A Polynomial-Time Branching Procedure for the Multiprocessor Scheduling Problem

Ricardo C. Corrêa[1] and Afonso Ferreira[2]

[1] Departamento de Computação, Universidade Fedeal do Ceará,
Campus do Pici, Bloco 910, 60455-760, Fortaleza, CE, Brazil,
correa@lia.ufc.br

[2] CNRS. Projet SLOOP, INRIA Sophia Antipolis, 2004, Route des Lucioles,
06902 Sophia Antipolis Cedex, France

Abstract. We present and analyze a branching procedure suitable for best-first 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.

1 Introduction

Nowadays, most *multiprocessor systems* consist of a set $\mathcal{P} = \{p_1, p_2, \cdots, p_m\}$ of $m > 1$ identical processors with distributed memory and communicating exclusively by message passing through a completely interconnected network. A common feature of many of the algorithms designed for these systems is that they can be described in terms of a set of *modules* to be executed under a number of *precedence constraints*. A precedence constraint between two modules determines that one module must finish its execution *before* the other module starts its execution. In this paper, we consider the following *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 (or *makespan*) is *minimized* (for an overview of this optimization problem, see [1, 3, 8] and references therein).

We call *schedule* any solution to the MSP. In general, finding an optimal schedule to an instance of the MSP is computationally hard [3, 4]. The branch-and-bound approach has been used to find exact or approximate solutions for the MSP (for further details, including computational results, see [2, 5, 6]). Briefly speaking, a *branch-and-bound* algorithm searches for an optimal schedule by recursively constructing a search tree, whose root is the initial problem, internal nodes are *subproblems*, and the leaves are schedules. This search tree is built using a *branching procedure* that takes a node as an input and generates its children, such that each child is a subproblem obtained by the scheduling of some tasks that were not yet scheduled in its parent. Although several techniques

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 272–279, 1999.
© Springer-Verlag Berlin Heidelberg 1999

can be used to reduce the number nodes of the search tree which are indeed visited during a branch-and-bound search [7], the branching procedure is called many times during a branch-and-bound algorithm. We study in this paper a new such procedure, rather than branch-and-bound algorithms as a whole. As a consequence, we are able to focus our work on the techniques required to speedup this important component of the branch-and-bound method.

The solutions reported in the literature are not satisfactory enough to deal with this central problem. Kasahara and Narita [5] have proposed such a procedure and applied it in a depth-first branch-and-bound algorithm. Unfortunately, their approach generates non-minimal schedules and the height of the search tree depends on the task costs. Thus, it is not suitable for the best-first search context intended to be covered by the branching procedure proposed in this paper.

More recently, another approach, based on state space search techniques, was proposed [2, 6]. In this case, only minimal schedules are generated. However, this branching procedure constructs a search **graph** (not necessarily a tree), which means that each subproblem can be obtained from different sequences of branchings from the initial problem (for more details, see Sections 3.1 and 3.2). For this reason, every new subproblem generated by the branching procedure should be compared with all subproblems previously generated during the search in order to eliminate duplicated versions of identical subproblems. This fact drastically affects the time and space complexities of the branching procedure.

In this paper, we propose a **polynomial time** branching procedure that generates a search **tree** (not a general graph, *i.e.* each subproblem is generated exactly once). In this search tree, an edge indicates the assignment of a task to a processor, the leaves are feasible schedules, and a path from the root to a leaf defines a total order on the task assignements. In addition, the branching procedure stablishes a set of total orders on the tasks verifying the precedence constraints. If a feasible schedule could be associated to more than one task assignment order (path in the search tree), then only the path which is coherent with one of the total orders stablished by the branching procedure is generated.

This strategy requires relatively small storage space, and the height of the generated search tree depends linearly upon the number of modules to be scheduled. Furthermore, the characteristics of our procedure enables it to be easily plugged into most existing branch-and-bound methods. The branch-and-bound algorithms thus obtained must outperform the original ones in most cases, provided that either the new branching procedure is faster (*e.g.*, when compared to [2, 6]) or the height of the search tree is smaller (*e.g.*, when compared to [5]).

