on-line scheduling of unit execution time (UET) tasks. We distinguish several levels of knowledge about newly arriving tasks and analyze performance guarantees of list scheduling algorithms under various dynamic assumptions. We show that, depending on the available knowledge, the well-known Graham's bound can be improved. We distinguish three levels of knowledge available for dynamic scheduling.

Level I. At each point of time we have knowledge about the tasks that are ready to be started. We also know the duration of tasks (we restrict ourselves to UETs). Though new tasks can arrive at any moment, tasks are started at integer times.

Level II. In addition to level I, the information whether or not immediate successors of each ready task exist is available.

Level III. In addition to level I, the number of immediate successors of each ready task is known.

For analysis of the performance we make four different assumptions:

Assumption 1. The information of level I is available for analysis.

Assumption 2. The total number of tasks and the length of the longest chain in the task system are also available.

Assumption 3. The total number of tasks and the ratio of tasks without successor and tasks with immediate successors are available.

Assumption 4. The total number of tasks, number of tasks without successors and the length of the longest chain in the task system are available.

To motivate the introduction of the levels of knowledge available for dynamic scheduling (especially levels II and III) we can consider a scheduling of processes that spawn other processes. A parallel program can be represented by a dynamic precedence task graph (dynamic spawn task graph) that is constructed at run time. In such a model the set of tasks is not known exactly in advance. Each node can be considered as a process (an instruction or a group of instructions) that is able to spawn other processes. In this case the precedence relation between tasks is interpreted as *spawn precedence*. $T_i \prec T_j$ means that during the processing of T_i, T_j is being created. In this paper we assume that T_j can be started only after T_i is completed.

A process can create child processes dynamically via, for instance, the *fork*() system call. These in turn can create further child processes and so on. The knowledge of the number of immediate successors of each process becomes very important especially for distributed programs that use the spawn system call intensively. For instance, Mosix [1], a software package that enhance the Linux kernel with cluster computing capabilities, considers a number of such descendant processes to justify migration of the parent process for load balancing.

To estimate and verify the quality of a schedule obtained during the dynamic execution of the task system and to calculate performance bounds we perform an analysis under the above assumptions. Additional knowledge such as the total number of tasks, number of tasks without successors and the length of the longest chain in the task system could help to verify its worst case behavior and justify a particular choice of algorithm. Such an information can be obtained for instance by performing a postmortem analysis, by the prediction performance analysis on the PRAM, or by the analysis of task graphs on an unbounded number of processors.

Section 2 introduces the notation of scheduling problems. In Section 3 we review some known results for the deterministic case of scheduling of UET tasks with arbitrary precedence constrains on a set of identical processors, subject to minimizing makespan. Furthermore we show some new results that consider the length of the longest task chain in the precedence graph. In Section 4 we discuss algorithms for the same problem under different levels of knowledge available for scheduling. In Section 5 we present some experimental results.

2. Notation

Following the notation given in [2] for deterministic scheduling problems, we denote by $\mathcal{T} =$

$\{T_1, \ldots, T_n\}$ the set of n tasks. Assume that the order in which the tasks can be processed on m identical processors $P_1, \ldots, P_m$ is restricted by some precedence relation $\prec$ over the set $\mathcal{T}$. Furthermore, non-preemptive scheduling is assumed. Generally we deal with *greedy schedules*, i.e. in which tasks are assigned to processors as early as possible. This strategy does not allow a processor to be idle as long as there are tasks available for processing.

There are several possibilities to define schedules formally. For the problem types considered here it would suffice to specify for each task the processor it is processed on, and the start time. For our purposes, however, the following definition appears to be more suitable. A schedule is defined as a function $S : \mathbb{R}^{\geqslant 0} \to (\mathcal{T} \cup \{\varepsilon\})^m$, where ε, the *idle processor*, is used to express the situation that a processor does not process any task, and $S(t) = (T_{\alpha_1}, \ldots, T_{\alpha_m})$ with $T_{\alpha_j} \in \mathcal{T} \cup \{\varepsilon\}$ for $j = 1, \ldots, m$, specifies the tasks assigned to processors $P_1, \ldots, P_m$ at time t. Let $t_0 < t_1 < \cdots < t_f$ be the points of time where function S changes. We refer to the interval $[t_i, t_{i+1})$ as the ith (*time*) *slot* of S. Obviously, S is uniquely specified by the sequence $S(t_i) \,|\, i = 0, \ldots, t_{f-1}$. W.l.o.g. we assume that the first slot starts at time $t_0 = 0$. t_f is referred to as the *length* or *makespan* of S, denoted by $C(S)$ or shortly by C.

In the special case that tasks have unit processing times, the ith (*time*) *slot is* $[i, i+1)$.

The general objective in this paper is to find a schedule that executes the given set of tasks in minimal possible time (scheduling problem with minimizing makespan criterion). In the short three-field notation *machine* |*task*| *criterion* proposed by Graham et al. [3] and Błażewicz et al. [4], this problem is characterized as $P|\text{prec}, p_j = 1|C_{\max}$.

Before going into details we make some general remarks well known in the theory.

Since exact algorithms such as branch and bound, dynamic programming and other enumerative techniques are very time consuming for most problem classes (to which practical problems usually also belong), one may apply heuristic algorithms. These allow generating solutions in reasonable time, even for large problem instances, but, except for specific cases, there is no general way to make a statement about the quality of the solution. Another approach is list (or priority driven) strategies by which we consider algorithms that assign priorities to the tasks. The tasks may be understood as being arranged in the list in decreasing order of their priorities. A processor becoming idle "grabs" a task of the highest priority from the list in a greedy way. It is obvious that list algorithms are not optimal in general; therefore a careful analysis of the property of the priority assignment informing about their worst case performance would be useful.

Let I be a problem instance for $P|\text{prec}, p_j = 1|C_{\max}$, and let $S_{\text{opt}}(I)$ be an optimal schedule for I with makespan $C_{\text{opt}}(I)$. For a list algorithm A, let $S_A(I)$ be the corresponding schedule with makespan $C_A(I)$. With $r_A(I) = C_A(I)/C_{\text{opt}}(I)$ we denote the *performance ratio of algorithm A for instance I.* To measure the overall quality of A we define the *performance of A* by

$$r_A = \sup\{C_A(I)/C_{\text{opt}}(I) \,|\, I \text{ is a problem instance}\}.$$

We may sometimes wish to know how accurate list algorithms are in general. For the minimizing makespan criterion, for example, let $C_w(I)$ denote the longest makespan of a schedule obtained by applying any list algorithm on an instance I. The worst case behavior of list algorithms, denoted by

$$r_w := \sup\{r_A \,|\, A \text{ is an arbitrary list algorithm}\},$$

gives an upper bound on the performance of any list algorithm. By C_w $[C_{\text{opt}}]$ we denote the length of an arbitrary greedy [respectively: optimal] schedule.

