Multiprocessor scheduling with communication delays

Citation for published version (APA):
Veltman, B. (1993). *Multiprocessor scheduling with communication delays*. [Phd Thesis 2 (Research NOT TU/e / Graduation TU/e), Mathematics and Computer Science]. Centrum voor Wiskunde en Informatica. https://doi.org/10.6100/IR396672

DOI:
10.6100/IR396672

Document status and date:
Published: 01/01/1993

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

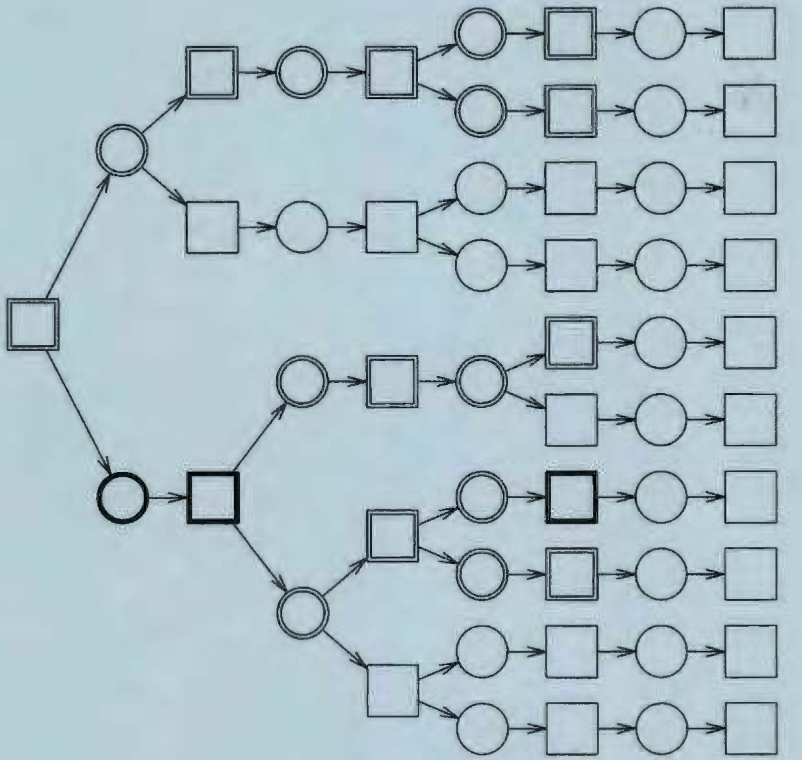
Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 26. Aug. 2022

MULTIPROCESSOR SCHEDULING
WITH
COMMUNICATION DELAYS



Bart Veltman

MULTIPROCESSOR SCHEDULING
WITH
COMMUNICATION DELAYS

MULTIPROCESSOR SCHEDULING WITH COMMUNICATION DELAYS

Proefschrift
ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus, prof. dr. J.H. van Lint, voor
een commissie aangewezen door het College van
Dekanen in het openbaar te verdedigen op
dinsdag 18 mei 1993 te 16.00 uur
door

Bart Veltman

geboren te Reinheim (BRD)

1993
CWI, Amsterdam

Dit proefschrift is goedgekeurd door
de promotor:

prof. dr. J. K. Lenstra.

Acknowledgements

The work presented here has been carried out at the CWI in Amsterdam as part of the *ScheduLink* subproject within the *ParTool* project. The ParTool project is partially supported by SPIN, a Dutch computer science stimulation program.

I am truly grateful to my supervisor, Jan Karel Lenstra, and my other co-authors, Han Hoogeveen, Ben Lageweg, Steef van de Velde en Marinus Veldhorst. I have learned a lot from their expertise. Working together has been a great pleasure.

Bart Veltman

Voor Hanna, Rob, Dirkje en Dineke

Table of contents

1. Introduction

From the late fifties onwards, many architectures for parallel computers have been proposed. Some models are useful from a theoretical point of view, but their realization is generally not feasible due to physical limitations; their main purpose is to help to design and analyze parallel algorithms. Others are more realistic and exist or are being built. Unfortunately, multiprocessors strongly differ from each other and, accordingly, there exists no general model that effectively describes the broad spectrum of feasible parallel architectures. Different classification schemes have been proposed, based on processor autonomy [Flynn, 1966], interprocessor communication [Schwartz, 1980], and mode of operation [Treleaven, Brownbridge, and Hopkins, 1982]. The diversity of models and existing architectures causes a series of problems when one wishes to take advantage of the computing power that multiprocessors offer. These problems can be summarized as follows. Important for programming a parallel computer is to preserve an algorithm's intrinsic parallelism when formalized in a programming language, to properly partition a program into tasks, and to assign the tasks to processors while respecting the information dependencies in between the tasks.

This thesis concerns the latter aspect: the new allocation and scheduling problems that have to be solved. These problems differ from the problems of classical sequencing and scheduling theory mainly in that interprocessor communication delays have to be taken into account.

In the literature, we distinguish two basically different approaches to handle communication delays. The first approach formulates the problem in graph theoretic terms; one speaks of the *mapping* problem [Bokhari, 1981]. The program graph is regarded as an undirected graph, where the vertices correspond to tasks and an (undirected) edge indicates that the adjacent tasks interact, that is, communicate with each other. The multiprocessor architecture is also regarded as an undirected graph, with nodes corresponding to processors. Processors are assigned to tasks. A mapping aims at reducing the total interprocessor communication time and balancing the workload of the processors, thus attempting to find an allocation that minimizes the overall completion time.

The second approach considers the allocation problem as a pure *scheduling* problem. It regards the program graph as an acyclic directed graph. Again, the vertices represent the tasks, but a (directed) arc indicates a one-way communication between a predecessor task and a successor task. A *schedule* is an allocation of each task to a time interval on one or more processors such that, among others, precedence constraints and communication delays are taken into account. It aims at minimizing the maximum or the average task completion time. We will take this second approach.

Eventually, it may be desirable to combine the mapping approach and the scheduling approach when allocating a parallel program to a multiprocessor. In that case, the combined approach would first schedule the tasks on a virtual architecture graph and next find a mapping of the virtual architecture graph onto the physical architecture of the multiprocessor [Kim, 1988].

Following the second approach, we address the allocation problems in the context of deterministic machine scheduling theory. Scheduling theory in general is concerned with the optimal allocation of scarce resources (processors) to activities (tasks) over time. The problems we consider are *deterministic* in the sense that all the information that defines an instance is known with certainty in advance. A complete formulation of the problem type to be considered in this thesis is given in Chapter 2.

Deterministic scheduling theory is part of the area of *combinatorial optimization*. Combinatorial optimization involves problems in which we have to choose the best from a discrete and often finite set of alternatives. The finiteness suggests the brute-force approach of *complete enumeration* to be effective: simply generate all feasible solutions, examine their costs, and select the best one. However, for realistic problems the time requirements of this method are prohibitive and we have to search for faster algorithms. The fundamental question is whether there exists an algorithm that solves a given problem to optimality in *polynomial time*. Algorithms that run in polynomial time are considered to be 'fast', and problems for which such an algorithm exists are said to be 'well-solved'. For other problems it has been shown that the existence of a polynomial-time algorithm is highly unlikely; these are the NP-*hard* problems. *Complexity theory* provides a mathematical framework in which computational problems can be classified as being solvable in polynomial time or NP-hard. The reader is referred to the textbook by Garey and Johnson [1979] for a detailed treatment of the subject. The complexity of many scheduling problems that come up in the context of programming a parallel computer is dealt with in Chapters 3-6. We will now give an overview of these chapters.

As indicated before, interprocessor *communication delays* form a major problem when one is programming a multiprocessor. Each task of a parallel program produces information which is in whole or in part required by one or more other tasks. The transmittal of information may induce several sorts of communication delays, depending on the amount of the information that is transferred. In Chapter 3, we study the simplest model that allows for communication delays: a set of unit-time tasks has to be processed subject to precedence constraints and unit-time communication delays. We consider two cases, in which the number of processors is restricted and unrestricted, respectively. For either case, we investigate the question whether there exists a schedule of length at most equal to a given threshold value. We also show that dynamic programming gives a polynomial-time algorithm in case the width of the precedence relation is fixed, i.e., part of the problem type. Finally, we show NP-hardness for the case that the precedence relation can be represented by a directed tree.

Communication delays may be reduced or even avoided by *task duplication*, that is, the creation of copies of a task. In Chapter 4, we investigate the trade-off between the optimal makespan of schedules with and without duplicated tasks. In general, task duplication can decrease the schedule length by a factor at most equal to the number of processors, even for tree-type precedence relations. However, in case of unit-time processing requirements and unit-time communication delays task duplication can help a factor of two, but no more.

Another aspect of multiprocessor scheduling is that a task may require more than one processor for its execution. Such tasks are referred to as *multiprocessor tasks*. In Chapter 5 we investigate the computational complexity of scheduling multiprocessor tasks with *prespecified processor allocations*. Moreover, we investigate the complexity when various additional task characteristics are involved, such as precedence constraints and release dates.

In Chapter 6 we explore a hybrid variant, in which some tasks are to be allocated to either of two processors and others have a prespecified allocation to a single processor. The multiprocessor architecture is a two-stage pipeline, where the first stage consists of two independent identical processors and the second stage consists of a single processor. This problem can be viewed as an extension of the classical *two-stage flow shop problem*. We establish its NP-hardness in the strong sense.

The general model described in Chapter 2 involves multiprocessor tasks, possibly with prespecified processor allocations, and allows for communication delays and task duplication. From the analyses of Chapters 3-6, we may conclude that it is unlikely that fast algorithms exist that solve the scheduling problem in its most general form to optimality. One is confined to take an approximative approach. *Tosca*, our tunable off-line scheduling algorithm, embodies such an approach. It has been developed as a tool to support the scheduling of parallel programs on distributed memory architectures. Tosca's purpose is to assist in the design and analysis of schedules of a given computation graph on a given processor model, allowing for communication delays. Tosca can be used to obtain performance predictions with respect to a program under development; given a decomposition of such a program, a schedule measures its quality.

Tosca constructs schedules for instances that may consist of multiprocessor tasks, possibly with prespecified processor allocations. It allows for communication delays, but does not apply task duplication. Tasks may be grouped into families. Tasks that belong to the same family must be executed by the same collection of processors. Tosca tries to find a reasonable solution in a reasonable amount of time by *bounded enumeration*. In principle, a schedule can be constructed by iteratively selecting the next task to schedule, allocating a collection of processors to it, and starting the task as early as possible on that collection. The various possible choices can be represented by an enumeration tree. The process of bounded enumeration considers only part of this enumeration tree. It consists of a number of stages. At each stage a task and a processor allocation for that task are selected. In order to select this task and allocation, Tosca generates a subtree of the enumeration tree. The subtree is determined by three parameters (which control the width and the depth of the subtree), two priority rules (for choosing good tasks and allocations), and a lower bound rule (in order to eliminate unpromising branches). The leaves of the subtree are evaluated according to an evaluation rule. A task-allocation pair that leads to a leaf of minimum value is selected. Tosca is tunable, since it enables the user to control the speed of the solution method and the quality of the schedules produced. First, by adjusting the three parameters the user influences the size of the subtree that is computed. Second, the user has to define two priority rules; one for selecting tasks and another for selecting processors. These rules may be part of a given set of rules or are of the user's making. Third, the user has to specify a lower bound rule and an evaluation rule. A detailed description of Tosca's *methodology* is given in Chapter 7.

Tosca is equipped with a simple user interface. All the information is presented in alphanumerical manner. The man-machine interaction is menu driven, so that at any moment all feasible commands are visible. Tosca's *implementation* is described in Sections 8.1 and 8.2. Together with Section 7.3, these sections can be seen as a manual for the use of Tosca. Tosca has been tested on four classes of problem instances: layered precedence relations, series parallel precedence relations, arbitrary precedence relations, and two precedence relations from practice. In addition to the precedence relations, we generated data sets, processing times, and task sizes. The corresponding four *problem generators* are described in Section 8.3. For the instances that were generated, we applied list scheduling with a number of different priority rules to construct initial schedules. Next we tried to build better schedules by use of bounded enumeration with a more restricted number of priority rules. In Section 8.4 we report on these *experiments*.

As an illustration of the models and methodology described in this thesis, especially those concerning Tosca, we present a small *example* in Chapter 9. Amongst others, it illustrates the aspects of communication delays, multiprocessor tasks, list scheduling and bounded enumeration.

Chapter 2 is a substantial revision and extension of:
B. Veltman, B.J. Lageweg, J.K. Lenstra (1990). Multiprocessor scheduling with communication delays. *Parallel Comput. 16*, 173-182.

Chapter 3 is based on:
J.A. Hoogeveen, J.K. Lenstra, B. Veltman (1992). *Three, four, five, six, or the complexity of scheduling with communication delays*, Report BS-R9229, CWI, Amsterdam;
J.K. Lenstra, M. Veldhorst, B. Veltman (1993). *The complexity of scheduling trees with communication delays*, in preparation.

Chapter 5 is based on:
J.A. Hoogeveen, S.L. van de Velde, B. Veltman (1993). Complexity of scheduling multiprocessor tasks with prespecified processor allocations. *Discrete Appl. Math.*, to appear.

Chapter 6 is based on:
J.A. Hoogeveen, J.K. Lenstra, B. Veltman (1993). *Minimizing makespan in a multiprocessor flow shop is strongly* NP-*hard*, in preparation.

Chapters 7 through 8 are based on:
B. Veltman, B.J. Lageweg, J.K. Lenstra (1993). *Tosca: a tunable off-line scheduling algorithm*, in preparation.

2. A general model for parallel processor scheduling

As indicated in Chapter 1, the subject of this thesis is the study of the allocation of program modules or tasks to parallel processors in the context of deterministic machine scheduling theory. A multiprocessor architecture can be represented by an undirected graph. Tasks can be processed on various subgraphs of the multiprocessor graph. Data dependencies define a precedence relation on the task set. The transmittal of data may induce several sorts of communication delays. These delays may be reduced or even avoided by task duplication. We search for an allocation of tasks to processors that minimizes the maximum or total completion time.

In this chapter, we formulate our scheduling model, we propose a classification that extends the scheme of Graham, Lawler, Lenstra and Rinnooy Kan [1979], and we review the available literature.

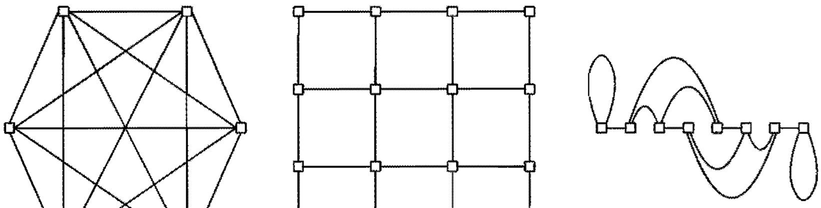
2.1. The processor model

The multiprocessor chosen consists of a collection of *m processors*, each provided with a *local memory* and mutually connected by an *intercommunication network*. The multiprocessor architecture can be represented by an undirected graph. Several examples are given in Figure 2.1; cf. Kindervater and Lenstra [1988]. The nodes of such a graph correspond to the processors of the architecture it represents. Transmitting data from one processor to another is considered as an independent event, which does not influence the availability of the processors on the transmittal path. In case of a shared memory, the assumption of having local memory only overestimates the communication delays.

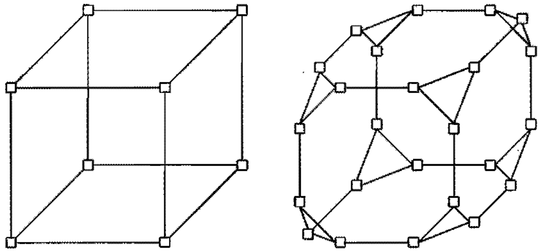
2.2. The program model

A parallel program is represented by means of an acyclic directed graph. The nodes of this program graph correspond to the modules in which the program is decomposed; they are called *tasks*. Each task produces *information*, which is in whole or in part required by one or more other tasks. These data dependencies impose a *precedence relation* on the task set; that is, whenever a task requires information, it has to succeed the tasks that deliver this information. The arcs of the graph represent these precedence constraints. The transmittal of information may induce several sorts of *communication delays*, which will be discussed in the next section. *Task duplication*, that is, the creation of copies of a task, might reduce such communication delays. Task duplication is discussed in Section 2.4.

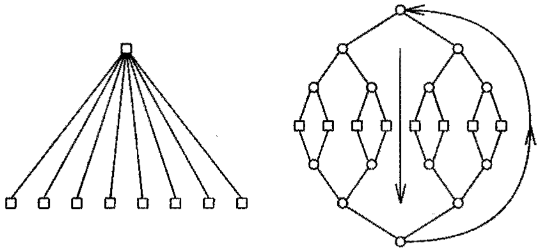
The task set is partitioned into a number of *families*. Each task belongs to exactly one family. A task can be processed on various *subgraphs* of the



(*i*) Complete network. (*ii*) Mesh connected network. (*iii*) Perfect shuffle network.



(*iv*) Cube connected network. (*v*) Cube connected cycles network.



(*vi*) Master-slave network. (*vii*) Binary trees network.

Figure 2.1. Seven interconnection networks.

multiprocessor graph. Tasks that belong to the same family have to be executed by the same subgraph of the multiprocessor graph. We assume that for each family a collection of subgraphs on which its tasks can be processed is specified, and that for each task in that family and each of its subgraphs a corresponding *processing time* is given. If the processors of the architecture are identical, then for each task the processing times related to isomorphic subgraphs are equal. For instance, one may think of a collection of subhypercubes of a hypercube system of processors, or a collection of submeshes of a mesh connected system. Another possibility occurs when each task can be processed on any subgraph of a given family-dependent size.

If *preemption* is allowed, then the processing of any operation may be interrupted and resumed at a later time. Although task splitting may induce communication delays, it may also decrease the cost of a schedule with respect to one or more criteria. We will not explore the aspect of communication delays that are induced by preemption in detail, but concentrate on communication delays between precedence-related tasks.