The remainder of the paper is organized as follows. 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 and the complexity analysis of our branching procedure are given in Section 4. Finally, concluding remarks are given in Section 5.

2 Definitions and Statements

Each module of the program to be scheduled is called a *task*. The program is described by a (connected) *directed acyclic graph (DAG)*, whose vertices represent the $n > 0$ 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 precedence relation between the tasks t_{i_1} and t_{i_2}, which means that t_{i_1} must finish its execution and send a message to t_{i_2} which must be received before the execution of t_{i_2}. In this case, t_{i_1} is called the *immediate predecessor* of t_{i_2}. The task t_1 is the only one with no immediate predecessors. The execution of any task on any processor and the communication between any pair of tasks costs one time unit each. One may notice that this assumption of identical costs does not yield any loss of generality in terms of the structural issues considered in the rest of the paper.

A *schedule* is a mapping $S_n : \mathcal{T} \rightarrow \mathcal{P} \times \mathbb{N}$, which indicates that each task t_i is allocated to be executed on processor $p(i, S_n)$ with rank $r(i, S_n)$ in this processor. The schedules considered are those whose computation of the start time of each task takes into account three conditions, namely: (i) precedence relations; (ii) each processor executes at most one task at a time; and (iii) task preemptions are not allowed. Thus, given a schedule S_n, the computation of the introduction dates $d(i, S_n)$, for all $t_i \in \mathcal{T}$, follows the *list heuristic* whose principle is to schedule each task t_i to $p(i, S_n)$ according to its rank $r(i, S_n)$ [3]. In addition, $d(i, S_n)$ assumes the minimum possible value depending on the schedule of its immediate predecessors. We call these (partial) schedules *minimal (partial) schedules*.

Let us define some notation. A *partial schedule* where r tasks, $0 \leq r \leq n$, are scheduled is represented by S_r. An *initial task* is a task, scheduled on some processor p_j in S_r, with rank 1. 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. The set of *free tasks*, *i.e.*, the non-scheduled tasks of a given S_r whose all immediate predecessors have already been scheduled, is denoted by $FT(S_r)$. In Figure 1, an example of an instance of the MSP is shown, where many of the above parameters are illustrated. The MSP can be stated formally as the search of a minimal schedule S_n that minimizes the makespan $d(n, S_n) + 1$.

3 Enumerating All Minimal Schedules

In this section, we concentrate our discussion on enumerating all minimal schedules 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 a refinement on the branching procedure used.

An enumeration algorithm consists of a sequence of *iterations*. Throughout these iterations, the state of the list of partial schedules $\mathcal{L}$ evolves from the initial state $\{S_0\}$ (partial schedule with no tasks scheduled) until it becomes idle.

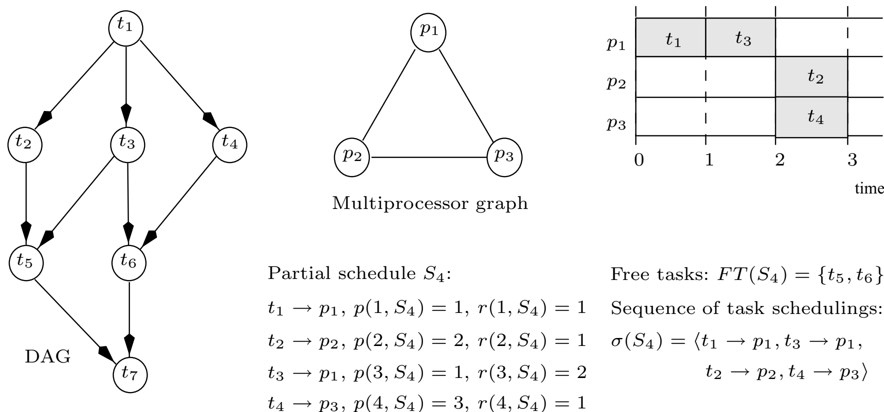

Fig. 1. Example of a scheduling problem and a partial schedule.

