Minimizing the Overhead For Some Tree-Scheduling Problems

Evripidis Bampis ‡ Frédéric Guinand † Denis Trystram †

‡LaMI, Université d'Evry, Boulevard des Coquibus, 91025 Evry, Cedex, France
bampis@lami.univ-evry.fr

†LMC-IMAG, 46 Avenue F. Viallet, 38031 Grenoble, Cedex, France
Frederic.Guinand@imag.fr Denis.Trystram@imag.fr

Abstract

This paper is devoted to the study of tree-scheduling problems within the execution model described by Anderson, Beame and Ruzzo [3]. We first prove the NP-completeness of the problem of minimizing the overhead for scheduling trees on m processors, and then we propose an algorithm that provides optimal schedules when complete trees are considered.

Key Words: overhead, scheduling, communications, trees.

1 Introduction

In this paper we present some new results concerning the problem of scheduling tree precedence task graphs. It is well-known that there exists no universal model for executing algorithms on parallel machines. The authors have reported the principal execution models in [5]. In the present paper, we concentrate the study on a model based on the notion of *overhead*. We prove that the general problem of scheduling trees within this model on m processors is strongly NP-complete. Hence, we present a polynomial-time algorithm that provides optimal schedules for the particular case of complete k-ary trees.

1.1 About the PRAM model

PRAM is one of the most popular abstraction of parallel machines. It models a shared-memory parallel machine where the access time from any processor to any memory location is constant. The abstraction provided by this model hides most of the problems arising with scheduling and communication delays. Although an important number of PRAM algorithms have been designed recently, this model is not representative of most of the latest parallel machines which are distributed-memory machines. Several ways are investigated for improving this model or its use. On the one hand, some researchers simulate the PRAM algorithms on real machines [11]. On the other hand, other researchers propose some extensions to the basic PRAM model [7, 2, 8, 9, 6, 13]. Two main approaches try to take into account architectural constraints. The

first is concerned with the communications and memory accesses, and the second studies the non uniformity of the execution environment.
Cole and Zajicek have introduced the Asynchronous PRAM model in order to point out the cost of synchronization [7]. They investigate algorithms such as Bitonic sort, Prefix Sum and Parallel Summation. For this latter one, the algorithm uses memory locations treated as a complete binary tree. They also propose an algorithm with low overhead to process the summation. This analysis is carried on within another model in which processes can be executed at different speeds [8]. About problems due to memory latency, Aggarwal, Chandra and Snir describe in [2] the LPRAM model (for Local-memory PRAM). Their model considers a PRAM with local memory and combines the architecture-based PRAM model with a language-based task graph model. Among the complexity measures considered as relevant, the authors point out the computation time ignoring the communication, and the overall communication delay. For general acyclic graphs both measures may not be achieved by the same schedule thus a tradeoff between them is required. The same kind of tradeoff has to be found within the model proposed by Anderson, Beame and Ruzzo [3]. The measure of schedule chosen is the overhead defined as the sum of the idle periods added to the communication times.

1.2 Overhead as criterion, Trees as graphs

In this work we focus our attention on this latter model. Minimizing the overhead is a very important issue in multiprocessor scheduling problems. The minimization of idle times requires dags with "lot" of parallelism i.e. fine grain dags while the minimization of the communication delays requires sequentialization in order to reduce data transfers i.e. coarse grain dags. The optimization problem can be expressed as the tradeoff between the idle time and the communication delays.
We are essentially interested in scheduling trees. Trees represent numerous mathematical applications such as, arithmetic expression evaluations, sorts, Parallel Summation, Parallel Prefix, but also computer applications such as, for example, Prolog programs or dynamic creation of tasks in parallel programming. We consider only in-trees, but all the described and proved results hold for out-trees.
In such a tree, the root is the only node with no successor. The level of a node is equal to one for the root. For a given node the level is equal to the length (the number of tasks) of the path that leads to the root. For instance, the nodes that are at level two are the immediate predecessors of the root.
We use the notation described in [12]. The machine environment, the job characteristics and the optimality criterion that together define a scheduling problem type are specified in terms of a three-field classification: $\alpha \mid \beta \mid \gamma$.

1.3 Outline of the paper

In the next section, the execution model, the tasks and the criterion are described. In the third section we consider the problem of scheduling trees on m processors and prove its NP-completeness. In the following section, we propose a polynomial time algorithm that produces optimal schedules for complete trees on m processors.