2.3. Communication

The information a task needs (or produces) has to be (or becomes) available on all the processors handling this task. The size of this data determines the communication times.

If two tasks J_k and J_l both succeed a task J_j, then they might partly use the same information from task J_j. Under the condition that the memory capacity of a processor is adequate, only one transmission of this common information is needed if J_k and J_l are scheduled on the same subgraph of the multiprocessor graph. It is therefore important to determine the *data set* a task needs from each of its predecessors. The transfer of data between J_j and J_k can be represented by associating a data set with the arc (J_j, J_k) of the transitive closure of the program graph. This would generally lead to the specification of $\Theta(n^2)$ sets, if there are n tasks. Another possibility is to associate two sets I_j and O_j with each task J_j, representing the data that this task requires and delivers, respectively. This requires $\Theta(n)$ sets. The intersection $O_j \cap I_k$ gives the data dependency of tasks J_j and J_k.

Each information set has a *weight*, which is specified by a function $c: 2^D \rightarrow N$, where D is the set containing all information. This function gives the time needed to transmit data from one processor to another, regarded as independent of the processors involved. Let $U \in 2^D$ be a data set and let $\{U_1, U_2, \ldots, U_u\}$ be a partition of U. We assume that U can be transmitted in such a way that $\cup_{i=1}^{t} U_i$ is available when a time period of length at most $c(\cup_{i=1}^{t} U_i)$ has elapsed, for each t with $1 \le t \le u$. We also assume that $c(\emptyset) = 0$ and that $c(U) \le c(W)$ for all $U \subset W \in 2^D$. These conditions state that a data set U can be transmitted in such a way that a subset of U becomes available no later than when this subset would be transmitted on its own.

J_k	I_k	$P(k,2)$	$U(2,1,k)$	$c(U(2,1,k))$
J_2	$\{a,b\}$	$\{2\}$	$\{a,b\}$	2
J_3	$\{a,c\}$	$\{2,3\}$	$\{a,b,c\}$	5

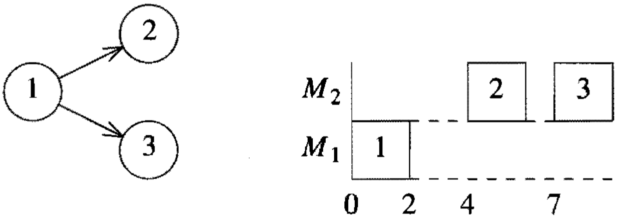


Figure 2.2. Communication delays.

Interprocessor *communication* occurs when a task J_k needs information from a predecessor J_j and makes use of at least one processor that is not used by J_j. Let M_i be such a processor. Let $F(j)$ denote the set of successors of J_j and, given a schedule, let $P(k,i)$ denote the set of tasks scheduled on M_i before and including J_k. Prior to the execution of J_k, the data set $U(i,j,k) = \cup_{l \in F(j) \cap P(k,i)} (O_j \cap I_l)$ has to be transmitted to M_i, since not only J_k but also each successor of J_j that precedes J_k on M_i requires its own data set. The time gap in between the completion of J_j (at time C_j) and the start of J_k (at time S_k) has to allow for the transmission of $U(i,j,k)$, as illustrated in Figure 2.2. The communication time is given by $c(U(i,j,k))$. For feasibility it is required that $S_k - C_j \geq c(U(i,j,k))$. At the risk of laboring the obvious, let it be mentioned that the communication time is schedule-dependent.

Sometimes one wishes to disregard the data sets and simply to associate a communication delay with each pair of tasks. That is, a (predecessor, successor) pair of tasks (J_j, J_k) assigned to different processors needs a communication time of a given duration c_{jk}. The communication time is of length c_{j*} if it depends on the broadcasting task only, it is of length c_{*k} if it depends on the receiving task only. Finally, it may be of constant length c, independent of the tasks.

2.4. **Task duplication**

If one manages to execute all predecessors of a task on all of the processors handling that task, then one may reduce or even avoid communication delays. This can be done by *task duplication*, that is, the creation of copies of a task.

Consider the example given in Figure 2.2. An optimal schedule for these three tasks without duplication takes six time units, whereas an optimal

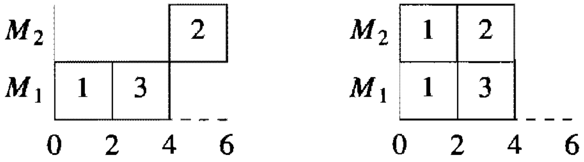


Figure 2.3. Task duplication.

schedule with task duplication is of length four, as illustrated in Figure 2.3. In the latter, task J_1 is executed twice: once by processor M_1 and once by processor M_2. This enables tasks J_2 and J_3 to be executed without any form of communication delay; task J_2 receives its information from the copy of task J_1 that is executed by M_2 and J_3 receives its information from the copy of J_1 that is processed by M_1.

Let J_j and J_k be such that $J_j \rightarrow J_k$. In a feasible schedule each copy of J_k has to receive the information it needs for processing in time, that is, there has to be a copy of J_j such that the time gap between the completion of this copy of J_j and the start of the copy of J_k allows for the transmission of the required information.

2.5. Classification

In general, m processors M_i $(i=1,\ldots,m)$ have to process n tasks J_j $(j=1,\ldots,n)$. A *schedule* is an allocation of (each copy of) a task to a time interval on one or more processors. A schedule is *feasible* if no two of these time intervals on the same processor overlap and if, in addition, it meets a number of specific requirements concerning the processor environment and the task characteristics (e.g., precedence constraints and communication delays). A schedule is *optimal* if it minimizes a given optimality criterion. The processor environment, the task characteristics and the optimality criterion that together define a problem type, are specified in terms of a three-field classification $\alpha \mid \beta \mid \gamma$, which is specified below. Let $\circ$ denote the empty symbol.

2.5.1. *Processor environment*

The first field $\alpha = \alpha_1 \alpha_2$ specifies the processor environment. The characterization $\alpha_1 = P$ indicates that the processors are *identical parallel processors*. The characterization $\bar{P}$ indicates that, in addition, the number of processors is not restricted; e.g., $m \geq n$ is sufficient in case of single-processor tasks.

If α_2 is a positive integer, then m is a constant, equal to α_2; it is specified as part of the problem type. If $\alpha_2 = \circ$, then m is a variable, the value of which is specified as part of the problem instance.

2.5.2. *Task characteristics*

The second field $\beta \subset \{\beta_1, \ldots, \beta_8\}$ indicates a number of task characteristics, which are defined as follows.

1. $\beta_1 \in \{prec, tree, chain, \circ\}$.

   $\beta_1 = prec$: A *precedence relation* $\rightarrow$ is imposed on the task set due to data dependencies. It is denoted by an acyclic directed graph G with vertex set $\{1, \ldots, n\}$. If G contains a directed path from j to k, then we write $J_j \rightarrow J_k$ and require that J_j has been completed before J_k can start.

   $\beta_1 = tree$: G is a *rooted tree* with either outdegree at most one for each vertex or indegree at most one for each vertex.

   $\beta_1 = chain$: G is a collection of vertex-disjoint chains.

   $\beta_1 = \circ$: No data dependencies occur, so that the precedence relation is empty.

2. $\beta_2 \in \{com, c_{jk}, c_{j*}, c_{*k}, c, c=1, \circ\}$

   This characteristic concerns the communication delays that occur due to data dependencies. To indicate this, one has to write β_2 directly after β_1.

   $\beta_2 = com$: Communication delays are derived from given data sets and a given weight function, as described in Section 2.3. In all the other cases, the communication delays are explicitly specified.

   $\beta_2 = c_{jk}$: Whenever $J_j \rightarrow J_k$ and J_j and J_k are assigned to different processors, a communication delay of a given duration c_{jk} occurs.

   $\beta_2 = c_{j*}$: The communication delays depend on the broadcasting task only.

   $\beta_2 = c_{*k}$: The communication delays depend on the receiving task only.

   $\beta_2 = c$: The communication delays are equal.

   $\beta_2 = c=1$: Each communication delay takes unit time.

   $\beta_2 = \circ$: No communication delays occur (which does not imply that no data dependencies occur).

3. $\beta_3 \in \{dup, \circ\}$.

   $\beta_3 = dup$: Task duplication is allowed.

   $\beta_3 = \circ$: Task duplication is not allowed.

4. $\beta_4 \in \{fam, \circ\}$.

   $\beta_4 = fam$: The number of distinct families is strictly less than the number of tasks.

   $\beta_4 = \circ$: Each family consists of a single task.

5. $\beta_5 \in \{any, set, size, cube, mesh, fix, \circ\}$.

   $\beta_5 = any$: The tasks of each family can be processed on any subgraph of the multiprocessor graph.

$\beta_5 = set$: Each family has its own collection of subgraphs of the multiprocessor graph on which its tasks can be processed.

$\beta_5 = size$: The tasks of each family can be processed on any subgraph of a given family-dependent size.

$\beta_5 = cube$: The tasks of each family can be processed on a subhypercube of given family-dependent dimension.

$\beta_5 = mesh$: The tasks of each family can be processed on a submesh of given family-dependent size.

$\beta_5 = fix$: The tasks of each family can be processed on exactly one subgraph.

$\beta_5 = \circ$: Each task can be processed on any single processor.

6. $\beta_6 \in \{\circ, p_j{=}1\}$.

   $\beta_6 = \circ$: For each task and each subgraph on which it can be processed, a processing time is specified.

   $\beta_6 = p_j{=}1$: Each task has a unit processing requirement.

7. $\beta_7 \in \{pmtn, \circ\}$.

   $\beta_7 = pmtn$: Preemption of tasks is allowed.

   $\beta_7 = \circ$: Preemption is not allowed.

8. $\beta_8 \in \{c, c{=}1, \circ\}$.

   This characteristic concerns the communication delays that occur due to preemption. To indicate this, one has to write β_8 directly after β_7.

   $\beta_8 = c$: When a task is preempted and resumed on a different processor, a communication delay of constant length occurs.

   $\beta_8 = c = 1$: Each communication delay caused by preemption takes unit time.

   $\beta_8 = \circ$: Preemption causes no communication delays.

2.5.3. *Optimality criterion*

The third field γ refers to the optimality criterion. In any schedule, each task J_j has a *completion time* C_j. A traditional optimality criterion involves the minimization of the *maximum completion time* or *makespan* $C_{\max} = \max_{1 \le j \le n} C_j$. Another popular criterion is the *total completion time* $\Sigma C_j = \Sigma_{j=1}^{n} C_j$.

The optimal value of γ will be denoted by γ^*, and the value produced by an (approximation) algorithm A by $\gamma(A)$. If $\gamma(A) \le \rho\gamma^*$ for all instances of a problem, then we say that A is a ρ-approximation algorithm for the problem.

2.6. **Literature review**

Practical experience makes it clear that some computational problems are easier to solve than others. Complexity theory provides a mathematical framework in which computational problems can be classified as being *solvable in* *polynomial time* or NP-*hard*. The reader is referred to the book by Garey and Johnson [1979] for a detailed treatment of the subject. In reviewing the literature, we will assume that the reader is familiar with the basic concepts of complexity theory. As a general reference on sequencing and scheduling, we mention the survey of deterministic machine scheduling theory by Lawler, Lenstra, Rinnooy Kan and Shmoys [1989], which updates the previous survey by Graham, Lawler, Lenstra and Rinnooy Kan [1979]. An earlier review of the literature on scheduling multiprocessor tasks with communication delays was given by Veltman, Lageweg, and Lenstra [1990].

2.6.1. *Single-processor tasks and communication delays*

The first NP-hardness proof for $P\,|prec,c{=}1,p_j{=}1\,|\,C_{\max}$ is due to Rayward-Smith [1987A]. Hoogeveen, Lenstra and Veltman [1992] show by a reduction from Clique that even the problem of deciding if there exists a feasible schedule of length at most 4 is NP-complete; see also Section 3.1. This result implies that, for $P\,|prec,c{=}1,p_j{=}1\,|\,C_{\max}$, there is no polynomial ρ-approximation algorithm for any ρ<5/4, unless P=NP. Their reduction also implies that $P\,|prec,c{=}1,p_j{=}1\,|\,\Sigma C_j$ is NP-hard. Picouleau [1991A] shows that the problem of deciding whether an instance has a schedule of length at most 3 is solvable in polynomial time; see also Section 3.1.

Hoogeveen, Lenstra and Veltman [1992] also study the variant $\bar{P}\,|prec,c{=}1,p_j{=}1\,|\,C_{\max}$ for which the number of processors is not restrictively small; see also Section 3.2. By use of an integer programming formulation, they show that the problem of deciding if there exists a feasible schedule of length at most 5 is solvable in polynomial time. A reduction from 3-Satisfiability shows that the problem of deciding if there exists a feasible schedule of length at most 6 is NP-complete. As a consequence, there exists no polynomial-time algorithm with performance bound smaller than 7/6 for $\bar{P}\,|prec,c{=}1,p_j{=}1\,|\,C_{\max}$, unless P=NP.

Rayward-Smith [1987A] analyzes the quality of *greedy* schedules (*G*) for problem instances of the type $P\,|prec,c{=}1,p_j{=}1\,|\,C_{\max}$. A schedule is said to be greedy if no processor remains idle if there is a task available; list scheduling, for example, produces greedy schedules. It is proved that $C_{\max}(G)/C^*_{\max} \leq 3-2/m$. To this end, various concepts are introduced. As indicated in Section 2.2, a directed graph or *digraph* represents the precedence relation. The nodes of this graph correspond to the tasks. The *depth* of a node is defined as the number of nodes on a longest path from any source to that node. A *layer* of a digraph comprises all nodes of equal depth. A digraph is *layered* if every node that is not a source has all of its parents in the same layer.

A layered digraph is *(n,m)-layered* if it has n layers, all terminal nodes are in the nth layer, and m layers are such that all of their nodes have more than one parent. A precedence relation is *(n,m)-layered* if the corresponding directed graph is (n,m)-layered. It takes at least time $n+m$ to schedule tasks with (n,m)-layered precedence constraints. Given a greedy schedule, let t be a point in time when one or more processors are idle. The tasks processed after t have at least one predecessor processed at $t-1$ or t. Moreover, if all processors are idle at t, then every task processed after t must have at least two predecessors processed at $t-1$. Therefore, from a greedy schedule, a layered digraph can be extracted. Some computations then yield the above result. Note that for problem instances of the type $\bar{P}\,|\,prec,c{=}1,p_j{=}1\,|\,C_{\max}$ it is trivial to see that $C_{\max}(G)/C^*_{\max} \leq 2-1/d$ holds, where d is defined as the number of nodes on a longest path from any source to any sink.

We have seen that $P\,|\,prec,c{=}1,p_j{=}1\,|\,C_{\max}$ is NP-hard. It is an open question whether this remains true for any constant value of $m \geq 2$. The problem is well solved, however, if the *width* of the precedence graph is fixed; see Section 3.3. Two elements $j,k \in V$ of an acyclic directed graph $G=(V,A)$ are said to be *incomparable* if neither $(j,k) \in A$ nor $(k,j) \in A$. The width of G is the largest number of pairwise incomparable elements of G.

Hu [1961] shows that critical path scheduling constructs optimal schedules in polynomial time for $P\,|\,tree,p_j{=}1\,|\,C_{\max}$. Surprisingly, $P\,|\,tree,c{=}1,p_j{=}1\,|\,C_{\max}$ is NP-hard, as Lenstra, Veldhorst, and Veltman [1993] show by a reduction from Satisfiability; see also Section 3.4. By use of dynamic programming, Varvarigou, Roychowdhury, and Kailath [1992] show that $Pm\,|\,tree,c{=}1,p_j{=}1\,|\,C_{\max}$ is solvable in polynomial time. The case of an unrestrictively large number of processors, $\bar{P}\,|\,tree,c{=}1,p_j{=}1\,|\,C_{\max}$, is solvable in $O(n)$ time [Chrétienne, 1989].

Picouleau [1992] gives a polynomial-time algorithm to solve $P\,|\,prec,c{=}1,p_j{=}1\,|\,C_{\max}$ if the precedence relation is of the *interval*-type. Each task is associated with an interval of a linearly ordered universe. Task J_j precedes task J_k, i.e. $J_j \rightarrow J_k$, if and only if the interval associated with J_j is entirely to the left of the interval associated with J_k.

The variant $\bar{P}$ in which the number of processors is not restrictively small has been well studied. Chrétienne [1992] shows NP-hardness for $\bar{P}\,|\,prec,c\,|\,C_{\max}$ with *send-receive*-type precedence relations; see Figure 2.4. Jakoby and Reischuk [1992] show by a reduction from Exact-3-Cover that $\bar{P}\,|\,tree,c,p_j{=}1\,|\,C_{\max}$ is NP-hard, even for intrees where each task has indegree at most 2. In addition, they study two classes for which the precedence relation can be represented by a *binary* tree. In a binary tree, each task is either

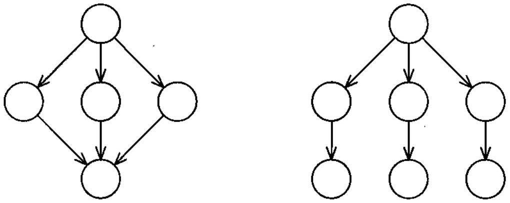


Figure 2.4. A send-receive and a harpoon-type precedence relation.

a leaf or has indegree (or outdegree) equal to 2. It is shown that $\bar{P}\,|\,tree,c_{jk},p_j{=}1\,|\,C_{\max}$, where the tree is of the *binary*-type, is NP-hard by a reduction from Exact-3-cover. Picouleau [1992] shows that $\bar{P}\,|\,tree,c_{jk}\,|\,C_{\max}$ is solvable in $O(n\log n)$ time for trees of depth 1. Together, Chrétienne and Picouleau [1991] show NP-hardness for $\bar{P}\,|\,tree,c_{jk}\,|\,C_{\max}$ with *harpoon*-type precedence relations, as illustrated in Figure 2.4.

