RAIRO Operations Research
RAIRO Oper. Res. **37** (2003) 1-16
DOI: 10.1051/ro:2003011

PARALLEL MACHINE SCHEDULING WITH UNCERTAIN COMMUNICATION DELAYS *

Aziz Moukrim[1], Eric Sanlaville[2] and Frédéric Guinand[3]

Communicated by Bernard Lemaire

Abstract. This paper is concerned with scheduling when the data are not fully known before the execution. In that case computing a complete schedule off-line with estimated data may lead to poor performances. Some flexibility must be added to the scheduling process. We propose to start from a partial schedule and to postpone the complete scheduling until execution, thus introducing what we call a stabilization scheme. This is applied to the m machine problem with communication delays: in our model an estimation of the delay is known at compile time; but disturbances due to network contention, link failures, ... may occur at execution time. Hence the processor assignment and a partial sequencing on each processor are determined off-line. Some theoretical results for tree-like precedence constraints and an experimental study show the interest of this approach compared with fully on-line scheduling.

Keywords. Parallel computing, scheduling with communication delays, disturbances on communication delays, list scheduling, flexibility.

Mathematics Subject Classification. 90B35, 90B25.

Received October, 2001.

* *An earlier partial version of this work has been presented at EUROPAR 99.*

[1] HeuDiaSyC, UMR 6599 du CNRS, Université de Technologie de Compiègne, Centre de Recherches de Royallieu, BP. 20529, 60205 Compiègne Cedex, France;
e-mail: Aziz.Moukrim@hds.utc.fr

[2] LIMOS, UMR 6158 du CNRS, Université de Clermont-Ferrand 2, Campus des Cézeaux, 63177 Aubière Cedex, France; e-mail: Eric.Sanlaville@math.univ-bpclermont.fr

[3] LIH, Université du Havre, 25 rue Philippe Lebon, BP. 5405, 76058 Le Havre Cedex, France; e-mail: Frederic.Guinand@univ-lehavre.fr

1. Introduction

The classical m machine scheduling problem is central for parallel computing issues. However it is essential to introduce the communication delays between tasks executed on different processors. Several models including communication delays for both shared and distributed memory multiprocessor systems have been proposed [1, 16]. If two tasks T_i and T_j are executed by two different processors, there is a delay between the end of T_i and the beginning of T_j due to data transfer between the two processors. The problem is NP-complete even for unit execution and communication times (the UECT problem), on an arbitrary number of processors or on an unlimited number of processors (see the pioneering work of Rayward–Smith [17], or the survey of Chrétienne and Picouleau [3]). However, some problems were found to be polynomial, especially for tree-like precedence constraints (see for instance [2, 9]).

In many models the communication delays only depend on the source and destination tasks, not on the communication network. The general assumption is that this network is fully connected, and that the lengths of the links are equal [7, 18]. This is rarely the case for a real machine: the network topology and the conflict of the communication links, may largely influence the delays, not to speak of communication failures. Some scheduling methods take into account the topology as in [11]. It is also possible to use more precise models (see [15, 19] for simultaneous scheduling and routing) but the performance analysis is then very difficult. In general, building an accurate model of the network is much complicated and entails a very intricate optimization problem.

A possible answer is to add an element of uncertainty in the model concerning the communication delays. This is the chosen approach in this paper: an estimation of the communication delays is known at compile time, allowing to compute a schedule. But the actual delays are not known before the execution. Building the complete schedule before the execution is then inadvisable. Conversely, postponing it to the execution time proves unsatisfactory, as we are then condemned to somewhat myopic algorithms. The method presented is a trade-off between these two approaches.

Scheduling with uncertainties on the data was not studied until recently. There is however a large amount of work concerning operations research problems with uncertain data, but it is mainly concerned with sensitivity analysis as introduced for linear programming. Recently, Kouvelis and Yu [13] proposed a unified approach for coping with uncertainty. For scheduling, we are concerned with robustness (or stability) and flexibility. A scheduling algorithm is robust, or stable, if its performances are not too much affected by disturbances on the data (see [8] and [20]). Flexibility means it is acceptable to modify a previously computed schedule to adapt it to the real data.

To our knowledge, there is no previous work fully devoted to dealing with uncertainties in parallel machine scheduling. However, in [8], the authors give some preliminary sensitivity results concerning the maximum performance degradation of any algorithm, depending on the maximum and minimum ratios between real

and estimated communication delays. Kolen *et al.* [12] measure the sensitivity of some list schedules for parallel machines case without communications. Their sensitivity criterion is the number of task assignment changes. By contrast, there are numerous works on workshop environments. Kouvelis and Yu [13] might be a good introduction. Remark that Wu *et al.* [21] apply to the job shop (criterion: weighted tardiness) ideas similar to those we independently present in this paper, that is fixing off line a partial order among the tasks.