2 The execution model

2.1 Presentation

A program is represented by a graph whose vertices are the elementary instructions (the **tasks**) and the arcs are the precedence relations. Any task requires **one unit of time** to be completed. The **preemption is not allowed**. The execution of a task can be decomposed into three basic phases:

- Loading the data from the shared-memory.
- Computing the task.
- Storing the results in the shared-memory.

However, considering the **locality of data**, it is possible to execute several tasks within only one loading-storing memory access.

Local overlap of communications by computations is not allowed (a processor cannot simultaneously compute a task and communicate with the memory), but global overlap, due to the **asynchronous working mode**, is allowed (one processor can compute some task while some other processor communicate with the memory). Any access to the global memory requires a constant time, whatever the volume of the data transfer is.

The goal of the scheduling algorithm for this model is to minimize the parallel execution time, also called **makespan**. This time is defined as the a sum of the sequential time (equal to the number of tasks) T_{seq}, the total idle periods T_{idle} and the communication times T_{com}, divided by the number of processors (denoted by m).

$$T_{par} = \frac{T_{seq} + T_{idle} + T_{com}}{m}$$

As both T_{seq} and m are schedule-independent, the minimization of the parallel time is equivalent to the minimization of the overhead $(T_{idle} + T_{com})$.
In order to reduce T_{com}, a strategy consists in gathering elementary tasks in elements called **jobs**. Although this operation reduces the number of tasks, it increases in general the number of idle periods by reducing the potential parallelism. In order to reduce T_{idle}, load-balancing is necessary implying in general a large number of tasks. So, it appears that finding a good schedule is equivalent to determine a good **tradeoff** between **gathering** and **load-balancing**.

The operation of gathering induces a new graph whose vertices are jobs: the *schedule graph*. Each vertex-job is valuated by the number of elementary tasks that it contains plus one for the corresponding communication. If job J_1 contains a task T and job J_2 a task T' and T precedes T' in the initial DAG, then an arc exists from J_1 to J_2. The schedule graph must be acyclic and thus the gathering of the tasks into jobs must be done carefully. The effect of gathering elementary tasks is to decrease the communication cost because of the constant time needed for any memory access.

Let us emphasize that a job cannot send or receive data during its execution. This implies some restrictions about the possible gatherings (all the arcs between two jobs must have the same orientation).

Let us now detail an example to illustrate the notions relative to this execution model.

2.2 Example

We present below an example of scheduling a data flow graph composed of 7 elementary tasks (figure 1). The performance of a "good" scheduling algorithm on this model lies on the measure of the overhead. A job corresponds to a task in the first schedule. The overhead is equal to 9 (2 idle periods plus 7 jobs).

In the second example, we define 3 jobs as depicted in Figure 1. Job J_1 is composed of the three tasks T_1, T_2 and T_4. It precedes the job J_3 which introduces an extra precedence constraint between T_4 and T_7. In the schedule graph, J_1 and J_2 are independent and precede both J_3. The idle time increases but the number of jobs decreases. We obtain an overhead equal to 7 (4 idle times plus 3 communications). The last schedule is obtained by breaking job J_3 responsible of some idle times. It is optimal since it minimizes the overhead (equal to 5).

