RAIRO Operations Research
RAIRO Oper. Res. **37** (2003) 145-156
DOI: 10.1051/ro:2003018

SCHEDULING PRECEDENCE TASK GRAPHS WITH DISTURBANCES*

Apurv Gupta[1], Gilles Parmentier[1] and Denis Trystram[1]

Communicated by Jacques Carlier

Abstract. In this paper we consider the problem of scheduling precedence task graphs in parallel processing where there can be disturbances in computation and communication times. Such a phenomenon often occurs in practice, due to our inability to exactly predict the time because of system intrusion like cache miss and packet transmission time in mediums like ethernet etc. We propose a method based on the addition of some extra edges to protect the initial scheduling from performing badly due to such changes and provide an upper bound on the performance guarantee for the scheduling algorithms. Moreover, this construction guarantees a result at least as good as the result obtained for the initial static scheduling. We also show that this construction is a minimal set in context of partially on-line scheduling.

Keywords. Parallel processing, scheduling, stability, uncertainty, communication delays.

1. Introduction

This paper investigates the effects of disturbances on scheduling algorithms during the execution of a parallel program. This problem, which is one of the most challenging problem in Parallel Processing, corresponds to determine a date when

Received December, 2000.

* *This research was supported by the INRIA exchange program with indian Institute of Technology in 1999.*

[1] ID – IMAG, Antenne ENSIMAG, ZIRST, 51 avenue Jean Kuntzmann, 38330 Montbonnot Saint-Martin, France; e-mail: Denis.Trystram@imag.fr

© EDP Sciences 2003

a task will start its execution and a processor location where it will take place. For most applications, the computations can entirely be determined at compile time. For irregular applications, these times may be difficult to predict with accuracy. In the past it was often sufficient to run such a scheduling algorithm for obtaining good performances (according to what it was expected). However, the apparition of new parallel systems and execution supports have introduced some unpredictable behaviors appearing at run time. The multiple levels of memory hierarchy makes it also more difficult to predict the performances of local computations (inside the processors).

One of the most studied problem in parallel computing is the minimization of the parallel time, also called *makespan*, of the application described by n tasks subject to precedence constraints on m processors. This problem is known to be NP-complete in all but a few very restricted cases, see for example [1]. Weights, which represent the computation times, are associated to the tasks of the program, and whenever two tasks are scheduled on different processors and they need to communicate, a communication cost is associated to the edge between them.

There exist many efficient heuristics for the scheduling problem in the literature. They make use of the information regarding the weights of the tasks and the edges. But one issue that has not been addressed until recently is the *stability* of these algorithms, where there are imprecisions in the estimates of computation and communication times and these may change at run time.

For example, if the communication takes place over a network where there may be contention for transmission, the communication delays are non-deterministic. As regards to the computation time, the nodes may be multi users and system intrusions render our estimates only approximate. Also this value is only approximated in most cases by the intrusion of monitoring tools. Building a good theoretical model for such complex systems is a very intricate problem and the analysis is in general intractable [10].

The main question in this context is: *"What happens to a schedule if the initial estimates are not good?"*. People usually distinguish between static (at compile time) and on-line (at run time) policies. We propose here an intermediate approach where only estimations are known at compile time. From this partial knowledge, a scheduling solution can be computed statically and then, potentially corrected at run time. One possibility is simply to analyze the variation of the makespan relatively to the disturbances on the estimated values, which occur at run time. This is known as the *sensitivity analysis*. In this case, no further decisions are allowed on-line. There exists a nice general result from Gerasoulis and Yang [3] for the sensitivity, which establishes that if a scheduling algorithm has a performance guarantee of B, then, the sensitivity remains bounded by a linear factor of B. The coarser the grain of the task graphs, the closer to 1 is this factor.

Another way for this problem is *robustness*, find an *a priori* solution which behaves well whatever the on-line disturbances may be, for a certain known set of possible disturbances [6]. Robustness is an optimization problem, seeking for a solution with good performances and low sensitivity. However, some anomalies can not always be avoided, in these cases, we need a correction process.

In the opposite, stabilization is the process whereby the on-line scheduler ajusts the static schedule at run time. Some solutions have been proposed for guaranty-ing the respect of deadlines in relation to real-time systems, see for instance [8,9]. Recently, this work has drawn attention in the context of disturbances on the communication times by Guinand *et al.* [4]. This is one of the only existing practical method. Their approach is to use partially on-line algorithms (*i.e.* a 2-phases approach based on on-line sequencing after a statically fixed allocation of the tasks). The schedule is stabilized by adding extra edges between some tasks allocated inside the same processors. Although some theoretical works are presented in this work on specific task graphs, most of this study is based on experiments.