The paper is organized as follows. In Section 2, the model is stated precisely, and the different algorithmic approaches are presented. The choice of two phase methods, processor assignment then sequencing, is justified. Section 3 presents a new partially on-line sequencing policy adapted to on-line disturbances. It is coupled with an assigment algorithm from the deterministic case to provide a flexible scheduling algorithm, what we call a stabilization scheme. Sensitivity analyses for special cases are presented in Section 4 (fork and join graphs) and in Section 5 (tree, unlimited number of processors). An experimental comparison is conducted and interpreted in Section 6: for a fixed assignment, our approach is compared with on-line scheduling.

2. Preliminaries

2.1. Model and definitions

We consider the schedule of a set of n tasks $V = \{T_1, \ldots, T_n\}$ subject to precedence relations denoted $T_i \prec T_j$, on m identical processors. $G = (V, \prec)$ is the task graph. Its height $h(G)$ is the number of arcs in its longest path. Preemption (the execution of a task may be interrupted) and task duplication are not allowed. One processor may not execute more than one task at a time but can perform computations while receiving or sending data. The duration of task T_i is p_i. If two tasks T_i and T_j verify $T_i \prec T_j$ and are executed on two different processors, there is a minimum delay between the end of T_i and the beginning of T_j. The communication delay between tasks executed on the same processor is neglected. At compile time, the communication delays are estimated as $\tilde{c}_{ij}$ time units. It is expected however (see Sect. 6) that these estimations are well correlated with the actual values c_{ij} (communication delays at execution time). The goal is to minimize the maximum task completion time, or *makespan*.

A schedule is composed of the *assignment* of each task to one processor and of the *sequencing* (or ordering) of the tasks assigned to one processor. A scheduling algorithm provides a schedule from a given task graph and a given processor network. We shall distinguish between *off-line* algorithms and *on-line* algorithms. An algorithm is off-line if the schedule is determined before the execution begins. An algorithm is on-line if the schedule is built during the execution; the set of rules allowing to build the final schedule is then called a policy. In our model, the communication delays c_{ij} are only known at execution time, once the data transfer is completed. Hence, for a given schedule, the beginning times of the tasks are not known before the execution.

In order to take into account the estimated delays a trade-off between off-line and on-line scheduling consists in *partially on-line scheduling*, that is, after some off-line processing, the schedule is built on-line.

Note that we are not within the framework of (strong) on-line scheduling (see for instance [6]) where nothing is known about the task durations themselves before their execution. Then all policies, even for independent tasks, will have bad worst case performance ratios.

2.2. Different approaches for scheduling with communication delays

Basically, there are two types of methods, one phase methods that compute the assignment and the sequencing simultaneously, and two phase methods that first compute the assignment, then the sequencing for each processor.

2.2.1. *List-scheduling approaches*

Without communication delay the so called List Schedules (LS) are often used as they provide good average performances, even as the worst case performance ratios are bad. Remember LS schedules are obtained from a complete priority order of the tasks. In most cases the choice is based upon Critical Paths (CP) computation. When a processor is idle, the ready task (all its predecessors are already executed) of top priority is executed on this processor. A way to tackle scheduling with communication delays is to adapt list scheduling. Then the concept of ready task must be precised. A task is *ready on processor* π at time t if it can be immediately executed on that processor at that time (all data from its predecessors have arrived to π). This extension is called ETF for Earliest Task First scheduling, following the notation of Hwang *et al.* [11]. The algorithm of Coffman and Graham [4] has been adapted in [10] and its performances analyzed.

2.2.2. *Clustering approaches*

Another proposed method is the clustering of tasks to build a pre-assignment [7, 18]. The idea is to cluster tasks between which the communication delays would be high. Initially, each task forms a cluster. At each step two clusters are merged, until another merging would increase the makespan. When the number of processors m is limited, the merging must continue until the number of clusters is less than or equal to m. The sequencing for each processor is then obtained by applying the CP rule.

More recently, Djordjevic and Tosic [5] proposed a new approach called chaining, trying to combine modified list scheduling and clustering. It is a one phase method as ETF. But for each processor, the position of a task in the sequencing is not determined before the algorithm termination. The time complexity of this method is significantly large.

2.2.3. *Limits of these approaches*

The above methods suppose the complete knowledge of the communication delays. Now if unknown disturbances modify these durations, one may question their efficiency. ETF schedules might be computed fully on-line, at least for simple priority rules derived from the critical path. But the first drawback is the difficulty to build good priority rules before the assignment. Moreover, fully on-line policies meet additional problems: if the communication delays are not known precisely at compile time, the ready time of a task is not known before the communication is achieved. But if this task is not assigned yet, the information relevant to that task should be sent to all processors, to guaranty its earliest starting time. This is of course very costly in time and memory space, and may lead to network contention.

The clustering approach would better fit our needs since the assignment might be computed off-line, and the sequencing on-line. But it suffers from other limitations. The merging process will result in large makespans when m is small. It is also poorly suited to different distances between processors.

It follows from the above discussion that the best idea is to compute the assignment off-line by one method or another (anyway, finding the optimal assignment is known to be NP-complete [18]). Then the sequencing is computed on-line so that the impact of communication delay disturbances is minimized. The on-line computation should be very fast to remain negligible with regard to the processing times and communication delays themselves. But it can be distributed, thus avoiding fastidious information exchanges between any processor and some "master" processor. Note that finding the optimal schedule when the assignment is fixed is NP-hard even for 2 processors and $UECT$ hypotheses [3].

