Scheduling with Communication Delays and On-line Disturbances

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

[1] Université du Havre 25 rue Philippe Lebon
BP 540 76058 Le Havre cedex
[2] LIMOS, Université de Clermont-Ferrand 2 Campus des Cézeaux
63177 Aubière Cedex FRANCE

Abstract. This paper considers the problem of scheduling tasks on multiprocessors. Two tasks bound by a precedence constraint and executed by two different processors must communicate. The resulting delay depends on the tasks and on the processor network. 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. Algorithms computing separately the processor assignment and the sequencing on each processor are considered. We propose a partially on-line sequencing algorithm based on critical paths to cope with the possible disturbances. Some theoretical results and a statistical study show the interest of this approach compared with fully on-line sequencing.

1 Introduction

With the growing importance of Parallel Computing issues, the classical problem of scheduling n tasks subject to precedence constraints on m processors in the minimum amount of time attracted renewed attention. Several models including communication delays for both shared and distributed memory multiprocessor systems have been proposed and widely studied [10, 1]. However 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 UET-UCT problem), on an arbitrary number of processors or on an unlimited number (see the pioneering work of Rayward Smith [11], or the survey of Chrétienne and Picouleau [3]).

Moreover, 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 ([12, 5]). This is rarely the case for a real machine: the network topology and the contention 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 [7]. It is also possible to use more precise models (see [9, 13] 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 intricated optimization problem. In that model also, routing problems are so important that they are treated by themselves, for a fixed assignment and sequencing.

The implicit assumption of the theoretical approach is that approximation algorithms construct schedules that in practice will behave well, even on real topologies. There are two ways to verify that assumption. The first is to implement these methods on real parallel computers, and to make real tests. Aside from the practicality of this approach, it will bring results valid for one machine only. The second is to remain inside the theoretical framework, adding only an element of uncertainty (a disturbance) on the communication times. this is what we propose in this paper: an approximation of the communication delays is known per advance, allowing to compute a schedule. But the real 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.

The paper is organized as follows. In section 2, the notations are stated precisely, and the different algorithmic approaches are presented. The choice of two phase methods, processor assignment then sequencing, is justified. Disturbances on the communication delays during the execution are taken into account when sequencing. Section 3 presents a new partially on-line sequencing policy adapted to these disturbances. Some theoretical results for special cases are presented in section 4, and an experimental comparison is conducted and interpreted in section 5 : for a fixed assignment, our approach is compared with off-line and on-line schedulings. The results of our study are summarized in section 6.

2 Preliminaries

2.1 Model and Definitions

We consider the schedule of n tasks $T_1, \ldots, T_n$ subject to precedence relations denoted $T_i < T_j$, on m identical processors. Preemption (the execution of a task may not be interrupted) and task duplication are not allowed, and 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 $T_i < T_j$ 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 becomes null. At compile time, the communication delays are estimated as $\tilde{c}_{ij}$ time units. It is expected however (see section 5) that these estimations are well correlated with the real values c_{ij} (at execution time). The goal is to minimize the maximum task completion time, or *makespan*, denoted $C_{\max}(S)$ for schedule S.

A schedule is composed of the *assignment* of each task to one processor and of the *sequencing* (or ordering) of the tasks assigned to the same processor. A scheduling algorithm provides a complete 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 the data transfer during the execution. In order to take into account the estimated delays a trade-off between off-line and on-line scheduling consists in a *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 [2]) where nothing is known of the task durations themselves before their execution. Then all policies, even for independent tasks, will have large performance upper bounds.

2.2 Different approaches for scheduling

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.

List-Scheduling approaches Without communication delay the so called List Schedules (LS) are often used as they provide good average performances, even as the performance bounds are poor. Remember LS schedules are obtained from a complete priority order of the tasks; 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 [7].

Clustering approaches So far the question of the priority computation was left aside. In most cases without communication delays the choice is based upon Critical Paths (CP) computation. For each task the length of its longest path to a final task is computed, and the priority is proportional to this length. CP based list schedules are optimal in the UET with no communication delay case when the task graph is a tree (a restricted version is optimal for two machines and arbitrary precedences). Again, to extend this to communication delays is not easy: If the search for the critical path overlooks the communication times, it looses much of its meaning, particularly if the communication times are sufficiently large with regard to processing times (fine grain parallelism). On the other hand, computing the critical paths with all the durations may not be better, as two successive tasks may be executed on the same processor, thus zeroing the communication time.