However, there have been only little theoretical investigations on this problem except the bound of the sensitivity of [3]. Also as far as we know, the issue of disturbances in computation time has not been addressed so far, except for some applications in the different context of real-time [7,9]. In this paper, we provide a theoretical framework for partially on-line scheduling algorithms and identify the minimal set of extra edges to add on the task graph that are necessary to ensure the performance guarantee under the severest disturbances.

We present in more details the problem and state the model precisely in Section 2. In Section 3, we describe the new stabilization algorithm and prove its performance guarantee. Finally, we conclude by presenting some perspectives.

2. Preliminaries

We consider the problem of scheduling a parallel application described by a directed acyclic graph $G = (V, E)$. There is a weight associated with each node of the graph representing the amount of computation time needed to perform that computation. The weights associated with the edges denote the time taken for communication, *i.e.* the data to exchange between processors. The set $V = \{1, \ldots, n\}$ describes the tasks to be scheduled and $E \subseteq V \times V$ the precedence constraints among them. Each task i must perform on a processor during time p_i without preemption. Data dependence between tasks i and j is represented by a directed edge $(i, j) \in E$. When tasks i and j are not assigned to the same processor, a communication delay C_{ij} is assumed for data transfer between these two tasks.

A schedule σ is an application such that each task $i \in V$ is assigned to be executed on a processor, denoted $\pi(i)$ and is associated a **starting time** S_i where the following holds:

$\forall i, j, (i, j) \in E, S_j \geq S_i + p_i + C_{i,j}$ where $C_{i,j} = 0$ if $\pi(i) = \pi(j)$.
Correspondingly, we define the **finish time** of task i by $F_i = S_i + p_i$.

For a given graph G, we denote by $\tilde{G}$ the graph whose structure is the same as that of G, it differs only in the weights associated with the nodes and the edges. These new weights are denoted respectively by $\tilde{p}_i$ and $\tilde{C}_{i,j}$.

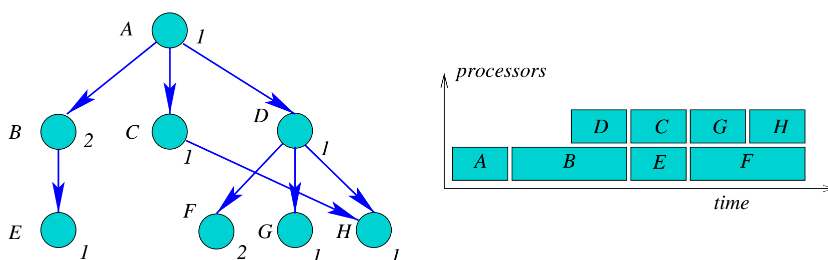

FIGURE 1. Illustration of a schedule for a given graph on two processors (the numbers beside the tasks represent the execution times and all the communications are set to 1).

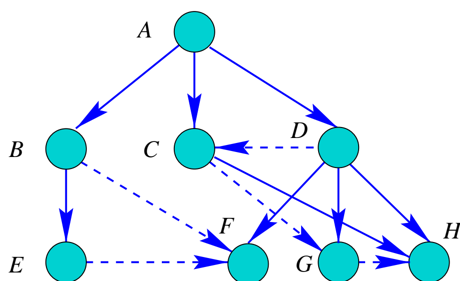

FIGURE 2. Execution graph (G_σ).

Figure 1 is a complete schedule example for a given graph.

Definition 1. Given a graph G and a schedule σ determined on it, we denote by G_σ the **execution graph** of G under σ obtained by adding edges from task i to j if i is executed before j on the same processor and there is not already an edge between task i and task j. These edges are called **pseudo-edges**. They are represented in dashed lines in Figure 2 on the same example as before.

Let us emphasize that the purpose of a pseudo-edge (i, j) is to ensure that task i gets executed before j even if there is no precedence constraint between them.

Note that G_σ has many transitive edges. It should be noted however that transitive edges do not hamper the flexibility of execution order. It would be sufficient to add an edge (i, j) if i is executed just before j. But the previous definitions will simplify the presentation of the proofs.

