Núcleo de Computação Eletrônica

Relatório Técnico

A Polynomial-Time Branching Procedure for the Multiprocessor Scheduling Problem

Ricardo C. Corrêa
Afonso Ferreira

NCE - 04/96
novembro

Universidade Federal do Rio de Janeiro

A polynomial-time branching procedure for the multiprocessor scheduling problem*

Ricardo C. Corrêa[†] and Afonso Ferreira[‡]

Abstract

In this paper, we present and analyze a branching procedure suitable for branch-and-bound algorithms for solving *multiprocessor scheduling problems.* The originality of this branching procedure resides mainly in its ability to enumerate all feasible solutions without generating duplicated subproblems. This procedure is shown to be polynomial in time and space complexities. The main applications of such branching procedure are instances of the MSP where the costs are large because the height of the search tree is linear on the number of tasks to be scheduled. This in opposition to another branching procedure in the literature that generates a search tree whose height is porportional to the costs of the tasks.

---

*This work was partially supported by DRET, the project Stratagème of the French CNRS, and the Human Capital and Mobility project SCOOP - Solving Combinatorial Optimization Problems in Parallel - of the European Union.

[†] Part of this work was done when this author was a Ph.D. student at the Laboratoire de Modélisation et Calcul, IMAG, Grenoble, France. Currently at Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, Caixa Postal 2324, CEP 20001-970, Rio de Janeiro, RJ, Brazil, correa@nce.ufrj.br. Partially supported by a CNPq fellowship and by the PROTEM-CC-II project ProMet of the CNPq (Brazil).

[‡] LIP – ENS-Lyon, CNRS URA 1398, 46, allée d'Italie, 69364 Lyon Cedex 07, France. Part of this work was done when visiting the School of Computer Science at Carleton University. Partially supported by a Région Rhône Alpes International Research Fellowship and NSERC.

1 Introduction

Nowadays, most *multiprocessor systems* consist of a set of identical processors with distributed memory. Each processor has its own memory, and two processors communicate exclusively by message passing through an interconnection network. This kind of system is been widely used in a broad spectrum of scientific and industrial applications with excellent cost effectiveness. From this cost effectiveness point of view, and considering the vast investment in sequential algorithms of such applications, we are lead to the direction of automatically parallelizing these sequential algorithms in order to run them in multiprocessor systems. This is the main motivation to the optimization problem considered in this paper, namely the *multiprocessor scheduling problem.*

An important feature of many of the sequential algorithms mentioned in the previous paragraph is that they can be described in terms of a set of modules to be executed under a number of *precedence constraints.* Each module requires a certain amount of time to be executed. A precedence constraint between two modules determines that one module must finish its execution *before* the other module starts its execution. The most important question related to the execution of the modules verifying the precedence constraints in a multiprocessor system concerns the minimum execution time with a minimum number of processors. Due to the importance of this optimization problem, it has been extensively studied by a large number of researchers. Given its difficulty, most of the authors dedicated more attention to ways of efficiently solving problems corresponding to weaker versions of the optimization problem introduced above (for an overview, see [1, 2, 4, 9, 11] and references therein). In this paper, we consider the following and slightly easier version of the problem, where the characteristics of the multiprocessor system are available as input (as they would, in reality).

> **Multiprocessor Scheduling Problem (MSP)** – Given a *program*, which must be divided in *communicating modules* to be executed in a *given* multiprocessor system under a number of *precedence constraints*, *schedule* these modules to the processors of the multiprocessor system such that the program's execution time is *minimized.*

In order to computationally solve the MSP, we suppose that we are able to provide, beforehand, a precise description of the program in terms of computation cost of the modules and their precedence relations. The modules are supposed to interact, and these interactions take place through communications between modules. Thus, we suppose that the communication cost between interacting modules are known beforehand. Therefore, in order to be executed, each module of the program must be scheduled on some processor of the multiprocessor system. Consequently, we also suppose that we know, beforehand, the performance features of the processors and of the interconnection network. Notice that the performance features of the interconnection network are important since modules interacting in the program may be scheduled to different processors, which leads these processors to communicate during the execution of the program.

We call *schedule* any solution to the MSP. In general, finding an optimal schedule to an instance of the MSP is computationally hard because the problem belongs to NP-hard in the strong sense. Considering the time complexity, there is no pseudopolynomial optimization or fully polynomial time approximation algorithm, unless $P = NP$ [4, 5, 6, 12, 14].

Consequently, it is of great importance the development of efficient techniques to give "reasonably good" *suboptimal schedules* for MSP instances. We mean by a *suboptimal schedule* any schedule that is not demonstrated to be non-optimal, i.e., the suboptimal schedule of a given instance of MSP is the best known schedule of this instance. Thus, a suboptimal schedule remains suboptimal until a better schedule is found or it is proved to be optimal. In general, the techniques used in this context consist of heuristics that do not examine all feasible schedules in order to provide the optimal one [1, 3, 11]. However, only for special cases there are theoretical results claiming that a specific heuristic always produces a schedule that is "within a factor of x of optimal." These special cases very often use models that do not correspond realistically to the multiprocessor system or to the sequential program at hand [11]. Consequently, a "reasonably good" suboptimal schedule for the model can become extremely poor when the program is actually executed on the multiprocessor system.

In the light of these considerations, an interesting topic of research is the formulation of heuristics for efficiently solving MSP instances using reasonable practical models in order to computationally provide a schedule and a certificate x guaranteeing that the schedule is "within a factor of x of optimal." From the parallelizer point of view, such a schedule must correspond to an actual good schedule [8]. In this approach, the branch-and-bound principle seems to be adequate given that it is based on successive improvements of lower and upper bounds for the problem at hand [10]. In general, the best-first strategy is more appropriate in terms of its ability to generate lower bounds closer to the value of an optimal schedule.

Briefly speaking, a branch-and-bound algorithm searches for an optimal schedule by recursively constructing a search tree using a branching procedure. Each node of this search tree should be viewed as the initial problem where some tasks have been scheduled. A branching procedure takes a subproblem as an input and generate a set of new subproblems. Each new subproblem corresponds to the original one plus some tasks scheduled. The root of the search tree is the initial problem, and the leaves are schedules. A depth-first branch-and-bound algorithm was proposed in [7]. In this algorithm, the branching procedure builds a search tree whose height depends on the time for execution of each module. In situations where these times for execution are relatively large, the height of the search tree becomes huge. In this paper, we propose a branching procedure that generates a search tree whose height depends linearly on the number of modules to be scheduled. Using this branching procedure, several different branch-and-bound search strategies can be implemented. For instance, a depth-first strategy as in [7] can be implemented. A positive feature of this strategy is that it requires relatively small storage space.