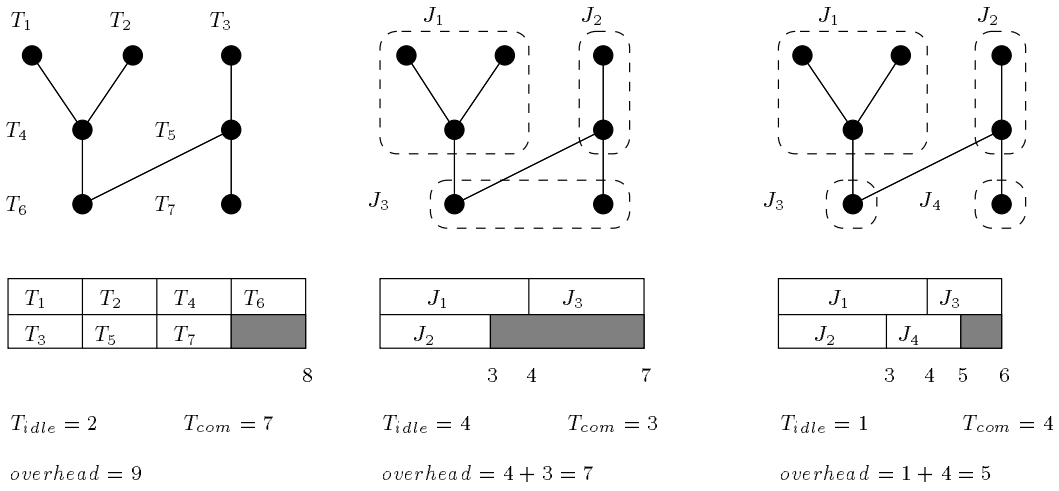

Figure 1: Example of three schedules for a given graph.

3 Scheduling Trees on m Processors is NP-complete

This section is devoted to the complexity study of scheduling trees on m processors within the overhead execution model. According to the previous notations, the problem can be denoted as:

$$P_m \mid \ trees,\ p_j = 1 \mid Overhead$$

Theorem
The problem of minimizing the overhead for scheduling trees of depth three on m processors is strongly NP-complete.

Proof
Our proof is based on a reduction from the well known NP-complete problem 3-partition [10]:

INSTANCE: Set A of $3m$ elements, a bound $B \in Z^+$ and a size $s(a) \in Z^+$ for each $a \in A$ such that $\frac{B}{4} < s(a) < \frac{B}{2}$ and such that $\sum_{a \in A} s(a) = mB$.

QUESTION: Can A be partitioned into m disjoint sets $A_1, A_2, \ldots, A_m$ such that for $1 \leq i \leq m$, $\sum_{a \in A_i} s(a) = B$? (note that each A_i must contain exactly three elements from A.)

Given an instance of 3-partition, we construct an in-tree with $s(a)$ leaves and depth one, and then we connect the roots of all such in-trees to a common root (our construction is based on a proof of Afrati et al. for the two parameters scheduling problem [1]).

Suppose that A can be partitioned into m disjoint sets A_1, A_2, ..., A_m of weight equal to B each. Then a schedule in $(B+6)$ time units can be directly constructed: Processor P_i executes one job containing the three in-trees whose leaves belong to A_i, for $1 \leq i \leq m$, while processor P_1 executes also a job containing the root of the tree. It is not difficult to verify that this schedule is of minimum length, thus leading to the minimization of the overhead.

Conversely, suppose that there is a schedule with length no more than $(B+6)$. Clearly, the root of the tree must constitute an individual job. Thus during the last two time units (execution of the root and the corresponding communication) of any optimal schedule, only one processor can be active. The best way to execute the remaining $m(B+3)$ tasks is to partition them into m independent jobs of size $(B+3)$ each. The only way to do this is to gather three independent in-trees into each job. This partition corresponds to a 3-partition of the set A, as required. □

The above construction can be slightly modified by removing the root of the tree i.e. by considering only the set of in-trees. Then using the same reasoning as above one can easily prove (simplifying considerably the proof of [15]) that the problem of minimizing the overhead for bipartite graphs of depth one on m processors, is strongly NP-complete.

4 Scheduling Complete Trees on m Processors

In the previous section we proved the NP-completeness of the problem: $P_m \mid intree,\ p_j = 1 \mid Overhead$. In this section we address the problem of scheduling complete trees on m processors. The first asymptotically optimal algorithm for scheduling a complete binary tree on$m$ processors was proposed by Bampis and König resulting to an overhead in $O(m \log m)$ [4] . Here, we present an algorithm leading to a schedule that is no more only asymptotically optimal, but reaches the exact optimal value of the minimum makespan for complete k-ary trees.
We first describe the principle of the algorithm and then prove the optimality of the built schedule.

4.1 Preliminary result

Assume the number of processors is equal to m, the algorithm considers that a complete k-ary tree of height h can be divided in two independent regions. The first one begins with the root of the tree and finishes at level $l_{inf} - 1$, where l_{inf} is such that $k^{l_{inf}-2} < m \leq k^{l_{inf}-1}$. The second region consists in the remaining tasks of the tree, and thus it contains $k^{l_{inf}-1}$ complete

k-ary trees of height $h - (l_{inf} - 1)$. A very natural algorithm for minimizing the overhead can be designed based on this observation.