Each iteration starts selecting a partial schedule $S_)r$ from the list $\mathcal{L}$. Whenever a partial schedule is selected, it is deleted from $\mathcal{L}$. Then, a branching procedure 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. Finally, the iteration ends inserting the partial schedules generated into the list $\mathcal{L}$. When a partial schedule S_r is inserted into $\mathcal{L}$, it becomes available for further selection.

3.1 All Minimal Schedules Branching Rule

The following branching rule implements an algorithm that enumerates each minimal schedule at least once, although a number of partial schedules can be enumerated more than once. One could modify the enumeration algorithm to employ an A* strategy to eliminate the occurrence of duplicated partial schedules, as proposed in [2, 6]. Our approach, to be presented in Section 3.2, is to refine this section's branching procedure, so that it prevents duplicated partial schedules from being generated at all.

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 free task $t_i \in FT(S_{r-1})$ to a processor p_j. We represent the scheduling of this task by $t_i \rightarrow p_j$. The following rule leads the enumeration algorithm to enumerate all minimal schedules at least once (see Lemma 1 in Section 4). The branching procedure adopted in [2, 6] is based on this rule.

Branching rule 1 *Given a partial schedule S_r, $n > r \geq 0$, every partial schedule S_{r+1} consisting of S_r plus $t_i \rightarrow p_j$, for all $p_j \in \mathcal{P}$ and $t_i \in FT(S_r)$, is generated.*

Denote by $\Sigma(B_1)$ the multiset of partial schedules enumerated by the enumeration algorithm with a branching procedure based on B_1. This is not necessarily a set because some partial schedules can occur several times in $\Sigma(B_1)$.

3.2 Avoiding Processor Permutations and Duplications

A first drawback of the previous enumeration algorithm is that "equivalent" partial schedules can be enumerated. This equivalence relation is related to processor permutations. A partial schedule $S_r^{'}$, denoted $S_r' = \Pi(S_r)$, is a *processor permutation* of another partial schedule S_r, $n \geq r \geq 0$, if there is a permutation $\pi : \{1, 2, \cdots, m\} \rightarrow \{1, 2, \cdots, m\}$ of the processors such that $p(t_i, S_r') = \pi(p(t_i, S_r))$ and $r(t_i, S_r') = r(t_i, S_r)$, for all t_i scheduled in S_r. For the sake of illustration, the first partial schedule in Figure 1 could be modified by exchanging processors p_2 and p_3, which corresponds to a processor permutation where $\pi(2) = 3$ and $\pi(3) = 2$.

Branching rule 1 suffers from a second drawback. Let $\sigma(S_r)$ represent the sequence of task schedulings leading S_0 to a partial schedule S_r, i.e., $\sigma(S_r) = \langle t_{i_1} \rightarrow p_{j_1}, \cdots, t_{i_{r-1}} \rightarrow p_{j_{r-1}}, t_{i_r} \rightarrow p_{j_r} \rangle$. A partial schedule S_r can be generated from two (or more) distinguished partial schedules if S_r can be generated by two (or more) sequences of task schedulings. Recall again Figure 1. Besides the sequence of task schedulings shown in that figure, rule B_1 allows the partial schedule S_4 to be generated with $\langle t_1 \rightarrow p_1, t_2 \rightarrow p_2, t_4 \rightarrow p_3, t_3 \rightarrow p_1 \rangle$. The next branching rule eliminates these two anomalies.

Branching rule 2 *Given a partial schedule S_r, $n > r \geq 0$, every partial schedule S_{r+1} consisting of S_r plus $t_i \rightarrow p_j$, for all $p_j \in \mathcal{P}$ and $t_i \in FT(S_r)$, is generated if and only if the following conditions hold:*

i. there is no empty processor $p_{j'}$ in S_r, $j' < j$, where a empty processor *is a processor with no task scheduled to it; and*
ii. $r = 0$ or if $l > \max\{k \mid k < r$ and $((i_k, i) \in A$ or $(i_k \rightarrow j_r) \in \sigma(S_r))\}$, then $i > i_l$.