We denote by $PT(G_\sigma)$ the makespan of the execution graph G_σ. Note that a schedule also specifies the processor allocation for each task.

3. A NEW APPROACH

3.1. PRINCIPLE

Let us informally describe the principle of our new approach. To tackle the problem of scheduling with on-line disturbances, most of the classical existing scheduling methods like fully on-line scheduling and static scheduling are unsatisfactory.

For on-line policies, since the arrival time of a task is not known at compile time, this task can not be assigned to a processor and there is a need to exchange informations at run time, which is generally costly. We already discussed the limitations of the purely static approach as regards to disturbances. When some knowledge about the tasks is known, a 2-phases clustering approach seems more reasonable for coping up with the disturbances. Here, the assignment is determined at compile time and the sequencing is performed on-line on each processor according to a ready list. However, there are problems with this basic 2-phases approach. For instance, if a high priority task is not executed at the time determined by the static schedule, due to a slight communication delay, the makespan will increase beyond what could have been achieved with static execution, *i.e.* in order execution. Figure 3 shows such a case on the same example as in Figure 2 with one perturbated communication (between tasks A and C). Thus, the execution order of tasks D and C is inverted. Such cases can be avoided by adding pseudo-edges (the precise description will be detailed later). The idea is to wait for high-priority tasks to become ready like in the static schedule. In this case, an on-line sequencing policy seems to be able to cope up with the disturbances very well.

The proposed method based on the addition of adequate pseudo-edges would help to achieve the flexibility of on-line sequencing keeping the same performance guarantee as for off-line static schedules. A good performance can be achieved for static schedules since we can even pay a heavy price and use higher complexity algorithms because they are computed at compile time. If we are guaranteed to stay close to this schedule, we expect to obtain good performances even in case of disturbances occurring at run time.

The rest of the paper is devoted to characterizing such tasks and finding a minimal set of additional pseudo-edges.

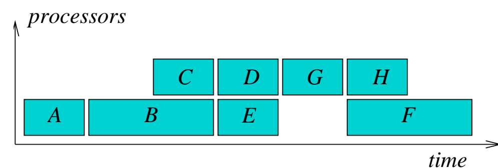

FIGURE 3. Illustration of a case where a basic 2-steps scheduling performs badly (tasks C and D are exchanged when the communication time between A and D increases slightly $(1+\epsilon)$).

3.2. Characterization of high-priority tasks

For the sake of clarification of further proofs, it is helpful to introduce some definitions:

Definition 2. For a given schedule σ, we call a task i **Permutable** *iff* the following two conditions hold:

- task i communicates with some task j scheduled on another processor. That is: $(i, j) \in E$ in G and $\pi_i \neq \pi_j$ in σ;
- there exists at least one task j that is scheduled in σ after task i on the same processor, and i and j are independent.

Definition 3. The set of tasks with which a permutable task i can possibly permutate is denoted as $\mathbf{P}(i)$.

In Figure 2, the communicating tasks are A and D. $\mathrm{P}(A)$ is empty, but we have $\mathrm{P}(D) = C$.

We claim that the only pseudo-edges that are necessary to add are those going from a permutable task i to tasks in $\mathrm{P}(i)$.

3.3. Description of the algorithm

The algorithm presented informally before can be expressed as follows:

- **Phase 1.** Compute an off-line Schedule using any well-known algorithm like ETF [5] or DSC [2].
  For each permutable task i, add pseudo-edges between i and the tasks of $\mathrm{P}(i)$.
- **Phase 2.** Each processor executes the tasks according to a ready-list maintained locally.

4. Theoretical analysis

4.1. Preliminaries

Let G be the estimated (given) graph and σ denote the schedule determined by the static scheduling algorithm in Phase 1 of the algorithm. Let $\tilde{G}$ be the disturbed graph with weights corresponding to the actual values of computation costs and communication delays, which are present at run time. Let $G_{\phi(\sigma)}$ denote the graph obtained by adding pseudo-edges in G from i to j iff i is a permutable task and $j \in \mathrm{P}(i)$. Figure 4 illustrates this definition on the graph of Figure 2. $\tilde{G}_{\phi(\sigma)}$ is defined as usual as the perturbated graph of $G_{\phi(\sigma)}$.

Some properties may be observed that will be helpful for the theoretical analysis.