For analyzing some given schedule S, the following notation may be useful: At some time $t \geqslant 0$, the number of tasks being processed is denoted by n_t, and $z_t := m - n_t$ is the number of idle processors at time t. An *idle interval* is an interval during which some processor does not process any task. Recall that in a greedy schedule, an idle interval can only occur if no task is available for processing. Let $z = \sum z_t$ denote the sum of lengths of idle intervals in the schedule within the time span 0 (start time of the schedule) and $C_{\max}$ (end of the schedule). Likewise, z_{opt}, z_A and z_w are the corresponding respective sums for an optimal

schedule, a schedule generated by algorithm A, and a general greedy schedule.

Given an instance, we denote by L_c the longest chain in the precedence graph, and by l_c its length.

3. Properties of the deterministic case

To construct an optimal off-line schedule, we may apply an exact algorithm. For example for $P|\text{prec}, p_j = 1|C_{\max}$, if prec is an in-tree or out-tree, Hu's level algorithm [5] is optimal and of linear time complexity. El-Rewini et al. [6] consider an algorithm for scheduling interval-ordered tasks on an arbitrary number of processors. The properties of interval orders make it possible to apply a simple greedy algorithm that calculates the number of immediate successors as a priority. It leads to an optimal schedule when all tasks have the UET. This algorithm solves the $P|\text{interval- ordered}, p_j = 1|C_{\max}$ for interval order (V,E) in $O(|E| + |V|)$ time (see also [7]). Chen and Liu [8] and Kunde [9] discussed the performance of the level algorithm when applied to general precedence graphs. For two processors and general precedences (i.e. $P2|\text{prec}, p_j = 1|C_{\max}$), the problem can be solved in quadratic time [10]. Garey and Johnson [11] devised an optimal algorithm for the same problem. If the precedence relation is of bounded width k (i.e. each subset of $k + 1$ tasks has at least one pair of dependent tasks) the problem $P|\text{width-}k, p_j = 1|C_{\max}$ can be solved in $O(n^k)$ time [12].

We know, however, that the more general problem $P|\text{prec}, p_j = 1|C_{\max}$ is NP-hard, hence, one way to get solutions in reasonable time is to apply approximation algorithms. For list scheduling algorithms Graham [13] showed that the performance bound $r = C_{\text{list}}/C_{\text{opt}}$ is $2 - 1/m$. The application of the critical path algorithm to solve $P|\text{prec}, p_j = 1|C_{\max}$ has been analyzed by Kunde [9], and the following bound have been proved: $r = 2 - (1/m - 1)$ for $m \geqslant 3$. The algorithm given by Braschi and Trystram [14] is slightly better, its bound is $r = 2 - (2/m) - ((m - 3)/(m \cdot C_{\max}))$ for $m \geqslant 3$. Regarding the time complexity of scheduling algorithms we have to emphasize that even the problem $Pm|\text{prec}, p_j = 1|C_{\max}$ (with a fixed number of processors) is still of unknown time complexity despite the fact that many papers have been devoted to solving various cases of precedence constraints (see [2] for more details).

In the following we summarize some simple known properties of optimal and greedy schedules under the deterministic assumption and discuss a strategy that appears to be useful for dynamic cases. We restrict ourselves to the case of UETs. For any arbitrary schedule it is obvious that $C = (n + z)/m$, where z is the sum of lengths of idle intervals in the schedule.

Define the *layer function* $\lambda : T \to \mathbb{N}^0$ as in [5] recursively by $\lambda(T) = 1$ if T has no predecessors, and $\lambda(T) := \max\{\lambda(T') + 1 \mid T' \prec T\}$. Let $\Lambda_i := \{T \mid \lambda(T) = i\}$, and $n_i := |\Lambda_i|$ $(i = 1, \ldots, l_c)$, where l_c is the length of the longest chain in the task system. Refer to Λ_i as the ith *layer*. The layer and co-level are coincided for UET task system.

We start our analysis by considering a simple scheduling algorithm.

Primitive strategy. Schedule the tasks of each layer separately: starting with the first layer, the tasks of Λ_1 are assigned arbitrarily in a greedy way to the processors. The number of time slots required for the tasks of Λ_1 is $\lceil n_1/m \rceil$. Then the tasks of Λ_2 are assigned by starting a new time slot, and so on.

Notice that precedences will not be violated by this strategy. Furthermore, for each layer $1, \ldots, l_c$, there will be at most one time slot that is not completely filled with tasks, i.e. has between 1 and $m - 1$ idle processors. Suppose the number n_i of tasks the ith layer Λ_i is smaller than or equal to m. Then the primitive strategy algorithm fills the tasks of Λ_i in a single time slot, thus using n_i processors, and leaves $m - n_i$ processors idle. On the other hand, if $n_i > m$, the algorithm fills $\lfloor n_i/m \rfloor$ slots completely. If there are some tasks left (i.e. if $n_i \bmod m > 0$), then an additional slot will be used for the remaining $n_i \bmod m$ tasks; this slot has $m - (n_i \bmod m) > 0$ idle processors.

Obviously a schedule constructed by this algorithm will not necessarily be greedy. The reason is that the number of tasks in a layer is not an integer multiple of m in general, and the primitive strategy does not take tasks from the next layer. In the attempt to improve the schedule, depending on the structure of the precedence relation and the order in which the tasks of a layer are scheduled, we

might be able to shift some of the tasks of a layer to an earlier time in the schedule. Hence, the length of a greedy schedule is less than or equal to the length of a schedule obtained by the primitive strategy.