ETF Scheduling is a one phase method. Another proposed method is the clustering of tasks to build a pre-assignment [5, 12]. The idea is to cluster tasks between which the communication costs would be high. Initially, each task forms a cluster. At each step two clusters are merged, until another merging would increase the makespan. This approach seems particularly successful in the case of a large number of processors (infinite for practical purposes). When the number of processors is limited, the merging must continue but could entail larger makespans. It is also poorly suited to different distances between processors. The sequencing for each processor is then obtained by applying the Critical Path rule.

More recently, Djordjevic and Tosic [4] proposed a new approach called chaining, trying to combine modified list scheduling and clustering. It is a one phase method as ETF. But tasks are not scheduled sequentially. At each step a task is assigned to one processor, and is included into the chain formed by the tasks already assigned to that processor. After adding these chains to the original task graph (and zeroing the corresponding communication times), a current critical task for that graph is assigned next. The time complexity of this method is significantly large.

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. However we have remarked above the severe drawbacks of this approach, due to the difficulty of building good priority lists. More than that, fully on-line policies meet additional problems: if the communication delays are not known precisely per advance, the ready time of a task is not known before the communication is achieved. But if this task is not assigned yet, it supposes to send the information relevant to that task to all processors! This is impossible in practice. Furthermore, if tasks are pre-assigned before execution, computation of the sequencing (ie, choice of the task to be scheduled next) can be distributed, thus avoiding fastidious information exchanges between any processor and some “master” processor.

The clustering approach would better fit our needs except for its above mentioned limitations. Indeed the assignment might be computed off-line, and the sequencing on-line. By contrast the chaining approach is off-line. However it might be used to build a (hopefully good) assignment, the sequencing being done on-line.

It follows from the above discussion that the best idea is to compute the assignment off-line by one method or another (finding the optimal assignment is known to be NP-complete [12]). 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. Note that finding the optimal schedule when the assignment is fixed is NP-hard even for 2 processors and $UET-UCT$ hypotheses [3].

Our goal here is to study the different possibilities once an assignment is fixed. This assignment might be obtained by clustering, or chaining, or simply by ETF. In both last cases, the assignment is kept but the sequencing is not used. In the statistical tests of section 5, the last option is taken, as it seems more adapted than clustering to different processor distances, and is of lower complexity than chaining.

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

3.1 New approach based on Partially on-line sequencing

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 (but with fixed assignment). 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 :

step 1 compute a fixed assignment based on the $\tilde{c}_{ij}$'s.
step 2 compute a partial order $<_p$ including $<$, 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 partial order $<_p$.

In what follows the choices for each step, used in the next sections, are detailed.

3.2 Detailed partially on-line algorithm

As said previously the assignment of step 1 is built by an ETF policy. We said that Critical Path based policies seemed best suited, as is shown by empirical tests with known communication delays [14], and also by bound results (see [7] and [6] for a study of the extension of the Coffman–Graham priority rule to communication delays). 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 to a final task, including processing times and communication delays (between tasks to be processed on different processors), the processing time of the final task 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^*. If the processing time of T_i is included in the longest path, the heuristic is just called RCP.

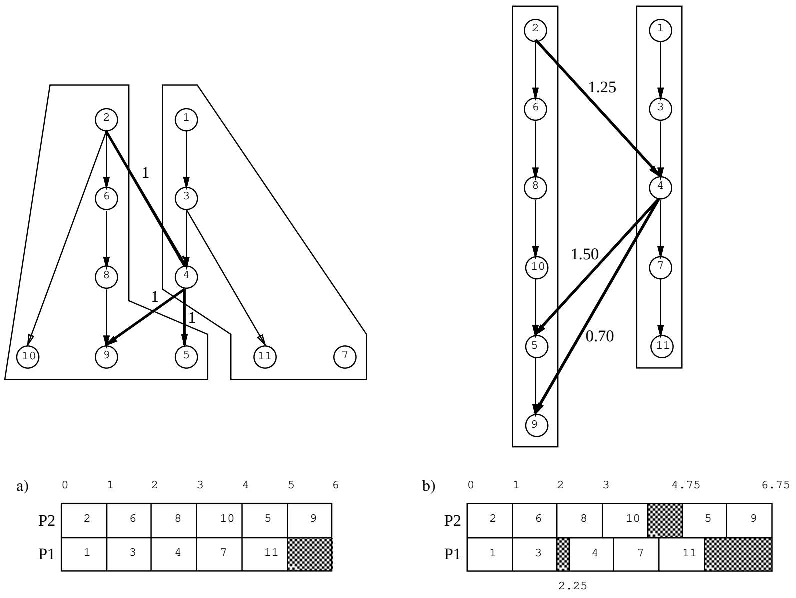

Fig. 1. RCP^* schedule before the execution and associated completely off-line schedule

The partial order of step 2 is obtained as follows : In step 1 a complete schedule based upon the $\tilde{c}_{ij}$'s has been built at compile time. Suppose two tasks T_i and T_j assigned to the same processor, such that T_i has larger priority than T_j and is sequenced before T_j. To avoid that a small disturbance leads to execute T_j before T_i during the execution time, *a precedence relation from T_i to T_j is added.* In step 3, RCP^* is again used as sequencing on-line policy to get the complete schedule.