We consider the set of tasks $1, \ldots, n_q$ scheduled on a single processor q. There may be some idle periods in processor q if no task has arrived and it can not execute any other local task. We consider the total order built after the addition of pseudo-edges between tasks that were scheduled successively if no edge already

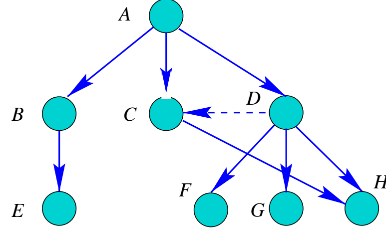

FIGURE 4. Graph $G_{\phi(\sigma)}$.

exists between them. This corresponds to the minimal linear extension of the initial execution order of σ. Without loss of generality we assume that the tasks are scheduled in the order $1, \ldots, n_q$.

Property 1. Any ready-list scheduling algorithm executed on one processor is optimal when no pseudo-edge has been added.

Property 2. If we remove all the pseudo-edges except those linked with one particular task, say i, and use ready-list scheduling, then the following holds:

- S_i (and hence F_i) does not increase;
- the makespan also does not increase.

Proof. The proof is easy and comes directly from Property 1. If we first remove all the edges emanating from the tasks scheduled before i, then the starting time of i does not decrease. This is because it may be considered as scheduling tasks $1, \ldots, i-1$. The makespan is less and hence, S_i does not increase. Now if the pseudo-edges emanating from tasks $i+1, \ldots, n_q$ are removed, then the starting time of i is not affected, since i would have been executed before any of these tasks start its execution. □

Property 3. If the availability date of some of the tasks is decreased, the makespan can not increase.

Proof. This is straightforward because we consider a single processor scheduling. □

Definition 4. We call **Reduction** the process of removing some pseudo-edges in a graph.

4.2. PERFORMANCE GUARANTEE

The theorem below gives a performance guarantee on the parallel time of our algorithm.

Theorem 1. *For any precedence tasks graph G scheduled by σ, we have:*

$$PT(\tilde{G}_{\phi(\sigma)}) \leq PT(\tilde{G}_{\sigma}).$$

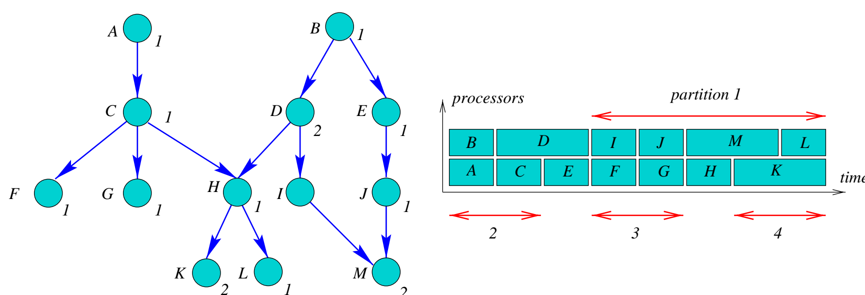

FIGURE 5. Illustration of the partitions introduced by communicating tasks.

Proof. We prove this result by applying a series of reductions, which consists of removing the pseudo-edges, on $\tilde{G}_\sigma$ to get $\tilde{G}_{\phi(\sigma)}$. At each of these steps, we will verify the required property that the parallel time of the reduced graph is not greater than $PT(\tilde{G}_\sigma)$. Actually, we shall prove a stronger assertion that each step of reduction, the parallel time cannot increase, which implies the required assertion.

To start with, choose any processor. The communicating tasks scheduled on this processor partition the set of tasks into ordered subsets, each of them having no communicating task. Figure 5 illustrates how to obtain these sets for a graph and an associated schedule (there are 4 communicating tasks, namely, B, D, E and H, leading to 4 ordered sets).

The process of reduction consists in removing all the pseudo-edges emanating from the tasks belonging to the partition. There are two cases to consider within the partitions depending if the resulting order is a total order or not. If it is a total order, the scheduling of the tasks in the partition is optimal from Property 1. If it is a partial order, the tasks are labelled in respect to the availability dates of the tasks that receive some data from communicating tasks (Fig. 6 illustrates this case on the first partition on the same example as before). This gives the priority for the ready list scheduling for Phase 2 of the algorithm. According to Property 3, the makespan will not increase.

Thus, the completion times will not increase for the tasks inside the partitions on each processor. Now, it remains to prove that the starting times of all the communicating tasks in the processor have also not increased. Let consider a communicating task, namely i. Again, there are two cases to consider if i receives or not some data from elsewhere.