Lemma 1. *The following facts are known (see, e.g. [15]):*
1. *For any instance of $P|\text{prec}, p_j = 1|C_{\max}$, the number of z_w idle intervals in any greedy schedule is limited by $l_c(m-1)$.*
2. $C_w \leqslant (n - l_c)/m + l_c$.
3. $(n + z_{\text{opt}})/m \geqslant l_c$.
4. *For an arbitrary task graph, $C_w - C_{\text{opt}} \leqslant l_c(m-1)/m$.*

Consider a schedule generated by the primitive strategy and the corresponding greedy schedule. Since at most one time slot with idle intervals can occur for each layer, the sum of idle intervals is $z_w \leqslant l_c(m-1)$. If $C_w = l_c$, then the schedule is already greedy, and each slot has at least one task (that of the chain L_c). If $C_w > l_c$ then there must be a layer Λ_i with $n_i > m$. In this case, the constructed schedule is not necessarily greedy, and some of the tasks of the next layer might be shifted to an earlier time in the schedule. Denote by S_i the sequence of time slots used by the primitive strategy for the tasks of Λ_i, $i = 1, \ldots, l_c$. If S_i has an idle interval, processing a task of Λ_{i+1} in S_i would decrease the number of idle intervals in S_i, but increase the number of idle intervals in S_{i+1}. Thus, making the schedule greedy will not increase the total length of the schedule, and hence the upper bound $l_c(m-1)$ remains the same for greedy schedules.

With $C_w = (n + z_w)/m$ and $z_w \leqslant l_c(m-1)$, we get $C_w \leqslant (n + l_c(m-1))/m = (n - l_c)/m + l_c$, that is similar to Brent's lemma [16]. If $C_{\text{opt}} = l_c$, then $(n + z_{\text{opt}})/m = l_c$. If $C_{\text{opt}} > l_c$, then $(n + z_{\text{opt}}/m > l_c)$. The case of $C_{\text{opt}} < l_c$ is not possible, hence $l_c \leqslant (n + z_{\text{opt}})/m$. $C_w - C_{\text{opt}} = ((n + z_w) - (n + z_{\text{opt}}))/m = (z_w - z_{\text{opt}})/m$. Since $z_w \leqslant l_c(m-1)$ and $z_{\text{opt}} \geqslant 0$, the maximum difference is $l_c(m-1)/m$.

4. Scheduling under dynamic assumptions

According to the levels of available knowledge about the task system introduced in Section 1, we distinguish different algorithms and perform their analysis under the various assumptions.

4.1. Level I

Algorithm. On this level of knowledge we are only able to apply a general list strategy.

Analysis under Assumption 1. Before completion of a task, we do not have any information about tasks. Hence it may happen that the chosen order in which the ready tasks are processed turns out to be bad. As compared to the best sequencing (which of course we do not know at run-time, but can be verified afterwards), Lemma 1[(1), (2)] gives $r_w \leqslant 1 - 1/m + n/(n + z_{\text{opt}})$. For an arbitrary task graph $z_{\text{opt}} \geqslant 0$, hence $r_w \leqslant 2 - 1/m$. So we get a theoretical performance r_w for arbitrary task graphs as given by Graham's bound, $2 - 1/m$ [15].

Analysis under Assumption 2. We next adopt Assumption 2 and estimate r_w. Based on the known parameters n and l_c we are able to derive a more accurate performance bound of any greedy strategy.

Theorem 1. *Given a set $\mathcal{T}$ of UET tasks, the performance of the general list strategy can be estimated by*

$$r_w = \min\{r'_w, r''_w\} \quad \text{with } r'_w \leqslant 1 + \frac{l_c}{n}(m-1)$$
$$\text{and} \quad r''_w \leqslant 1 + \frac{1}{m}\left(\frac{n}{l_c} - 1\right). \tag{1}$$

Furthermore, r'_w is tight in the case of $l_c \leqslant n/m$, and r''_w is tight in the case of $l_c > n/m$.

Proof. From Lemma 1 we know $C_w \leqslant (n - l_c)/m + l_c$. Obvious lower bounds for C_{opt} are $C_{\text{opt}} \geqslant n/m$, and $C_{\text{opt}} \geqslant l_c$. From this the bounds r'_w and r''_w follow immediately. To prove tightness of the bounds, notice first that

- $r_w = 2 - 1/m$ in case of $l_c = n/m$,
- $r'_w < r''_w$ for $l_c < n/m$, and
- $r'_w > r''_w$ for $l_c > n/m$.

Choose the following problem instance: $n = km + l_c$ tasks, organized in $m + 1$ chains, one of length l_c [chain L_c] and m chains each of length

$k \in \mathbb{N}$. An optimal schedule is obtained by assigning the tasks of L_c to one processor, and the tasks of the other chains in a greedy way. The worst greedy schedule is obtained by scheduling first the tasks of the m chains, thus filling the first k time slots completely, and then assigning the tasks of L_c. The length of this schedule is $k + l_c$.

If we choose $l_c = n/m = km/(m-1)$, i.e. $n = km^2/(m-1)$, we get $C_{\text{opt}} = l_c$, and $r_w = 2 - 1/m$.
If $l_c < km/(m-1)$ we get $C_{\text{opt}} = l_c + (km - l_c(m-1))/m$; the ratio $r_w = C_w/C_{\text{opt}}$ in this case is $1 + l_c(m-1)/n = r'_w$.
If $l_c > km/(m-1)$ we get $C_{\text{opt}} = l_c$, and the ratio $r_w = C_w/C_{\text{opt}}$ is $(1 + (1/m)((n/l_c) - 1)) = r''_w$. □

Fig. 1 shows the dependency between r_w and the number m of machines. It shows that there is a trade-off for a given value of m, which depends on the number of tasks and the length of the critical path.

Fig. 2 shows the performance bound for task graphs with different l_c, when the number of machines m varies.

4.2. Levels II and III

In this section we study two algorithms that use additional information about the immediate successors of ready tasks: The first algorithm uses the fact of existence of immediate successors. The second algorithm takes the number of immediate successors into consideration. We show that the performance guaranties of both algorithms are the same.

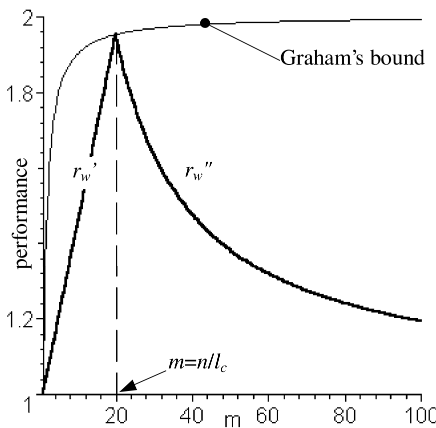

Fig. 1. Worst case ratio r_w, when keeping the number of tasks and the length l_c fixed, and varying the number machines ($n = 1000$, $l_c = 50$).