3. An algorithm for scheduling with on-line disturbances on communication delays

3.1. Stabilization scheme

For a fixed assignment, the natural way to deal with on-line disturbances on communication delays is to apply a fully on-line sequencing policy based on ETF. After all communication delays between tasks executed on a same processor are zeroed, relative priorities between tasks may be computed much more accurately.

It is expected however that this approach will result in bad choices. Suppose a communication delay is a bit larger than expected, so that at time t a task with high priority, say T_i, is not yet available. A fully on-line policy will not wait. It will instead, schedule some ready task T_j that might have much smaller priority. If T_i is ready for processing at $t + \epsilon$, for ϵ arbitrarily small, its execution will nonetheless be postponed until the end of T_j.

We propose the following general approach. It aims at stabilizing, that is, reducing the effect of disturbances on, the schedule computed off-line.

Step 1. Compute an off-line schedule based on the $\tilde{c}_{ij}$'s.

Step 2. Compute a partial order $\prec_p$ including $\prec$, by adding precedences between tasks assigned to a same processor.

Step 3. At execution time, use some ETF policy to get a complete schedule considering assignment of Step 1 and partial order of Step 2.

The choices done for each step in the rest of the paper are now detailed.

3.2. Detailed partially on-line algorithm

The schedule of Step 1 is built by an ETF algorithm, based on Critical Path priority. We said that Critical Path based policies seemed best suited, as shown by empirical tests with known communication delays [22], and also by bound results (see [11] and [10]). These empirical tests, and ours, show that the best priority rule is the following: for each task T_i compute $L^*(i)$, the longest path that starts at T_i, including processing times and communication delays, the processing time of the task without successor in the path but **not** the processing time of T_i. The priority of T_i is proportional with $L^*(i)$. The heuristic that sequences ready tasks using the above priority is called RCP^* in [22] (RCP if the processing time of T_i is included).

The partial order of Step 2 is obtained as follows. Suppose two tasks T_i and T_j are assigned to the same processor. If the two following conditions are respected:

(1) T_i has larger priority than T_j for RCP^*;
(2) T_i is sequenced before T_j in the schedule of step 1;

then a precedence relation is added from T_i to T_j. This will avoid that a small disturbance leads to execute T_j before T_i at execution time.

In Step 3, RCP^* is again used as sequencing on-line policy to get the complete schedule.

The resulting algorithm is called $PRCP^*$ for Partially on-line sequencing with RCP^*. The algorithm for which the sequencing is obtained Fully on-line by RCP^* is denoted by $FRCP^*$.

3.3. Example

Figures 1 and 2 show how our approach can outperform both fully off-line and fully on-line sequencing computations. A set of 11 unitary tasks is to be scheduled on two processors. All estimated communication delays are 1. Schedule a) is obtained at compile time by RCP^*. The resulting assignment is kept for scheduling with the actual communication delays. Schedule b) supposes the sequencing is also fixed, regardless of on-line disturbances (fully off-line schedule). Schedule c) is obtained by $PRCP^*$, and schedule d) by $FRCP^*$.

After zeroing the internal communication delays, only $\tilde{c}_{24}$, $\tilde{c}_{45}$, and $\tilde{c}_{49}$ remain (bold arcs). The following precedences are added by our algorithm: for processor P_1, $T_4 \prec_p T_7$, $T_4 \prec_p T_{11}$, and for processor P_2, $T_8 \prec_p T_5$, and $T_8 \prec_p T_{10}$ (dashed arcs). Consider now the following actual communication delays: $c_{24} = 1.25$, $c_{45} = 1.50$, and $c_{49} = 0.70$. At time 2, task T_4 is not ready for processing, but T_7 is. $FRCP^*$ executes T_7 whereas $PRCP^*$, due to the partial order $\prec_p$, waits for T_4. At time 4

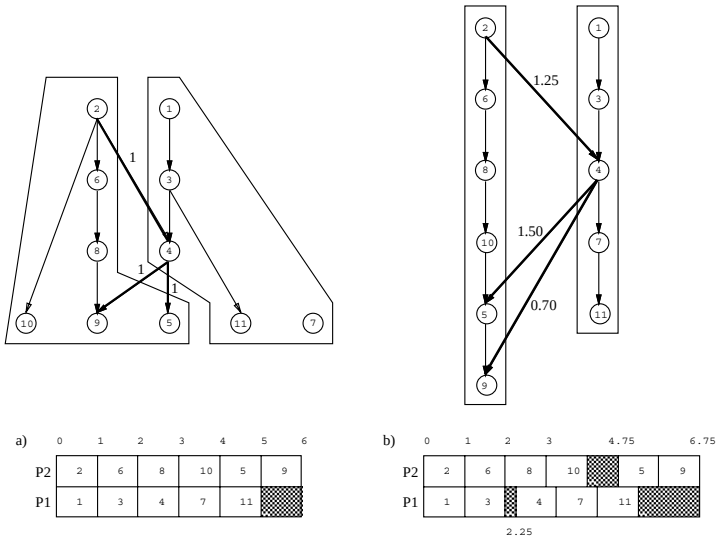

FIGURE 1. RCP^* schedule at compile time and associated completely off-line schedule.