An important characteristic of parallel algorithms is the relative cost of communication and computing. If the interprocessor communication is time consuming, then algorithms need to have a high computation/communication ratio to be efficient; we speak of *coarse-grained* parallelism. In case of *fine-grained* parallelism, the interprocessor communication time is usually in the order of an arithmetic operation.

Independently, Gerasoulis and Yang [1992], and Picouleau [1991B] study $\bar{P}\,|prec,c_{jk}\,|\,C_{\max}$ for coarse-grained instances. The *granularity* g of an instance is defined by $g=\min_j p_j/\max_{(j,k)} c_{jk}$. An instance models a coarse-grained algorithm if $g\geq 1$. Picouleau proves that the problem of deciding whether an instance of the subclass $\bar{P}\,|prec,c,g\geq 1\,|\,C_{\max}$ with $c\leq 1$ has a schedule of length at most $5{+}3c$ is NP-hard. This result can be improved upon using the techniques of Section 3.2: even the problem of deciding whether an instance has a schedule of length at most $6c$ is NP-hard. In both papers a $1{+}1/g$ -approximation algorithm is given for instances of the general problem type. Chrétienne [1989] and Anger, Hwang, and Chow [1990] note that $\bar{P}\,|\,tree,c_{jk},g\geq 1\,|\,C_{\max}$ is solvable in $O(n)$ time.

Chrétienne and Picouleau [1991] use a less restrictive definition of granularity. The *grain* $g(k)$ of a task J_k is defined by $g(k)=\min_{j\in P(k)}p_j/\max_{j\in P(k)}c_{jk}$, where $P(k)$ is the set of predecessors of J_k. An instance models a coarse-grained algorithm if $g(j)\geq 1$ for all $j=1,\ldots,n$. If the precedence relation is of the *bipartite*-type or the *series-parallel*-type, then $\bar{P}\,|prec,c_{jk},g(j){\geq}1\,|\,C_{\max}$ is solvable in polynomial time.

Duplication of tasks can be used to reduce or even avoid communication delays. The NP-hardness proof of $P\,|prec,c{=}1,p_j{=}1\,|\,C_{\max}$ [Hoogeveen, Lenstra, and Veltman, 1992] implies that the problem of deciding whether an instance of $P\,|prec,c{=}1,dup,p_j{=}1\,|\,C_{\max}$ has a schedule of length at most 4 is NP-complete, too. As a consequence, neither of these problems has a polynomial approximation scheme, unless P=NP.

Papadimitriou and Yannakakis [1990] prove that the unrestricted variant $\bar{P}\,|prec,c,dup,p_j{=}1\,|\,C_{\max}$ is NP-hard. In addition they derive a 2-approximation algorithm for the more general problem $\bar{P}\,|prec,c_{jk},dup\,|\,C_{\max}$. The algorithm determines a set of tasks T_j and computes a lower bound b_j on the starting time, for each task J_j. It is shown that if the task set T_j is assigned to the same processor as J_j, and its tasks and J_j are started as early as possible, then J_j starts no later than $2b_j$. The computation of the lower bounds and the task sets is as follows. Zero lower bounds and empty task sets are assigned to tasks without predecessors. For any task J_k other than a source task, consider its predecessors. For each predecessor J_j of J_k, define f_j by $f_j{=}b_j{+}p_j{+}c_{jk}$. Sort the predecessor set in decreasing order of f, that is $f_{j_1}\geq\cdots\geq f_{j_q}$. Given an integer y satisfying $f_{j_i}\geq y\geq f_{j_{i+1}}$, define task set $T_k(y)$ by $T_k(y){+}\{J_{j_1},\ldots,J_{j_i}\}$ and consider the following single-processor scheduling problem with release dates on i tasks $L_1,\ldots,L_i$. The release date of a task is the point in time at which it becomes available for processing. Task L_l ($l{=}1,\ldots,i$) corresponds to task J_{j_l}, that is, it has processing time $p_l{=}p_{j_l}$ and release date $r_l{=}b_{j_l}$. Let $C_{\max}(y)$ denote the minimum makespan of this single-processor scheduling problem. Define b_k as the least integer y such that $y\geq C_{\max}(y)$, and define task set T_k by $T_k{=}T_k(b_k)$. Now, each task J_j is assigned to a distinct processor and is preceded by (copies of) the tasks that belong to T_j. The tasks are scheduled in the order in which they become available, given the precedence constraints and the communication delays. This 2-approximation algorithm takes $O(n^2(e{+}n\log n))$ time, where e denotes the number of precedence constraints.

Colin and Chrétienne [1990] observe that this method generates optimal schedules in $O(n^2)$ time for coarse-grained problem instances. Their essential argument is the following: if the task grains $g(k)$ are such that $g(k)\geq 1$ for all $k{=}1,\ldots,n$, then each task set T_k consists of at most one predecessor task J_j. The precedence constraints $J_j{\rightarrow}J_k$, where $\{J_j\}=T_k$, form a spanning forest of outtrees. It is observed that the assignment of each path, from a root to a leaf, to a single processor determines an optimal schedule.

For $\bar{P}\,|prec,c,dup\,|\,C_{\max}$, dynamic programming gives an $O(n^{c+1})$ time algorithm [Jung, Kirousis, and Spirakis, 1989].

Rayward-Smith [1987B] allows preemption at integer points in time and studies $\bar{P}\,|\,pmtn, c\,|\,C_{\max}$. He observes that the communication delays increase $C^*_{\max}$ by at most $c-1$. Thus, $\bar{P}\,|\,pmtn, c=1\,|\,C_{\max}$ is solvable in polynomial time by McNaughton's *wrap-around rule* [McNaughton, 1959]. Surprisingly, for any fixed $c \geq 2$, the problem is NP-hard, which is proved by a reduction from 3-Partition. For the special case that all processing times are at most $C^*_{\max} - c$, the wrap-around algorithm will also yield a valid c-delay schedule.

Finally, we will give an overview of some related models.

Picouleau [1992] studies a variant of $\bar{P}\,|\,tree, c_{jk}\,|\,C_{\max}$ where the precedence relation can be represented by a tree of depth 1 and a distance function is specified. Given a pair of processors M_h, M_i, their *distance* d_{hi} is defined by $d_{hi} = |\,h-i\,|$. A communication delay of duration $c_{jk} d_{hi}$ occurs if M_h and M_i execute J_j and J_k, respectively. The problem is shown to be NP-hard by a reduction from Partition.

El-Rewini and Lewis [1990] consider a variant of $P\,|\,prec, c_{jk}, dup\,|\,C_{\max}$. Again, a distance is given for each pair of processors and *contention* (leading to message-routing problems) is taken into account. Contention is the event that two or more data transshipments simultaneously have to pass a single communication channel. The *level* of a task is defined as the longest path in the precedence graph from this task to a sink, taking into account processing times and communication delays. They propose an algorithm that lexicographically orders the tasks according to their level and number of successors. It recursively chooses among the available tasks one with highest order. In essence, it is a priority based scheduling algorithm. A tool for scheduling parallel programs is introduced, called *Task Grapher*. It implements a number of priority based scheduling algorithms. The deliverables of Task Grapher are Gantt charts, performance charts, simulation animations, and critical path analysis.

Kim [1988] studies $P\,|\,prec, c_{jk}\,|\,C_{\max}$. His approach starts by reducing the program graph, by merging nodes with high internode communication cost through the iterative use of a *critical path* algorithm. This (undirected) graph is then mapped to a multiprocessor graph. Numerical results are given.

Sarkar [1989] defines a graphical representation for parallel programs and a cost model for multiprocessors. Together with frequency information obtained from execution profiles, these models give rise to a scheme for compile-time cost assignment of execution times and communication sizes in a program. Most attention is paid to the partitioning of a parallel program, which is outside the scope of this thesis.

Kruatrachue and Lewis [1988] treat the problem $P\,|prec,c_{jk},dup\,|\,C_{\max}$. Most attention is paid to the *grain size problem*: how to partition a parallel program into concurrent modules in order to obtain the shortest possible schedule length. A scheduling algorithm allowing for duplication is used to schedule fine-grained instances. Coarse-grained instance are formed by packing nodes located contiguously on the same processor. Then an operating system scheduler can be used to construct a schedule at runtime.

Papadimitriou and Ullman [1987] attempt to minimize the communication overhead for problem instances of the type $P\,|prec,c{=}1,dup,p_j{=}1\,|\,C_{\max}$, where the precedence relation is of *grid*-type. They show that any schedule that computes all tasks of an $n{\times}n$ grid or diamond graph has a total communication overhead of C and takes time T, where $(C+n)T=\Omega(n^3)$.

2.6.2. *Multiprocessor tasks*

The problems and algorithms mentioned above deal with tasks that are processed on a single processor and focus on communication delays. The papers discussed below disregard the notion of communication and concentrate on tasks that may require more than one processor at the same time.

Examples of such tasks are the pairwise tests that processors may perform in order to prevent a partial or total failure of the multiprocessor system. Each of the tests can be viewed as a biprocessor task with a prespecified processor allocation. In order to execute the diagnosis as fast as possible, one has to solve a problem of the type $P\,|fix,p_j{=}1\,|\,C_{\max}$ where each task is a biprocessor task. Krawczyk and Kubale [1985] show that this problem is NP-hard by a reduction from Chromatic Index. Hoogeveen, Van de Velde, and Veltman [1992] do not restrict themselves to biprocessor tasks. They show that even the problem of deciding whether an instance has a schedule of length at most 3 is NP-complete. As a consequence, there exists no polynomial-time algorithm with performance bound smaller than 4/3 for $P\,|fix,p_j{=}1\,|\,C_{\max}$, unless P=NP. If the number of processors m is fixed, i.e., $Pm\,|fix,p_j{=}1\,|\,C_{\max}$, then the problem is solvable in polynomial time through an integer programming formulation with a fixed number of variables [Hoogeveen, Van de Velde, and Veltman, 1992].

It is easy to see that $P2\,|fix\,|\,C_{\max}$ is solvable in polynomial time. However, Blazewicz, Dell'Olmo, Drozdowski, and Speranza [1992] show that $P3\,|fix\,|\,C_{\max}$ is strongly NP-hard. Hoogeveen, Van de Velde, and Veltman [1992] consider a *block-constraint*, which decrees that all *bi*processor tasks that require the same processors are scheduled consecutively. They show that $P3\,|fix\,|\,C_{\max}$ subject to this block-constraint is solvable in pseudopolynomial time. Under a stronger version of the block-constraint, where *all* tasks of the same type are scheduled consecutively, there exists a 4/3-approximation algorithm [Blazewicz, Dell'Olmo, Drozdowski, and Speranza, 1992].

Hoogeveen, Van de Velde, and Veltman [1992] also show that $P2 \mid chain, fix, p_j{=}1 \mid C_{\max}$ is NP-hard even for single-processor tasks only. This leaves little hope of finding polynomial-time optimization algorithms if precedence constraints are imposed, although $P2 \mid prec, fix, p_j{=}1 \mid C_{\max}$ is solvable in $O(n \log n)$ time in case of single-processor tasks and a precedence relation of the *interval*-type [Kellerer and Woeginger, 1992].

The introduction of release dates has a similar inconvenient effect on the problem's complexity. The problem $P2 \mid fix, r_j \mid C_{\max}$ is NP-hard in the strong sense. The complexity of the case of unit processing times, that is, $Pm \mid fix, r_j, p_j{=}1 \mid C_{\max}$, is still open. However, if the number of distinct release dates is fixed, then the problem is solvable in polynomial time through an integer programming formulation with a fixed number of variables. All these results are due to Hoogeveen, Van de Velde, and Veltman [1992]; see also Section 5.1.

Two *branch and bound* approaches for $P \mid fix \mid C_{\max}$ have been proposed. Bozoki and Richard [1970] concentrate on *incompatibility*; two tasks are *incompatible* if they have at least one processor in common. Lower bounds for the optimal makespan are the maximum amount of processing time that is required by a single processor, and the maximum amount of processing time required by tasks that are mutually incompatible. Upper bounds are obtained by list scheduling according to priority rules such as *shortest processing time* (*SPT*) and *maximum degree of competition* (*MDC*). The degree of competition of a task represents the number of tasks incompatible with it. *MDC* gives tasks with large degree of competition priority over tasks with low degree, breaking ties by use of *SPT*. In branching, an *acceptable* subset of tasks that yield smallest lower bounds is selected at each decision moment t. A set of tasks is acceptable if the tasks are mutually compatible, each task of the set is compatible with each task that is in process at time t, and each task is incompatible with at least one task terminating at t. Bianco, Dell'Olmo, and Speranza [1991] follow a graph-theoretical approach. In addition to proposing a branch and bound algorithm, they determine a class of polynomially solvable instances that corresponds to the class of *comparability graphs*. A comparability graph is an undirected graph that is transitively orientable.

Krawczyk and Kubale [1985] present an approximation algorithm for $P \mid fix \mid C_{\max}$ with biprocessor tasks only, which has worst case bound $4(d-1)/d$, where d is the maximum degree of competition.

Hoogeveen, Van de Velde, and Veltman [1992] consider a second criterion: minimizing the sum of the task completion times; see also Section 5.2. This objective function is often interpreted as a measure of the average time a task is in the multiprocessor system. In general, this criterion leads to severe computational difficulties.

Their main result is establishing NP-hardness in the ordinary sense for $P2 \mid fix \mid \Sigma C_j$. The question whether this problem is solvable in pseudopolynomial time or NP-hard in the strong sense still has to be resolved. The weighted version, however, is shown to be NP-hard in the strong sense. The problem $P3 \mid fix \mid \Sigma C_j$ is also NP-hard in the strong sense. The problem with unit-time processing times is NP-hard in the strong sense if the number of processors is part of the problem instance, but the complexity is still open in case of a fixed number of processors. As could be expected, the introduction of precedence constraints does not simplify the computational complexity. It is shown that even the mildest non-trivial problem of this type, with two processors, unit processing times, and chain-type precedence constraints, is NP-hard in the strong sense. As for the introduction of release dates, Lenstra, Rinnooy Kan, and Brucker [1977] show that even the single-processor problem $1 \mid r_j \mid \Sigma C_j$ is strongly NP-hard.

Dobson and Karmarkar [1989] develop integer programming formulations for $P \mid fix \mid \Sigma w_j C_j$. They apply Langrangian relaxation to obtain lower bounds and an approximation algorithm. The relaxation has a nice intuitive interpretation. Every task J_j that is to execute on more than one processor is split into subtasks, one for each processor it is executed on, and the task weights are divided among the subtasks. For a fixed multiplier, the remaining minimization problem is simply m single-processor minimum weighted flow time problems. These can be solved in $O(mn\log n)$ time [Conway, Maxwell, and Miller, 1967]. Next, the multipliers are adjusted in order to move the subtasks to a common starting time.

Li and Cheng [1990] study the scheduling of tasks on a mesh-connected network of processors; cf. Figure 2.1. Due to the relationship with 2-dimensional bin packing, $P \mid mesh, p_j=1 \mid C_{\max}$ has no polynomial-time 2-approximation algorithm, unless P=NP. A $5+4p/(8q-p)$-approximation algorithm for scheduling tasks on a mesh of size $p \times q$, with $q \leq p \leq 8q$, is given. A 5-approximation algorithm results if each task requires a square submesh. If the size $a_j \times b_j$ of the submesh required by J_j $(j=1,\ldots,n)$ is restricted to $a_j \leq p/k$ and $b_j \leq q/k$, where $k \geq 3$, then these bounds can be reduced to $2+2/(k-2)$ and $2+(2k-1)/(k-1)^2$,respectively.

Several papers are devoted to the cube connected network of processors; cf. Figure 2.1. Chen and Lai [1988A] give a worst-case analysis of *largest dimension, longest processing time list scheduling* (*LDLPT*) for $P \mid cube \mid C_{\max}$. They show that $C_{\max}(LDLPT)/C^*_{\max} \leq 2-1/m$. *LDLPT* scheduling is an extension of Graham's *longest processing time scheduling* algorithm (*LPT*) [Graham, 1966]. It considers the given tasks one at a time in lexicographical order of nonincreasing dimension of the subcubes and processing times, with each task assigned to a subcube that is earliest available.

For the preemptive problem $P \mid cube, pmtn \mid C_{\max}$, Chen and Lai [1988B] give an $O(n^2)$ algorithm that produces a schedule in which each task meets a given deadline, if such a schedule exists. The algorithm considers the tasks one at a time in order of nonincreasing dimension. It builds up a *stairlike* schedule. A schedule is stairlike if a nonincreasing function $f: \{1, \ldots, m\} \rightarrow N$ exists such that each processor M_i is busy up to time $f(i)$ and idle afterwards. The number of preemptions is at most $n(n-1)/2$. By binary search over the deadline values, an optimal schedule is obtained in $O(n^2(\log n + \log \max_j p_j))$ time.

Ahuja and Zhu [1990] also study $P \mid cube, pmtn \mid C_{\max}$ and present an $O(n \log n)$ algorithm to decide whether the tasks can be completed by a given deadline T. Instead of building up stairlike schedules, this algorithm produces *pseudostairlike* schedules. Given a schedule, let t_i be such that processor M_i is busy for $[0, t_i]$ and free for $[t_i, T]$. A schedule is pseudostairlike if $t_i < t_h < T$ implies $h < i$, for any two processors M_h and M_i. Again, the tasks are ordered according to nonincreasing dimension. Dealing with J_j, the algorithm recursively searches for the highest i such that $p_j > T - t_i$. It schedules J_j on processors $M_{i-(2^{d_j}-1)}, \ldots, M_i$ in the time slot $[t_i, T]$, and on $M_{i+1}, \ldots, M_{i+2^{d_j}-1}$ in the time slot $[t_{i+1}, p_j-(T-t_i)]$. By a combination of this algorithm and *binary search*, $C^*_{\max}$ can be determined in $O(n \log n \log(n + \max_j p_j))$ time. Furthermore, since each task except the first is preempted at most once, the algorithm creates no more than $n-1$ preemptions, and this bound is tight.