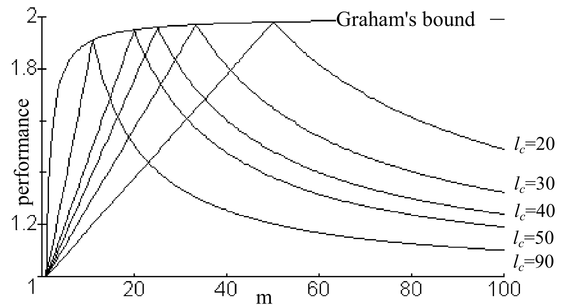

Fig. 2. Worst case ratio r_w, varying the number m of machines and the length l_c of the longest chain ($n = 1000$).

Immediate successors first (*ISF*) *algorithm*. Let us now assume that at each point of time we have additional knowledge about the fact of the existence of immediate successors of all ready tasks. The following algorithm makes use of the available information by giving tasks with immediate successors a higher priority as compared to those without successor.

Algorithm 1 (*ISF algorithm*). Maintain two lists of current available tasks: one contains the tasks with immediate successors, and the other those without successors. The tasks of the first list are processed prior to the tasks of the second list. If the processing of a task T is started, it is erased from the list. Upon completion of the task T its immediate successor tasks become available and are entered in the corresponding list according to the existence of their immediate successors.

begin
Let $Q^{>0}$ and $Q^{=0}$ be the lists of tasks that are ready to be started, with and without immediate successors, respectively;
repeat
for each completed task *do*
Insert its ready immediate successor tasks in $Q^{>0}$ or $Q^{=0}$, according to the existence of their immediate successors; – a task is *ready* if all its immediate predecessors are completed

for each idle processor *do*
if $Q^{>0} \neq \emptyset$
then Assign the first task T from $Q^{>0}$ for processing
else Assign the first task T from $Q^{=0}$ for processing;
Remove T from its list;
until $Q^{>0}$ and $Q^{=0}$ are empty;
end;

Maximum number of immediate successors first (MISF) algorithm. A version of ISF, referred to as MISF, is obtained if only one list of tasks is used where the tasks are sorted in decreasing order of the number of immediate successors.

Algorithm 2 (*MISF algorithm*)
begin
Let $Q = (T_1, \ldots, T_r)$ be the sorted list of tasks that are ready to be started;
repeat
for each completed task *do*
Insert its ready immediate successor tasks in Q in decreasing order of their number of immediate successors;
for each idle processor *do*
Assign the first task T from Q for processing, and remove T from Q;
until Q is empty;
end;

Analysis under Assumption 1. The advantage of ISF and MISF as compared to an arbitrary greedy strategy (as can only be applied in case of Assumption 1) is that the execution of a task with no successor is delayed until there are not enough other tasks to fill a whole time slot. To see that the bound of Theorem 1 is tight for the levels II and III, consider the following example: let q be an arbitrary integer, and be given m chains, each of length $(m-1)q$, and one chain of length mq. The total number of tasks is m^2q, and it is possible to schedule the tasks optimally in mq time. Notice that $l_c = n/m$. Since all tasks, except the last $m+1$ tasks of the chains, have exactly one immediate successor, the algorithm could start with the tasks of the shorter chains and execute them except the last m ones. This takes $(m-1)q-1$ time slots. Then ISF or MISF processes the tasks of the long chain, altogether with the m remaining tasks of the short chains; this requires mq time slots. So altogether we get a schedule of length $(m-1)q-1+mq = 2mq-q-1$. With this the performance of ISF or MISF is

$$r_{\mathrm{ISF}} = \frac{C_{\mathrm{ISF}}}{C_{\mathrm{opt}}} = \frac{2mq-q-1}{mq} = 2 - \frac{1}{m} - \frac{1}{mq},$$

which tends to $2-1/m$ for large values of q. We see from this example that ISF and MISF have the same worst case bound under Assumption 1.

Analysis under Assumption 2. This analysis is reduced to the analysis under Assumption 2 with available knowledge of level I.

Analysis under Assumption 3. Under this assumption additional information about the ratio of tasks without successor and tasks with immediate successors of the task graph is available.

Denote by $\mathscr{T}^0$ the set of tasks without successor (out-degree = 0, "leaves"), and let $\mathscr{T}^{>0} := \mathscr{T} - \mathscr{T}^0$ the set of tasks with immediate successors (out-degree > 0, "inner tasks"). Let $n^0 := |\mathscr{T}^0|$, $n^{>0} := |\mathscr{T}^{>0}| = n - n^0$, and assume that n^0 and $n^{>0}$ are more than 0. Let denote by s'' the number of slots filled only with leaves.

Consider the special case where $s'' = 1$. In view of Lemma 1 $[Z_w \leqslant l_c(m-1)]$, $s''=1$ implies that the number of tasks with out-degree = 0 is $n^0 \leqslant Z_w = l_c(m-1)$.

Lemma 2. *If* $s''=1$, *then* $r_{\mathrm{ISF}} \leqslant (1/1+\sigma) + 1 - 1/m + 1/n$, *where* $\sigma = n^0/n^{>0}$ *is the ratio of out-degree* = 0 *tasks and out-degree* > 0 *tasks.*

Proof. If $s''=1$, then the tasks with out-degree = 0 are processed together with L_c, and the ISF algorithm gives $Z_{\mathrm{ISF}} \leqslant z_w - n^0$. Hence

$$\begin{aligned} C_{\mathrm{ISF}} &\leqslant (n + l_c(m-1) - (n^0-1))/m \\ &= (n + ml_c - l_c - n^0 + 1)/m \\ &= (n - l_c - n^0 + 1)/m + l_c. \end{aligned}$$

We get for the performance ratio

$$r_{\mathrm{ISF}} \leqslant (n - n^0 + 1)/n + (1 - 1/m) l_c / C_{\mathrm{opt}},$$

i.e. using the lower bound l_c for C_{opt},

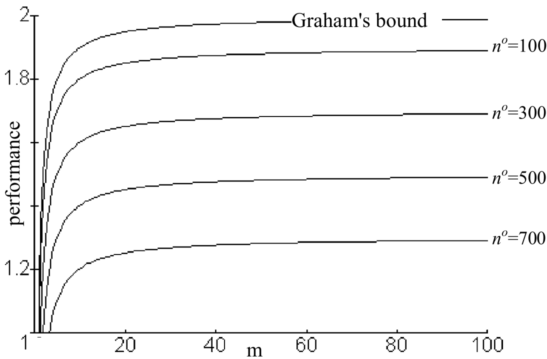

Fig. 3. Worst case ratio r_{ISF}, varying the number m of machines and the number n^0 $(n = 1000)$.

$$r_{\mathrm{ISF}} \leqslant \frac{1}{1+\sigma} + 1 - \frac{1}{m} + \frac{1}{n}. \qquad \square$$

For example, if $\mathscr{T}^0$ and $\mathscr{T}^{>0}$ have the same cardinality, then $\sigma = 1$, and $r_{\mathrm{ISF}} < 3/2 - 1/m + 1/n$.