- (a) The easy case is when i does not receive any communication. Then, i is a root of the graph or it follows directly a partition on its processor. The same arguments as before considering that i belongs to this partition proves that its starting time can not increase.
- (b) If i receives data from another communicating task (say k), if it is it-self a successor of a communicating task, we proceed by a backwards

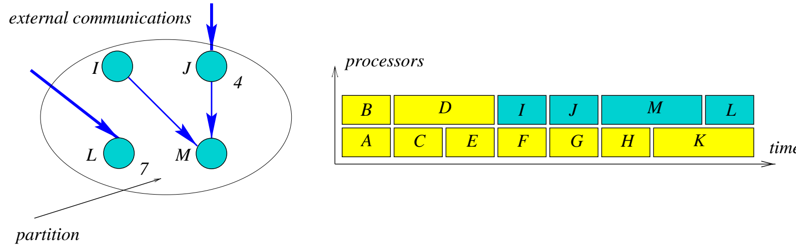

FIGURE 6. Labelling of the tasks inside a partition according to their availability dates when the resulting order is not a total order.

induction until reaching a task that receives no data from elsewhere, and then, we come back to case (a). □

Let us now consider the implications of Theorem 1. The theorem, combined with the bound in [3], implies that there is a guarantee on our algorithm. On the other hand, let remark that since we have been able to relax many of the constraints in $\tilde{G}_{\phi(\sigma)}$ compared to $\tilde{G}_\sigma$, we will obtain an improved performance.

In addition, we will also prove in the next section that this set of pseudo-edges is minimal set.

4.3. Minimal set of additional constraints

Theorem 2. *The set of pseudo-edges emanating at the Permutable tasks is a minimal set of pseudo-edges to guarantee the bound of Theorem 1. In other words, no pseudo-edge from $G_{\phi(\sigma)}$ can be removed without hindering the bound of Theorem 1.*

Proof. The proof consists in showing that if a pseudo-edge from task i to task j were to be removed in $G_{\phi(\sigma)}$, there exists a possible configuration of the resulting graph $\tilde{G}_{\phi(\sigma)}$ such that the $PT(\tilde{G}_{\phi(\sigma)}) > PT(\tilde{G}_\sigma)$.

Consider a pseudo-edge from i to j in our construction, then task j could have executed before i if the availability date of i were to be delayed beyond the availability date of j. Consider such a disturbance. Let k be one of the tasks to which task i were communicating (for instance, tasks D, C and F of Fig. 1 are such tasks).

The permutation between i and j will delay the execution of k. The resulting effect will be to increase k of at least p_j. If the new makespan is still lower than $PT(\tilde{G}_\sigma)$ (this will be the case for instance for the graph of Fig. 3 with $p_k = 1$), then, consider the new perturbation on p_k leading to a makespan greater than $PT(\tilde{G}_\sigma)$. Clearly, this perturbation violates the guarantee of Theorem 1, thus the set of pseudo-edges is minimal. □

5. Experiments

5.1. Context

We report in this section some experiments obtained from simulations for assessing the good practical behaviour of the strong stability algorithm (denoted by SSA) versus ETF or any basic on-line algorithm (denoted by ON-LINE). The experiments have been carried out on a usual PC machine. Each test has been performed over 1000 randomly generated instances. The structures of the precedence task graphs have been created by layers as in [2] for DSC within the parallel programming environment PYRROS: the number of layers is a random number and the number of connected nodes between the layers is randomly chosen and decreases as the distance between the layers increases. The number of processors is fixed and did not show a big importance. The size of the graphs was taken from 100 to 500, depending on the number of processors. The percentage of over and under perturbances was fixed and can vary over an interval of magnitude. For instance, for 100 tasks, we considered 80% of disturbances, 20 tasks are over estimated and 60 tasks are under estimated. We tried other ratios (like inverting proportions of over and under estimated tasks) but results are the same. The simulations were conducted in both cases of balanced communications and large communication delays.

We concentrate on large communication times (ten times larger than the execution times) as it is the most important case in practice. In each table we report the number of times SSA is better than the other algorithms for 1000 executions. We consider small and large disturbances. Small perturbations means that each time t (communication, computation and both) can be disturb with a probability of 0.8 in the interval $[\frac{3t}{4}, \frac{6t}{5}]$. Similarly, large perturbations means that each time t can be disturb with a probability of 0.25 in the interval $[\frac{t}{2}, 2t]$. As said previously, the distribution of the disturbances within the intervals is not homogenous (25% of over-estimated and 75% of under-estimated times).