Shen and Reingold [1991] perform some preprocessing in the sense that the tasks are lexicographically ordered according to nonincreasing dimension of the subcubes and nondecreasing processing times (*LDSPT*). They also build up pseudostairlike schedules, but their algorithm to construct optimal schedules has $O(m^2 n^2)$ time complexity.

Sometimes one wishes to disregard the multiprocessor architecture and simply associate a size with each task to indicate that a task can be processed on any subgraph of that size. Du and Leung [1989] show that $P5 \mid size \mid C_{\max}$ with sizes belonging to $\{1,2,3\}$ is strongly NP-hard. Blazewicz, Drabowski and Weglarz [1986] pay attention to unit-length tasks. They present an $O(n)$ algorithm for solving $P \mid size, p_j{=}1 \mid C_{\max}$, where the tasks require either one or k processors. After calculating the optimal makespan, it schedules the k-processor tasks first and the single-processor tasks next. For the problem with sizes belonging to $\{1,2,\ldots,k\}$, an *integer programming* formulation leads to the observation that for fixed k the problem is solvable in polynomial time. However, if k is specified as part of the problem instance, then the problem remains strongly NP-hard.

For the preemptive case, Blazewicz, Weglarz and Drabowski [1984] propose an $O(n \log n)$ algorithm for solving the special case of $P \mid size, pmtn \mid C_{\max}$ in which the tasks require either one or two processors for processing. An initial step computes $C^*_{\max}$ without giving an optimal schedule. Subsequently, the biprocessor tasks are scheduled using McNaughton's *wrap-around rule* [McNaughton, 1959]. A modification of this rule schedules the single-processor tasks one at a time in order of nonincreasing processing times. In Blazewicz, Drabowski and Weglarz [1986] this result is extended to an $O(n \log n)$ time algorithm for the special case of $P \mid size, pmtn \mid C_{\max}$ in which the tasks require either one or k processors. A *linear programming* formulation shows that for any fixed number of processors the problem $Pm \mid size, pmtn \mid C_{\max}$ with sizes belonging to $\{1,2,\ldots,k\}$ is solvable in polynomial time.

When precedence constraints are imposed, a reduction from 3-Partition shows that $P2 \mid chain, size \mid C_{\max}$ is strongly NP-hard [Du and Leung, 1989]. For the case of unit-length tasks and only two processors, $P2 \mid prec, size, p_j{=}1 \mid C_{\max}$, Lloyd [1981] presents a polynomial-time algorithm. He also proves that the three-processor variant is NP-hard and that *list scheduling* leads to an approximation algorithm for $P \mid prec, size, p_j{=}1 \mid C_{\max}$ with performance bound $(2m - s_{\max})/(m - s_{\max} + 1)$, where $s_{\max}$ is the maximum task size.

Blazewicz, Drozdowski, Schmidt, and de Werra [1992] study scheduling problems for a multiprocessor built up of uniform k-tuples of identical parallel processors; the processing time of J_j is the ratio p_j/q_i, where q_i is the speed of the slowest processor that executes J_j. They show that this problem is solvable in polynomial time if the sizes s_j $(s_j{=}1,\ldots,n)$ are such that $s_j \in \{1,\ldots,k\}$, and $s_j \geq s_k$ implies $s_j/s_k \in Z^+$. For a fixed number of processors, a *linear programming* formulation leads to the observation that the problem is solvable in polynomial time if the task sizes are restricted to $1,\ldots,k$. These results extend those of Blazewicz, Drozdowski, Schmidt, and de Werra [1990] for $k{=}2$.

The most general case, in which each task can be processed on any subgraph of the multiprocessor graph, is studied by Du and Leung [1989]. A *dynamic programming* approach leads to the observation that $P2 | any | C_{\max}$ and $P3 | any | C_{\max}$ are solvable in pseudopolynomial time. Arbitrary schedules for instances of these problems can be transformed into so called *canonical schedules.* A canonical schedule on two processors is one that first processes the tasks using both processors. It is completely determined by three numbers: the total execution times of the single-processor tasks on processor M_1 and M_2 respectively, and the total execution time of the biprocessor tasks. For the case of three processors, similar observations are made. These characterizations are the basis for the development of the pseudopolynomial algorithms. The problem $P4 | any | C_{\max}$ remains open; no pseudopolynomial algorithm is given. For the preemptive case, they prove that $P | any,pmtn | C_{\max}$ is strongly NP-complete by a reduction from 3-Partition. With restriction to two processors, $P2 | any,pmtn | C_{\max}$ is still NP-complete, as is shown by a reduction from Partition. Using a result of Blazewicz, Drabowski and Weglarz [1986], Du and Leung show that for any fixed number of processors $Pm | any,pmtn | C_{\max}$ is also solvable in pseudopolynomial time. The basic idea of the algorithm is as follows. For each schedule S of $Pm | any,pmtn | C_{\max}$, there is a corresponding instance of $Pm | size,pmtn | C_{\max}$ with sizes belonging to $\{1, \ldots, k\}$, in which a task J_j is an l-processor task if it uses l processors with respect to S. An optimal schedule for the latter problem can be found in polynomial time. All that is needed is to generate optimal schedules for all instances of $Pm | size,pmtn | C_{\max}$ that correspond to schedules of $Pm | any,pmtn | C_{\max}$, and choose the shortest among all. It is shown by a dynamic programming approach that the number of schedules generated can be bounded from above by a pseudopolynomial function of the size of $Pm | any,pmtn | C_{\max}$.

Finally, a few words on scheduling problems of the type $P | set | C_{\max}$ restricted to single-processor tasks of unit length. Chang and Lee [1988] use *matching* techniques to construct optimal solutions in $O(n^2 m^2)$ time. Chen and Chin [1989] construct optimal solutions in $O(\min(\sqrt{n}, m) nm \log n)$ time by use of a *network flow* formulation.

Kellerer and Woeginger [1992] impose precedence constraints on the task set. After establishing NP-hardness for $P2 | prec,set,p_j{=}1 | C_{\max}$ with single-processor tasks, they concentrate on precedence relations of the *interval*-type. For these, they show that $P2 | prec,set,p_j{=}1 | C_{\max}$ is solvable in $O(n^2 \sqrt{n})$ time, and that $P | prec,set,p_j{=}1 | C_{\max}$ is solvable in $O(n \log n)$ time in case for each task J_j a processor M_{i_j} is given such that J_j can be executed by any processor of the set $\{M_{i_j}, \ldots, M_m\}$. The complexity of the general problem $P \mid prec, set, p_j{=}1 \mid C_{\max}$ for single-processor tasks with a precedence relation of the interval-type is still open.

3. Communication delays

In this chapter we study the simplest model that allows for communication delays. A set of unit-time tasks has to be processed on identical parallel processors subject to precedence constraints and unit-time communication delays. We are interested in the minimization of the makespan. There are two variants of the problem, depending on whether the number of processors is restricted or not. Using the three-field notation scheme we denote the first variant by $P\,|prec,c=1,p_j=1|\,C_{\max}$ and the second variant by $\bar{P}\,|prec,c=1,p_j=1\,|\,C_{\max}$.

The $P\,|prec,c=1,p_j=1|\,C_{\max}$ problem was first addressed by Rayward-Smith [1987], who established NP-hardness and showed that the length of an *active* schedule is at most equal to $3-2/m$ times the optimal makespan. A schedule is active if no task can start earlier without increasing the start time of another task.

Picouleau [1991B] also considered $P\,|prec,c=1,p_j=1|\,C_{\max}$ and showed that the problem of deciding whether an instance has a schedule of length at most 3 is decidable in polynomial time. For the case of an unrestricted number of processors, he established NP-completeness for the problem of deciding whether an instance has a schedule of length at most 8 [Picouleau, 1991A].

In the first two sections we study the same type of questions as investigated by Picouleau: for what deadline b can one determine in polynomial time if a schedule of length at most b exists? In Section 3.1, we give our own version of the proof that the restricted variant of the problem is polynomially solvable if $b \leq 3$ and show NP-completeness if $b \geq 4$. In Section 3.2, we show for the unrestricted variant that the problem is polynomially solvable if $b \leq 5$ and NP-complete if $b \geq 6$. These results are due to Hoogeveen, Lenstra, and Veltman [1992].

As a consequence, there exists no polynomial-time algorithm with performance bound smaller than 5/4 for $P\,|prec,c=1,p_j=1|\,C_{\max}$ and no polynomial-time algorithm with performance bound smaller than 7/6 for $\bar{P}\,|prec,c=1,p_j=1|\,C_{\max}$, unless P = NP. Thus, neither of these problems has a polynomial approximation scheme, unless P = NP.

In Sections 3.3 and 3.4 we study special types of precedence relations. First, we show that dynamic programming results in a polynomial-time algorithm in case the width of the precedence relation is fixed, i.e., part of the problem type. Second, we show that $P\,|tree,c=1,p_j=1\,|\,C_{\max}$ is NP-hard.

A few open problems remain. The complexity of $Pm\,|prec,c=1,p_j=1\,|\,C_{\max}$ is unknown to us, even for $m=2$, and it is a challenging open problem to approximate an optimal schedule for $P\,|prec,c=1,p_j=1|\,C_{\max}$ appreciably better that a factor of 3 in polynomial time. Variations on list scheduling that construct active schedules may not help, as is shown by an example due to Hurkens [1992]; see Section 3.5.

3.1. The restricted variant

In this section, we start by showing that the problem of deciding whether an instance has a schedule of length at most 3 is decidable in polynomial time. This problem was already solved by Picouleau [1991B]. Next, we prove NP-completeness of the problem of deciding whether an instance has a schedule of length at most 4, even for the special case that the precedence relation has the form of a *bipartite* graph.

Theorem 3.1. *The problem of deciding whether an instance of* $P \mid prec, c=1, p_j=1 \mid C_{\max}$ *has a schedule of length at most* 3 *is solvable in polynomial time.*

Proof. Given an instance of $P \mid prec, c=1, p_j=1 \mid C_{\max}$, we first check whether some trivial necessary constraints for the existence of a feasible schedule of length at most 3 are satisfied. These are the constraints that there are no paths in the graph of length more than 3, that there are no more than $3m$ tasks, and that no two paths of length 3 interfere or share a task. Subsequently, we delete the isolated tasks from the instance; they will be dealt with later.

Our approach to check the existence of a feasible schedule of length at most 3 consists of two steps. We first assign the tasks to time slots. Then the tasks are assigned to the processors by which they have to be executed. The first step proceeds in such a way that the number of processors needed in the second step is minimized.

We first deal with the paths of length 3. The tasks in a path of length 3 are entirely assigned to a single processor. As no two paths of length 3 interfere, the second task in a chain has only one predecessor and only one successor. The first task in a path of length 3 may be succeeded by several tasks without successors; these tasks are assigned to the third time slot. The third task in a path of length 3 may be preceded by several tasks without predecessors; these tasks are assigned to the first time slot. Furthermore, we assign the tasks with two or more successors to the first time slot and the tasks with two or more predecessors to the third time slot.

The tasks that still have to be assigned either belong to isolated chains of length 2 or are the leaves of a rooted intree or outtree with depth at most equal to 2. In case of a chain of length 2, the tasks can be assigned either to the time slots 1 and 2, or to the time slots 2 and 3, or to the time slots 1 and 3; if a task is assigned to the second time slot, then the other task in the chain has to be executed by the same processor. In case of a rooted intree, at most one of the tasks can be assigned to time slot 2 and all other tasks (except the root) must be assigned to time slot 1, whereas in case of a rooted outtree at most one task can be assigned to time slot 2 and all other tasks (except the root) must be assigned to time slot 3. A straightforward approach finds an assignment of the tasks belonging to chains and trees to time slots such that the maximum number of tasks assigned to a single time slot is minimized. If this number exceeds the number of available processors, then clearly the instance has no schedule of length at most 3.

Given an assignment of tasks to time slots, a feasible schedule is constructed in the following way. First assign each task that is to be processed in the second time slot to a processor and assign its predecessor or successor to the same processor. The remaining tasks can be scheduled on arbitrary processors according to the time slot assignment. Finally, the isolated tasks can be used to fill the empty slots. ☐

Theorem 3.2. *The problem of deciding whether an instance of* $P \mid prec, c=1, p_j=1 \mid C_{\max}$ *has a schedule of length at most* 4 *is* NP-*complete, even for bipartite precedence relations.*

Proof. Our proof is based on a reduction from the NP-complete problem Clique and extends the proof by Lenstra and Rinnooy Kan [1978] for the variant without communication delays, $P \mid prec, p_j=1 \mid C_{\max}$. The Clique problem is defined as follows:

Clique
Given a graph $G=(V,E)$ and an integer k, does G have a complete subgraph on k vertices?

Given an instance of Clique, define the number of edges in a clique of size k by $l=k(k-1)/2$ and define $\overline{m}=\max\{\,|V|+l-k,\ |E|-l\,\}$. We construct the following instance of $P \mid prec, c=1, p_j=1 \mid C_{\max}$. There are $m=2(\overline{m}+1)$ processors, which have to process $4m$ tasks. Each vertex $v \in V$ corresponds to a pair of vertex tasks J_v and K_v, and each edge $e \in E$ corresponds to an edge task L_e; we introduce precedence constraints $J_v \to K_v$, and $J_v \to L_e$ if v is incident to e. In addition, we define $4m-2|V|-|E|$ dummy tasks: there are $m-k$ tasks of type W, $m-|V|$ of type X, $m-|V|+k-l$ of type Y, and $m-|E|+l$ of type Z. The precedence constraints between these dummy tasks are such that all W

tasks should precede all Y and Z tasks, and all X tasks should precede all Z tasks.

Suppose that G contains a clique of size k. Then a schedule of length at most 4 is obtained by scheduling the tasks according to the pattern given in Figure 3.1. Here J, K, and L stand for the tasks of type J_v, K_v, and L_e, respectively, J_{clique} (K_{clique}) denotes the set of tasks of type J (K) corresponding to the clique vertices, and L_{clique} denotes the set of tasks of type L corresponding to the clique edges.

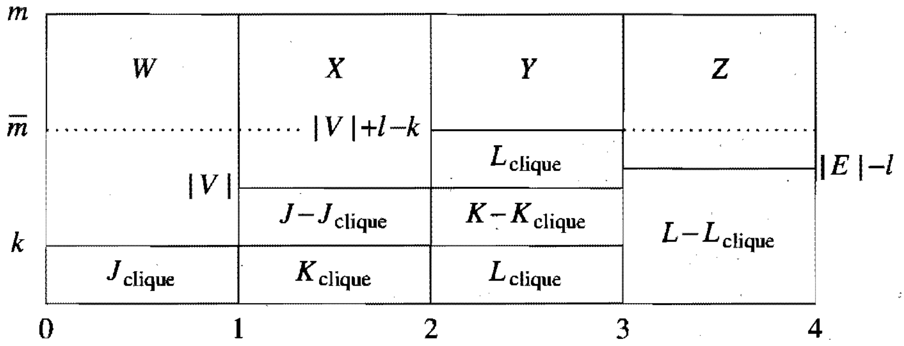


Figure 3.1. Schedule of length 4.

Conversely, suppose that there exists a feasible schedule σ of length at most 4. We will show that in any such schedule the non-dummy tasks processed in time slot 1 correspond to the vertices of a clique of size k. The W tasks are processed in time slot 1 in σ, since they must precede all of the tasks of types Y and Z, of which there are at least $m+2$. A similar argument shows that the Z tasks are processed in time slot 4 in σ. It follows immediately from these observations that the tasks of types X and Y are processed in σ in the time periods $[0,2]$ and $[2,4]$, respectively.

As the number of tasks is exactly equal to $4m$, σ does not contain any idle time; hence, next to the tasks of type W and X, exactly $k+|V|$ vertex tasks must be processed in time period $[0,2]$. As the vertex tasks of type J have to precede the corresponding vertex tasks of type K, we know that no more than $|V|$ vertex tasks are processed in time slot 2 in σ. This observation, combined with the observation that all X tasks are processed in time period $[0,2]$, implies that σ processes all X tasks in time slot 2, that k vertex tasks of type J are processed in time slot 1, and that the corresponding vertex tasks of type K and the remaining vertex tasks of type J are processed in time slot 2. The set of tasks that are processed in time slot 3 consists of Y tasks, edge tasks L that have both predecessors processed in time slot 1, vertex tasks of type K, and L tasks with one predecessor in time slot 1 and one predecessor in time slot 2; the total number of these tasks is equal to m, as σ contains no idle time. Note that both the K tasks and the L tasks with one predecessor in time slot 2 must be scheduled immediately after their preceding task of type J, implying that the number of these tasks is at most $|V|-k$. Hence, there are at least l edge tasks with both predecessors processed in time slot 1, implying that the k vertices corresponding to the k vertex tasks that are processed in time slot 1 induce a complete subgraph of G. □

Corollary 3.1. *For* $P\,|prec,c=1,p_j=1\,|\,C_{\max}$ *there exists no polynomial-time algorithm with performance bound smaller than* 5/4, *unless* P=NP. □

3.2. The unrestricted variant

This section concerns the variant for which the number of processors is not restrictively small. We first show that the problem of deciding whether an instance has a schedule of length at most 5 is solvable in polynomial time. Next we show that the problem of deciding whether an instance has a schedule of length at most 6 is NP-complete.

Theorem 3.3. *The problem of deciding whether an instance of* $\bar{P}\,|prec,c=1,p_j=1\,|\,C_{\max}$ *has a schedule of length at most 5 is solvable in polynomial time.*

Proof. Given an arbitrary instance of the problem $\bar{P}\,|prec,c=1,p_j=1\,|\,C_{\max}$, we first check whether some obviously necessary constraints hold. These are that the graph contains no path of length more than 5 and that there are no two interfering paths of length 5. Suppose that these constraints are satisfied. Then it is easy to see that each task that does not belong to a path of length 4 can be assigned to a processor and time slot without violating any constraint. We now present a polynomial-time algorithm that checks whether a given set of paths of length 4 fits into a feasible schedule of length at most 5.