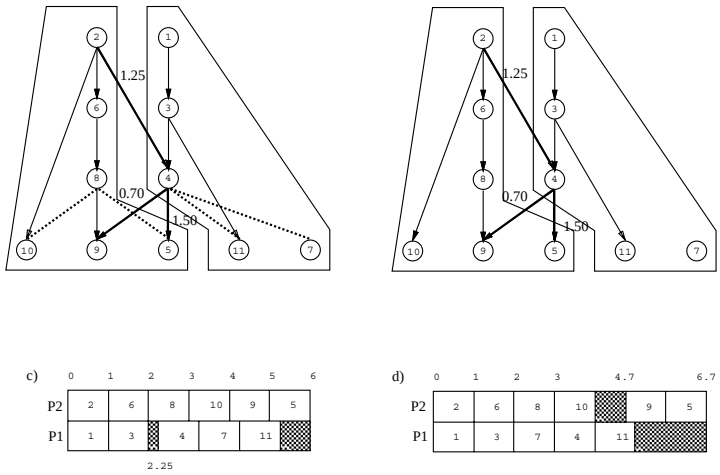

FIGURE 2. $PRCP^*$ and $FRCP^*$ schedules.

for $PRCP^*$, processor P_1 is available, and T_9 is ready. No additional precedence was added between T_5 and T_9 as they have the same priority, hence T_9 is executed and then T_5 is ready, so that P_2 has no idle time. For $FRCP^*$, exchanging T_7 and T_4 results in idleness for P_1, as T_4 is indeed critical.

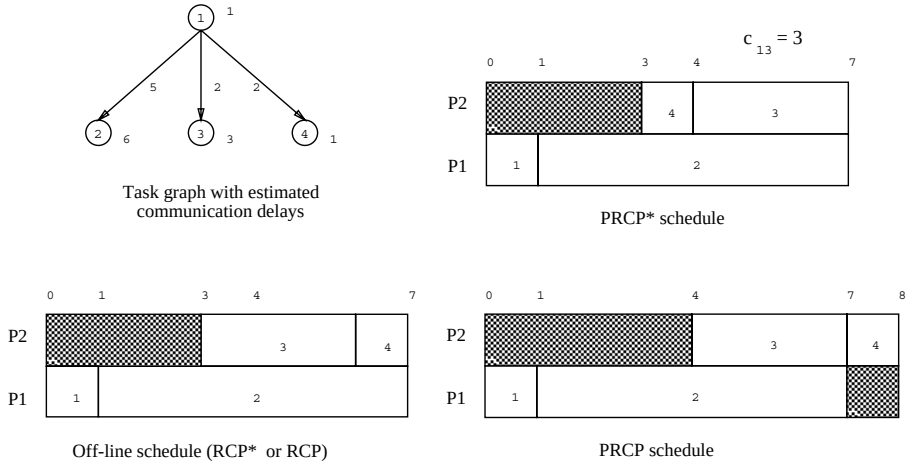

FIGURE 3. *PRCP* versus PRCP* for a fork graph

In what follows we consider the merits of the different strategies for sequencing, once the assignment has been fixed. Hence optimality is to be understood within that framework: minimum makespan among all sequencing policies, for a fixed assignment. There is little hope to obtain theoretical results (optimality, bounds) for anything but very special cases. Some of them are presented in Sections 4 and 5.

4. Optimality of $PRCP^*$ for fork and join graphs

4.1. Fork graphs

Fork task graphs are considered (see Fig. 3). One task T_1 is the immediate predecessor of all the others, which are final tasks. In the case of fixed communication delays, RCP or RCP^* find the minimum makespan (see [22]).

Adding disturbances do not much complicate. After the assignment a group of final tasks is to be sequenced on the same processor as T_1, say P_1. All communication delays are zeroed, hence the completion time on P_1 is the sum of the completion times of all these tasks, and is independent of the chosen policy. Consider now a group of final tasks executed on another processor. Each has a ready time (or release date), $r_i = p_1 + c_{1i}$, and a processing time p_i. Minimizing the completion time on this processor is equivalent to minimizing the makespan of a set of independent tasks subject to different release dates on one machine. This easy problem may be solved by sequencing the tasks by increasing release dates and processing them as soon as possible in that order. If a fully on-line policy is used, it will do precisely that, whatever priority is given to the tasks. On the other