In the first region, we try to minimize the idle time by exploiting entirely the potential parallelism of the tree. So, in this region, each job contains only one task. In the second region, we try to minimize both the number of jobs and the number of idle time.

Before detailing our algorithm let us prove the following lemma:

Lemma 1

The optimal makespan for scheduling a complete k-ary tree on a number of processors greater than or equal to the number of leaves of the tree is obtained by considering each task as an individual job.

Proof
Without loss of generality, we can consider a number of processors equal to the number of leaves: $m = k^{h-1}$.
Let us first recall that the problem of minimizing the overhead within the ABR model is equivalent to the problem of minimizing the makespan. Let us calculate the overhead in the case where each task is considered as a job. The obtained schedule has the form of stairs (see Fig. 2).

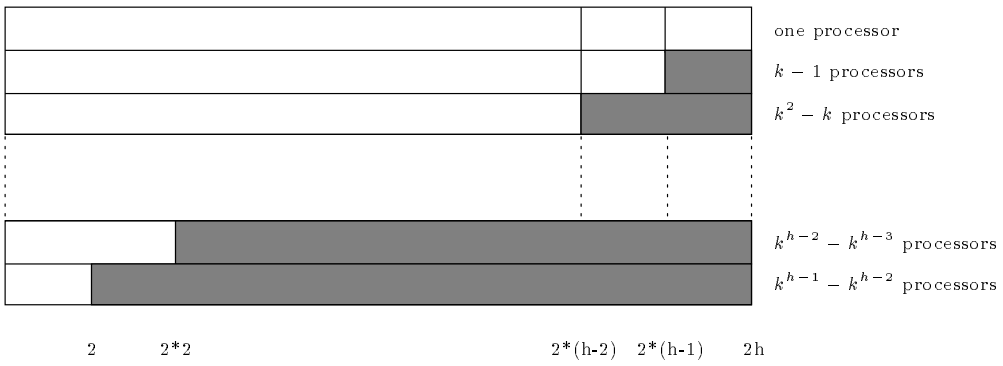

Figure 2: Shape of the schedule.

For a complete k-ary tree of height h, and a gathering of the tasks into jobs minimizing the parallel execution time. We focus our attention on the job that contains the root.
We notice that since this last job contains the root, during its execution all the other processors are idle. Suppose this job contains i tasks (including the root), it is then obvious that:

$$T_{par} \geq (i+1) + \left\lceil \frac{\frac{k^h-1}{k-1} - i}{k^{h-1}} \right\rceil + 1,$$

where the first term is equal to the number of tasks included in the last executed job plus one time unit for the communication, and the second term gives a lower bound of the execution time of all the remaining tasks (in the best case the remaining tasks are evenly distributed among the processors). Notice that the parallel execution time is minimized for the smaller possible

size of the last job i.e. for $i = 1$.
Thus, in a gathering reaching the minimum parallel execution time, the size of the last job must be not greater than two (one unit time task and the corresponding communication). If we consider the problem of scheduling each of the k complete k-ary subtrees of level 2 on $\frac{m}{k}$ processors, the previous result is still valid. The same property could be verified by recurrence at each level, from level one (containing only the root) to level h (the leaves level), thus, an optimal schedule is obtained when each job contains only one task. □

4.2 Description of the algorithm

We now present with more detail the algorithm applied to the second region. Let l_{inf} be the first level of the tree with more than m tasks, and N_2 the number of tasks of the second region. $N_2 = k^{l_{inf}-1} \times \frac{k^{h-l_{inf}+2} - 1}{k-1}$.

Case 1

If the number of complete subtrees at level l_{inf} is a multiple of m, then the algorithm allocates $k^{l_{inf}-1}$ div m complete subtrees gathered into one job to each processor (Figure 3). In such a situation, the schedule for the second region is clearly optimal.