Define $\Sigma(B_2)$ as the set of partial schedules enumerated using branching rule B_2. We shall formally see in the next section that $\Sigma(B_2)$ is indeed a set and that all minimal partial schedules are generated nonetheless. Let us illustrate the application of B_2 with an example. Partial schedule S_4 in Figure 1 can be generated with B_2 following the sequence $\sigma(S_4)$ indicated in that figure. However, any sequence starting with $\langle t_1 \rightarrow p_1, t_4 \rightarrow p_3 \rangle$ is eliminated by condition i, while the sequences starting with $\langle t_1 \rightarrow p_1, t_3 \rightarrow p_1, t_2 \rightarrow p_2 \rangle$ are eliminated by condition ii.

4 Correctness of the Enumeration Algorithms

In this section, we provide a proof of correctness of the enumeration algorithms. The lemma below concerns processor permutations and branching rule 1.

Lemma 1. *For all $n \geq r \geq 0$, every minimal partial schedule S_r is in $\Sigma(B_1)$.*

Proof. This lemma is trivially demonstrated by induction on r. □

In the following lemma, branching rule 2 is used in order to assure that at most one processor permutation of each partial scheduling of $\Sigma(B_1)$ is generated.

Lemma 2. *Let $\Pi_1(S_r)$ and $\Pi_2(S_r)$ be two different processor permutations of some $S_r \in \Sigma(B_1)$. Then $\Pi_1(S_r) \in \Sigma(B_2) \Rightarrow \Pi_2(S_r) \notin \Sigma(B_2)$.*

Proof. The lemma is trivially valid if S_r contains only one non-empty processor. Otherwise, let t_{i_a}, $1 \leq a < r$, be the initial task of a processor p_j and t_{i_b}, $a < b \leq r$, be the initial task of a processor $p_{j'}$, $j < j'$, in S_r. Then $(t_{i_a}, t_{i_b}) \in A$ or $i_a < i_b$. Using this fact, the proof of the lemma is by contradiction. Suppose now that $S_r \in \Sigma(B_1)$ and a processor permutation $\Pi_1(S_r) \in \Sigma(B_2)$. Still, take a processor permutation $\Pi_2(S_r)$, $\Pi_1(S_r) \neq \Pi_2(S_r)$. Let $t_{i_1}, t_{i_2}, \cdots, t_{i_c}$, $c \leq m$, be the initial tasks of all non-empty processors $p_1, p_2, \cdots, p_c$ in S_r. Thus, these are the initial tasks of processors $p_{\pi_1(1)}, p_{\pi_1(2)}, \cdots, p_{\pi_1(c)}$, respectively, in $\Pi_1(S_r)$, and $p_{\pi_2(1)}, p_{\pi_2(2)}, \cdots, p_{\pi_2(c)}$, respectively, in $\Pi_2(S_r)$. It follows that $R(S_r) = R(\Pi_1(S_r)) = R(\Pi_2(S_r))$. Consider two indices d and e, $1 \leq d, e \leq c$, such that $\pi_1(d) < \pi_1(e)$ and $\pi_2(d) > \pi_2(e)$. Such a pair d, e exists since $\Pi_1(S_r) \neq \Pi_2(S_r)$. It steems from the fact mentioned at the begining of the proof that $((t_{i_d}, t_{i_e}) \in A$ or $i_d < i_e)$ and $((t_{i_e}, t_{i_d}) \in A$ or $i_e < i_d)$, which contradicts the assumption that both $\Pi(S_r)$ and $\Pi_2(S_r)$ satisfies branching rule 2. □

The following lemma says that branching rule 2 allows the generation of all minimal partial schedules, as an immediate corollary of the fact that at least one processor permutation of every partial schedule is indeed generated.