Fig. 3 shows the behavior of r_{ISF} when varying the number of machines and the number n^0 of tasks without successors, while keeping the number of tasks fixed.

Analysis under Assumption 4. If we have knowledge about the number of tasks, the length of the longest chain and the number of tasks without successors, we are able to derive better estimations for the performance of MISF and ISF.

ISF-schedules. The ISF-strategy has the property that it does not distinguish between different out-degrees > 0. In extreme, the ISF-algorithm could give all tasks of out-degree > 0 and not being in L_c a higher priority. We assume that the algorithm chooses the tasks of L_c at the latest possible time for processing, and let S_{ISF} be a schedule generated by the ISF-algorithm in this way. We discuss some simple properties of S_{ISF}.

Analyzing the sequence of time slots of S_{ISF}, we see that there are time slots that have a task of L_c (some longest chain of length l_c), and there may be other slots that have no task of L_c. Consider a slot s_i that has no task of L_c, and assume that there is a later slot s_j $(j > i)$ in S_{ISF} that has a task T_k of L_c. If T_k has out-degree > 0, then s_i can neither have an idle interval nor can contain a task of out-degree $= 0$, because otherwise the ISF-strategy would already have scheduled T_k in s_i.

As a consequence, if $l_c \geqslant 2$, the only idle intervals that may occur in S_{ISF} are found in those slots where a task of L_c is scheduled. Notice that the number of time slots with idle intervals is bounded from above by l_c.

Assume that $l_c \geqslant 2$, and let the $(l_c - 1)$th task of L_c be scheduled at some time t (i.e. in slot t). Then consider the time slots from 0 to $t - 1$ that contain no task of L_c. We know from the above discussion that in these slots, we find only tasks of out-degree > 0, and no idle tasks. With s' we denote the number of these time slots, i.e. $s' = t - l_c + 2$.

Next we claim that in the schedule S_{ISF} we may find only tasks of degree $= 0$ from time $t + 1$ on. Suppose a task of degree > 0 is scheduled at time $t' \geqslant t + 1$. This, however, leads to a contradiction with the assumption that the tasks of L_c are scheduled with the smallest priority among the tasks with out-degree > 0, i.e. at the latest possible time.

We distinguish three kinds of slots, those containing (a) a task of L_c, (b) no task of L_c, and (c) only leaves. We know that l_c is the number of slots, each with a task from L_c, and possibly other tasks from $\mathscr{T}^{>0}$ or $\mathscr{T}^0$.

Let

s' be the number of slots with tasks from $\mathscr{T}^{>0} - L_c$.

s'' be the number of slots filled only with leaves.

Obviously, $C_{\mathrm{ISF}} = s' + s'' + l_c - 1$. We subtracted 1 because one slot counted in l_c is filled only with leaves, and thus is also counted in s''. To optimize schedule S_{ISF}, we should exchange the positions of certain tasks, but there is no advantage to shift leaves to earlier slots. Hence the number of s'' slots in S_{ISF} will not be smaller in an optimal schedule, and as a consequence, $C_{\mathrm{opt}} \geqslant l_c + s'' - 1$.

Theorem 2. *The performance of the ISF-algorithm can be estimated as* $r_{\mathrm{ISF}} \leqslant \min\{r'_{\mathrm{ISF}}, r''_{\mathrm{ISF}}\}$, *where*

$$r'_{\mathrm{ISF}} = 1 + \frac{l_c}{n}(m-1) - \frac{n^0 - m(s''-1) - 1}{n}$$

$$\text{and} \quad r''_{\mathrm{ISF}} = 1 + \frac{1}{m}\left(\frac{n}{l_c} - 1\right) - \frac{n^0 - 1}{m l_c}. \qquad (2)$$

Proof. There are two lower bounds for C_{opt}. The first bound can be estimated as $C_{\mathrm{opt}} \geqslant n/m$, and the other bound is $C_{\mathrm{opt}} \geqslant l_c + s'' - 1$.

Let us first derive an upper bound for r_{ISF} by using the first upper bound for $C_{\mathrm{opt}} \geqslant n/m$. With $n^{>0} = n - n^0 \geqslant s'm + l_c - 1$ we get

$$r'_{\mathrm{ISF}} \leqslant (s' + s'' + l_c - 1)\frac{m}{n} \leqslant \frac{n^{>0}m(s''-1) + 1 + (m-1)l_c}{m} \cdot \frac{m}{n} = 1 + \frac{n^{>0}m(s''-1) + 1 - n}{n} + \frac{l_c}{n}(m-1),$$

which gives

$$r'_{\mathrm{ISF}} \leqslant 1 + \frac{l_c}{n}(m-1) - \frac{n^0 - m(s''-1) - 1}{n}.$$

Notice that, because of $n^0 \geqslant m(s''-1) + 1$, this bound is better than the bound r_w of Theorem 1 (see also Fig. 4).

Next we derive an upper bound for r_{ISF} by using $C_{\mathrm{opt}} \geqslant l_c + s'' - 1$:

$$r''_{\mathrm{ISF}} \leqslant \frac{s' + s'' + l_c - 1}{l_c + s'' - 1} = 1 + \frac{s'}{l_c + s'' - 1} \leqslant 1 + \frac{s'}{l_c}$$

(because $s'' - 1 \geqslant 0$). We get

$$r''_{\mathrm{ISF}} \leqslant 1 + \frac{n - l_c + 1 - n^0}{ml_c} = 1 + \frac{1}{m}\left(\frac{n}{l_c} - 1\right) - \frac{n^0 - 1}{ml_c}. \qquad \square$$

From Fig. 5 we see that for increasing values of n^0, and while keeping n fixed, the performance faster approaches the lower limit 1.