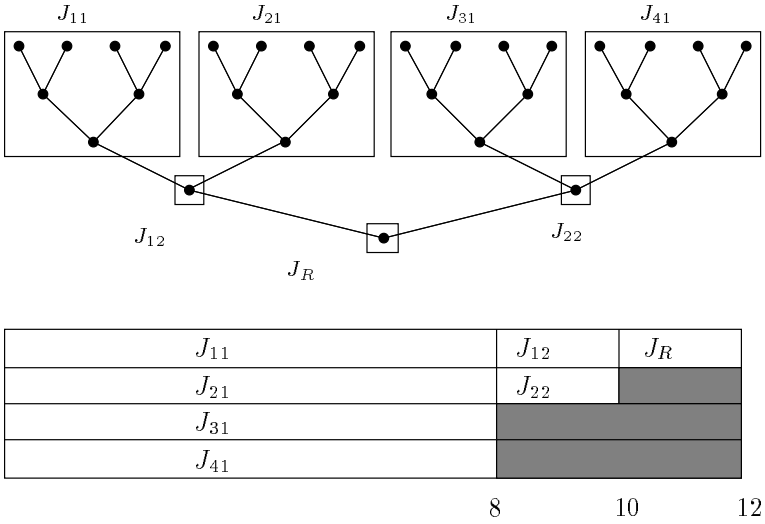

Figure 3: When $k^{l_{inf}}$ is proportional to m.

Case 2

If the number of tasks belonging to this second region is a multiple of m, the algorithm tries to evenly distribute the load, in order to produce a schedule without idle time. This is done in the following way:
(1) Allocate to $k^{l_{inf}-1}$ mod m processors $k^{l_{inf}-1}$ div $(m) + 1$ complete subtrees with a height equal to $h - l_{inf}$.
(2) Allocate to the other processors $k^{l_{inf}-1}$ div m complete subtrees of same height and add

some leaves in order to have jobs of equal length.
The algorithm creates exactly two jobs per processor (Figure 4).

Assume that the m first jobs contain a tasks and the m second jobs contain b tasks. If this schedule is not optimal then it is possible to find another schedule with a finishing time equal to $a + b + 1$. But, as the total number of tasks is equal to $m(a + b)$, in such a schedule each processor would be allocated only one job and this is possible if and only if $k^{l_{inf}-1}$ is a multiple of m (first case). Then, the schedules provided by the algorithm for the second region are optimal if $N_2 \bmod m = 0$.

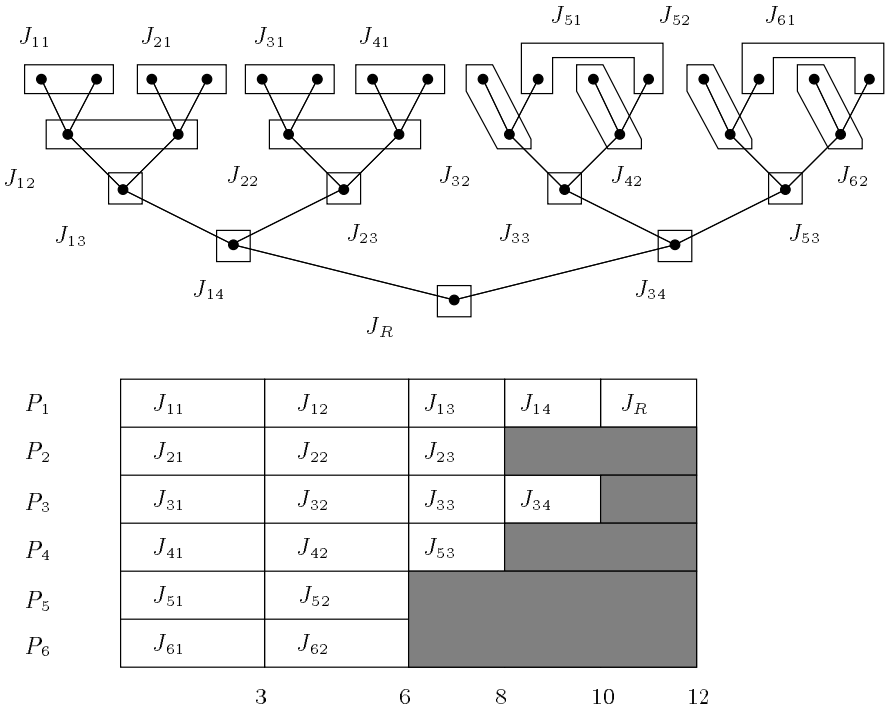

Figure 4: When N_2 is proportional to m.

Case 3

If $N_2 \bmod m \neq 0$ the problem is more complex. However, the idea is the same, the algorithm tries to find a partition of the tasks such that all the processors could finish at the same time. As $N_2 \bmod m \neq 0$, some processors are allocated $a+b$ tasks while the others are allocated $a+b+1$ tasks.