Let $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$ denote a path of length 4. Without loss of generality, J_1 and J_4 can be processed in the first and last time slot, respectively. We develop an algorithm to check in polynomial time whether there exists a feasible assignment of the middle tasks J_2 and J_3 to time slots, while observing the constraint that two dependent tasks that are assigned to two consecutive time slots must be performed by the same processor. We distinguish a number of cases.

Suppose that J_2 has at least two predecessors, implying that J_2 cannot start

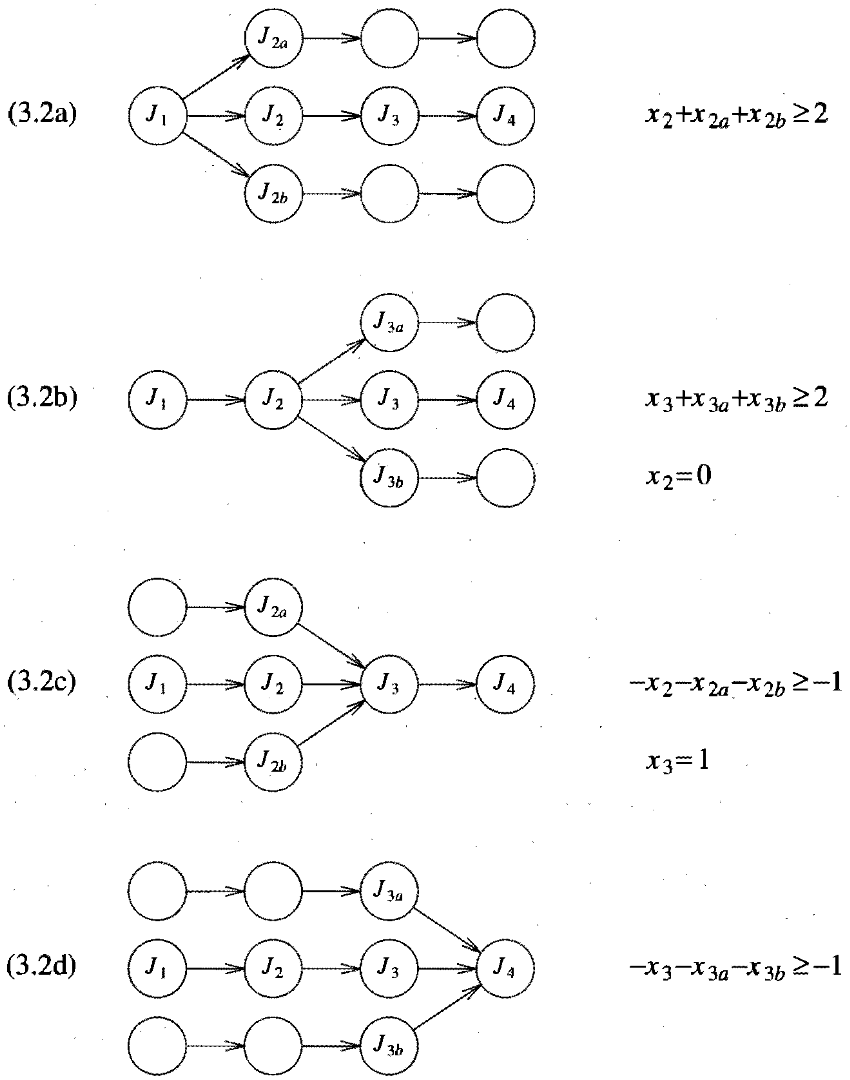


Figure 3.2. The four cases.

before time 2; then we have to assign J_2, J_3, and J_4 to the last three time slots and they have to be performed by the same processor. Similarly, if J_3 has at least two successors, then it has to be executed in time slot 3 and J_1 and J_2 have to be executed by the same processor in the first and second time slot, respectively.

The other cases require a more intricate procedure. From now on, J_2 has one predecessor and J_3 has one successor. For each unscheduled task J_j, we define the *depth* d_j as the number of tasks, in a path of length 4, that precede this task; thus $d_2 = 1$ and $d_3 = 2$. As J_j starts at time d_j or d_j+1 in any feasible schedule of length at most 5, we have that $S_j = d_j + x_j$, with $x_j \in \{0,1\}$. The problem of assigning feasible start times to the tasks J_j can thus be formulated as a problem of assigning feasible binary values to the variables x_j.

Consider the case depicted in Figure 3.2a. Let V denote the set of immediate successors of J_1 that belong to a path of length 4; in this case we have $V = \{J_2, J_{2a}, J_{2b}\}$. As at most one of the tasks in V can be executed in time slot 2, the x variables corresponding to the tasks in V must satisfy the constraint $\Sigma_{j \in V} x_j \geq |V| - 1$.

Three other cases we distinguish are illustrated in Figures 3.2b-d. Analogous observations lead to similar constraints for each case; these constraints are shown next to the graph. Note that in case 3.2b we have already assigned J_2 to time slot 2, and that in case 3.2c task J_3 has already been assigned to time slot 4. If two dependent tasks J_2 and J_3 have not yet been assigned to time slots, then we have to add the constraint $x_3 - x_2 \geq 0$ to ensure consistency. Note that each binary solution that satisfies all constraints derived for the cases 3.2a through 3.2d induces a feasible schedule of length at most 5; let $Ax \geq b$ denote the set of constraints.

It is easily verified that every column of A contains at most one +1 and at most one −1 entry, implying that A is a network matrix [Schrijver, 1986]. Hence, if we add the inequalities $0 \leq x_j \leq 1$, then the constraint matrix remains totally unimodular and the polyhedron $\{x \mid 0 \leq x \leq 1; Ax \geq b\}$ is integral. As we can decide in polynomial time whether the polyhedron is empty, the problem whether a given instance of $\bar{P} \mid prec, c = 1, p_j = 1 \mid C_{\max}$ has a schedule of length at most 5 is decidable in polynomial time. □

Theorem 3.4. *The problem of deciding whether an instance of* $\bar{P} \mid prec, c=1, p_j=1 \mid C_{\max}$ *has a schedule of length at most* 6 *is* NP-*complete.*

Proof. Our proof is based on a reduction from the NP-complete problem 3-Satisfiability.

3-Satisfiability
Given a set U of variables and a collection C of clauses over U such that each clause $c \in C$ has $|c| = 3$, does there exist a truth assignment for C?

Given an instance (U, C) of 3-Satisfiability, we construct the following instance of $\bar{P} \mid prec, c = 1, p_j = 1 \mid C_{\max}$. For each variable x we introduce six

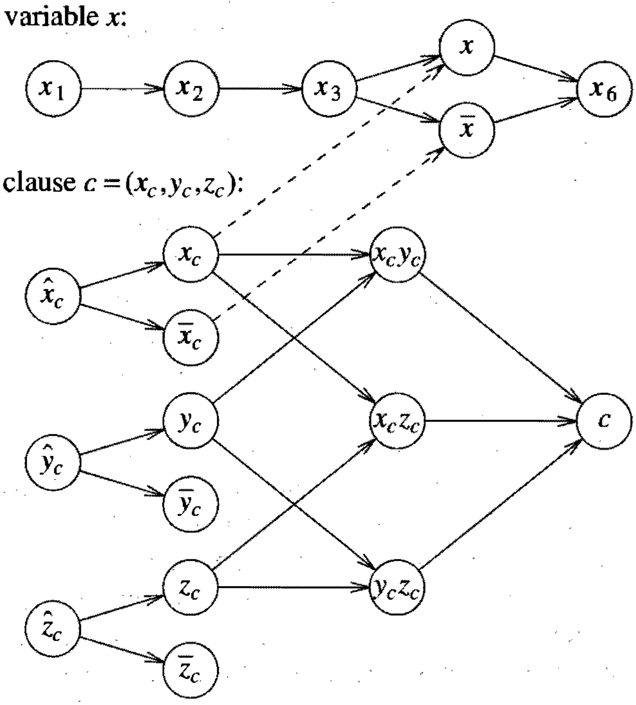


Figure 3.3. Variable tasks and clause tasks.

tasks: $x_1, x_2, x_3, x, \bar{x}$, and x_6; the precedence constraints between these tasks are given in Figure 3.3. For each clause $c=(x_c,y_c,z_c)$, where the literals x_c, y_c, and z_c are occurrences of negated or unnegated variables, we introduce thirteen tasks: $\hat{x}_c, \hat{y}_c, \hat{z}_c, x_c, \bar{x}_c, y_c, \bar{y}_c, z_c, \bar{z}_c, x_cy_c, x_cz_c, y_cz_c$, and c; the precedence constraints between these tasks are also given in Figure 3.3. We further introduce precedence constraints between the variable tasks and the clause tasks. If the occurrence of variable x in c is unnegated, then x_c precedes the variable task x and $\bar{x}_c$ precedes the variable task $\bar{x}$, as illustrated in Figure 3.3. If the occurrence of variable x in c is negated, then x_c precedes the variable task $\bar{x}$ and $\bar{x}_c$ precedes the variable task x. Thus, x_c represents the occurrence of variable x in clause c; it precedes the corresponding variable task.

We start by making two essential observations. First, note that in a schedule of length at most 6 there are exactly two ways to schedule the tasks corresponding to a variable x, depending upon whether variable task x is scheduled in time slot 4 and variable task $\bar{x}$ in time slot 5 or the other way around. In both cases, the tasks x_1, x_2, and x_3 have to be performed by the same processor as the variable task that is scheduled in time slot 4 and the task

x_6 has to be performed by the same processor as the variable task scheduled in time slot 5. Second, note that in order to schedule the clause tasks corresponding to clause $c=(x_c,y_c,z_c)$ within six time units at least one of the tasks x_c, y_c, and z_c must be scheduled in time slot 2.

Suppose that a truth assignment for C exists. Then a schedule of length at most 6 is obtained by scheduling the variable task x in time slot 4 if variable x is true and in time slot 5 otherwise. If the literal x occurring in clause c is true, then x_c is scheduled in time slot 2 on the same processor as $\hat{x}_c$, and $\bar{x}_c$ is scheduled in time slot 3; if the literal x in c is false, then the task x_c is scheduled in time slot 3 and $\bar{x}_c$ in time slot 2. The other tasks are scheduled in a greedy manner. As every clause c contains at least one true literal, each clause task c will be completed by time 6.

Conversely, suppose that there exists a schedule of length at most 6. We will show that there exists a truth assignment for the instance of 3-Satisfiability. Define a variable x as true if the corresponding variable task x is processed in time slot 4, and false otherwise. Without loss of generality, suppose that variable task x is executed in time slot 4. Each unnegated occurrence of x must be scheduled in time slot 2 and each negated occurrence of x in slot 3, implying that all literals are assigned values consistently. As each clause task c has been completed at time 6, we know that each clause contains at least one true literal.
☐

Corollary 3.2. *For $\bar{P}\,|prec,c=1,p_j=1\,|\,C_{\max}$ there exists no polynomial-time algorithm with performance bound smaller than* 7/6, *unless* P=NP. ☐

3.3. A dynamic programming formulation

Given a precedence relation, let $G=(V,A)$ be a directed graph that represents the transitive closure of the relation. Two elements $J_j,J_k\in V$ are said to be *incomparable* if neither $(J_j,J_k)\in A$ nor $(J_k,J_j)\in A$. The *width* w of the precedence relation is the largest number of pairwise incomparable elements of the graph G. Möhring [1989] showed that $P\,|prec,p_j=1\,|\,C_{\max}$ with a precedence relation of fixed width is solvable in polynomial time by dynamic programming. This result can be extended for problems of the type $P\,|prec,c=1,p_j=1\,|\,C_{\max}$ with a precedence relation of fixed width, in the following manner.

A subset $I\subseteq V$ is an *order ideal* if $J_k\in I$ and $J_j\to J_k$ imply that $J_j\in I$. The number of order ideals of G is bounded by $O(n^w)$; this bound is tight if G consists of w parallel chains. A feasible schedule σ of length $C_{\max}(\sigma)$ can be viewed as a sequence of $C_{\max}(\sigma)$ columns. Every initial part of σ with in its last column a task set T_1 corresponds to an order ideal I_1. Adding a next column to the part of the schedule associated with (I_1,T_1) results in an (order ideal, column) pair (I_2,T_2) with $I_1 \subseteq I_2$ and $T_2 = I_2 - I_1$. This implies that the construction of a feasible schedule can be viewed as following a path in the digraph D associated with the states (I,T). The vertices of D correspond to the states and the arcs represent possible transitions. An arc $(I_1,T_1) \rightarrow (I_2,T_2)$ occurs if and only if the transition it represents satisfies the following conditions. First, it always has to respect the constraints $I_1 \subseteq I_2$ and $T_2 = I_2 - I_1$. Second, if $T_1 = \emptyset$, then it also has to respect the constraints

(1) $1 \leq |T_2| \leq m$, and
(2) the elements of T_2 are pairwise incomparable.

If $T_1 \neq \emptyset$, then the transition either has to respect the constraint $I_2 = I_1$, or the previously mentioned constraints (1) and (2) as well as the constraints

(3) for each $J_j \in T_1$ there is at most one $J_k \in T_2$ such that $J_j \rightarrow J_k$, and
(4) for each $J_k \in T_2$ there is at most one $J_j \in T_1$ such that $J_j \rightarrow J_k$.

Any path from $(\emptyset,\emptyset)$ to any state (I,T) corresponds to a schedule for the induced suborder on I with the tasks of T in its last column. The digraph D can be used for a recursive computation of an optimal schedule. It fulfills the recursion property

$$C^*_{\max}(I_2,T_2) = \min \{ 1 + C^*_{\max}(I_1,T_1) \mid (I_1,T_1) \rightarrow (I_2,T_2) \text{ is an arc of } D \},$$

where $C^*_{\max}(I,T)$ denotes the optimal value associated with the pair (I,T). Thus, the globally optimal solution can be computed along with the transition graph D. It follows that this dynamic programming procedure takes at most as many steps as there are arcs in D.

Each state (I_2,T_2) corresponds to a pair of ideals (I_1,I_2), because $T_2 = I_2 - I_1$. From the above observations we conclude that there are $O(n^{2w})$ states. Each transition corresponds to a choice of a task set out of at most w tasks. Thus, given an initial state there are at most 2^w transitions to another state. It follows that the dynamic programming algorithm takes $O(2^w n^{2w})$ time. We have proven the following theorem.

Theorem 3.5. *Given an instance of* $P \mid prec, c{=}1, p_j{=}1 \mid C_{\max}$ *with a fixed-width precedence relation, one can construct an optimal schedule in polynomial time.* □

3.4. Trees

Hu [1961] gave an $O(n)$ time algorithm to solve $P\,|\,tree,p_j{=}1\,|\,C_{\max}$. It is a critical path scheduling algorithm: the next task chosen is one that heads the longest current chain of unexecuted tasks. Lenstra, Veldhorst, and Veltman [1993] apply list scheduling in order to construct optimal schedules for instances of the type $P2\,|\,tree,c{=}1,p_j{=}1\,|\,C_{\max}$ in $O(n)$ time. They also show that $P\,|\,tree,c{=}1,p_j{=}1\,|\,C_{\max}$ is NP-hard. The last result is presented below.

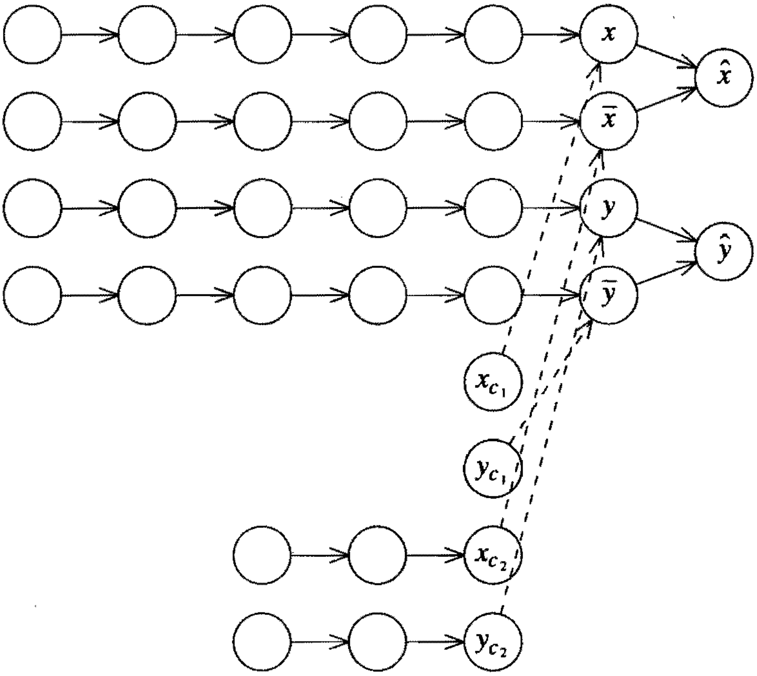


Figure 3.4. Variable tasks and clause tasks corresponding to a Satisfiability instance with $c_1 = (x,\bar{y})$ and $c_2 = (\bar{x},y)$.

Theorem 3.6. *The* $P\,|\,tree,c{=}1,p_j{=}1\,|\,C_{\max}$ *problem is* NP-*hard.*

Proof. The proof is based on a reduction from the NP-complete problem Satisfiability.

Satisfiability

Given a set U of variables and a collection C of clauses over U, does there exist a truth assignment for C?

Given an instance (U,C) of Satisfiability, define a threshold value b as $b = 2|C|+4$ and let the number of processors be given by $m = 2|U| + \Sigma_{c \in C}|c|+1$.