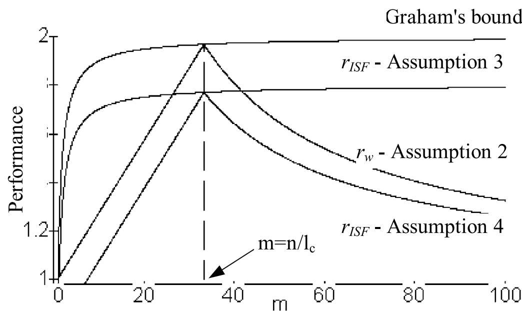

Fig. 4. Performance ratios keeping the number of tasks n, l_c, and n^0 fixed, and varying the number m of machines ($n = 1000$, $l_c = 30$, $n^0 = 200$, $s'' = 1$).

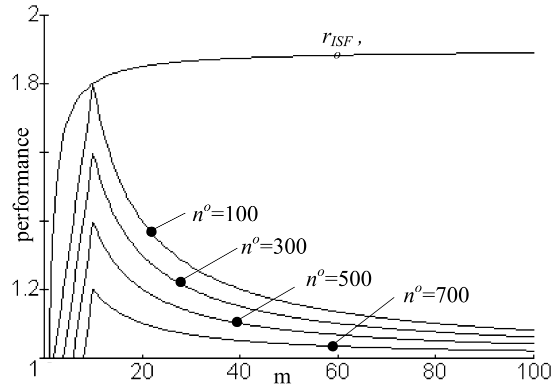

Fig. 5. Worst case ratio r_{ISF}, varying the number of machines m and the number n^0 ($n = 1000$).

5. Experimental studies

We present results of experiments conducted by Rodríguez Alcántar [17] and Tchernykh to study the average behavior of the performance of MISF in comparison to ARB (an arbitrary/random list strategy), MICF (minimum co-level first), MACF (maximum co-level first) dynamic list scheduling heuristics, and Hu's heuristic (MLF) that gives an optimal solution for trees of UET tasks. MICF is similar to FIFO scheduling that explores the task graph in a width-first manner, and MACF is similar to LIFO scheduling, i.e. a depth-first execution. The experiments analyze the average behavior of these algorithms for problem instances of $P|\mathrm{prec}, P_j = 1|C_{\max}$. Two kinds of experiments are performed, first under the assumption of out-forests, and then for arbitrary precedence constraints.

5.1. Out forests

In this section, the comparison test consists of applying dynamic heuristics to out-trees/out-forests. Tree structures occur in a wide variety of applications, including some form of divide and conquer algorithms, recursive calculations organized as a tree, such as summation, π-product, algorithms for tree architectures, parallel evaluation of arithmetical expressions, assembly line scheduling problems, network unification [18,19],

fork graphs and other. Notice that Hu's optimal algorithm is of no use because only local information about task dependencies is available in dynamic situations. In the simulation, we analyze out-forests with 20, 100, 1000 tasks. For each of these numbers, 100 task instances were randomly generated. Average values of some parameters of the generated set of task graphs are shown in Table 3. The performance ratio is calculated separately for each heuristic. The mean and standard deviation of the performance ratio are measured for each test. As a measure of the solution quality, the deviation from MISF is analyzed for each of three algorithms. For a given heuristic A, the PRI deviation from the MISF is defined as average of $(r_A - r_{\mathrm{MISF}}/r_{\mathrm{MISF}}) \times 100$, and the worst deviation as maximum of PRI for the set of instances. PRI stands for the average percentage of the performance ratio improvement gained by the algorithm MISF over algorithm A.

Figs. 6 and 8 show average performance ratios for out-forests with 100 and 1000 tasks, respectively, versus the number of processors. Figs. 7 and 9 summarize the deviation of the average performance ratios of MICF, MACF and arbitrary list scheduling from MISF for out-forests with, respectively, 100 and 1000 tasks.

Notice that the behavior of the average performance ratios (see Figs. 6 and 8) are similar to the behavior of the upper bounds claimed in Theorems 1 and 2 for arbitrary list scheduling and MISF. The performance ratio and improvement increase as the number of m increases up to the number of processors $m \approx n/\mathrm{AVG}(l_c)$, where

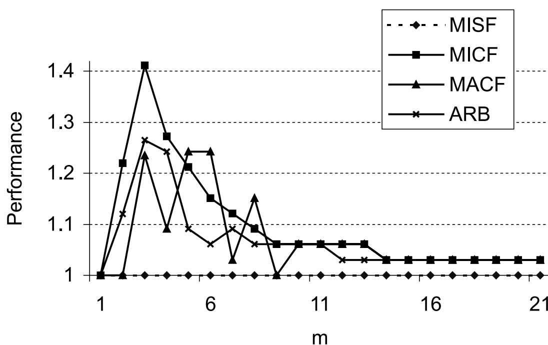

Fig. 6. Average performance ratio for graphs with 100 tasks.

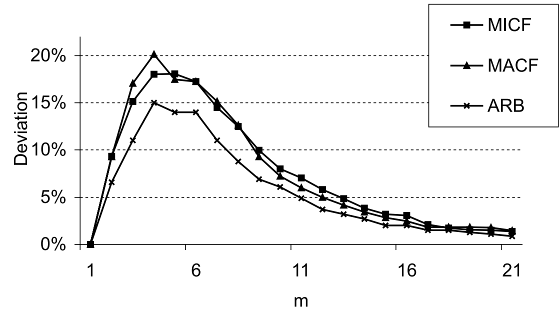

Fig. 7. Percentage of the average performance ratio improvement for graphs with 100 tasks.

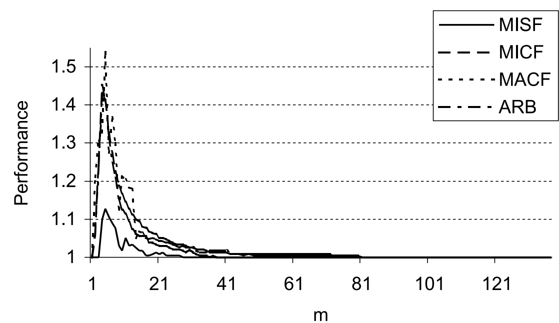

Fig. 8. Average performance ratio for graphs with 1000 tasks.

