Scheduling with Communication Delays and On-Line Disturbances

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

[1] LIMOS, Université de Clermont-II, 63177 Aubière Cedex France
{moukrim, sanlavil}@ucfma.univ-bpclermont.fr
[2] LIH, Université du Havre, BP 540 76058 Le Havre Cedex France
guinand@fst.univ-lehavre.fr

Abstract. This paper considers the problem of scheduling tasks on multiprocessors. Two tasks linked 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 scheduling algorithm based on critical paths to cope with the possible disturbances. Some theoretical results and an experimental study show the interest of this approach compared with fully on-line scheduling.

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 [1, 8]. 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 (see the pioneering work of Rayward Smith [9], or the survey of Chrétienne and Picouleau [2]).

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 ([3, 10]). 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 [5]. It is also possible to use more precise models (see [7, 11] for simultaneous scheduling and routing) but the performance analysis is then

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 350–357, 1999.

very difficult. In general, building an accurate model of the network is much complicated and entails a very intricated optimization problem.

This paper proposes the following approach: 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.

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. 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 on-line scheduling.

2 Preliminaries

Model and Definitions We consider the schedule of n tasks $T_1, \ldots, T_n$ subject to precedence relations denoted $T_i \prec T_j$, on m identical processors. 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 section 5) 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 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 complete 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. 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.

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. 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 [5].

Clustering approaches Another proposed method is the clustering of tasks to build a pre-assignment [3, 10]. 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 m. The sequencing for each processor is then obtained by applying the CP rule.

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. The first drawback is the difficulty to build good priority lists. 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 impossible in practice.

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 [10]). 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 [2].

3 An Algorithm for Scheduling with On-Line Disturbances

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. 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 (the choices for each step are detailed next).

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.

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 [12], and also by bound results (see [5], and [4] 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^* in [12] (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 $FRCP^*$.

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

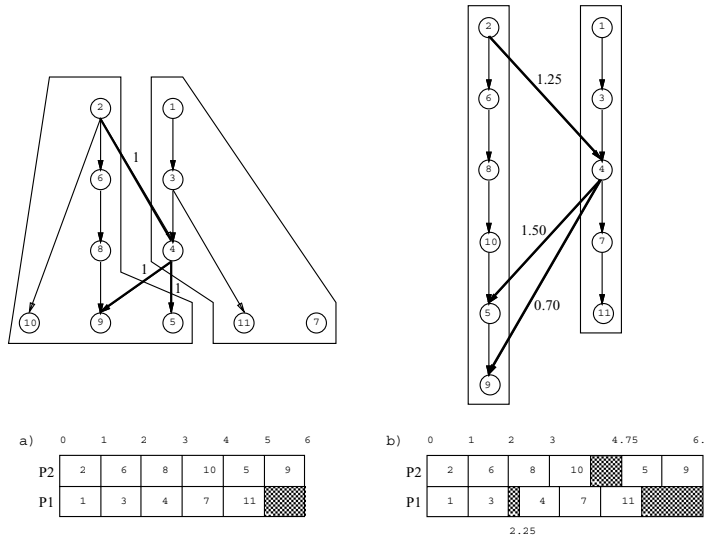

Fig. 1. RCP^* schedule at compile time and associated completely off-line schedule

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 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. Two of them are presented in section 4.

4 Optimality of $PRCP^*$ for Fork and Join Graphs

Fork graphs In that case one task T_1 is the immediate predecessor of all the others, which are final tasks. For fixed communication delays, RCP or RCP^* find the minimum makespan (see [12]).