Idle times can be avoided if and only if $a+b+1$ corresponds to a multiple of $\frac{k^{h-l_{inf}+2}-1}{k-1}$ (the number of tasks in a complete tree of height $h - l_{inf} + 1$). In such a case, the processors owning $a+b+1$ tasks are allocated $(a+b+1)\ div\ \frac{k^{h-l_{inf}+2}-1}{k-1}$ complete subtrees and the algorithm of Case 2 is applied in order to assign the remaining tasks to the remaining processors (Figure 5). As previously, these schedules are optimal since the number of jobs cannot be reduced without creating new idle times due to the precedence constraints.

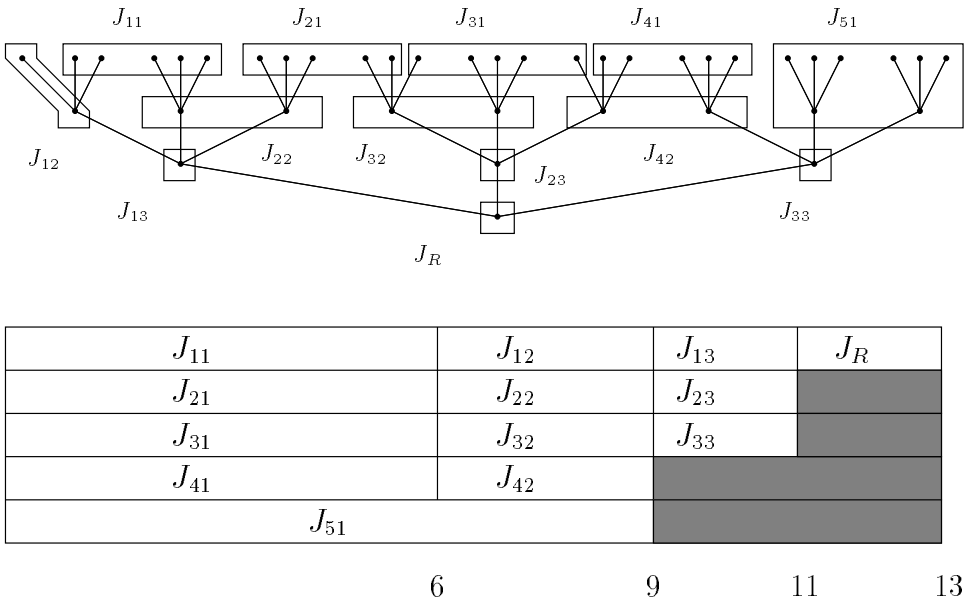

Figure 5: A special case

For all these cases, an optimal global schedule is formed by concatenating the schedule of the first and the schedule of the second region without any modification.

Lemma 2

If the schedule obtained for the second region is without any idle time and if its number of jobs is minimum then, the global schedule formed by both schedules (for the first and for the second region) is optimal.

Proof
The number of jobs in both the first and the second region is minimum, and the same holds for the number of idle times. So, the only solution to decrease the overhead would be to gather some jobs at the border. Since all the jobs of the second region finish at the same time, gathering jobs belonging to different regions would lead to new precedence constraints among them and would lead to the creation of a number of idle periods larger than the number of saved jobs, indeed, for each saved job at least two idle periods are created. □

Case 4

The idle times cannot always be removed in the third case as shown in Figure 6. In such a case, some processors finish their second job one unit of time after the others. Although these schedules are optimal for the second region, Lemma 2 cannot be applied.

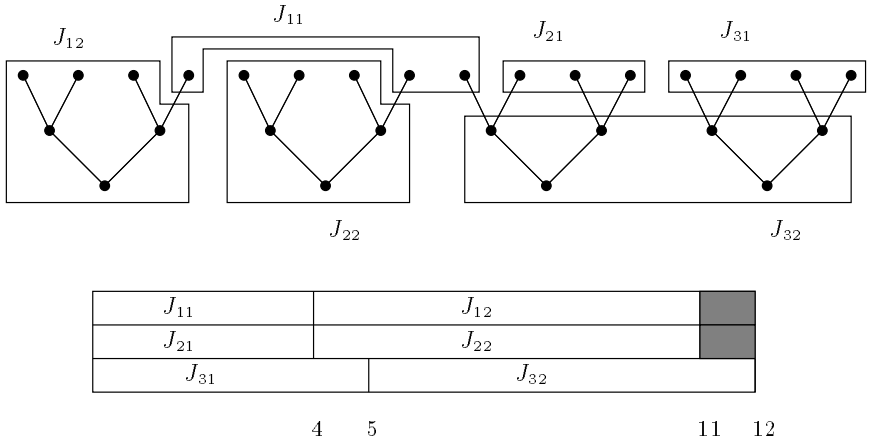

Figure 6: Optimal non synchronized schedule for the second region.