For each variable x we introduce a task $\hat{x}$ and two *variable chains* consisting of $b-2$ tasks each; one of these chains corresponds to the literal x and the other corresponds to the literal $\bar{x}$. Both chains precede $\hat{x}$, as illustrated in Figure 3.4. Let $c_1, \ldots, c_{|C|}$ be an arbitrary ordering of the clauses in C. For each clause c_i $(i = 1, \ldots, |C|)$ we introduce $|c_i|$ *clause chains* consisting of $2i-1$ tasks each; there is a one-to-one correspondence between these chains and the literals that constitute c_i. We introduce precedence constraints between the variable tasks and the clause tasks, as follows. If the occurrence of variable x in c_i is unnegated, then the last task (x_{c_i}) of the clause chain corresponding to this occurrence has to precede the last task of the variable chain corresponding to the literal x. If the occurrence of x in c_i is negated, then x_{c_i} has to precede the last task of the variable chain corresponding to $\bar{x}$. For an illustration see the dashed lines in Figure 3.4.

Finally, we introduce a total of $b + |U| + \Sigma_{i=1}^{|C|} \{ |c_i|(b-2i-3)+1 \}$ dummy tasks, which form a number of chains. First, there is a single chain of length b. Second, there are $|U|$ unit length chains, each consisting of a single task. Third, for each clause c_i there are $|c_i|$ chains of dummy tasks; one is of length $b-2i-2$ and the other chains are of length $b-2i-3$. For each dummy chain of length l, $l < b$, its last task has to precede the $(l+2)$nd task of the dummy chain of length b. Hence, in a feasible schedule of length b each dummy chain is scheduled on a single processor, the first task of such a chain is executed in time slot 1, and the execution of the chain is without interruption.

Suppose that a truth assignment for the Satisfiability instance (U,C) exists. Given such a truth assignment, one can construct a schedule of length b as follows. If variable x is true, then the variable task x is performed in time slot $b-1$ on the same processor as $\hat{x}$ and the variable task $\bar{x}$ is performed in time slot $b-2$. If variable x is false, then the variable task $\bar{x}$ is performed in time slot $b-1$ on the same processor as $\hat{x}$ and the variable task x is performed in time slot $b-2$. If x_{c_i} is an unnegated occurrence of x in c_i and variable x is true, then x_{c_i} is performed in time slot $b-3$. Now, the tasks corresponding to clause c_i are executed by the same processors that execute dummy chains of length $b-2i-2$ and $b-2i-3$. Thus, given the assignment of the variable tasks one can easily construct a feasible schedule of length b; for an illustration see Figure 3.5.

Conversely, suppose that there exists a feasible schedule of length at most b. The dummy tasks impose a structure on such a schedule, as illustrated in Figure 3.5 by the black dots. At most $|U|$ non-dummy tasks can be processed in

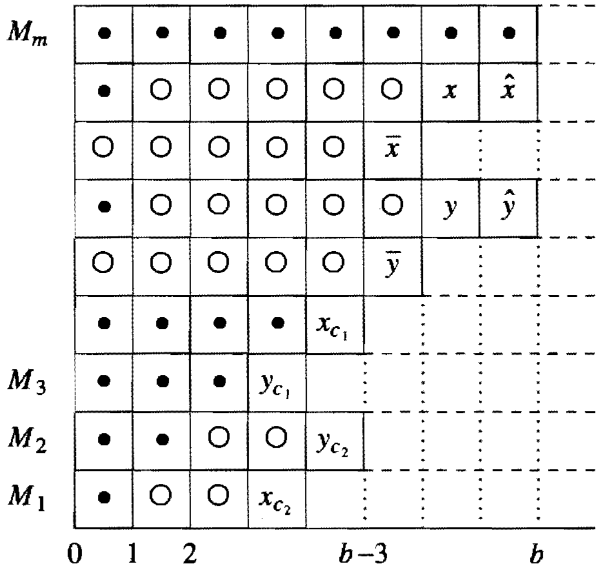


Figure 3.5. A schedule of length b.

time slot 1. Given a variable x, at least one of the two chains corresponding to x has to begin its execution at 0. Thus, for each variable x exactly one of its chains is processed in time period $[0,b-2]$, and exactly one of its chains is processed in time period $[1,b-1]$. The latter chain is executed by the same processor as $\hat{x}$; the literal corresponding to this chain is regarded to be true. Given a clause c_i, the corresponding clause chains can only be executed by processors that execute dummy chains of length at most $b-2i-2$. It follows that the clause chains of c_i are executed by the processor that executes a dummy chain of length $b-2i-2$ and the $|c_i|-1$ processors that each execute a dummy chain of length $b-2i-3$. At least one of these clause chains completes in time slot $b-3$; it has to precede a true literal, since otherwise the schedule would not be feasible. Hence, from the schedule we can derive a truth assignment for the corresponding Satisfiability instance. □

3.5. Two open problems

As indicated in the introduction of this chapter, Rayward-Smith [1987] showed that the length of an active schedule is at most $3-2/m$ times the optimal makespan. It is a challenging open problem to approximate an optimal solution appreciably better than a factor of 3 in polynomial time. Critical path scheduling, for instance, does not improve upon this bound. Critical path scheduling is a special form of list scheduling: it gives priority to the task that precedes the longest chain of other tasks. The following example, due to Hurkens [1992], shows that critical path scheduling is, asymptotically, a 3-approximation algorithm and thereby no better than Rayward-Smith's algorithm.

A total of $n=(m+2)q$ tasks have to be processed by m processors. The precedence constraints are such that task J_{jk} precedes task $J_{(j+1)k}$, for $j=1,\ldots,q-1$ and $k=1,\ldots,m+2$, and task J_{j1} precedes tasks $J_{(j+1)k}$, for $j=1,\ldots,q-1$ and $k=2,\ldots,m+2$; cf. Figure 3.6. Critical path scheduling may generate a schedule of length $C_{\max}(CP)=(3q-1)m$, whereas the optimal makespan is $C^*_{\max}=(m+2)(q-1)+2m-1$. It follows that $C_{\max}(CP)/C^*_{\max}=3m/(m+2)$.

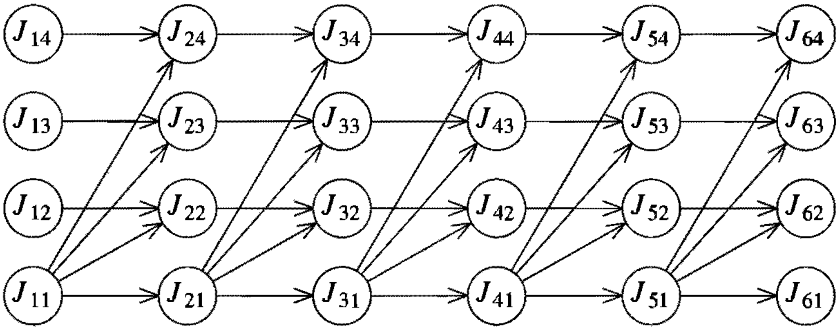


Figure 3.6. A bad example for critical path scheduling.

Another open problem is the complexity of $Pm\,|prec,c=1,p_j=1\,|\,C_{\max}$, even for $m=2$. In case of no communication delays, $P2\,|prec,p_j=1\,|\,C_{\max}$ is solvable in polynomial time [Fujii, Kasami, and Ninomiya, 1969, 1971], but the complexity of $P3\,|prec,p_j=1\,|\,C_{\max}$ is not yet determined.

4. Task duplication

In this brief chapter we study scheduling problems of the type $P\mid prec,c_{jk},dup\mid C_{\max}$ in their most general form. As indicated in Section 2.4, the duplication of tasks may reduce or avoid communication delays that would occur otherwise. We are interested in the possible profit task duplication offers compared to the case that it is not allowed. In general, task duplication can decrease the schedule length by a factor of at most m, even for tree-type precedence relations. However, in case of unit-time processing requirements and unit-time communication delays, task duplication can help a factor of two, but no more. Note that from the proof of Theorem 3.2 it follows that $P\mid prec,c=1,dup,p_j=1\mid C_{\max}$ is NP-hard.

4.1. The potential profit

From a computational point of view, $P\mid intree,c_{jk}\mid C_{\max}$ and $P\mid outtree,c_{jk}\mid C_{\max}$ are the same; one speaks of the problem type $P\mid tree,c_{jk}\mid C_{\max}$. This is no longer the case if duplication is allowed. In case of precedence relations of the intree-type, duplication will not help. Thus, for $P\mid intree,c_{jk},dup\mid C_{\max}$ it holds that $C^*/C_d^*=1$, where C_d^* is the optimal makespan allowing for task duplication and C^* is the optimal makespan without duplication. In case of outtrees duplication can be very profitable.

Theorem 4.1. *For $P\mid prec,c_{jk},dup\mid C_{\max}$ duplication can help a factor of at most m. This bound is tight for $P\mid outtree,c,dup,p_j=1\mid C_{\max}$ and $P\mid outtree,c=1,dup\mid C_{\max}$. For $P\mid prec,c=1,dup,p_j=1\mid C_{\max}$ duplication can help a factor of at most 2.*

Proof. Given an instance of $P\mid prec,c_{jk},dup\mid C_{\max}$, notice that $C_d^*\geq\Sigma p_j/m$ and $C^*\leq\Sigma p_j$. It follows that $C^*/C_d^*\leq m$. The examples below show that this bound is tight, even for subclasses of the problem type.

Given an instance of $P\mid prec,c=1,dup,p_j=1\mid C_{\max}$, one can transform an optimal schedule with duplication into a schedule of length $2C_d^*-1$ without duplication by inserting C_d^*-1 idle periods and deleting the duplicates. It follows that $C^*/C_d^*\leq 2$. □

$P\mid outtree,c,dup,p_j=1\mid C_{\max}$

A root J_r precedes m chains of k tasks each, as illustrated in Figure 4.1 for $m=3$ and $k=4$. The communication delay c is such that $c\geq km$. An optimal schedule allowing for duplication is of length $k+1$, whereas an optimal schedule without duplication has length $mk+1$. Thus, $C^*/C_d^*=(mk+1)/(k+1)$.

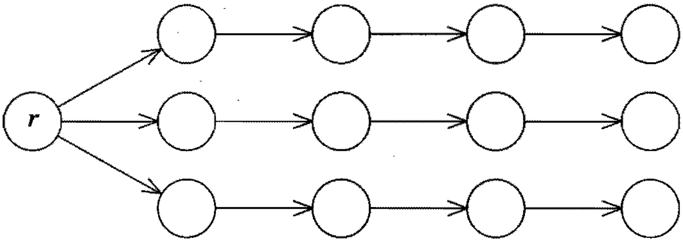


Figure 4.1. An instance of $P \mid outtree, c, dup, p_j{=}1 \mid C_{\max}$.

$P \mid outtree, c{=}1, dup \mid C_{\max}$

A root J_r precedes m mutually independent tasks, as illustrated in Figure 4.2 for $m{=}3$. The root has processing requirement 0 and each of the remaining tasks has processing requirement $1/m$. An optimal schedule without duplication executes all tasks on a single processor and is of length 1, whereas an optimal schedule with duplication is of length $1/m$. Thus, $C^*/C_d^*{=}m$.

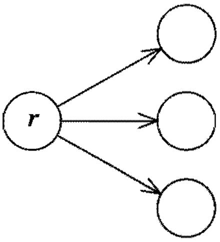


Figure 4.2. An instance of $P \mid outtree, c{=}1, dup \mid C_{\max}$.

$P \mid outtree, c{=}1, dup, p_j{=}1 \mid C_{\max}$

Let there be $m = 2^d$ processors, where d is a positive integer. The precedence relation is represented by means of a full binary tree on $2m-1$ nodes, as illustrated in Figure 4.3 for $m{=}4$. An optimal schedule with duplication is of length $1+\log m$, whereas an optimal schedule without duplication has makespan $1+2\log m$. Thus, $C^*/C_d^*{=}2-1/\log 2m$.

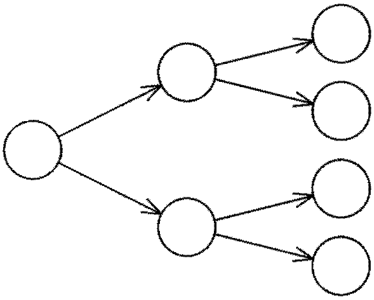


Figure 4.3. An instance of $P \mid outtree, c=1, dup, p_j=1 \mid C_{\max}$.

5. Multiprocessor tasks with prespecified processor allocations

The problems and algorithms mentioned in the previous chapters deal with tasks that are processed on a single processor and focus on communication delays. The following chapter disregards the notion of communication and concentrates on tasks that may require more than one processor. We assume that for each task a single subgraph is specified on which it has to be processed and we investigate the computational complexity of allocating tasks to time intervals. We are interested in two objectives: the minimization of the maximum and the total completion time.

We will investigate the complexity of two classes of problems denoted by $P \mid fix \mid C_{\max}$ and $P \mid fix \mid \Sigma C_j$, respectively. We refer to Section 2.5.2 for a literature review. The outline of this chapter is as follows.

Section 5.1 deals with the makespan criterion. The general problem with a fixed number m of processors is polynomially solvable if $m=2$, but NP-hard in the strong sense for $m \geq 3$. There are two well-solvable cases. The first one concerns the case of unit processing times; the problem is then solvable in polynomial time through an integer programming formulation with a fixed number of variables. The second one concerns the three-processor problem in which all multiprocessor tasks of the same type are decreed to be executed consecutively, the so-called *block-constraint*; this problem is solvable in $O(n \Sigma p_j)$ time. If the number of processors is part of the problem instance, then the problem with unit processing times is already NP-hard in the strong sense. In general, the introduction of precedence constraints or release dates leads to strong NP-hardness, with one exception: the problem with unit processing times in which both the number of processors and the number of distinct release dates are fixed is solvable in polynomial time through an integer programming formulation with a fixed number of variables. The computational complexity of the problem $Pm \mid fix, r_j, p_j{=}1 \mid C_{\max}$ is still open.

Section 5.2 deals with the total completion time criterion. In general, this criterion leads to severe computational difficulties. The problem is NP-hard in the ordinary sense for $m=2$ and in the strong sense for $m=3$. The weighted version and the problem with precedence constraints are already NP-hard in the strong sense for $m=2$. The problem with unit time processing times is NP-hard in the strong sense if the number of processors is part of the problem instance, but still open in case of a fixed number of processors. Another open problem is $Pm \mid fix, r_j, p_j{=}1 \mid \Sigma C_j$.

5.1. Makespan

In this section, we investigate the computational complexity of minimizing the makespan. If no precedence relation is specified, then we may discard the tasks that need all the processors for execution, since they can be scheduled ahead of the other ones. Hence, the two-processor problem without precedence constraints is simply solved by scheduling each single-processor task on its processor without causing idle time.

5.1.1. *Three processors and the block-constraint*

The block-constraint decrees that all biprocessor tasks of the same type are scheduled consecutively. As this boils down to the case that there is at most one biprocessor task of each type, we replace all biprocessor tasks of the same type by one task of this type with processing time equal to the sum of the individual processing times. The biprocessor task that requires M_2 and M_3 is named a task of type A and its processing time is denoted by p_A. Correspondingly, the biprocessor task that requires M_1 and M_3 and the biprocessor task that requires M_1 and M_2 are said to be of type B and C, respectively; their processing times are denoted by p_B and p_C.

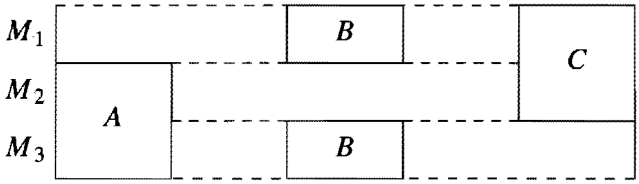


Figure 5.1. A schedule satisfying the block-constraint.

Theorem 5.1. *The problem* $P3\,|\,fix\,|\,C_{\max}$ *subject to the block-constraint is* NP-*hard in the ordinary sense.*

Proof. We will show that $P3\,|\,fix\,|\,C_{\max}$ subject to the block-constraint is NP-hard by a reduction from the NP-complete problem Partition.

Partition
Given a multiset $N=\{a_1,\ldots,a_n\}$ of n integers, is it possible to partition N into two disjoint subsets that have equal sum $b=\Sigma_{j\in N}a_j/2$?

Given an instance of Partition, define for each $j\in N$ a task J_j that requires M_1 for execution and has processing time $p_j=a_j$. In addition, we introduce five *separation tasks* that create two time slots of length b on M_1. The tasks J_A, J_B, and J_C, each with processing time b, are of the type A, B, and C, respectively. The two single-processor tasks J_{n+1} and J_{n+2}, each with processing time $2b$, have to be executed by M_2 and M_3, respectively.

Note that each processor has a load of $4b$, which implies that $4b$ is a lower bound on the makespan of any feasible schedule. We will show that Partition has a solution if and only if there exists a schedule for the corresponding instance of $P3\,|\,fix\,|\,C_{\max}$ with $C_{\max} \leq 4b$.

Suppose that there exists a subset $S \subset N$ such that $\Sigma_{j\in S} a_j = \Sigma_{j\in N-S} a_j = b$. A schedule of length $C_{\max} = 4b$ then exists, as is illustrated in Figure 5.2.

Conversely, notice that only four possibilities exist to schedule the tasks J_{n+1}, J_{n+2}, J_A, J_B, and J_C in a time interval of length $4b$. Each of these possibilities leaves two separated idle periods of length b on processor M_1, in which the tasks J_j with $j \in N$ must be processed. Thus, if there exists a schedule of length $C_{\max} = 4b$, then there is a subset $S \subset N$ such that $\Sigma_{j\in S} a_j = \Sigma_{j\in N-S} a_j$.