The following sections are organized as follows. A more formal statement of the problem and the mathematical definitions used in this paper are presented in Section 2. In Section 3, we describe an enumeration algorithm based on our branching procedure. The proof of correctness of this enumeration algorithm is given in Section 4, which yields that our branching procedure performs correctly in constructing the search tree. Section 5 is devoted to the complexity analysis of our branching procedure. Finally, the conclusions and remarks for further research are given in Section 6.

2 Definitions and statements

In this section, the MSP is formally stated.

2.1 The multiprocessor system

In this formulation, the following assumptions on multiprocessor systems are made.

I. We are given a set of m identical processors $\mathcal{P} = \{p_1, \cdots, p_m\}$.

II. The interconnection network is a fully connected network of identical communication links.

III. Each processor executes at most one task at a time.

IV. Task preemptions are not allowed.

V. Each processor can compute and communicate through several of its links simultaneously.

These assumptions can be considered "realistic" since they can be verified in several commercial parallel computers, as IBM SP2, Cray T3D, or CM 5. With relation to assumption II, we consider that the interconnection network can be physically or virtually fully connected. What is required is that the communication cost is independent of the processor sending and receiving a message.

2.2 The task digraph

We now define more formally the parameters characterizing a program. Each module of the program to be scheduled is called a *task*. A task is said to be *scheduled* when it is allocated to be executed on a processor at a given start time. The program is described by a (connected) *directed acyclic graph (DAG)*, whose vertices represent the n tasks $\mathcal{T} = \{t_1, \cdots, t_n\}$ to be scheduled and edges represent the *precedence relations* between pairs of tasks. An edge (t_{i_1}, t_{i_2}) in the DAG is equivalent to a communication between the tasks t_{i_1} and t_{i_2}, taking place after the execution of t_{i_1} and before the execution of t_{i_2}. In this case, t_{i_1} is called the *immediate predecessor* of t_{i_2}, which itself is the *immediate successor* of t_{i_1}. The task t_1 is the only one with no immediate predecessors and t_n is the only task with no immediate successors. A *path* $< t_{i_1}, t_{i_k} >$ is a sequence of vertices $\{t_{i_1}, t_{i_2}, \cdots, t_{i_k}\}$, $1 < k \leq n$, such that each t_{i_h} is an immediate predecessor of $t_{i_{h+1}}$, $1 \leq h < k$. We say that a task t_{i_1} is a *predecessor* of another task t_{i_k} if there exists a path $< t_{i_1}, t_{i_k} >$ in the DAG. Similarly, t_{i_k} is called a *successor* of t_{i_1}.

In order to evaluate and compare schedules, we assign costs to tasks and communications. The execution of each task t_i on any processor costs $e(i)$, which is a linear combination of the number of elementary operations related to t_i and the time to execute each elementary operation on any processor. The communication between two tasks t_{i_1} and t_{i_2} costs $c(i_1, i_2)$ if t_{i_1} and t_{i_2} are scheduled on different processors, and zero otherwise. In the former case, $c(i_1, i_2)$ is a combination of the number of bytes communicated and the performance parameters of the communication links. This combination can be linear or not, depending on the model adopted to the interconnection network. In our experiments, we only consider a linear combination (see Section 6).

2.3 Minimal schedules

The schedules considered are those whose computation of the start times for the tasks is done in a special way. We only consider the earliest start time of each task t_i on any processor p_j taking into account the precedence relations. For each task t_i, denote $p(t_i, S_r)$ and $r(t_i, S_r)$, respectively, the processor and the rank in this processor of t_i under the schedule S_r. The computation of the introduction dates for the tasks in the schedule S_r follows a *list heuristic* whose principle is to schedule each task t_i to $p(t_i, S_r)$ according to its rank $r(t_i, S_r)$. In addition, the task is scheduled as soon as possible depending on the schedule of its immediate predecessors. We call these (partial) schedules *minimal (partial) schedules* (for example, S_r of Figure 1).

A list heuristic builds a schedule step by step. At each step, the tasks that can be scheduled are those whose all predecessors have already been scheduled (free tasks). Then, we choose one of such tasks, say t_i, according to a certain rule R_1. Additionaly, we choose, according to another rule R_2, a processor, say p_j, to which t_i will be scheduled. We then schedule t_i to p_j as soon as possible. This algorithm finishes when all tasks have been scheduled. At an iteration k of this algorithm, let $O(k)$ be the set of tasks remaining to be scheduled, and $O_F(k)$ be the set of free tasks from $O(k)$. Initially, $O(0) = \mathcal{T}$ and $O_F(0) = \{t_1\}$. Thus, at an iteration $k > 0$, we choose a task t_i from $O_F(k)$, we take it out from both $O(k)$ and $O_F(k)$, and we schedule it to $p(t_i, S_r)$, as soon as possible. This algorithm finishes when $O_F(k) = \emptyset$. It is clear that the schedule obtained is minimal with respect to the makespan.

2.4 The multiprocessor scheduling problem

In what follows, we use the notation below:

S_r: represents a *partial schedule* where r tasks, $0 \leq r \leq n$, are scheduled. For an example, see Figure 1. An *initial task* is a task, scheduled on some processor p_j in S_r, with the smallest start time among the tasks scheduled on p_j in S_r. Similarly, a *terminal task* is that with the largest completion time. A schedule S_n is said to be *attainable from* S_r if the tasks that are scheduled in S_r are scheduled in S_n on the same processor and with the same start time as in S_r. Notice that to each partial schedule S_r is associated a set of schedules attainable from S_r. If the tasks $t_{i_1}, t_{i_2}, \cdots, t_{i_r}$ are scheduled in S_r, then the set $NT(S_r) = \mathcal{T} \backslash \{t_{i_1}, t_{i_2}, \cdots, t_{i_r}\}$ is the set of *non-scheduled* tasks of S_r;

$FT(S_r)$: set of *free tasks*, i.e., the non-scheduled tasks of a given S_r whose all predecessors have already been scheduled. For an illustration, see Figure 1;

$S_r(j)$: set of tasks scheduled on p_j in S_r;

$g(S_r)$: load of S_r, given by its time to completion or *makespan*,

$$g(S_r) = \max_{1 \leq j \leq m} g(S_r(j)),$$

where $g(S_r(j))$ is the total execution time of the tasks scheduled on p_j. See Figure 1.