The analysis of the experimental results is detailed below.

5.2. Analysis

We focus on the behaviour of SSA in regard to a pure static algorithm, namely ETF and a simple greedy on-line algorithm. In the following tables, we report the number of times SSA performs better than the other algorithm, among 1000 instances.

- The experiments show that there is no significant differences between SSA and ETF for communications and computations in the same order of magnitude (equal communication and computation times).
- As shown in Table 1, left, SSA is slightly better than ETF for small graphs and much better for larger graphs.
- SSA performs much better than ON-LINE (see Tab. 1, right). The left subcolums correspond to the number of times where SSA is better than

ON-LINE. The right ones correspond to the number of times where ON-LINE is better than SSA. In remaining cases, both algorithms perform the same. It seems that ON-LINE performs slightly better on bigger graph, in this case it is better than SSA in 30% of the case instead of 20% of the case for small graphs.

Table 1. Left, SSA *vs.* ETF, large communication times (range 1−10), right, SSA *vs.* ON-LINE, large communication times (range 1−10).

pert.	small	large
$n = 100$, $m = 8$		
	+	+
comm.	105	115
comp.	140	129
both	125	126
$n = 500$, $m = 8$		
comm.	440	417
comp.	417	417
both	445	401
$n = 500$, $m = 16$		
comm.	396	443
comp.	426	403
both	440	415

pert.	small		large	
$n = 100$, $m = 8$				
	+	−	+	−
comm.	625	217	618	215
comp.	626	229	627	226
both	667	187	620	229
$n = 500$, $m = 8$				
comm.	596	361	629	317
comp.	633	327	614	335
both	604	359	617	335
$n = 500$, $m = 16$				
comm.	625	321	620	329
comp.	590	363	602	362
both	593	350	593	351

6. Conclusion

We have presented in this work an approach to deal with on-line disturbances in computation and communication times with an upper bound on its performance. The process is quite simple and, as shown in the simulation experiments, can be useful for correcting the bad effects of disturbances. Remark that only few results exist in this direction today. This work intents to investigate this field.

Some promising results have been achieved, however, the investigation is in a preliminary stage. It remains to study if techniques with better performance bounds may be obtained in the case where a limit on the maximum disturbance is assumed to be known at compile time. Another interesting perspective is to study the average behavior of heuristics that are less constrained than the solution we proposed in this paper (for instance by allowing more flexibility while adding pseudo-constraints). It is also of interest to investigate methods that may allow to change the processor assignment during run time without loss of efficiency. Another investigation direction is to study the impact of the initial scheduling algorithm.

References

[1] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt and J. Weglarz, *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, 3rd edn. (1996).

[2] A. Gerasoulis and T. Yang, Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel Distrib. Syst.* **5** (1994) 951-967.

[3] A. Gerasoulis, J. Jiao and T. Yang, Applications of graph scheduling techniques in parallelizing irregular scientific computation, in *Parallel Algorithms for Irregular Problems: State of the Art*, edited by A. Ferreira and J.D.P. Rolim, Chapter 13. Kluwer Academic Publishers, Netherlands (1995) 245-267.

[4] F. Guinand, A. Moukrim and E. Sanlaville, Scheduling With Communication Delays and On-Line Disturbances, in *Proc. of the European Conference on Parallel Computing, EuroPar'99, Aug. 31–Sept. 3, Toulouse (France).* Springer-Verlag, *Lecture Notes in Comput. Sci.* **1685** (1999).

[5] J.-J. Hwang, Y.-C. Chow, F.D. Anger and C.-Y. Lee, Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.* **18** (1989) 244-257.

[6] P. Kouvelis and G. Yu, *Robust Discrete Optimization and its Applications.* Kluwer Academic Publishers (1997).

[7] R.M. Kieckhafer, J.S. Deogun and A.W. Krings, The performance of inherently stable multiprocessor list scheduler. *Real Time Syst.* **15** (1998) 5-39.

[8] A.W. Krings and M. Dror, Real-time dispatching: Scheduling stability and precedence. *Int. J. Found. Comput. Sci.* **10** (1999) 313-327.

[9] G.K. Manacher, Production and stabilization of real-time task schedules. *J. ACM* **14** (1967) 439-465.

[10] C. Papadimitriou and M. Yannakakis, Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.* **19** (1990) 322-328.

To access this journal online:
www.edpsciences.org