We conclude that $P3\,|\,fix\,|\,C_{\max}$ is NP-hard in the ordinary sense. □

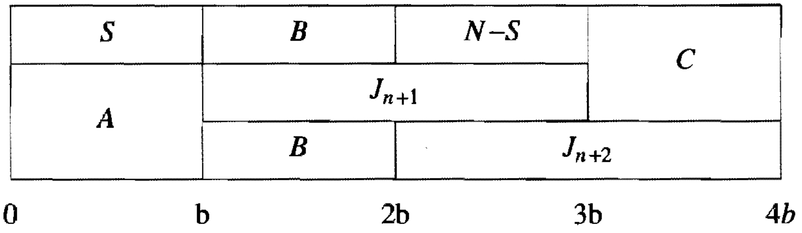


Figure 5.2. A schedule with partition sets S and $N-S$.

Theorem 5.2. *The problem $P3\,|\,fix\,|\,C_{\max}$ subject to the block-constraint is solvable in pseudopolynomial time.*

Proof. We propose an algorithm for this problem that requires $O(n\Sigma_{j\in N} p_j)$ time and space. For $i=1,2,3$, let T_i denote the set of indices of tasks that require only M_i for processing, and $n_i = |T_i|$. In addition, we define $p(S) = \Sigma_{j\in S} p_j$.

Using an interchange argument, we can transform any optimal schedule into an optimal schedule with some biprocessor task scheduled first and some other biprocessor task scheduled last. Suppose for the moment that these tasks are of type A and C, respectively; a B-type task is then scheduled somewhere in between. Any feasible schedule of this type, referred to as an *ABC*-schedule, is completely specified by the subsets $Q_1 \subseteq T_1$ and $Q_3 \subseteq T_3$ scheduled before the B-type task; see Figure 5.3.

For an *ABC*-schedule with given subsets Q_1 and Q_3, the earliest start time of the task of type B is

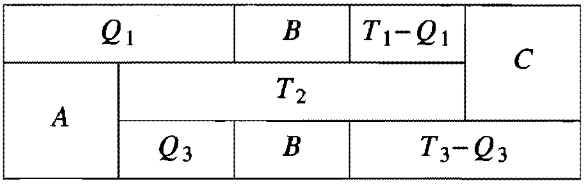


Figure 5.3. Structure of an *ABC* schedule.

$$S_B(Q_1,Q_3)=\max\{p(Q_1),p_A+p(Q_3)\}.$$

The earliest start time of the task of type C is then

$$S_C(Q_1,Q_3)=\max\{S_B(Q_1,Q_3)+p_B+p(T_1-Q_1),p_A+p(T_2)\}.$$

The minimal length of such a schedule is therefore

$$(1)\quad C_{\max}(Q_1,Q_3)=\max\{S_C(Q_1,Q_3)+p_C,\,S_B(Q_1,Q_3)+p_B+p(T_3-Q_3)\}.$$

Hence, the minimal length of an *ABC*-schedule is determined by $p(Q_1)$ and $p(Q_3)$. In other words, the length of an optimal *ABC*-schedule is equal to the minimum of $C_{\max}(Q_1,Q_3)$ over all possible values of $p(Q_1)$ and $p(Q_3)$. Due to symmetry, we can transform any *ABC*-schedule into an *CBA*-schedule of the same length. The only other types of schedules of interest to us are therefore the *BAC* and *ACB*-schedules. Similar arguments show that the length of an optimal *BAC*-schedule is equal to the minimum of $C_{\max}(Q_2,Q_3)$ over all possible values of $p(Q_2)$ and $p(Q_3)$, and that the length of an optimal *ACB*-schedule is equal to the minimum of $C_{\max}(Q_1,Q_2)$ over all possible values of $p(Q_1)$ and $p(Q_2)$.

For $i=1,2,3$, we compute all possible values that $p(Q_i)$ can assume in $O(n_i p(T_i))$ time and space by a standard dynamic programming algorithm of the type also used for the knapsack and the subset-sum problems; see e.g. Martello and Toth [1990]. If these values are put in sorted lists, then all possible values that $S_B(Q_1,Q_3)$ can assume are computed in $O(p(Q_1)+p(Q_3))$ time and space. The minimum of $C_{\max}(Q_1,Q_3)$ over $p(Q_1)$ and $p(Q_3)$ is then determined by evaluating expression (1) for each possible combination of $p(Q_1)$ and $p(Q_3)$; this takes $O(p(T_1)+p(T_3))$ time.

The lengths of the optimal *BAC* and *ACB*-schedules are determined similarly. The overall minimum then follows immediately, and an optimal schedule is determined by backtracing. Since $n_i \le n$ and $p(T_i) \le \Sigma_{j\in N} p_j$ for each i, it takes $O(n\Sigma_{j\in N}p_j)$ time and space to find an optimal schedule. □

5.1.2. *Strong NP-hardness for the general 3-processor problem*

Theorem 5.3. *The problem* $P3 \mid fix \mid C_{\max}$ *is* NP-*hard in the strong sense.*

Proof. The proof is based upon a reduction from the strongly NP-complete problem 3-Partition. It is similar to an earlier proof by Blazewicz, Dell'Olmo, Drozdowski, and Speranza [1992].

3-Partition

Given an integer b and a multiset $N = \{a_1, \ldots, a_{3n}\}$ of $3n$ positive integers with $b/4 < a_j < b/2$ and $\Sigma_{j=1}^{n} a_j = nb$, is there a partition of N into n mutually disjoint subsets $N_1, \ldots, N_n$ such that the elements in N_j add up to b, for $j = 1, \ldots, n$?

number	allocation	processing time
n	M_2 & M_3 (type A)	p_A
n	M_1 & M_3 (type B)	p_B
n	M_1 & M_2 (type C)	p_C
1	M_1	p_A+b
$n-1$	M_1	p_A+b+p_z
n	M_1	p_y
$n-1$	M_2	p_z
n	M_2	p_B+b+p_y
1	M_3	p_C+p_y
$n-1$	M_3	$p_C+p_y+p_z$

Table 5.1. Separation tasks for $P3 \mid fix \mid C_{\max}$.

Given an instance of 3-Partition, we construct the following instance of $P3 \mid fix \mid C_{\max}$. There are $3n$ single-processor tasks J_j that correspond to the elements of 3-Partition; these tasks have to be executed by M_3 and their processing time is equal to a_j, for $j = 1, \ldots, 3n$. In addition, there are $3n$ biprocessor *separation tasks* and $5n-1$ single-processor *separation tasks*; there processing times and processing requirements are defined in Table 5.1. Here we define

$$p_B = (n+1)b,$$

$$p_y = (n+1)(b+p_B),$$

$$p_z=(n+1)(b+p_B+p_y),$$

$$p_C=(n+1)(b+p_B+p_y+p_z),$$

$$p_A=(n+1)(b+p_B+p_y+p_z+p_C).$$

Note that each processor has a processing load equal to $\gamma=n\,(p_A+p_B+p_C+p_y+p_z+b)-p_z$, which implies that γ is a lower bound on the makespan of any schedule. We will show that 3-Partition has an affirmative answer if and only if there exists a schedule with makespan at most γ for the corresponding instance of $P3\,|\,fix\,|\,C_{\max}$.

If 3-Partition has an affirmative answer, then a schedule with makespan $C_{\max}\le\gamma$ exists, as is illustrated in Figure 5.4.

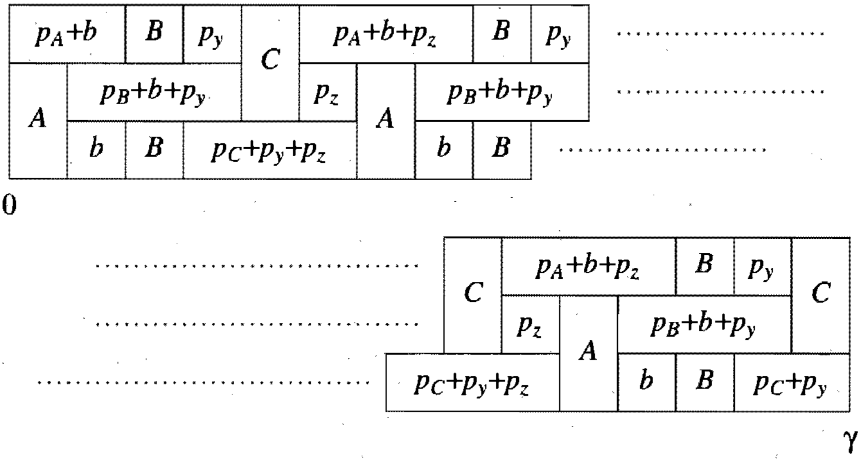


Figure 5.4. Structure for $P3\,|\,fix\,|\,C_{\max}$: $ABCAB\cdots CABC$.

Conversely, suppose that $C^*_{\max}\le\gamma$. Note that a schedule with makespan γ has no idle time. To avoid idle time at the start of a biprocessor task, both processors on which it has to be executed must have equal load. Hence, at the start of a task of type A, there exist a set $T\subset N$ and integers $\kappa_1,\kappa_2,\kappa_4,\kappa_5,\kappa_6,\kappa_7\in\{0,\ldots,n\}$ $\kappa_3\in\{0,\ldots,n-1\}$, and $\kappa_8\in\{0,1\}$, such that

$$\begin{aligned}(2)\quad & \kappa_1p_A+\kappa_2p_C+\kappa_3p_z+\kappa_4(p_B+b+p_y)=\\ & \kappa_5p_A+\kappa_6p_B+\kappa_7(p_C+p_y+p_z)+\kappa_8(p_C+p_y)+\Sigma_{j\in T}p_j.\end{aligned}$$

Due to the choice of the processing times of the separation tasks, we draw the following conclusions:

- the sum $\Sigma_{j\in T}\,p_j$ is a multiple of b, since p_A, p_B, p_C, p_y, and p_z are multiples of b,

- $\Sigma_{j \in T} p_j = \kappa_4 b$, since all other terms are multiples of $(n+1)b$,
- $\kappa_1 = \kappa_5$, since $p_A > n(p_C + p_z + p_y + p_B + b)$,
- $\kappa_2 = \kappa_7 + \kappa_8$, since $p_C > n(p_z + p_y + p_B + b)$,
- $\kappa_3 = \kappa_7$, since $p_z > n(p_y + p_B + b)$,
- $\kappa_4 = \kappa_7 + \kappa_8$, since $p_y > n(p_B + b)$, and
- $\kappa_4 = \kappa_6$, since $\Sigma_{j \in T} p_j = \kappa_4 b$.

It follows that

(3) $\kappa_1 = \kappa_5, \ \kappa_2 = \kappa_4 = \kappa_6 = \kappa_7 + \kappa_8, \ \kappa_3 = \kappa_7, \ \kappa_4 b = \Sigma_{j \in T} p_j.$

Analogous computations lead to similar relations that should hold at the start of a task of type *B* and *C*, respectively.

We will make extensive use of these relations in our analysis of the form that a schedule with makespan γ should have. Using an interchange argument, we see that there exists an optimal schedule in which a biprocessor task starts at time 0. We analyze the case that the first biprocessor task is of type *B* and that the next biprocessor task of another type is of type *A*; this case will be denoted as case *BA*. Hence, we have that no tasks of type *A* and *C* and at least one task of type *B* have been executed at the start of the first task of type *A*: $\kappa_1 = \kappa_2 = \kappa_5 = 0$ and $\kappa_6 \geq 1$. Expression (3), however, decrees that $\kappa_2 = \kappa_5$, which yields a contradiction. Therefore, case *BA* cannot occur. A continued application of this argument shows that any schedule with makespan γ should have the form as displayed in Figure 5.4, or its mirror image. A schedule with this structure determines *n* separate periods of length *b* on processor M_3, in which the remaining single-processor tasks have to be scheduled. These tasks correspond to the $3n$ elements of 3-Partition. We conclude that, if a schedule of length γ exists, then a solution to 3-Partition is obtained by taking the partition of *N* as defined by the schedule. We conclude that $P3 \mid fix \mid C_{\max}$ is NP-hard in the strong sense. ☐

5.1.3. *Unit execution times, release dates, and precedence constraints*

In this section, we show that the $Pm \mid fix, p_j = 1 \mid C_{\max}$ problem is solvable in polynomial time by providing an integer linear programming formulation with a fixed number of variables; a problem that allows such a formulation is solvable in polynomial time [H.W. Lenstra, Jr., 1983]. A similar approach is given by Blazewicz, Drabowski and Weglarz [1986].

Consider an arbitrary instance of the problem. There are at most $M = 2^m - 1$ tasks of a different type; let these types be numbered $1, \ldots, M$. We can denote the instance by a vector $b = (b_1, \ldots, b_M)$ in which component b_j indicates the number of tasks of type *j*. A collection of tasks is called *compatible* if all these tasks can be executed in parallel; hence, a compatible collection of tasks contains at most one task of each type. A compatible collection is denoted by a $\{0,1\}$-vector c of length M with $c_j=1$ if the collection contains a task of type j and zero otherwise. There are at most $K=2^M-1$ different compatible collections; this number is fixed, as M is fixed. Let the collections be numbered $1,\ldots,K$; let the vectors indicating the collections be denoted by $c_1,\ldots,c_K$. The problem of finding a schedule of minimal length is then equivalent to the problem of finding a decomposition of this instance into a minimum number of compatible collections. Formally, we wish to minimize $\Sigma_{j=1}^{K}x_j$ subject to $\Sigma_{j=1}^{K}c_jx_j=b$, x_j integer and nonnegative. As the number of variables in this integer linear programming formulation is fixed, we have proven the following theorem.

Theorem 5.4. *The* $Pm\,|\,fix,p_j=1\,|\,C_{\max}$ *problem is solvable in polynomial time.* ☐

If the number of processors is specified as part of the problem type, implying that this number is no longer fixed, then things get worse from a complexity point of view. This is stated in the following theorem.

Theorem 5.5. *The problem of deciding whether an instance of* $P\,|\,fix,p_j=1\,|\,C_{\max}$ *has a schedule of length at most* 3 *is* NP-*hard in the strong sense.*

Proof. The proof is based upon a reduction from the strongly NP-complete problem Graph 3-Colorability. A similar approach is used by Blazewicz, Lenstra, and Rinnooy Kan [1983].

Graph 3-Colorability
Given a graph $G=(V,E)$, does there exist a 3-coloring, that is, a function $f:V\rightarrow\{1,2,3\}$ such that $f(u)\neq f(v)$ whenever $\{u,v\}\in E$?

Given an arbitrary instance $G=(V,E)$ of Graph 3-Colorability, we construct the following instance of $P\,|\,fix,p_j=1\,|\,C_{\max}$. There are $|V|$ tasks $J_1,\ldots,J_{|V|}$ and $|E|$ processors $M_1,\ldots,M_{|E|}$. A task J_u has to be processed by M_e if $u\in e$. We claim that there exists a 3-coloring for G if and only if there exists a schedule of length at most 3. Suppose that a 3-coloring of G exists. Since no two nodes u and v with the same color are adjacent, the corresponding tasks J_u and J_v require different processors. Hence, all tasks that correspond to identically colored nodes can be executed in parallel. This generates a schedule with length no more than 3. Conversely, in a schedule with length at most 3 we have that the nodes corresponding to tasks scheduled in time period t $(t=1,2,3)$ are independent; therefore, these nodes can be given the same color. This leads to a 3-coloring of G. Thus, $P\,|\,fix\,,p_j=1\,|\,C_{\max}$ is NP-hard in the strong sense. □

Corollary 5.1. *For* $P\,|\,fix,p_j=1\,|\,C_{\max}$, *there exists no polynomial approximation algorithm with performance ratio smaller than* 4/3, *unless* P = NP. □

The introduction of precedence constraints leaves little hope of finding polynomial-time optimization algorithms. Even the two-processor problem with unit execution times and the simplest possible precedence relation structure, a collection of vertex-disjoint chains, is already NP-hard in the strong sense.

Theorem 5.6. *The* $P2\,|\,chain,fix,p_j=1\,|\,C_{\max}$ *problem is* NP-*hard in the strong sense.*

Proof. The proof is based upon a reduction from 3-Partition and follows an approach of Blazewicz, Lenstra, and Rinnooy Kan [1983]. Given an arbitrary instance of 3-Partition, we construct the following instance of $P2\,|\,chain,fix,p_j=1\,|\,C_{\max}$. Each element a_j corresponds to a chain K_j of $2a_j$ tasks; the first part consists of a_j tasks that have to be executed by M_1 and the second part also consists of a_j tasks that have to be executed by M_2. In addition, there is a chain L of $2nb$ tasks; groups of b tasks have to be alternately executed by M_2 and M_1.

Suppose that there exists a partition of N into $N_1,\ldots,N_n$ that yields an affirmative answer to 3-Partition. A feasible schedule with makespan no more than $2nb$ is then obtained as follows. The chain L is scheduled according to its requirements; the execution of L is completed at time $2nb$. Now M_1 and M_2 are idle in the intervals $[2(i-1)b,(2i-1)b]$ and $[(2i-1)b,2ib]$ $(i=1,\ldots,n)$, respectively. For each $i\in\{1,\ldots,n\}$ it is now possible to schedule the three chains corresponding to the elements of N_i in $[2(i-1)b,(2i-1)b]$ and $[(2i-1)b,2ib]$.

Conversely, suppose that there exists a feasible schedule with makespan no more than $2nb$. It is clear that this schedule contains no idle time. Let N_i be the index set of the chains K_j that are completed in the interval $[(2i-1)b,2ib]$. It is impossible that $\Sigma_{j\in N_1}a_j>b$ due to the definition of N_1. The case $\Sigma_{j\in N_1}a_j<b$

cannot occur either, since this would lead to idle time in $[b, 2b]$. Therefore, we must have that $\Sigma_{j\in N_1} a_j = b$. Through a repetition of this argument, it can be easily proven that $N_1, \ldots, N_n$ constitutes a solution to 3-Partition. □

The introduction of release dates has a similar inconvenient effect on the computational complexity.