In Figure 1, an example of an instance of the MSP consisting of a DAG with five tasks to be scheduled to three processors is shown, where many of the above parameters are illustrated.

The MSP can be stated formally as the search of a schedule that minimizes the makespan. The MSP can hence be stated as follows.

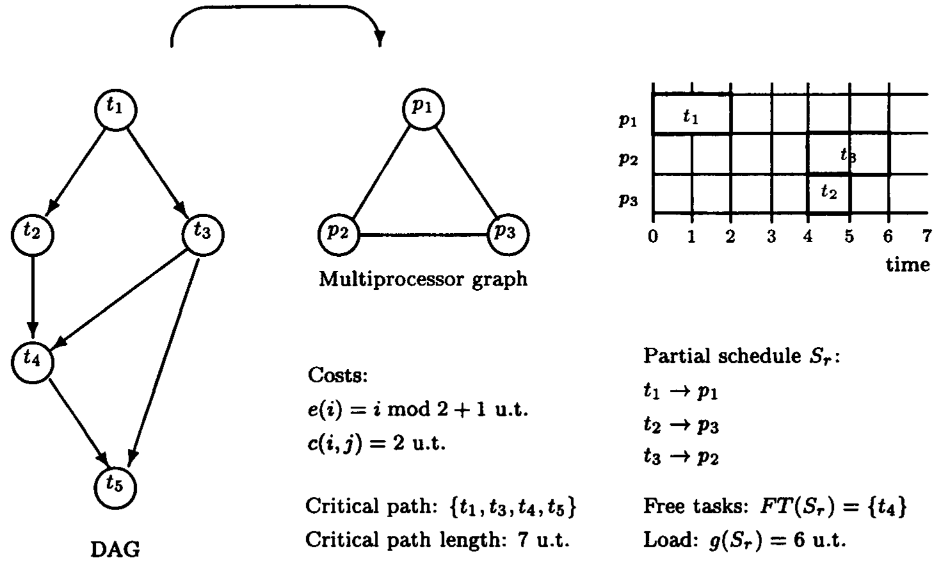


Figure 1: Example of a scheduling problem.

$$\begin{array}{ll} \textit{minimize} & g(S_n) \\ \textit{such that} & S_n \text{ is a minimal schedule.} \end{array}$$

3 All minimal schedules enumeration algorithm

In this section, we concentrate our discussion on enumerating all minimal schedules based on a branchig procedure in such a way that each partial schedule is enumerated exactly once. Starting from an algorithm that enumerates all minimal schedules at least once, we reach an enumeration algorithm where each partial schedule is visited exactly once, by two successive refinements on the branching procedure used.

An enumeration algorithm consists of a sequence of *iterations*, as can be seen in Figure 2. In each iteration, a partial schedule S_r in a *list* $\mathcal{L}$ is *selected* and *split*. Informally speaking, splitting a partial schedule S_r means that if $r = n$, then it is a minimal schedule that is eliminated from the enumeration; otherwise, the set of schedules attainable from S_r is split, which generates a number of new partial schedules to be *inserted* in $\mathcal{L}$. Each schedule attainable from S_r is also attainable from some of the new partial schedules. The execution of the algorithm starts with the list $\mathcal{L}$ containing the partial schedule S_0. The execution finishes when $\mathcal{L} = \emptyset$, meaning that all schedules were enumerated. More formally, an iteration of the enumeration algorithm is composed of three rules:

1. *Selection of partial schedules:* the rule for selecting partial schedules from the list $\mathcal{L}$. See line 2 of the algorithm in Figure 2.

2. *Splitting:* given a partial schedule S_r, this rule generates a set of new partial schedules $S^1_{r+1}, S^2_{r+1}, \cdots, S^l_{r+1}$, each of which different from the others and consisting of S_r plus exactly one task $t_i \in FT(S_r)$ scheduled on some processor. Notice that the set of schedules attainable from each new schedule represents a subset of the set of schedules attainable from S_r. This rule is applyed as in line 3 of the algorithm in Figure 2, and corresponds to the branching procedure.

3. *Insertion of partial schedules:* the rule for inserting partial schedules into the list $\mathcal{L}$,as shown in lines 1 and 4 of the algorithm in Figure 2.

In what follows, we specify the implementation of each of the rules defined above.

```
    list $\mathcal{L}$;   /* initially empty */

    algorithm all_minimal($S_0$):
    list $SPL$;
    partial schedule $S_r$;
1.      insertion($\{S_0\}, \mathcal{L}$);
        while $\mathcal{L} \neq \emptyset$ do
2.          $S_r \leftarrow$ selection($\mathcal{L}$);
            if ($r = n$) then
                eliminate $S_n$;
            else
3.              $SPL \leftarrow$ splitting($S_r$);
4.          insertion($SPL, \mathcal{L}$);
```

Figure 2: Sequential enumeration of schedules.

3.1 All minimal schedules rules

The following rules implement an algorithm that enumerates all minimal schedules at least once.

3.1.1 Selection

Select the first partial schedule from the list $\mathcal{L}$ in a LIFO (*last in first out*) order.

3.1.2 Splitting

To implement this rule, the tasks are ordered in the decreasing order of their *levels* in the DAG. The *level* lv_i of a task t_i, $i \leq n$, is defined to be the longest path length from t_i to t_n (t_n is at level $e(n)$). To each immediate successor t_{i_k} of t_i corresponds a path c_k from t_i to

t_n. In mathematical terms, the level of t_i is defined as

$$lv_i = \max_k \sum_{t_j \in c_k} e(j).$$

These levels can be computed in $O(n^2)$ time by a longest path algorithm [13]. For the tasks in the same level, the tasks having the largest number of immediate successors are ordered first. Thus, for the sake of simplicity, given two tasks t_{i_1} and t_{i_2}, if

$$(lv_{i_1} < lv_{i_2}) \text{ or } ((lv_{i_1} = lv_{i_2}) \text{ and } (i_1 > i_2)),$$

we say that the level of t_{i_1} is *smaller* than the level of t_{i_2}. Similarly, the level of t_{i_1} is *larger* than the level of t_{i_2} if

$$(lv_{i_1} > lv_{i_2}) \text{ or } ((lv_{i_1} = lv_{i_2}) \text{ and } (i_1 < i_2)).$$

We consider that the tasks are previously ordered, and that this order is expressed by the lexicographic order of the tasks in such a way that, for two tasks t_{i_1} and t_{i_2}, if $i_2 > i_1$ then the level of t_{i_1} is larger than the level of t_{i_2}.