The resulting algorithm is called $PRCP^*$ (Partially on-line sequencing with RCP^*). A sequencing 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 UET tasks is to be scheduled on two processors. all estimated communication delays are 1. Schedule a) is obtained before the execution by RCP^*. The resulting assignment is kept for scheduling with the real 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, $T_4 <_p T_7$, $T_4 <_p T_{11}$, and for processor P_2, $T_8 <_p T_5$, and $T_8 <_p T_{10}$

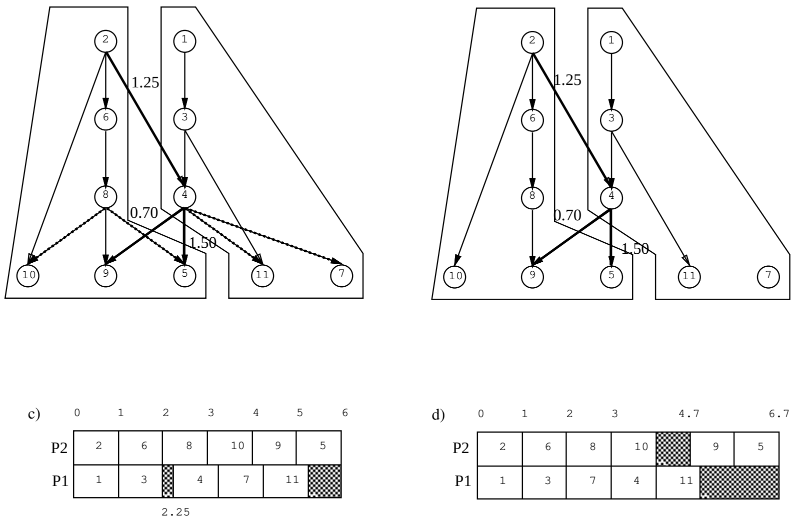

Fig. 2. $PRCP^*$ and $FRCP^*$ schedules

(dashed arcs). Consider now the following real 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 $<_p$ waits for T_4. At time 4 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.

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 listed in section 4.

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

4.1 Fork graphs

Fork task graphs are considered (see figure 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 [14]).

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

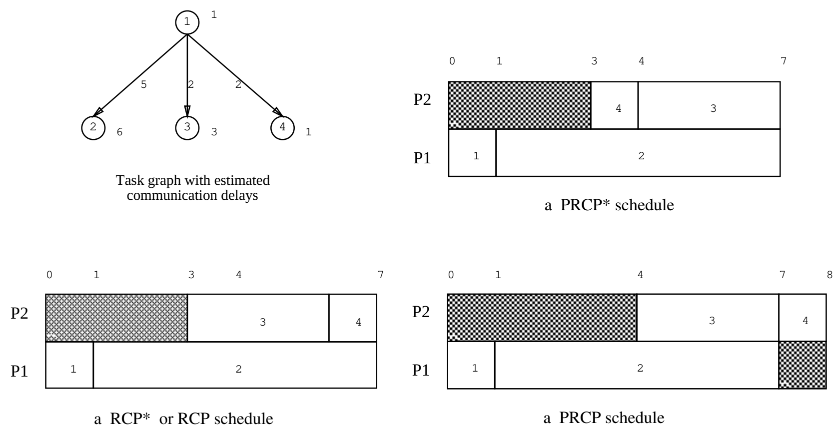

Fig. 3. $PRCP^*$ versus $PRCP$ for a fork graph

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 $r_i = p_1 + c_{1i}$, 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 ordering 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 hand, a partially off-line policy 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 RCP^*, not by RCP. It is another argument for $PRCP^*$: it never adds any precedence between final tasks.