However, sometimes, it is possible to join the schedules from the first and from the second region such that these overloaded processors can be synchronized with the other during the execution of the higher jobs of the schedule of the first region. We consider the algorithm of the second case. The condition is: $k^{l_{inf}-1}\ div\ (m)+1=k$, where $k^{l_{inf}-1}\ div\ (m)+1$ is the number of tasks belonging to level l_{inf} and allocated to the most loaded processors. Indeed, within this condition, and for all these processors, the last job issued from the schedule of the second region can be gathered with the first executed job of the schedule of the first region as illustrated in Figure 7.

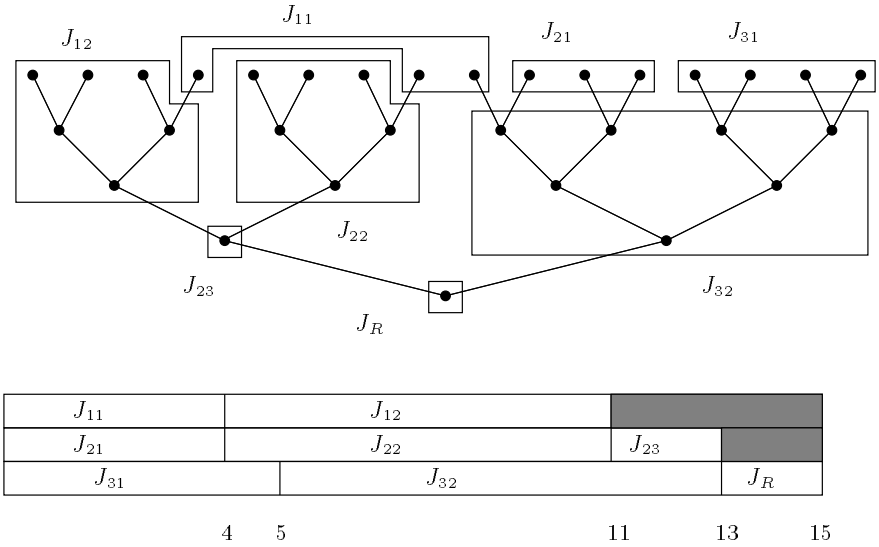

Figure 7: Optimal schedule of the special case

When all the previous conditions are not verified, ie, $N_2 \bmod m \neq 0$, a+b+1 $\bmod \dfrac{k^{h-l_{inf}+2}-1}{k-1} \neq 0$ and $k^{l_{inf}-1}\ div\ m\ +\ 1 \neq k$, any gathering of jobs at the border cannot decrease the makespan of the whole schedule. Moreover, as both the schedules of the first and of the second region are

optimal, the whole schedule obtained by concatenation is optimal. One such example is shown in Figure 8.