Lemma 3. *For each partial schedule $S_r \in \Sigma(B_1)$, there exists a sequence $\sigma(\Pi(S_r))$ such that $\Pi(S_r)$ is a processor permutation of S_r and $\Pi(S_r) \in \Sigma(B_2)$.*

Proof. We demonstrate the lemma by induction on r. Again, the lemma is trivially verified for $r = 0$. Suppose a sequence $\sigma(S_r) = \langle t_{i_1} \rightarrow p_{j_1}, \cdots, t_{i_r} \rightarrow p_{j_r} \rangle$ of task schedulings leading to S_r. If $\sigma(S_r)$ is generated with B_2, then the lemma is proved. Otherwise, let $S_{r-1} \in \Sigma(B_1)$ be a partial schedule obtained with the task schedulings $t_{i_1} \rightarrow p_{j_1}, \cdots, t_{i_{r-1}} \rightarrow p_{j_{r-1}}$ performed in some rank preserving order, that is, an order that preserves, in S_{r-1}, the rank of $t_{i_1}, \cdots, t_{i_{r-1}}$ in S_r. By the induction hypothesis, there exists a processor permutation $\Pi(S_{r-1}) \in \Sigma(B_2)$. Denote S'_r the partial schedule obtained with $\sigma(S'_r) = \langle \sigma(\Pi(S_{r-1})), t_{i_r} \rightarrow p_{\pi(j_r)} \rangle$, where $\sigma(\Pi(S_{r-1}))$ is a sequence of task schedulings generated with B_2 leading to $\Pi(S_{r-1})$. Again, if $S'_r \in \Sigma(B_2)$, then the lemma is proved. Otherwise, we will exhibit another sequence leading to S'_r that is generated.

Rename the tasks scheduled in S'_r such that $\sigma(S'_r) = \langle t_{i'_1} \rightarrow p_{j'_1}, \cdots, t_{i'_r} \rightarrow p_{j'_r} \rangle$. By definition of S'_r, $i'_r = i_r$ and $j'_r = \pi(j_r)$. Since we have assumed that $S'_r \notin \Sigma(B_2)$, it follows that $(t_{i'_{r-1}}, t_{i'_r})$ is not an arc in the DAG. However,

by the induction hypothesis, there exists a processor permutation $\Pi(S'_{r-1})$ of a partial schedule S'_{r-1} generated with some rank preserving sequence of the tasks schedulings $t_{i'_1} \rightarrow p_{j'_1}, \cdots, t_{i'_{r-2}} \rightarrow p_{j'_{r-2}}, t_{i'_r} \rightarrow p_{j'_r}$ that is generated by a sequence $\sigma(\Pi(S'_{r-1}))$. This processor permutation Π is chosen in such a way that if $p_{\pi(j_{r-1})}$ is empty, then processors $p_1, p_2, \cdots, p_{\pi(j'_{r-1})}$ are not empty. We can apply the induction hypothesis here because all immediate predecessors of $t_{i'_r}$ are in $\{t_{i'_1}, \cdots, t_{i'_{r-2}}\}$ due to $S_r \in \Sigma(B_1)$ and $(t_{i'_{r-1}}, t_{i'_r}) \notin R(S'_r)$. Certainly, these arguments and $S'_r \in \Sigma(B_2)$ imply that $i'_r < i'_{r-1}$ and $\pi(j'_r) \neq \pi(j'_{r-1})$. Thus, if the sequence $\langle \sigma(\Pi(S'_{r-1})), t_{i'_{r-1}} \rightarrow p_{\pi(j'_{r-1})} \rangle$ is generated, then it is a processor permutation of S_r. Otherwise, we can follow recursively the same arguments until finding a sequence $\sigma(\Pi(S'_q))$, for some $l < q \leq r-1$, such that a processor permutation $\langle \sigma(\Pi(S'_{q-1})), t_{i_r} \rightarrow p_{\pi(j_r)}, t_{i'_q} \rightarrow p_{\pi(j'_q)}, \cdots, t_{i'_{r-1}} \rightarrow p_{\pi(j'_{r-1})} \rangle$ of S_r is generated. □

In what follows, we analyze the role played by B_2 in avoiding duplications.