4.2 Join graphs

RCP^* is optimal for Join task graph (simply reverse the fork graph, see figure 4) and fixed communication delays, whereas RCP is optimal only if all tasks have the same duration (see [14]). However this is no longer true if the communication

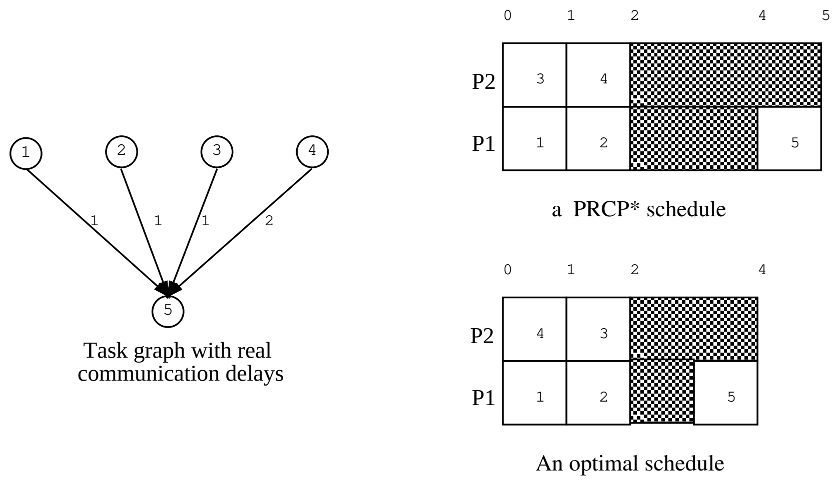

Fig. 4. Example of join graph

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 monotonic property is kept between expected and real communication delays, $PRCP^*$ is optimal. Indeed consider the following property on the disturbances (task T_n is final):

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

Theorem 2. *If the communication delays respect the local monotonic 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 c_{ij}'s, hence for the $\tilde{c}_{in}$'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 UET-UCT case and an arbitrary number of processors (see [8]).

n	m	$Mean\ Imp$ P/F	nb P/F	nb F/P	$Max\ Imp$ P/F	$Max\ Imp$ F/P
50	3	0.93	28	6	5.65	1.17
50	4	0.59	28	0	5.00	0.00
50	10	0.44	22	4	4.83	1.82
100	3	1.35	68	6	5.17	1.76
100	4	0.77	48	12	7.10	1.21
100	10	0.24	20	2	2.61	0.60
150	3	0.75	54	26	5.47	1.66
150	4	0.85	52	4	3.44	0.45
150	10	0.18	16	8	1.72	0.62
200	3	0.53	66	18	4.32	4.62
200	4	0.85	62	6	3.37	0.29
200	10	0.39	36	4	2.73	0.26
250	3	1.14	68	18	5.01	0.69
250	4	0.37	56	6	3.03	0.42
250	10	0.69	58	0	3.29	0.00

Table 1. Task graphs with large communication delays

5 Experimental Results

As said previously, if the $\tilde{c}_{ij}$'s are too poor approximations of the real communication delays, the policy of choosing any ready task for execution might prove as well as another. Hence the following assumptions are done: 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.

400 graphs are randomly generated, half relatively large with respect to the size of antichains (set of independent tasks), half strongly connected, thus having small antichains and long chains. The presented results are for the first case. Then the maximum size of an antichain is in $[10, 20]$. The number of vertices varies from 50 to 250, and 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 three times for a given graph to obtain LCT, SCT and UETUCT durations (Large and Small communications times, and Unit Execution and 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

n	m	$Mean\ Imp$ P/F	nb P/F	nb F/P	$Max\ Imp$ P/F	$Max\ Imp$ F/P
50	3	0.07	4	6	2.26	0.42
50	4	1.71	38	0	11.90*	0.00
50	10	0.11	12	0	1.25	0.00
100	3	0.04	16	26	1.50	2.37
100	4	2.16	66	0	7.79	0.00
100	10	0.47	34	0	3.04	0.00
150	3	0.34	48	18	1.39	0.25
150	4	2.45	86	2	6.57	0.30
150	10	0.25	20	4	1.69	0.34
200	3	0.14	34	46	1.57	0.30
200	4	1.23	92	4	5.38	0.14
200	10	0.63	48	0	3.90	0.00
250	3	0.28	64	16	1.00	0.61
250	4	1.73	92	2	4.68	0.10
250	10	0.34	46	0	1.29	0.00

Table 2. Task graphs with small communication delays

graph, 5 duration sets of each type are tested. The number of processors from 3 to 10.

Table 1. shows the results for Large Communication Times and large task graphs. n is the number of vertices, m the number of processors. The other columns give the mean percentage of improvement from $PRCP^*$ to $FRCP^*$, the number of times $PRCP^*$ was better, $FRCP^*$ was better, and the maximum improvement (in percentage) obtained by $PRCP^*$ and by $FRCP^*$, respectively. Tables 2. and 3. display the same informations for SCT and UETUCT duration sets.