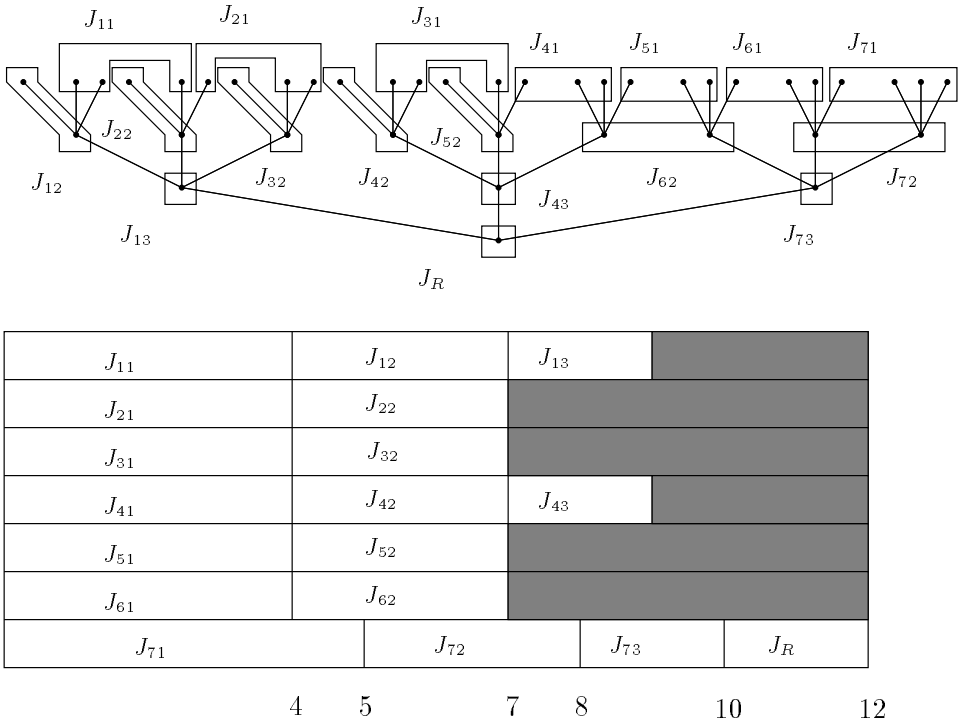

Figure 8: No reduction of the overhead is possible in this case.

5 Conclusion

We have considered in this paper the problem of scheduling tree-shaped precedence task graphs assuming the execution model introduced by Anderson, Beame and Ruzzo [3], model based on the notion of overhead.

We proved the NP-completeness of the problem of minimizing the overhead for trees on m processors. Then we derive, for the particular case of complete trees, a polynomial time algorithm which provides optimal schedules.

Some problems remain open about the complexity of the scheduling of trees on only two processors. Moreover, it would be interesting to evaluate the performances of clustering strategies designed for other execution models allowing the overlap assumption, because, clustering seems to be the best solution to obtain good schedules within this model.

References

[1] F. Afrati, C.H. Papadimitriou, G. Papageorgiou, *Scheduling DAGs to minimize time and computation,* Proc. of the Aegian Workshop on Computing (AWOC) 134-138 (1988).

[2] A. Aggarwal, A. K. Chandra, M. Snir, *Communication Complexity of PRAMs,* Theoretical Computer Science **71** 3-28 (1990).

[3] R.J. Anderson, P. Beame, w. Ruzzo, *Low Overhead Parallel Schedules for Task Graphs,* Proc. SPAA 66-75 (1990).

[4] E. Bampis, *L'impact des communications sur la complexité des algorithmes parallèles,* Ph D Thesis, Univ. of Paris-Sud, France (1993).

[5] E. Bampis, F. Guinand , D. Trystram, *Some Models for Scheduling Parallel Programs with Communication Delays,* To Appear in Discrete Applied Mathematics.

[6] E. Bampis, J-C. König, D. Trystram, *Optimal Parallel Execution of Complete Binary Trees and Grids into most Popular Interconnection Networks,* Theoretical Computer Science **147** 1-18 (1995).

[7] R. Cole, O. Zajicek, *The APRAM: Incorporating Asynchrony into the PRAM Model,* Proc. of STOC (1989).

[8] R. Cole, O. Zajicek, *The Expected Advantage of Asynchrony,* Proc. of STOC (1990).

[9] M. Cosnard, A. Ferreira, *Designing parallel non numerical algorithms,* in Parallel Computing '91, Eds. D.J. Evans et coll., North Holland 3–18 (1991).

[10] M. R. Garey, D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness* Freeman (1979).

[11] T. J. Harris, *A Survey of PRAM simulation techniques,* ACM computing surveys **26** 2 187-206 (1994).

[12] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys, *Sequencing and scheduling: algorithms and complexity,* D. Reidel Publishing Company M.A.H. Dempster, J.K. Lenstra, A.H.G. Rinnooy Kan Editors, Amsterdam (1989).

[13] C. Martel, A. Raghunathan, *Asynchronous PRAM with Memory Latency,* Journal of Parallel and Distributed Computing **23** 10-26 (1994).

[14] V.J. Rayward–Smith, *UET Scheduling with Unit Interprocessor Communication Delays,* Discrete Applied Mathematics **18** 55-71 (1987).

[15] R. Saad, *Overhead in parallel schedules,* Unpublished Manuscript.