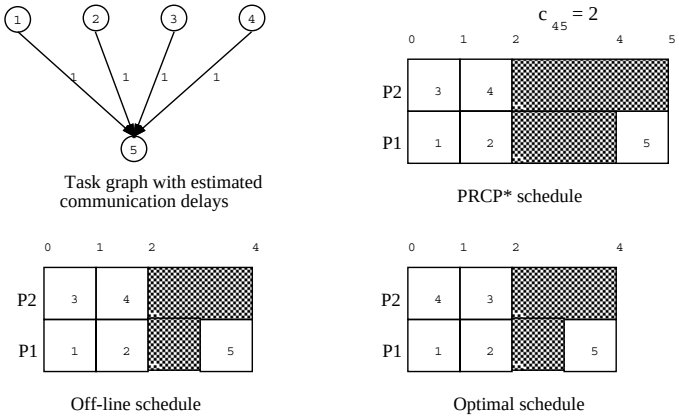

FIGURE 4. Example of join graph.

hand, a partially off-line algorithm may add some precedence relations between these tasks. This may lead to unwanted idleness. Indeed, suppose it is used with RCP. It may enforce a precedence between T_i and T_j if $\tilde{c}_{1i} \leq \tilde{c}_{1j}$, and $p_i > p_j$. During the execution you may have $c_{1i} > c_{1j}$, and the processor may remain idle, waiting for T_i to be ready. But if RCP^* is used, all priorities of the final tasks are equal, hence no precedence will be added, which guaranty optimality. This result is stated as a theorem below:

Theorem 1. *A $PRCP^*$ sequencing is optimal for a fixed assignment when the task graph is a fork graph.*

The difference between $PRCP^*$ and $PRCP$ is illustrated in Figure 3: suppose $c_{13} = 3$ and c_{14} is unchanged, then exchanging T_4 and T_3 is allowed by $PRCP^*$, not by $PRCP$.

4.2. Join graphs

RCP^* is optimal for join task graph (simply reverse the fork graph, see Fig. 4) and fixed communication delays, whereas RCP is optimal only if all tasks have the same duration (see [22]). However this is no longer true if the communication delays are allowed to vary. In fact in that case there is no optimal policy. The sequencing of the initial tasks must be done using the $\tilde{c}_{in}$'s. During the execution the respective order of these communication delays may change, thus implying a task exchange to keep optimality; but of course the initial tasks are already processed. This remains true even if restrictive hypotheses are done on the size of the disturbances, and is illustrated in the example of Figure 4: all initial durations are 1, but during the execution one communication delay increases, $c_{45} = 2$. Only a "lucky guess" can find the optimal schedule.

However if some local monotonicity property is verified by expected and actual communication delays, $PRCP^*$ is optimal. Indeed consider the following property

on the disturbances (task T_n is final):

(T_i and T_j are assigned to the same processor and $\tilde{c}_{in} \leq \tilde{c}_{jn}$) $\Rightarrow$ $c_{in} \leq c_{jn}$.

Theorem 2. *If the communication delays respect the local monotonicity property, then $PRCP^*$ sequencing is optimal for a fixed assignment.*

Proof. Any sequencing respecting the non increasing order of the $\tilde{c}_{in}$'s on all processors is optimal, as the induced release date for T_n is then minimized. $PRCP^*$ respects this order for the $\tilde{c}_{ij}$'s, hence for the c_{in}'s because of the property. □

This property may hold when, for instance, the disturbances depend only on the source and target processors.

A natural extension of fork and join graphs is the trees. However the problem is already NP-complete in the UECT case and an arbitrary number of processors (see [14]). As mentionned earlier, things are more encouraging when the number of processors is supposed to be infinite.

5. Tree-like precedence constraints and unlimited number of processors

In this section, we consider an unlimited number of processors. The precedence graph G is an in-tree.

For any off-line schedule S of G computed with the estimated communication delays $\tilde{c}_{ij}$, we will denote by $\tilde{\omega}(S)$ the makespan of S and $\omega(S)$ the makespan computed with the actual communication delays c_{ij} of the schedule determined by our algorithm $PRCP^*$ based on the off-line schedule S. ω^* (resp. $\tilde{\omega}^*$) will denote the optimal makespan of G with the actual communication delays (resp. estimated communication delays).

In this section some additionnal definitions are used.

size of the disturbance: $\epsilon = \max_{i,j}(c_{ij} - \tilde{c}_{ij})$

number of leaves of G: $l(G)$, the number of tasks without predecessor in G where G is an in-tree.

communicating arc for S: an arc of G whose extremities are not executed on the same processor in S.

linear schedule: it assigns two independent tasks to different processors.

First, we will establish a result that is valid for any precedence graph.

Proposition 1. *Let G be a task graph with arbitrary estimated and actual communication delays. Then, for any linear schedule S, there exists a path $\mathcal{C}$ such that*

$$\omega(S) - \tilde{\omega}(S) \leq h_{\mathcal{C}} \times \epsilon$$

where $h_{\mathcal{C}}$ designates the number of communicating arcs in $\mathcal{C}$.