As we have already had the opportunity to discuss, each partial schedule S_r, $r \geq 1$, is generated from another partial schedule S_{r-1} by scheduling a task t_i from $FT(S_{r-1})$ to a processor p_j. We represent the scheduling of this task by $t_i \rightarrow p_j$. Let $\sigma(S_r)$ represent the sequence of task schedulings leading S_0 to a partial schedule S_r, i.e.,

$$\sigma(S_r) = < t_{i_1} \to p_{j_1}, \cdots, t_{i_{r-1}} \to p_{j_{r-1}}, t_{i_r} \to p_{j_r} > .$$

For an illustration of these parameters, see Figure 3. The concepts of *ancestor*^{k} and *founder* shall be introduced in Section 3.3.

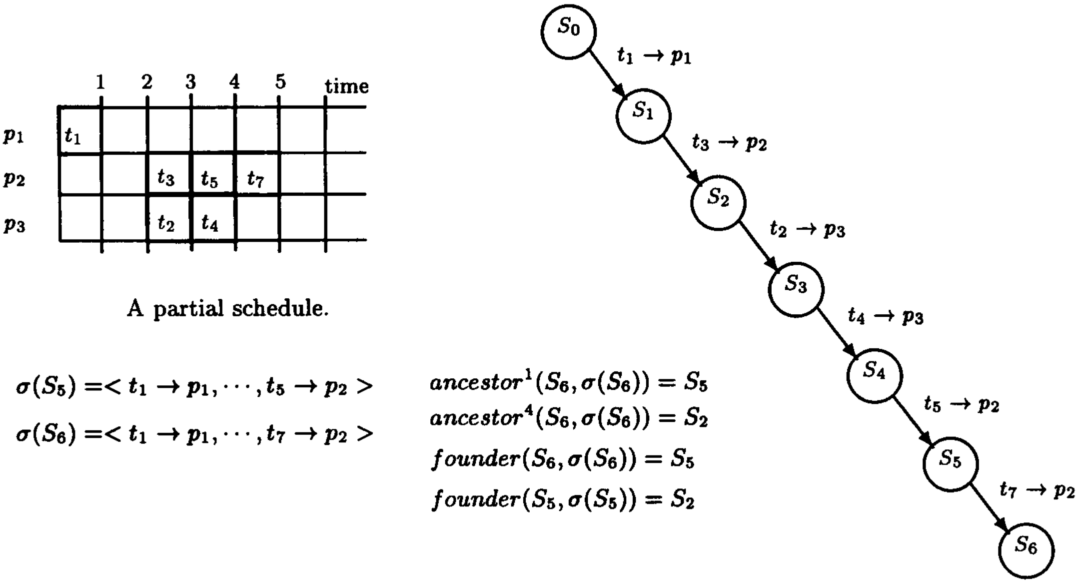


Figure 3: Example of a sequence of splittings.

Each splitting of a partial schedule S_r corresponds to the scheduling of the tasks in $FT(S_r)$ as follows.

Branching rule 1 *Given a partial schedule S_r, $r \geq 1$, and a sequence $\sigma(S_r)$, the splitting rule generates every partial schedule S_{r+1} for which $\sigma(S_{r+1}) = < \sigma(S_r), t_i \to p_j >$ such that $t_i \in FT(S_r)$.*

The idea behind this definition is that the task t_i is scheduled on p_j if and only if t_i is a free task. It is clear that this algorithm enumerates all minimal schedules at least once.

3.1.3 Insertion

Inserts the partial schedules generated by a splitting into the list $\mathcal{L}$.

3.2 Avoiding processor permutations

A drawback of the previous enumeration algorithm is that "equivalent" partial schedules can be generated. In this section, we deal with *processor permutations.*

Definition 1 (Processor permutation) *A partial schedule S_r' is a* processor permutation *of another partial schedule S_r if there is a permutation*

$$\{p_{\pi(1)}, p_{\pi(2)}, \cdots, p_{\pi(m)}\}$$

of the processors such that $S_r'(j_k) = S_r(\pi(k))$, for $k = 1, 2, \cdots, m$.

For an illustration, see Figure 4. The first partial schedule in that figure could be modified by exchanging processors p_2 and p_3, which corresponds to a processor permutation where $\pi(2) = 3$ and $\pi(3) = 2$. In order to avoid these situations, we redefine the splitting rule.

Branching rule 2 *Given a partial schedule S_r, $r \geq 1$, and a sequence $\sigma(S_r)$, the splitting rule generates every partial schedule S_{r+1} for which $\sigma(S_{r+1}) =< \sigma(S_r), t_i \to p_j >$ such that:*

i. $t_i \in FT(S_r)$*; and*

ii. there is no free processor $p_{j'}$, $j' < j$.

In Section 4, we demonstrate the correctness of this splitting rule in avoiding processor permutations.

3.3 Avoiding intersections

A partial schedule S_r could be generated from two (or more) different partial schedules if S_r could be generated by scheduling more than one of its terminal tasks. Recall again Figure 4. The partial schedule S_6 of this figure could be generated by scheduling t_7 on p_2 or t_4 on p_3. We shall define a third splitting rule avoiding this undesirable situation. We need some more formalism.

Definition 2 (Intersection) *If a partial schedule is generated from the splitting of two different partial schedules, then we say that there is an* intersection.

We denote $ancestor^1(S_r, \sigma(S_r))$ the partial schedule obtained by the sequence $\sigma(S_{r-1}) =< t_{i_1} \rightarrow p_{j_1}, \cdots, t_{i_{r-1}} \rightarrow p_{j_{r-1}} >$, for a given S_r. Recursively, we define

$$ancestor^k(S_r, \sigma(S_r)) = ancestor^1(ancestor^{k-1}(S_r, \sigma(S_r)), \sigma(S_{r-1})),$$

for some $k > 1$. Finally, the *founder* partial schedule of a given partial schedule S_r is defined in function of t_{i_r} as follows. If t_{i_r} is an initial task in S_r, then we call $founder(S_r, \sigma(Sr))$ the initial partial schedule S_0; otherwise, if t_{i_r} is not an initial task in S_r, then let t_{i_q} be the task scheduled to p_{j_r} immediately before t_{i_r} in S_r. Also, let S_q be the ancestor of S_r in $\sigma(S_r)$ generated by the scheduling of t_{i_q}. Then, we call $founder(S_r, \sigma(S_r))$ the partial schedule S_q. For an illustration of these parameters, see Figure 3.