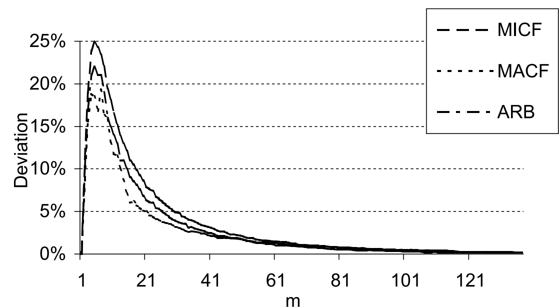

Fig. 9. Percentage of the average MISF performance ratio improvement for graphs with 1000 tasks.

$\mathrm{AVG}(l_c)$ is the average length of longest chains in the instances (see Fig. 4, where the peak of the bound is reached for $m = n/l_c$). After the peak the ratio decreases and tends to 1 while m gets close to $\max(n_i := |A_i|,\ i = 1, \ldots, l_c)$, where n_i is the number of tasks in the ith layer.

Table 1
Average performance ratio and percentage of the performance ratio standard deviation

n	MISF		MICF		MACF		ARB	
	PR	S.D. (%)	PR	S.D. (%)	PR	S.D. (%)	PR	S.D. (%)
20	1.0001	0.46	1.026	5.13	1.054	7.19	1.032	5.60
100	1.0001	1.05	1.096	8.48	1.070	10.29	1.067	7.50
1000	1.0060	0.98	1.030	4.30	1.032	4.94	1.023	4.05

Table 2
Maximum and average percentage of MISF performance ratio improvement over MACF, MICF heuristics and arbitrary list schedule

n	MACF (%)		MICF (%)		ARB (%)	
	Maximum	Average	Maximum	Average	Maximum	Average
20	9.75	2.57	12.48	2.69	6.60	1.65
100	18.08	7.56	20.13	7.53	15.00	5.63
1000	24.91	3.62	19.31	2.66	22.00	2.99

The latter reflects the fact that, as the number of processors increases, fewer tasks will be delayed due to processor limitation, and hence for sufficiently many processors, every greedy heuristic gives an optimal solution (or very near optimal on average).

Table 1 summarizes performance ratios and percentages of their standard deviations. We see that MISF gives more stable results with the smaller standard deviations as compared to the other heuristics. Notice that the deviation of MISF from optimal is at most 0.6% and remains almost optimal for the different problem sizes.

From Table 2 we conclude that MISF gives up to 9.75%, 18.08%, and 24.91% of the maximal improvement over other heuristics for respective 20, 100, and 1000 tasks in the graphs. Moreover, on average for different number of machines, it gives up to 7.56% of the improvement.

The worst deviation of MICF, MACF, and ARB from MISF reaches slightly more than 20%. It increases with the problem size (see Fig. 10). We can interpret this result in such a way that MISF handles bigger numbers of tasks better than other heuristics (Table 3).

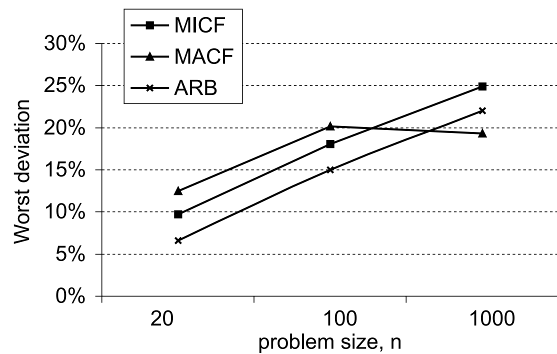

Fig. 10. Worst deviation vs. increasing problem size.

5.2. Arbitrary task graphs

In this section, we present results of experiments comparing heuristics applied to arbitrary task

Table 3
Average parameters of task graphs generated

n	$\max(n_i)$	# Trees in a forest	l_c	# Successors	n^0
20	6.99	3.13	7.78	4.48	9.69
100	20.54	7.07	25.55	10.87	44.02
1000	136.87	11.94	231.6	86.27	388.12

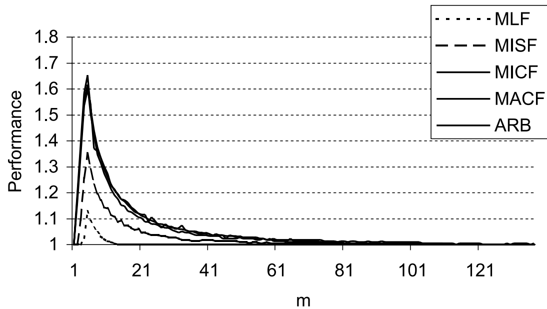

Fig. 11. Average performance ratio for graphs with 1000 tasks.

graphs (see Table 6). The competitive ratio is calculated over a lower bound of optimal solution that is the maximum of n/m and l_c. Fig. 11 shows the average performance for graphs with 1000 tasks for different numbers of processors. Fig. 12

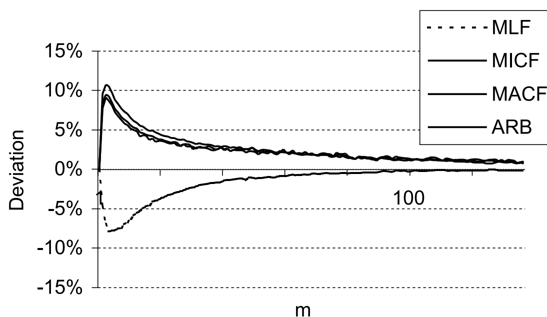

Fig. 12. Percentage of the average MISF performance ratio improvement for graphs with 1000 tasks.

summarizes the deviation of the average performance MLF, MICF, MACF, and ARB from MISF for graphs with 1000 tasks. Notice that the behavior of the average competitive ratios for arbitrary graphs are similar to the behavior of the performance ratios for out-forests (Tables 4–6).

6. Conclusion and future work

This paper addressed the problem of dynamic non-preemptive list scheduling of $P|\text{prec}, p_j = 1|C_{\max}$. We have analyzed the worst behavior of arbitrary list scheduling strategies, the ISF and MISF heuristics and presented their performance guaranties under the four different assumptions. We have shown that additional knowledge such as the total number of tasks, number of tasks without successors and the length of the longest chain in the task system allow a more accurate analysis of the performance bounds. Our new bounds reach the maximum value of Graham's bound in the point $m = n/l_c$, and tend to 1 while increasing the number of machines. For both regions, $m < n/l_c$ and $m > n/l_c$, bounds (1) and (2) are better.