$PRCP^*$ is always better in average, and the maximum improvement is much larger in general when it is better. The mean improvements are significant, as the assignment is the same for both policies. 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 improvements are larger for $m = 4$, this is due to the mean width of our graphs. For $m = 3$, there is less idleness for each processor as ready tasks are most of the time available (the results of a study with $m = 2$ with the same graphs are not reported, as both policies were equivalent most of the time). For $m = 10$, the assignment algorithm will balance the load so that the makespan slightly differs from the length of the longest chain, for both policies. They are also a bit more significant in the SCT and UETUCT case, where it may be interesting to wait for a task with large priority, instead of immediately executing a task with large duration.

n	m	$Mean\ Imp$ P/F	nb P/F	nb F/P	$Max\ Imp$ P/F	$Max\ Imp$ F/P
50	3	1.83	50	22	9.42	3.36
50	4	2.64	50	0	9.13	0.00
50	10	0.23	6	0	4.40	0.00
100	3	1.17	50	14	8.01	2.75
100	4	2.26	76	0	5.33	0.00
100	10	0.87	38	0	4.91	0.00
150	3	0.99	60	20	4.87	3.01
150	4	1.88	90	0	4.36	0.00
150	10	0.47	24	0	5.33	0.00
200	3	1.01	72	18	3.83	1.83
200	4	2.13	84	0	5.01	0.00
200	10	0.90	62	0	3.20	0.00
250	3	0.46	44	34	5.19	1.85
250	4	2.15	96	0	5.13	0.00
250	10	0.98	76	2	4.23	0.67

Table 3. UET UCT task graphs

In the case of highly 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 $PRCP^*$ (with larger improvements for $m = 3$ processors, as expected). It is sometimes strictly better in 46% of the cases. $FRCP^*$ is never better in more than 10%, except in one case.

6 Conclusion

When uncertainties are assumed on the communication delays, separate computation of assignment and sequencing is necessary. Once the assignment is fixed, we propose to add a subset of precedences at the compile time (off-line), based on critical path computation. These precedences do not change the sequencing if the expected communication delays are respected. However they might entail some idleness on one processor if an expected high priority task is delayed. Experimental results show that fully on-line sequencing is outperformed by this approach.

References

1. Bampis E., F. Guinand and D. Trystram, *Some Models for Scheduling Parallel Programs with Communication Delays*, DAM, 51,pp. 5-24, 1997.

2. CHEN B., Van Vliet A., Woeginger G.J., *A lower bound for randomized on-line scheduling algorithms*, Inf. Proc. Letters, 51,pp. 219-222, 1994.
3. CHRÉTIENNE Ph., Picouleau C., *Scheduling with communication delays: a survey*, Scheduling Theory and its Applications, P. Chrétienne, E.G. Coffman, J.K. Lenstra, Z. Liu (Eds), John Wiley Ltd 1995.
4. DJORDJEVIC G.L., Tosic M.B., *A heuristic for scheduling task graphs with communication delays onto multiprocessors*, Parallel Computing, 22,pp. 1197-1214, 1992.
5. GERASOULIS A., Yang T., *A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors*, Journal of Parallel and Distributed Computing,16,pp. 276-291, 1992.
6. HANEN C, Munier A, *Performance of Coffman Graham schedule in the presence of unit communication delays*, accepted in *D.A.M.*, 1998.
7. HWANG J.J., Chow Y.C., Anger F.D., Lee C.Y., *Scheduling precedence graphs in systems with interprocessor communication times*, SIAM J. Comput., 18(2), pp. 244-257, 1989.
8. LENSTRA J.K., Veldhorst, M., Veltman B., *The complexity of scheduling trees with communication delays*, J. of Algorithms 20, pp. 157-173, 1996.
9. MOUKRIM A., Quilliot A., *Scheduling with communication delays and data routing in Message Passing Architectures*, Springer Lecture Notes in Computer Science, vol. 1388, pp. 438-451, 1998.
10. PAPADIMITRIOU C.H. and Yannakakis M. *Towards an Architecture-Independent Analysis of Parallel Algorithms*, SIAM J. Comput., 19(2), pp. 322-328, 1990.
11. RAYWARD-SMITH V.J., *UET scheduling with interprocessor communication delays*, Discrete Applied Mathematics, 18, 55-71, 1986.
12. SARKAR V., *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, The MIT Press, 1989.
13. SIH G.C., Lee E.A., *A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures*, IEEE Transaction on Parallel and Distributed Systems, 4, pp. 279-301, 1993.
14. YANG T. and Gerasoulis A., *List scheduling with and without communication delay*, Parallel Computing, 19, pp 1321-1344, 1993.

This article was processed using the LaTeX macro package with LLNCS style