An important notion in the splitting operation is the *dependence relation* of task schedulings. Informally, given a partial schedule S_r and two tasks, namely t_{i_1} and t_{i_2}, scheduled in S_r, t_{i_1} is said to be *dependent* of t_{i_2} in S_r if S_r cannot be constructed by scheduling t_{i_1} before scheduling t_{i_2}. More formally, let S_r be a partial schedule, and t_{i_1} and t_{i_2} be two tasks scheduled on p_{j_1} and p_{j_2}, $j_1 \neq j_2$, respectively, in S_r. The task t_{i_1} is *dependent* of t_{i_2} in S_r if (t_{i_2}, t_{i_1}) belongs to the digraph $D(S_r) = (\mathcal{T}, A(S_r))$, where:

$\mathcal{T}$ is the set of tasks; and

$A(S_r)$ is the set of arcs formed by:

1. the transitive closure of the arcs in the DAG and
2. every arc $(t_i, t_{i'})$, if t_i and $t_{i'}$ are scheduled to the same processor in S_r, and t_i is scheduled before $t_{i'}$.

Notice that the dependence relation just defined is transitive and antisymmetric. For the sake of simplicity, we say that t_{i_1} is dependent of t_{i_2} or that t_{i_1} depends of t_{i_2} and we often omit the corresponding partial schedule where it is clear by the context. As an example, consider the partial schedule S_7 in Figure 4. In this case, t_4 depends of t_9 because t_9 is scheduled on p_1 before t_2, and p_2 is a predecessor of t_4. Then, considering S_7, t_4 cannot be scheduled before scheduling t_9.

Each splitting of a partial schedule S_r corresponds to the scheduling of the tasks in $FT(S_r)$ as follows. It is assumed that the representation of each partial schedule contains its corresponding sequence of task schedules.

Branching rule 3 *Given a partial schedule S_r, $r \geq 1$, and a sequence $\sigma(S_r)$, the splitting rule generates every partial schedule S_{r+1} for which $\sigma(S_{r+1}) =< \sigma(S_r), t_i \rightarrow p_j >$ such that:*

i. $t_i \in FT(S_r)$; and

ii. there is no free processor $p_{j'}$, $j' < j$; and

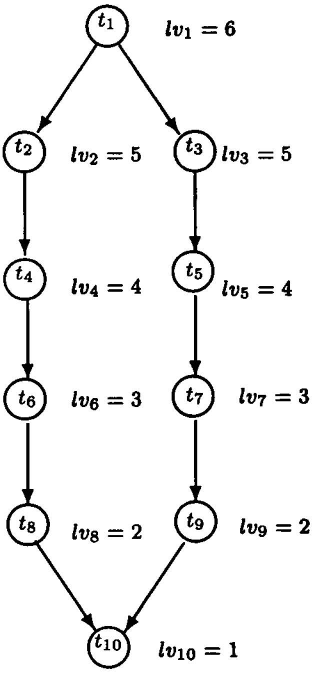
iii. every task $t_{i'}$ scheduled in $\sigma(S_r)$ from $founder(S_r, \sigma(S_r))$ until S_r is such that:

- *$i' < i$; or*
- *t_i is dependent of $t_{i'}$.*

The idea behind this definition is that the task t_i is scheduled to p_j if and only if $\sigma(S_r)$ is the only possibility for the generation of S_r. Hence, if S_r can be generated by another sequence $\sigma'(S_r)$, then the splitting does not generate S_r from $\sigma(S_r)$. Intuitively, we can see that this avoids situations where the same partial schedule is generated several times during an enumeration process. We shall formally see in the next section that this is indeed the case and that all minimal partial schedules are generated nonetheless.

Example 1 *Figure 4 shows a DAG, whose costs are all equal to 1, and two partial schedules. The first one can be generated by scheduling t_4 on p_3 using the splitting rule 3. We can observe that, when scheduling t_4, the tasks t_5, t_7 and t_9 have already been scheduled. However, the scheduling of t_4 is dependent of the scheduling of t_5, t_7 and t_9 since t_2, an immediate predecessor of t_4, is scheduled on p_1 after t_9, which is itself an immediate successor of t_7.*

On the other hand, the second partial schedule cannot be generated by scheduling t_4 on p_3 according to the splitting rule 3 because, in this case, the scheduling of t_4 is not dependent *of t_7 (t_2 is scheduled to p_3). Nevertheless, this second partial schedule can be generated by scheduling t_7 on p_2.*



	1	2	3	4	5	6	7	8	9	10
p_1	t_1						t_9	t_2		
p_2			t_3	t_5	t_7					
p_3										t_4

S_7: A partial schedule that can be generated by scheduling t_4.

p_1	t_1									
p_2			t_3	t_5	t_7					
p_3			t_2	t_4						

S_6: A partial schedule that cannot be generated by scheduling t_4.

Figure 4: Example of splitting.

4 Correctness of the enumeration algorithm

In this section, we provide a proof of correctness of the enumeration algorithm, which guarantees that splitting rule 3 performs correctly, i.e., all minimal partial schedules are generated, but that neither processor permutations nor intersections are generated. The first lemma concerns processor permutations and splitting rule 2.

Lemma 1 *If splitting rule 2 is used during an execution of the enumeration algorithm then, for every (minimal) partial schedule S_r, either it is generated or a processor permutation of*

S_r is generated.

Proof: We demonstrate the lemma by induction on r. For $r = 0$, the lemma is trivially verified since there is no tasks scheduled. Consider a partial schedule S_r, and let t_i and p_j be any terminal task in S_r and the processor to which t_i is scheduled, respectively. Supposing, without loss of generality, that $ancestor^1(S_r, \sigma(S_r))$, the partial schedule that differs from S_r by t_i (which is not scheduled in $ancestor^1(S_r, \sigma(S_r))$, is generated, we show that, if S_r is not generated, then a processor permutation of S_r is. Then, by contradiction, suppose that S_r violates the lemma, i.e. neither S_r is generated nor a permutation of S_r is generated. If S_r is not generated from $ancestor^1(S_r, \sigma(S_r))$ by scheduling t_i to p_j due to splitting rule 2 then there exists a free processor $p_{j'}$ in $ancestor^1(S_r, \sigma(S_r))$, $j' < j$. In this case, it is easy to see that the permutation of S_r where $\pi(j') = j$, $\pi(j) = j'$ and $\pi(\hat{\jmath}) = \hat{\jmath}$ for $\hat{\jmath} \neq j, j'$, is generated. Contradiction. □

In the following lemma, splitting rule 3 is used in order to assure that only one processor permutation of each partial scheduling is generated.