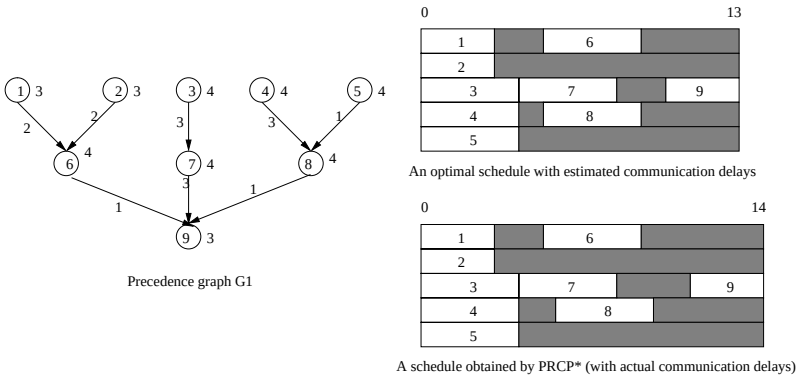

FIGURE 5.

Proof. Let G be a precedence graph with arbitrary estimated and actual communication delays and let S be a linear schedule. Denote $H_{\mathcal{C}}$ the set of communicating arcs of a path $\mathcal{C}$.

As the number of processors is unlimited, there exists a path $\mathcal{C}$ such that

$$\omega(S) = \sum_{i \in \mathcal{C}} p_i + \sum_{(i,j) \in H_{\mathcal{C}}} c_{ij}.$$

Now considering estimated delays we have:

$$\tilde{\omega}(S) \geq \sum_{i \in \mathcal{C}} p_i + \sum_{(i,j) \in H_{\mathcal{C}}} \tilde{c}_{ij}.$$

And finally:

$$\omega(S) - \tilde{\omega}(S) \leq \sum_{(i,j) \in H_{\mathcal{C}}} (c_{ij} - \tilde{c}_{ij}) \leq h_{\mathcal{C}} \times \epsilon. \qquad \square$$

When the task graph G is an in-tree with Small Communication Times ($\max_{i,j} \tilde{c}_{ij} \leq \min_i p_i$), Chrétienne (see [2]) shows that linear schedules are dominant, and proposes a polynomial time algorithm to determine an optimal schedule S^*. Suppose the in-tree is denoted by $\langle r; G_1, ..., G_q \rangle$ where r is the root of G and the G_k's are the subtrees of G whose roots are the tasks that immediately preceed r. Initialy, distinct processors are assigned to the leaves of G. Then, the algorithm proceeds recursively as follows: assuming that an optimal schedule is constructed for the different subtrees $G_k (k = 1, ..., q)$, the algorithm executes the root r after the root of a subtree G_{k^*} (on the same processor) such that the makespan is

minimum. One can immediately deduce the following properties of the resulting optimal schedule:

1. S^* is linear;
2. the number of processors that are used to obtain S^* is equal to $l(G)$;
3. each processor executes exactly a path of G that starts with a leaf.

Remark that this algorithm is in fact the application of ETF algorithm: each task is executed as soon as possible; there is no need for a priority rule as the number of available processors is unlimited. Now suppose S^* is used as the off-line schedule for $PRCP^*$. As it is linear, no precedence relation is added in step two of our stabilization scheme: the assignment *and* sequencing are fixed on-line.

Theorem 3. *Let G be an intree with estimated communication delays that are SCT (Small Communication Times). Then, the schedule S^* computed by Chretienne's algorithm is such that*

$$\omega(S^*) - \tilde{\omega}^* \leq \min(h(G), l(G) - 1) \times \epsilon.$$

Proof. Remember first that $\tilde{\omega}^* = \tilde{\omega}(S^*)$.

From Proposition 1, there is a path $\mathcal{C}$ such that

$$\omega(S^*) - \tilde{\omega}^* \leq h_{\mathcal{C}} \times \epsilon.$$

It follows from Property 3 and the fact that G is an intree that there is at most one communicating arc between any two processors. Therefore, for any path of G, the number of communicating arcs is strictly less than the number of used processors $l(G)$ (Property 2). It follows according to Proposition 1 that

$$\omega(S^*) - \tilde{\omega}^* \leq (l(G) - 1) \cdot \epsilon.$$

Of course $h_{\mathcal{C}} \leq h(G)$, and the result holds. □

Two examples (see Fig. 5 and Fig. 6) show that this bound is tight. G_1 and G_2 are two intrees where we have associated next to each task its processing time and next to each arc the estimated communication delay. The actual communication delays for G_1 are such that $c_{ij} = \tilde{c}_{ij}$ for any arc (i, j) except that $c_{58} = c_{89} = 1.5$. $h(G_1) = 2, l(G_1) = 5, \tilde{\omega}^* = 13, \omega(S^*) = 14$ and $\epsilon = 0.5$.

The actual communication delays for G_2 are such that $c_{ij} = \tilde{c}_{ij}$ for any arc (i, j) except that $c_{45} = 2.5$. $h(G_2) = 2, l(G_2) = 2, \tilde{\omega}^* = 5, \omega(S^*) = 6.5$ and $\epsilon = 1.5$.

Note that in the second example, actual communication delays are no more SCT. The theorem includes that case.