Lemma 4. *$\Sigma(B_2)$ is a set.*

Proof. By contradiction, let S_r be a partial schedule with $r > 0$ tasks scheduled such that B_2 induces two disjoint sequences $\sigma(S_r)$ and $\sigma'(S_r)$ leading from S_0 to S_r. If $r \leq 2$, a contradiction can be easily found. Assume $r > 2$ and let S_q, $r > q \geq 0$, be the most recent partial schedule in the intersection of $\sigma(S_r)$ and $\sigma'(S_r)$, and $\sigma(S_q)$ be its sequence of task schedulings. Denote by $t_i \rightarrow p_j$ the task scheduling performed from S_q in $\sigma(S_r)$. Still, denote by $t_{i'} \rightarrow p_{j'}$ the task schedulings from S_q in $\sigma'(S_r)$. Clearly, $i \neq i'$ and $j \neq j'$. It follows that $(t_i \rightarrow p_j) \in \sigma'(S_r)$ and $(t_{i'} \rightarrow p_{j'}) \in \sigma(S_r)$, since both $\sigma(S_r)$ and $\sigma'(S_r)$ lead S_q to S_r. Define S_{q_1} to be the partial schedule generated by $t_{i'} \rightarrow p_{j'}$ in $\sigma(S_r)$, and S_{q_2} to be the partial schedule generated by $t_i \rightarrow p_j$ in $\sigma'(S_r)$. We observe that either $t_{i'} \rightarrow p_{j'}$ occuring after $t_i \rightarrow p_j$ or $t_i \rightarrow p_j$ occuring after $t_{i'} \rightarrow p_{j'}$ violate condition ii of branching rule 2 in $\sigma(S_r)$ or $\sigma'(S_r)$, respectively, which is a contradiction. □

Here is the theorem that follows directly from Lemmas 2, 3 and 4.

Theorem 1. *If S_n is a minimal schedule, then $S_n \in \Sigma(B_2)$, and $\Sigma(B_2)$ is minimal with respect to this property.*

The time complexity of a branching procedure based on B_2 is determined by four components. First, examining the tasks in $FT(S_r)$ takes $O(n)$ time, while scheduling a free task to each one of the processors takes $O(m)$ time. Setting the new free tasks has time complexity $O(n)$. Finally, testing condition ii of branching rule 2 takes $O(n)$ time. Besides the $O(n^2)$ storage space required for representing the DAG, the storage space required by each new partial schedule generated with BRANCH_PARTIAL_SCHEDULE is $O(n)$ and corresponds to $FT(S_{r+1})$. This proves the following theorem.

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

5 Concluding Remarks

In this paper, we proposed a polynomial time and storage space branching procedure for the multiprocessor scheduling problem. This procedure can be implemented in a branch-and-bound algorithm, and it is suitable for this purpose either in case of unit task costs (this case was considered in the previous sections for the sole sake of simplicity) or when the task costs are arbitrary (which means that different tasks can be assigned different costs), implying in both cases an associated search tree whose height is equal to the number of tasks in the MSP instance. A best-first branch-and-bound algorithm using the branching procedure proposed in this paper will be effective provided that tight lower and upper bounds procedures are available. In addition, this new branching procedure is more adapted to parallel implementations of branch-and-bound algorithms than other branching procedures which generate each subproblem more than a once during the search because checking for duplicated subproblems in parallel slow-down the parallel algorithm significantly.

Acknowledgments We are grateful to anonymous referees for their useful suggestions. *Ricardo Corrêa* is partially supported by the brazilian agency CNPq. *Afonso Ferreira* is partially supported by a Région Rhône Alpes International Research Fellowship and NSERC.

References

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

[2] P.C. Chang and Y.S. Jiang. A State-Space Search Approach for Parallel Processor Scheduling Problems with Arbitrary Precedence Relations. *European Journal of Operational Research*, 77:208–223, 1994.

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

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

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

[6] Y.-K. Kwok and I. Ahmad. Otimal and near-optimal allocation of precedence-constrained tasks to parallel processors: defying the high complexity using effective search techniques. In *Proceedings Int. Conf. Parallel Processing*, 1998.

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

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