Lemma 2 *If splitting rule 3 is used during an execution of the enumeration algorithm then, for all partial schedules S_r, at most one processor permutation of S_r is generated during an enumeration process.*

Proof: An immediate consequence of the splitting rule 3 is that, for all partial schedules that are generated, if t_i is the initial task of a processor p_j and $t_{i'}$ is the initial task of a processor $p_{j'}$, $j' < j$, than either $i' < i$ or t_i is dependent of the initial task in $p_{j'}$. Using this fact, we prove the lemma by contradiction. Suppose that two processor permutations S_r and S_r' are generated, both satisfying the conditions in splitting rule 3. Let $t_{i(p_1)}, t_{i(p_2)}, \cdots, t_{i(p_n)}$ be the starting tasks of processors $p_1, p_2, \cdots, p_n$, respectively, in S_r, and $t_{i(p_{\pi(1)})}, t_{i(p_{\pi(2)})}, \cdots, t_{i(p_{\pi(n)})}$ the starting tasks in S'_r. Consider processor p_k such that $k \neq \pi(k)$. If $i(p_k) < i(p_{\pi(k)})$ (the situation where $i(p_k) > i(p_{\pi(k)})$ is analogous) then, in S_r, there are $a < n - k$ initial tasks greater than $i(p_{\pi(k)})$ or dependent of $t_{i(p_{\pi(k)})}$. However, in S'_r, there exist $n - k$ initial tasks greater than $i(p_{\pi(k)})$ or dependent of $t_{i(p_{\pi(k)})}$. Thus, since the precedence relation is antisymmetric, if the initial tasks of S_r are ordered according to splitting rule 3, then the same is not true for the initial tasks of S'_r. Contradiction. □

In what follows, we analyze the role played by splitting rule 3 in avoiding intersections. The following lemma says that this rule allows the generation of all minimal partial schedules.

Lemma 3 *If splitting rule 3 is used during an enumeration process, then every partial schedule S_r can be generated or a processor permutation of S_r is generated.*

Proof: From lemmas 1 and 2, we know that splitting rule 3 guarantees that at most one processor permutation can be generated. In order to demonstrate lemma 3, we will show that

$$\begin{array}{l}\text{for each partial schedule } S_r \text{ whose generation is allowed by splitting rule 2,}\\ \text{there exists a sequence } \sigma(S_r) \text{ that is generated with splitting rule 3.}\end{array} \tag{1}$$

We will demonstrate (1) by induction on r. Again, (1) is trivially verified for $r = 0$ since there is no task scheduled. We suppose (1) valid for all partial schedules containing $r - 1$ tasks scheduled, then we show that (1) is also valid for a partial schedule S_r, $r > 0$, whose generation is allowed by splitting rule 2. Suppose a sequence $\sigma(S_r)$ of task schedulings. If

$\sigma(S_r)$ can be generated with splitting rule 3, then the lemma is proved. Otherwise, we will exhibit another sequence that is generated.

Let $t_{i'} \rightarrow p_{j'}$ be a task scheduling in $\sigma(S_r)$ such that all tasks scheduled after $t_{i'}$ in $\sigma(S_r)$ are not dependent of $t_{i'}$. Such a task exists if splitting rule 3 is not satisfied. Let S_{r-1} be the partial schedule whose task schedulings are those of S_r but for $t_{i'} \rightarrow p_{j'}$. By lemma 1, there exists a processor permutation $\Pi(S_{r-1})$ that is generated using splitting rule 2 and, by the induction hypothesis, there exists a sequence $\sigma(\Pi(S_{r-1}))$ that is generated. Finally, we include $t_{i'} \rightarrow p_{\pi(j')}$ to $\sigma(\Pi(S_{r-1}))$, which can be done since all tasks on which $t_{i'}$ is dependent are scheduled in $\sigma(\Pi(S_{r-1}))$. □

Lemma 4 *If splitting rule 3 is used during the execution of the enumeration algorithm then there are no intersections.*

Proof: Once more, the proof is by induction on r, being the basis of the induction ($r = 0$) trivially satisfied. Suppose that splitting rule 3 renders the lemma valid for all partial schedules with less than r, $r \geq 1$, tasks scheduled. Let S_{r-1} be a partial schedule with $r-1$ tasks scheduled, and $\sigma(S_{r-1})$ its sequence of task schedulings (which is unique by the induction hypothesis). Additionally, let S_r and S'_r be two partial schedules with r tasks scheduled such that there exist two sequences $\sigma(S_r) =< \sigma(S_{r-1}), t_i \rightarrow p_j >$ and $\sigma(S'_r)$. Two cases are possible:

Case 1: The sequences of task schedulings generating S'_r do not include $\sigma(S_{r-1})$. Then, there exists a partial schedule S_{l-1}, $l < r$, such that $S_{l-1} \in \sigma(S_r)$ and $S_{l-1} \in$

$\sigma(S'_r)$, but $S'_l \notin \sigma(S'_r)$ and $S'_l \notin \sigma(S_r)$, where