The above theorem bounds the makespan increasing of Chretienne schedule by a function of the perturbation and the graph sizes. In the important case where actual communication delays are larger than estimated ones (*e.g.*, in case

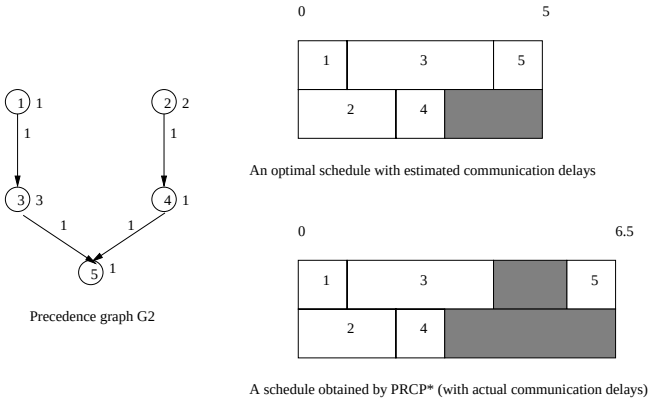

FIGURE 6.

of network contention), the following absolute bound holds:

Corollary 1. *Let G be an SCT-tree. If $c_{ij} \geq \tilde{c}_{ij}$ for any arc (i,j), then*

$$\omega(S^*) - \omega^* \leq \min(h(G), l(G) - 1) \cdot \epsilon.$$

Proof. In that case, $\omega^* \geq \tilde{\omega}^*$ which entails the result. □

The bound is tight. For instance if all processing times are equal to 1 in example 2, and $c_{45} = 1 + \epsilon = 2.5$, then $\omega(S^*) = 5.5$, and $\omega^* = 4 = \omega(S^*) - \epsilon$.

6. Experimental results for arbitrary task graphs

In this section, $PRCP^*$ and $FRCP^*$ are compared. The fully off-line approach is not considered, as it leaves no way to cope with disturbances.

As said previously, if the $\tilde{c}_{ij}$'s are too poor approximations of the actual communication delays, the policy of choosing any ready task for execution might prove as good as another. We chose to make the following assumptions: c_{ij} is obtained as $c_{ij} = \tilde{c}_{ij} + \pi_{ij}$, where π_{ij}, the disturbance, is null in fifty percent of the cases (after all, the communication delay might be correctly estimated sometimes!). When non zero, it is positive three times more often than negative (a disturbance being usually the result of a problem or a conflict during the message routing). Finally, its size is chosen at random and uniformly between 0 and 0.5. This insures that any disturbance is twice smaller than all execution times and communication delays, see below.

500 task graphs are randomly generated, half relatively wide with respect to the size of sets of independent tasks (wide graphs), half strongly connected, thus having small antichains and long chains. The results for the first case are presented. The *mean* numbers of edges are 103, 222, 335, 462, and 586 for graphs with respectively 50, 100, 150, 200 and 250 vertices. The durations are generated

TABLE 1. Results for wide task graphs, 3, 4 and 10 processors.

		LCT			*SCT*			*UECT*		
n	m	Mean	nb	Max	Mean	nb	Max	Mean	nb	Max
50	3	0.93	28	5.65	0.07	4	2.2	1.83	50	9.42
50	4	0.59	28	5.00	1.71	38	11.9*	2.64	50	9.13
50	10	0.44	22	4.83	0.11	12	1.25	0.23	6	4.40
100	3	1.35	68	5.17	0.04	16	1.50	1.17	50	8.01
100	4	0.77	48	7.10	2.16	66	7.79	2.26	76	5.33
100	10	0.24	20	2.61	0.47	34	3.04	0.87	38	4.91
150	3	0.75	54	5.47	0.34	48	1.39	0.99	60	4.87
150	4	0.85	52	3.44	2.45	86	6.57	1.88	90	4.36
150	10	0.18	16	1.72	0.25	20	1.69	0.47	24	5.33
200	3	0.53	66	4.32	0.14	34	1.57	1.01	72	3.83
200	4	0.85	62	3.37	1.23	92	5.38	2.13	84	5.01
200	10	0.39	36	2.73	0.63	48	3.90	0.90	62	3.20
250	3	1.14	68	5.01	0.28	64	1.00	0.46	44	5.19
250	4	0.37	56	3.03	1.73	92	4.68	2.15	96*	5.13
250	10	0.69	58	3.29	0.34	46	1.29	0.98	76	4.23

three times for a given graph to obtain LCT, SCT and UECT durations (Large and Small *estimated* Communications Times, and Unit Execution and estimated Communication Times, respectively). In the first case processing times are chosen uniformly in $[1, 5]$ and communication delays in $[5, 10]$, in the second case it is the opposite, in the third all durations are set to 1. For each graph, 5 duration sets of each type are tested.

The table displays the mean percentage of improvement of $PRCP^*$ with respect to $FRCP^*$, the number of times (in percentage) $PRCP^*$ was better, and the maximum improvement ratio obtained by $PRCP^*$.

$PRCP^*$ is always better in average. The number of times $PRCP^*$ is strictly better may reach 96% of the cases (example given, UECT, $n = 250, m = 4$). The mean improvements are significant, as the assignment is the same for both algorithms. Indeed an improvement of 5% is frequent and might be worth the trouble. A randomly generated case with improvement of 11.9% is reported.