We provided simulation results comparing the effectiveness of MISF against previously known dynamic heuristics. The experiments confirm an advantage of the MISF in the average case. The measurements are based on randomly generated out-forests, to cover a wide variety of tree char-

Table 4
Average performance ratio and percentage of the performance ratio standard deviation

n	MLF		MISF		MICF		MACF		ARB	
	PR	S.D. (%)	PR	S.D. (%)	PR	S.D. (%)	PR	S.D. (%)	PR	S.D. (%)
1000	1.0032	2.27	1.0260	4.41	1.0584	9.12	1.0582	9.31	1.0518	8.93

Table 5
Maximum and average percentage of MISF performance ratio improvement over MLF, MACF, MICF heuristics and arbitrary list schedule

n	MLF (%)		MICF (%)		MACF (%)		ARB (%)	
	Maximum	Average	Maximum	Average	Maximum	Average	Maximum	Average
1000	−7.85	−1.52	10.77	2.62	9.14	2.39	9.50	2.38

Table 6
Average parameters of task graphs generated

n	$\max(n_i)$	l_c	# Successors	n^0
1000	212.71	413.95	136.29	213.73

acteristics. This simulation results for dynamic scheduling are close to those of [20] for static scheduling for linear Algebra DAGs that used a modification of the known critical path-most immediate successor first (CP-MISF) heuristic, where, in contrast to our assumptions, information about the critical path is available. Experimental results in [20] show that CP-MISF is within 1% off the optimum. In our work, we assume that information about the critical path is not available, but the number of immediate successors is available for scheduling. The effectiveness of the MISF depends on several parameters of the task system; the key point is a sufficiently large number of tasks without successors. To fully validate the MISF heuristic for the average case, more accurate analysis of generation schemes for the computational testing that may not introduce biases into computational results should be done [21], and directed acyclic graphs (DAGs) generated from real world parallel applications have to be tested.

Further more accurate measurements using MISF for arbitrary task graphs comparing with optimal solutions are currently under way. Future investigations will be carried out for the ISF and MISF heuristics in the context of non-unit execution times and with or without task preemption.

Acknowledgements

The authors take pleasure in acknowledging Denis Trystram for his useful suggestions during the preparation of the paper, and the anonymous referees whose valuable remarks and comments helped to improve the paper. Part of this work was supported by CICESE (Centro de Investigación Científica y de Educación Superior de Ensenada) under project #634102, and by CONACYT (Consejo Nacional de Ciencia y Tecnología de México) under grant #32989-A.

References

[1] A. Barak, S. Guday, R. Wheeler, The MOSIX Distributed Operating System. Load Balancing for UNIX, Lecture Notes in Computer Science, vol. 672, Springer, Berlin, 1993.
[2] J. Błażewicz, K.H. Ecker, E. Pesch, G. Schmidt, J. Węglarz, Scheduling Computer and Manufacturing Processes, Springer, Berlin, 2001.
[3] R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling theory: A survey, Annals of Discrete Mathematics 5 (1979) 287–326.
[4] J. Błażewicz, J.K. Lenstra, A.H.G. Rinnooy Kan, Scheduling subject to resource constraints: Classification and complexity, Discrete Applied Mathematics 5 (1983) 11–24.
[5] T.C. Hu, Parallel sequencing, and assembly line problems, Operational Research 9 (1961) 841–848.
[6] El-Rewini, T. Lewis, H. Ali, Task Scheduling in Parallel and Distributed Systems, Prentice-Hall, Englewood Cliffs, NJ, 1994.
[7] C.H. Papadimitriou, M. Yannakakis, Scheduling interval-ordered tasks, SIAM Journal on Computing 8 (1979).
[8] N.F. Chen, C.L. Liu, On a class of scheduling algorithms for multiprocessors computing systems, in: T.Y. Feng (Ed.), Parallel Processing, Lecture Notes in Computer Science, vol. 24, Springer, Berlin, 1975, pp. 1–16.
[9] M. Kunde, Nonpreemptive LP-scheduling on homogeneous multiprocessor systems, SIAM Journal on Computing 10 (1) (1981).
[10] E.G. Coffman Jr., R.L. Graham, Optimal scheduling for two-processor systems, Acta Informatica 1 (1972) 200–213.
[11] M.R. Garey, D.S. Johnson, Two-processor scheduling with start times and deadlines, SIAM Journal on Computing 6 (1977) 416–426.
[12] K.H. Ecker, Scheduling of resource tasks, European Journal of Operational Research 115 (1999) 314–327.
[13] R.L. Graham, Bounds for certain multiprocessing anomalies, Bell System Technical Journal 45 (1966) 1563–1581.
[14] B. Braschi, D. Trystram, A new insight into the Coffman–Graham algorithm, SIAM Journal on Computing 23 (1994) 662–669.
[15] R.L. Graham, Bounds on multiprocessor timing anomalies, SIAM Journal on Applied Mathematics 17 (1969) 263–269.
[16] R.P. Brent, The parallel evaluation of arithmetic expressions in logarithmic time, in: J.F. Traub (Ed.), Complexity of Sequential and Parallel Numerical Algorithms, Academic Press, New York, 1973, pp. 83–102.
[17] E. Rodríguez Alcántar, Estudio Experimental de la Heurística MISF para Calendarización Dinámica de Tareas en Árboles y Bosques, CICESE, Ensenada, Baja California, México, 1999.
[18] A. Tchernykh, A. Stepanov, A. Lupenko, N. Tchernykh, Extraction and optimization of the implicit program parallelism by dynamic partial evaluation, in: pAs'97 The Second Aizu International Symposium on Parallel

Algorithms/Architecture Synthesis, IEEE Computer Society Press, Silver Spring, MD, 1997, pp. 322–338.
[19] A. Rodríguez Díaz, A. Tchernykh, K. Ecker, Non-deterministic scheduling in an abstract network machine, in: PCS'97 International Workshop on Parallel Computation and Scheduling, Ensenada, Baja California, Mexico, 1997, pp. 47–58.
[20] A. Gerasoulis, I. Nelken, Static scheduling for linear Algebra DAGs, in: Proceedings of the 4th Conference on Hypercubes, California, Monterey, Concurrent Computers and Application, vol. 1, 1989.
[21] N.G. Hall, M.E. Posner, Generating experimental data for computational testing with machine scheduling application. Operations Research, to appear.