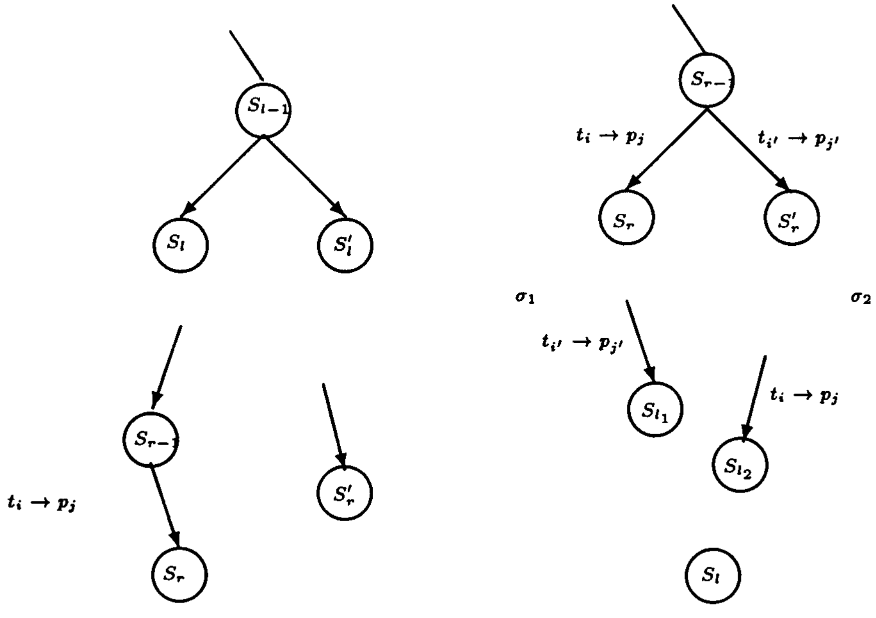
$$ancestor^1(S_l, \sigma(S_l)) = ancestor^1(S'_l, \sigma(S'_l)) = S_{l-1}.$$

See Figure 5. By the induction hypothesis, S_l and S'_l do not intersect. As a consequence, S_r and S'_r do not intersect because $S_l \subset S_r$ and $S'_l \subset S'_r$.

Case 2: The sequence $\sigma(S'_r)$ of task schedules generating S'_r contains $\sigma(S_{r-1})$. Let $\sigma(S'_r) =< \sigma(S_{r-1}), t_{i'} \rightarrow p_{j'} >$. By contradiction, let S_l, $l > r$, be a partial schedule such that there exist two disjoint sequences of task schedulings leading S_r and S'_r to S_l, respectively, as shown in Figure 5. Let σ_1 and σ_2 be such sequences. Clearly, $(t_{i'} \rightarrow p_{j'}) \in \sigma_1$ and $(t_i \rightarrow p_j) \in \sigma_2$ since S_l contains both S_r and S'_r. As shown in Figure 5, we define S_{l_1} to be the partial schedule generated by $t_{i'} \rightarrow p_{j'}$ in σ_1, and S_{l_2} to be the partial schedule generated by $t_i \rightarrow p_j$ in σ_2. Additionally, $\sigma(S_{l_1})$ and $\sigma(S_{l_2})$ being two sequences generating S_{l_1} (containing $\sigma(S_r)$ and $t_{i'} \rightarrow p_{j'}$) and S_{l_2} (containing $\sigma(S'_r)$ and $t_i \rightarrow p_j$), respectively, we observe that $founder(S_{l_1}, \sigma(S_{l_1}))$ and $founder(S_{l_2}, \sigma(S_{l_2}))$ are generated by task schedulings in $\sigma(S_{r-1})$ because $t_{i'}$ and t_i must be the first tasks scheduled to, respectively, $p_{j'}$ and p_j after S_{r-1}. Thus, $t_{i'} \rightarrow p_{j'}$ and $t_i \rightarrow p_j$ violate splitting rule 3 in σ_1 or σ_2. Contradiction.

□

We consider in the following theorem the general case where no other equivalence among the partial schedules than processor permutations and intersections occurs. For particular



Case 1

Case 2

Figure 5: Two cases in the proof of lemma 4.

instances where equivalence among the partial schedules can be identified, the splitting rule 3 is no more a necessary condition.

Theorem 1 *An enumeration of partial schedules generates a minimum number of (minimal) partial schedules if and only if splitting rule 3 or an equivalent is used.*

Proof: To prove the "if" assertion, we examine the situation where a weaker rule is used. In this case, partial schedules not generated using splitting rule 3 are generated. Then, by lemma 1, we know that all partial schedules are generated using splitting rule 3. Consequently, redundant partial schedules are generated in the case where the weaker rule is used (processor permutations or intersections).

For the necessary condition, we suppose that a stronger rule is used. In this case, contrary to the previous case, there exist some partial schedules that are not generated, but that are generated using splitting rule 3. Using lemmas 2 and 4, we verify that all cases of processor permutations and intersections are already avoided. Consequently, the stronger rule may avoid the generation of some minimal partial schedules. □

5 Complexity analysis

In this section, we show that a splitting procedure derived from the splitting rule 3 has polynomial time and space complexities. For this purpose, call two partial schedules S_r and S'_r *different* if S_r is not a processor permutation of S'_r, and S_r and S'_r do not intersect. Let us consider the following problem.

> **Splitting problem:** given an instance of the MSP and a generic set R of different partial schedules with exactly r tasks scheduled, $0 \leq r < n$, list all different partial schedules S_{r+1} such that $S_{r+1} = < \sigma(S_r), t_i \rightarrow p_j >$, where $S_r \in R$, $t_i \in FT(S_r)$ and $p_j \in \mathcal{P}$.

Clearly, a procedure that solves the splitting problem is able to enumerate all different partial schedules if applied recursively. In order to analize the complexity of enumerating all minimal schedules avoiding processor permutations and intersections based on the splitting rule 3, we will describe a splitting procedure derived from the splitting rule 3 and analyze its complexity. This splitting procedure solves the splitting problem, then its complexity is an upper bound for all splitting procedures solving the splitting problem. We define some specific functions to be used in the procedure in Figure 6. This procedure checks splitting rule 3. Suppose $\sigma(S_r) = < t_{i_1} \rightarrow p_{j_1}, \cdots, t_{i_r} \rightarrow p_{j_r} >$. Then, define the function

$prev(i_k, \sigma(S_r))$, $1 \leq k \leq r$, to be the function mapping i_k to i_{k-1}, if $k > 1$, or to -1 otherwise. Equivalently, define $next(i_k, \sigma(S_r)) = i_{k+1}$, for all $1 \leq k < r$. Additionaly, define $proc(i_k, \sigma(S_r))$, $1 \leq k \leq r$, as the processor on which t_{i_k} is scheduled in $\sigma(S_r)$, i.e., j_r.

```
        algorithm split($DAG, \sigma(S_r)$):
        integer mark[m];          /* all initialized with FALSE */
        integer $i, j, i'$;
            $i \leftarrow prev(i_r, \sigma(S_r))$;
            $j \leftarrow proc(i, \sigma(S_r))$;
1.          while ($i \geq 0$) AND ($j \neq j_r$) do
                if $i > i_r$ then
                    if NOT mark[j] then
                        $i' \leftarrow next(i, \sigma(S_r))$;
2.                      while ($i' \neq i_r$) AND (NOT mark[$proc(i', \sigma(S_r))$]) AND
                        (NOT ($t_i \rightarrow t_{i'}$))do
                            $i' \leftarrow next(i', \sigma(S_r))$;
                        if ($i' = i_r$) then
3.                          return FALSE;
                        else
                            mark[j] $\leftarrow$ TRUE;
            return TRUE;
```

Figure 6: Verifying splitting rule 3.