The results are more significant in the UECT case. Note that it may be interesting to wait for a task with large priority, instead of immediately executing a ready task (no-wait schedules are no more dominant when on-line disturbances occur).

In the case of strongly connected graphs, the differences are less significant, as more often as not, there is only one ready task at a time per processor. Hence the full results are not presented. However when there are differences they are in favor of $PRCP^*$.

7. Conclusion

In this paper we considered a way to build schedules when the data are uncertain. This domain has been little investigated: most works consider a fixed schedule and study its performance degradation; they propose a sensitivity analysis of existing algorithms. The goal of the approach proposed here for the m machine problem with uncertain communication delays is to introduce flexibility. It consists in a three step scheme: compute a first schedule with estimated delays, fix partially this schedule (assignment plus partial order), and determine the complete sequencing on each machine during execution. This stabilization scheme gives interesting results for arbitrary task graphs. We then considered tree-like precedence constraints. The scheme builds optimal schedules for fork graphs and join graphs when communication delays have some monotonicity property. When the number of processors is unlimited, Chrétienne's algorithm, optimal for fixed delays, is used in the first step of our scheme. As it is linear, the full schedule is in fact fixed off-line. Some sensitivity bounds are then proved.

Scheduling with uncertain data is a very promising research field. Similar stabilization schemes may be used for other scheduling problems, for instance when processing times also are uncertain, or when the machines are not identical, or not parallel (flow shops, job shops, ...).

References

[1] E. Bampis, F. Guinand and D. Trystram, Some Models for Scheduling Parallel Programs with Communication Delays. *Discrete Appl. Math.* **51** (1997) 5-24.

[2] Ph. Chrétienne, A polynomial algorithm to optimally schedule tasks on a virtual distributed system under tree-like precedence constraints. *EJOR* **43** (1989) 225-230.

[3] Ph. Chrétienne and C. Picouleau, Scheduling with communication delays: a survey, in *Scheduling Theory and its Applications*, edited by Ph. Chrétienne, E.G. Coffman, J.K. Lenstra and Z. Liu. John Wiley Ltd. (1995).

[4] E.G. Coffman Jr. and R.L. Graham, Optimal scheduling for two-processor systems. *Acta Informatica* **1** (1972) 200-213.

[5] G.L. Djordjevic and M.B. Tosic, A heuristic for scheduling task graphs with communication delays onto multiprocessors. *Parallel Comput.* **22** (1996) 1197-1214.

[6] G. Galambos and G.J. Woeginger, An on-line scheduling heuristic with better worst case ratio than Graham list scheduling. *SIAM J. Comput.* **22** (1993) 349-355.

[7] A. Gerasoulis and T. Yang, A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors. *J. Parallel Distributed Comput.* **16** (1992) 276-291.

[8] A. Gerasoulis and T. Yang, Application 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. Rolim. Kluwer Academic Publishers, The Netherlands (1995).

[9] F. Guinand and D. Trystram, Optimal scheduling of UECT trees on two processors. *RAIRO: Oper. Res.* **34** (2000) 131-144.

[10] C. Hanen and A. Munier, Performance of Coffman Graham schedule in the presence of unit communication delays. *Discrete Appl. Math.* **81** (1998) 93-108.

[11] 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.

[12] A.W.J. Kolen, A.H.G. Rinnooy Kan, C.P.M. Van Hoesel and A.P.M. Wagelmans, Sensitivity analysis of list scheduling heuristics. *Discrete Appl. Math.* **55** (1994) 145-162.
[13] P. Kouvelis and G. Yu, *Robust Discrete Optimization and Its Applications*. Kluwer Academic Publisher (1997).
[14] J.K. Lenstra, M. Veldhorst and B. Veltman, The complexity of scheduling trees with communication delays. *J. Algorithms* **20** (1996) 157-173.
[15] A. Moukrim and A. Quilliot, *Scheduling with communication delays and data routing in Message Passing Architectures*. Springer, *Lecture Notes in Comput. Sci.* **1388** (1998) 438-451.
[16] C.H. Papadimitriou and M. Yannakakis, Towards an Architecture-Independent Analysis of Parallel Algorithms. *SIAM J. Comput.* **19** (1990) 322-328.
[17] V.J. Rayward–Smith, UET scheduling with interprocessor communication delays. *Discrete Appl. Math.* **18** (1986) 55-71.
[18] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. The MIT Press (1989).
[19] G.C. Sih and E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distributed Systems* **4** (1993) 279-301.
[20] Y.N. Sotskov, A.P.M. Wagelmans and F. Werner, On the calculation of stability radius of an optimal or an approximate schedule. *Ann. O.R.* **83** (1998) 213-252.
[21] S.D. Wu, E. Byeon and R.H. Storer, A graph-theoretic decomposition of the job shop scheduling problem to achieve scheduling robustness. *Oper. Res.* **47** (1999) 113-124.
[22] T. Yang and A. Gerasoulis, List scheduling with and without communication delay. *Parallel Comput.* **19** (1993) 1321-1344.