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 com-

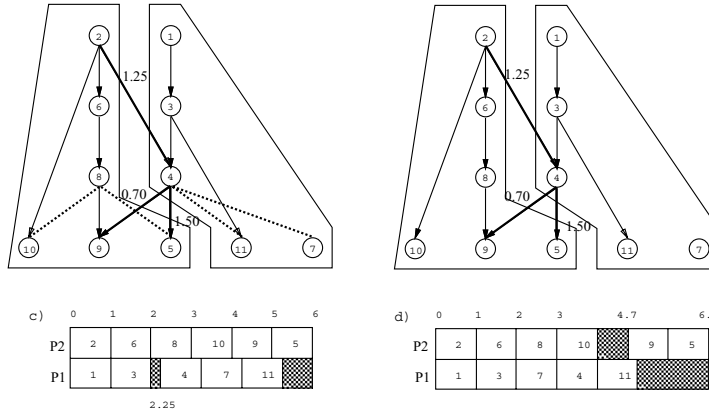

Fig. 2. $PRCP^*$ and $FRCP^*$ schedules

munication 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 hand, a partially off-line algorithm may add some precedence relations between these tasks. This may lead to unwanted idleness. Indeed, using RCP 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. However with RCP^*s, all priorities of the final tasks are equal, hence no precedence will be added, which guaranty optimality.

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

Join graphs RCP^* is optimal for join task graph (simply reverse the arcs of a fork graph) and fixed communication delays, whereas RCP is optimal only if all tasks have the same duration (see [12]). This is no longer true if the communication delays vary, as 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.
However if some local monotonic property is verified by expected and actual communication delays (for instance, when disturbances depend only on the source and target processors), $PRCP^*$ 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 $\tilde{c}_{in} \leq \tilde{c}_{jn}$) $\Rightarrow$ $c_{in} \leq c_{jn}$

		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

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

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 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. □

The trees are a natural extension of fork and join graphs. However the problem is already NP-complete in the UECT case and an arbitrary number of processors (see [6]).

5 Experimental Results

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. If the $\tilde{c}_{ij}$'s are poor approximations of the actual communication delays, the policy of choosing any ready task for execution might prove as well as another. Hence we admitted 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. When non zero, it is positive three times more often than negative. Finally, its size is chosen at random and uniformly between 0 and 0.5. 500 task graphs were 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 number of vertices varies from 50 to 250. The durations are generated 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 p_i is taken uniformly in $[1, 5]$ and $\tilde{c}_{ij}$ 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 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. The results are logically more significant in the SCT and UECT cases. In the case of strongly connected graphs, the differences are less significant, as often there is only one ready task at a time per processor. But when there are differences they are in favor of $PRCP^*$.

References

[1] Bampis E., Guinand F., Trystram D., *Some Models for Scheduling Parallel Programs with Communication Delays*, Discrete Applied Mathematics, 51, pp. 5–24, 1997.

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

[3] Gerasoulis A., Yang T., *A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors*, J. of Parallel and Distributed Computing, 16, pp. 276–291, 1992.

[4] Hanen C, Munier A, *Performance of Coffman Graham schedule in the presence of unit communication delays*, Discrete Applied Mathematics, 81, pp. 93–108, 1998.

[5] 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.

[6] Lenstra J.K., Veldhorst, M., Veltman B., *The complexity of scheduling trees with communication delays*, J. of Algorithms 20, pp. 157–173, 1996.

[7] Moukrim A., Quilliot A., *Scheduling with communication delays and data routing in Message Passing Architectures*, LNCS, vol. 1388, pp. 438–451, 1998.

[8] Papadimitriou C.H., Yannakakis M., *Towards an Architecture-Independent Analysis of Parallel Algorithms*, SIAM J. Comput., 19(2), pp. 322–328, 1990.

[9] Rayward-Smith V.J., *UET scheduling with interprocessor communication delays*, Discrete Applied Mathematics, 18, pp. 55–71, 1986.

[10] Sarkar V., Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors, The MIT Press, 1989.

[11] Sih G.C., Lee E.A., *A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures*, IEEE Trans. on Parallel and Distributed Systems, 4, pp. 279–301, 1993.

[12] Yang T., Gerasoulis A., *List scheduling with and without communication delay*, Parallel Computing, 19, pp 1321–1344, 1993.