Lemma 5 below says that the algorithm *splitting* in the Figure 7, which corresponds to our splitting procedure, solves the splitting problem when applied to all partial schedules in the set R.

Lemma 5 *For a given S_r, splitting rule 3 is verified if and only if branch returns TRUE.*

Proof: What procedure *branch* does is to check, for each task t_i from $founder(S_r, \sigma(S_r))$ until the last task scheduled in $\sigma(S_r)$ (t_{i_r}), whether $i > i_r$ or whether t_{i_r} depends of t_i in S_r. The latter is equivalent to check whether some of the immediate successors of t_i is scheduled on a processor whose terminal task t_{i_k} verifies the follwoing property: t_{i_r} is dependent of t_{i_k} in S_r. Since procedure *branch* examine the tasks in the loop of line 1 in the inverse order of their schedulings, then, for every processor $p_{j'}$ different from p_j, the first task examined among those scheduled on $p_{j'}$ is the terminal task. Additionaly, when an arbitrary task t_i examined in the loop of line 1, then its immediate successors scheduled in S_r have already been examined. Then, it follows that line 3 is executed if and only if splitting rule 3 is not verified. □

The time complexity of *branch* is determined by the loops in the lines 1 and 2. The time required by these loops in the worst case is bounded by $\sum_{l=1}^{r} l$, that is, $O(n^2)$. However, it is clear that the average case turns in time much smaller than the worst case. The storage requirements is $O(1)$.

```
        algorithm splitting($S_r$):
        partial schedule $S_{r+1}$;
        set of partial schedules $\mathcal{S}$;
            $\mathcal{S} \leftarrow \emptyset$;
1.          for each task $t_i \in FT(S_r)$ do
2.              for each processor $p_j \in \mathcal{P}$ s.t.
                there is no free processor $p_{j'}$, $j' < j$ do
3.                  if $branch(DAG, \sigma(S_r))$ then
                        $S_{r+1} \leftarrow S_r \cup (t_i \rightarrow p_j)$;
                        $FT(S_{r+1}) \leftarrow FT(S_r) \backslash \{t_i\}$;
4.                      $\sigma(S_{r+1}) \leftarrow < \sigma(S_r), t_i \rightarrow p_j >$;
                        $\mathcal{S} \leftarrow \mathcal{S} \cup \{S_{r+1}\}$;
            return $\mathcal{S}$;
```

Figure 7: The splitting procedure.

The time complexity of the splitting procedure is determined by four components. First, the loop in the line 1 is executed $O(n)$ times, while the loop in the line 2 is executed

$O(m)$ times. Checking splitting rule has time complexity $O(n^2)$ (line 3). Finally, setting $\sigma(S_{r+1})$ in line 4 takes $O(n)$ time. The additional storage space required by *splitting*, besides the $O(m + n^2)$ storage space required for representing the DAG and the multiprocessor system, corresponds to the variables $\mathcal{S}$ and $\sigma(S_{r+1})$. These storage space requirements are, respectively, $O(mn)$ and $O(n)$.

We have proved the following theorem, which indicates, as desired, that our splitting procedure requires polynomial time and storage space.

Theorem 2 *The time complexity of the splitting problem is $O(mn^3)$ and requires $O(mn)$ storage space.*

We conclude this section with the following recall: a splitting procedure based on the splitting rule 1 turns in $O(mn)$ in the worst and average cases, but generating a lot of processor permuations and intersections. Consequently, *splitting* performs much better in practice.

6 Concluding remarks

In this paper, considered the multiprocessor scheduling problem. Being an NP-hard problem in the strong sense, branch-and-bound algorithms appear to be an adequate method for finding approsimated solutions with proved accuracy. In order to efficiently implement such algorithms, the branching problem must be faced up. We have shown in this paper that this problem is polynomial in time and storage space. Therefore, we proposed a polynomial time and storage space branching procedure. This implies that a branch-and-bound algorithm whose associated search tree has height equal to the number of tasks in the MSP instance can be efficiently implemented. This in opposition to branching procedures in the literature where the height of the search tree is proportional to the costs of the tasks. A branch-and-bound algorithm using the branching procedure proposed in this paper applies mainly to instances of the MSP where the costs involved are large.

Acknowledgments

We would like to thank the substantial help granted by Gregory Mounié and Pascal Rebreyend. We are also grateful to Maria Cristina Boeres, João Paulo Kitajima and Vinod Rebello for their useful suggestions.

References

[1] T. Casavant and J. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2), February 1988.

[2] E. Coffman. *Computer and Job-Shop Scheduling Theory.* Wiley, New York, 1976.

[3] R. Corrêa, A. Ferreira, and P. Rebreyend. Integrating list heuristics into genetic algorithms for multiprocessor scheduling. In *IEEE Symposium on Parallel and Distributed Processing*, New Orleans, USA, October 1996. To appear.

[4] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures.* International Thomson Computer Press, 1995.

[5] M. Garey and D. Johnson. Strong np-completeness results: Motivation, example and implications. *Journal of the ACM*, 25:499–508, 1978.

[6] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. F. Freeman, 1979.

[7] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, November 1984.

[8] J.P. Kitajima. *Modèles Quantitatifs d'Algorithmes Parallèles.* PhD thesis, Institut National Polytechnique de Grenoble, October 1994.

[9] B. Malloy, E. Lloyd, and M. Soffa. Scheduling DAG's for asynchronous multiprocessor execution. *IEEE Transactions on Parallel and Distributed Systems*, 5(5), May 1994.

[10] L. Mitten. Branch-and-bound methods: General formulation and properties. *Operations Research*, 18:24–34, 1970. Errata in *Operations Research*, 19:550, 1971.

[11] M. Norman and P. Thanisch. Models of machines and computations for mapping in multicomputers. *ACM Computer Surveys*, 25(9):263–302, Sep 1993.

[12] C. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

[13] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall Inc., Englewood Cliffs, NJ, 1982.

[14] P. Pardalos and H. Wolkowicz, editors. *Quadratic assignments and related problems*, volume 16 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, 1994.