Theorem 5.7. *The* $P2\,|\,fix, r_j\,|\,C_{\max}$ *problem is* NP-*hard in the strong sense.*

Proof. The proof is again based upon a reduction from 3-Partition. Given an arbitrary instance of 3-Partition, we construct the following instance of $P2\,|\,fix, r_j\,|\,C_{\max}$. For each element a_j, we define a task J_j with $p_j = a_j$ and $r_j = 0$ that has to be executed by M_1. Furthermore, there are n tasks K_j with processing time b and release date $r_j = (j-1)(b+\varepsilon)$, for $j = 1, \ldots, n$ and ε sufficiently small; these tasks have to be executed by M_2. Finally, there are $n-1$ biprocessor tasks L_j with processing time ε and release date $r_j = jb + (j-1)\varepsilon$, for $l = 1, \ldots, n-1$. It is easy to see that 3-Partition has an affirmative answer if and only if there exists a feasible schedule for $P2\,|\,fix, r_j\,|\,C_{\max}$ with $C_{\max} \leq nb + (n-1)\varepsilon$. □

Consider the case $Pm\,|\,fix, r_j, p_j = 1\,|\,C_{\max}$ where the number s of distinct release dates is fixed. Analogously to our analysis of $Pm\,|\,fix, p_j = 1\,|\,C_{\max}$, we can transform any instance of $Pm\,|\,fix, r_j, p_j = 1\,|\,C_{\max}$ into an integer linear programming problem with a fixed number of variables. We have proven the following theorem.

Theorem 5.8. *The* $Pm\,|\,fix, r_j, p_j = 1\,|\,C_{\max}$ *problem with a fixed number of distinct release dates is solvable in polynomial time.* □

5.2. Sum of completion times

In this section, we investigate the computational complexity of our type of scheduling problems when we wish to minimize total completion time. Our main result is establishing NP-hardness in the ordinary sense for $P2\,|\,fix\,|\,\Sigma C_j$. The question whether this problem is solvable in pseudopolynomial time or NP-hard in the strong sense still has to be resolved. The weighted version, however, is NP-hard in the strong sense. We start with an easy observation. Given an instance, let the maximum processing time be denoted by $p_{\max} = \max_j p_j$.

Proposition 5.1. *There is an optimal schedule for $P \mid fix \mid \Sigma C_j$ in which the tasks that require all processors for execution during $p_{\max}$ time are scheduled last, if they exist.* □

Proof. Consider a schedule σ for $P \mid fix \mid \Sigma C_j$ in which the task J that needs all processors for execution during time $p_{\max}$ is not scheduled last. The interchange illustrated in Figure 5.5 generates a schedule σ^* with $\Sigma C_j(\sigma^*) \le \Sigma C_j(\sigma) + p(B) - \lceil p(B)/p_{\max} \rceil p_{\max} \le \Sigma C_j(\sigma)$, where $p(B) = \Sigma_{J_j \in B} p_j$. □

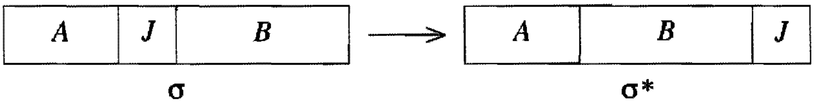


Figure 5.5. The interchange.

5.2.1. NP-*hardness for the* 2-*processor problem.*

Theorem 5.9. *The $P2 \mid fix \mid \Sigma C_j$ problem is* NP-*hard in the ordinary sense.*

Proof. Our proof is based upon a reduction from the NP-complete problem Even-Odd Partition.

Even-Odd Partition
Given a multiset of $2n$ positive integers $A = \{a_1, \ldots, a_{2n}\}$ such that $a_i < a_{i+1}$ $(i = 1, \ldots, 2n-1)$, is there a partition of N into two disjoint subsets A_1 and A_2 with equal sum $b = \Sigma_{i=1}^{2n} a_i/2$ and such that A_1 contains exactly one of $\{a_{2i-1}, a_{2i}\}$, for each $i = 1, \ldots, n$?

Given an instance of Even-Odd Partition, define $p = (n^2+1)b$, $q = n^2(n^2+1)(n+1)p$, and $r = \Sigma_{j=1}^{n}(n+j-1)(a_{2j-1}+a_{2j}) + n^2(n+1)b$. We construct the following instance of $P2 \mid fix \mid \Sigma C_j$. Each element a_j corresponds to a partition task J_j with processing time $p_j = nb + a_j$ that has to be executed by M_1. In addition, we define n^2+3 extra tasks. There are n^2 identical tasks Q_i $(i = 1, \ldots, n^2)$ with processing time $2p(n+1)$ that have to be executed by M_2, a task K with processing time p that has to be executed by M_2, a biprocessor task L with processing time p, and a task P with processing time $2p(n+1)$ that has to be executed by M_1. We will show that Even-Odd Partition is answered affirmatively if and only if there exists a schedule for the corresponding instance of $P2 \mid fix \mid \Sigma C_j$ with total completion time no more than the threshold

$$y=(2n^2+4n+8)p+q+r.$$

Suppose that there exist subsets A_1 and A_2 that lead to an affirmative answer to Even-Odd Partition. Then there exists a schedule σ^* with total completion time no more than y, as is illustrated in Figure 5.6: the completion times of the extra tasks add up to $(2n^2+2n+8)p+q$, the sum of the completion times of the partition tasks is equal to $2np+r$.

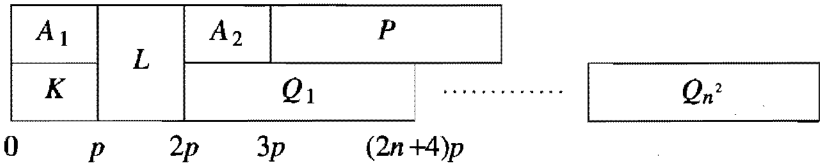


Figure 5.6. The schedule σ^* with partition sets A_1 and A_2.

Conversely, suppose that there exists a schedule σ with total completion time no more than y. We first show that the extra tasks in σ must be scheduled according to the pattern of Figure 5.6.

A straightforward computation shows that task P and the Q-tasks must be completed after all other tasks in σ. Suppose that task L precedes task K, and that m partition tasks are completed before L starts. Note that $m \leq n$; otherwise, task K could be scheduled parallel to the m partition tasks, without increasing the completion time of any other job. If we compare this schedule with σ^*, then task L turns out to be the only task with smaller completion time; this gain is more than offset by the increase of completion time of task K. Hence, in order to satisfy the threshold, the extra jobs must be scheduled according to the pattern of Figure 5.6.

We now show that, if $\Sigma C_j(\sigma) \leq y$, then the partition tasks must constitute an affirmative answer to Even-Odd Partition. First, suppose that the partition tasks before L in σ have total processing time smaller than p, implying that at most n partition tasks are scheduled before L. Then the total completion time of the partition jobs amounts to at least $r+2np$, the total completion time of the Q-tasks, task K, and task L is equal to the total completion time of these tasks in σ^*, and the completion time of task P is greater than $3p+(2n+2)p$, implying that the threshold is exceeded. Hence, the total processing time of the partition tasks before task L amounts to at least p.

Now suppose that m partition tasks with total processing time $p+x$ precede task L. Comparing σ with σ^* shows that the total completion time of the extra jobs in σ is $x(n^2+1)$ greater and that the difference in total completion time of the partition tasks is no more than $2p(n-m)+x(2n-m)$ in favor of σ. If $m=n$, then the difference in total completion time between σ^* and σ is at least equal to $x(n^2+1)-xn$ in favor of σ^*; $x>0$ then clearly implies that the threshold will be exceeded. In case $m>n$, we wish to show that $x(n^2+1)>2p(n-m)+x(2n-m)$, which boils down to showing that $x(n^2+1-2n+m)>2p(n-m)$. As the left-hand-side of the inequality is positive and the right-hand-side negative, we have that the case $m>n$ leads to an excess of the threshold. Hence, exactly n partition tasks with total processing time equal to p must precede task L in σ. The total completion time of the partition tasks is equal to $2np+n(p_{[1]1}+p_{[1]2})+\cdots+(p_{[n]1}+p_{[n]2})$, where $p_{[i]1}$ and $p_{[i]2}$ denote the processing time of the $[i]$th partition task before L and after L, respectively. It is easy to see that the threshold can only be met if $\{p_{[i]1},p_{[i]2}\}=\{p_{2i-1},p_{2i}\}$, for $i=1,\ldots,n$. Define A_1 and A_2 as the set of partition tasks before L and after L in σ, respectively. As the total processing time of the tasks in A_1 amounts to $n^2b+\Sigma_{A_1}a_j=p=(n^2+1)b$, we have that the corresponding subset of partition elements has sum equal to b. Furthermore, A_1 contains exactly one element from every pair $\{a_{2i-1},a_{2i}\}$; hence, the subsets A_1 and A_2 lead an affirmative answer to Even-Odd Partition. □

Theorem 5.10. *The* $P2|fix|\Sigma w_jC_j$ *problem is* NP-*hard in the strong sense.*

Proof. The proof is based upon a reduction from 3-Partition. Given an arbitrary instance of 3-Partition, we construct the following instance of $P2|fix|\Sigma w_jC_j$. Each element a_j corresponds to a task J_j with processing time a_j and unit weight that has to be executed by M_1. In addition, there are n tasks K_j with processing time b and weight $2(j+\alpha-1)\beta$ that have to be executed by M_2, and n_L biprocessor tasks L_j with processing time b and weight $(2j-1)\beta$, where $\alpha=3n(2n-1)$, $\beta=\alpha b$, and $n_L=\alpha+n-1$.

Suppose that there exists a partition of N into $N_1,\ldots,N_n$ that yields an affirmative answer to 3-Partition. A feasible schedule with sum of weighted completion times no more than

$$y=\beta+\Sigma_{k=1}^{n}w_k(2(n-k)+1)b+\Sigma_{l=1}^{\alpha}w_l(2n+\alpha-l)b+\Sigma_{l=\alpha+1}^{n_L}w_l(2n-2(l-\alpha))b$$

is then obtained by scheduling the tasks as illustrated in Figure 5.7.

Conversely, suppose that there exists a schedule σ with sum of weighted completion times no more than y. Straightforward computations show that the K-tasks and the L-tasks have to be scheduled as indicated in Figure 5.7 and that the tasks J_j have to be scheduled in the time slots parallel to the K-tasks. Let N_j denote the set of J-tasks that are scheduled parallel to K_j; the sets $N_1,\ldots,N_n$ constitute a solution to 3-Partition. □

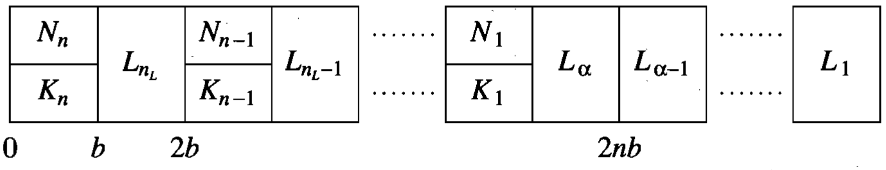


Figure 5.7. A schedule for $P2\,|\,fix\,|\,\Sigma w_jC_j$ with $\Sigma w_jC_j \le y$.

5.2.2. *Strong NP-hardness for the general* 3-*processor problem*

Theorem 5.11. *The* $P3\,|\,fix\,|\,\Sigma C_j$ *problem is* NP-*hard in the strong sense.*

Proof. The proof is based upon a reduction from the decision version of the $P3\,|\,fix\,|\,C_{\max}$ problem, which was shown to be NP-complete in Section 2.2. The decision version of $P3\,|\,fix\,|\,C_{\max}$ is defined as the following question: given an instance of $P3\,|\,fix\,|\,C_{\max}$ and a threshold b, does there exist a schedule σ with makespan no more than b?

Given an arbitrary instance of $P3\,|\,fix\,|\,C_{\max}$ and a threshold b, we construct the decision instance of $P3\,|\,fix\,|\,\Sigma C_j$ by adding $nb+1$ identical triprocessor tasks K_j with processing time $p_{\max}$. The corresponding threshold is equal to $y = nb + \Sigma_{k=1}^{nb+1}(b + kp_{\max})$.

Application of Proposition 5.1 shows that there is an optimal schedule with the K-tasks executed last. The number of K-tasks is such that the threshold will be exceeded if the first K-task starts later than b. Hence, the decision variant of $P3\,|\,fix\,|\,\Sigma C_j$ has an affirmative answer if and only if the decision variant of $P3\,|\,fix\,|\,C_{\max}$ has an affirmative answer.

Note that, the number of tasks needed in our reduction is pseudopolynomially bounded. We conclude that $P3\,|\,fix\,|\,\Sigma C_j$ is NP-hard in the strong sense.
□

5.2.3. *Unit execution times and precedence constraints*

In this section, we address the complexity of minimizing total completion time in case of unit processing times. We show that $P\,|\,fix, p_j = 1\,|\,\Sigma C_j$ is NP-hard in the strong sense; the complexity of this problem with a fixed number of processors is still open.

Theorem 5.12. *The* $P\,|\,fix, p_j = 1\,|\,\Sigma C_j$ *problem is* NP-*hard in the strong sense.*

Proof. The proof of this theorem is based upon a reduction from $P \mid fix, p_j = 1 \mid C_{\max}$; it proceeds along the same lines as the proof of the previous theorem. Given an instance of $P \mid fix, p_j = 1 \mid C_{\max}$, we add w tasks that require all processors for execution; application of Proposition 5.1 shows that these tasks can be assumed to be executed after all other tasks. By choosing w suitably large, we obtain the situation that the threshold of $P \mid fix, p_j = 1 \mid \Sigma C_j$ is exceeded if and only if the threshold of $P \mid fix, p_j = 1 \mid C_{\max}$ is exceeded. As the decision variant of $P \mid fix, p_j = 1 \mid C_{\max}$ is NP-complete in the strong sense and as w is polynomially bounded, we conclude that $P \mid fix, p_j = 1 \mid \Sigma C_j$ is NP-hard in the strong sense. □

As could be expected, the addition of precedence constraints does not have a positive effect on the computational complexity. We show that even the mildest non-trivial problem of this type, with two processors and chain-type precedence constraints, is NP-hard in the strong sense.

Theorem 5.13. *The* $P2 \mid chain, fix, p_j = 1 \mid \Sigma C_j$ *problem is* NP-*hard in the strong sense.*

Proof. The proof is based upon the same reduction as used in the proof of Theorem 5.6, only the threshold differs. As the number of tasks is equal to $2nb$, and as each task has unit processing time, an obvious lower bound on the total completion time is equal to $y = 2nb(2nb+1)$; this bound can only be attained by a schedule without idle time in which both processors execute nb tasks. Hence, there exists a schedule with total completion time no more than y if and only if there exists a schedule with makespan no more than b. We conclude that $P2 \mid chain, fix, p_j = 1 \mid \Sigma C_j$ is NP-hard in the strong sense. □

6. Chains of length 1, or the two-stage flow shop

In the previous chapter we dealt with prespecified processor allocations. In general, an instance may consist of tasks that still have to be allocated as well as tasks with a prespecified allocation. An example of such a situation occurs in a two-stage pipeline, where the first stage consists of two independent identical processors and the second stage is built up of a single processor. This model is described more precisely as follows.

A set of $2n$ tasks has to be processed by three processors. Each of the first n tasks J_j is to be executed by one of the first two processors M_1 and M_2. The remaining n tasks K_j belong to a single family, which is entirely executed by the third processor M_3. The tasks are related through a precedence relation that consists of a collection of chains of length 1; J_j precedes K_j for $j=1,\ldots,n$. Preemption of tasks is allowed, which means that the processing of a task may be interrupted and resumed at the same time on a different processor or at a different time on the same or a different processor. However, each task can be active on only one processor at a time. The objective is to minimize makespan.

The problem described above belongs to the class $P3 \mid chain,fam,set,pmtn \mid C_{\max}$. It is a special case of the class of *two-stage flow shop* scheduling problem, where each stage is executed by a number of processors. Related nonpreemptive problem types have been addressed by several researchers. Salvador [1970] investigates the problem of finding a permutation schedule of minimal length in an m-stage no-wait flow shop environment with an arbitrary number of parallel processors in every stage. Buten and Shen [1973] propose two heuristics for the two-stage problem with an arbitrary number of parallel processors in each stage and analyze their worst-case behavior, whereas Arthanri [1974] designs a branch-and-bound algorithm for the same problem.

We consider the problem from a complexity point of view. The problem can be seen as an immediate generalization of the preemptive version of the classical two-stage flow shop problem, which has a single processor at each stage, and the problem of minimizing makespan on two parallel processors allowing for preemption, $P2 \mid pmtn \mid C_{\max}$. For the classical two-stage flow shop problem, Johnson [1954] derived an $O(n \log n)$ time algorithm; it also constructs optimal schedules for the preemptive version, since preemption does not help. Mc-Naughton's [1959] wrap-around rule solves $P \mid pmtn \mid C_{\max}$ in $O(n)$ time. However $P2 \mid \mid C_{\max}$ is NP-hard, so that the nonpreemptive problem $P3 \mid chain,fam,set \mid C_{\max}$ is NP-hard as well. In this chapter, we show that the preemptive problem described above is NP-hard in the strong sense.

The organization of this brief chapter is as follows. We first show that the problem $P3|chain,fam,set,pmtn\ |C_{\max}$ is NP-hard in the ordinary sense in Section 6.1. In Section 6.2, we use the basic idea of that proof to obtain the more general result that the problem is NP-hard in the strong sense.

6.1. **Ordinary NP-hardness**

In this section, we show ordinary NP-hardness of the problem $P3|chain,fam,set,pmtn\ |C_{\max}$ by a reduction from Partition. This reduction will provide the insight needed to establish NP-hardness in the strong sense.

Partition

Given a multiset $N=\{a_1,\ldots,a_n\}$ of n integers, is it possible to partition N into two disjoint subsets that have equal sum $b=\Sigma_{j\in N}a_j/2$?

Given an instance of Partition, construct the following instance of $P3|chain,fam,set,pmtn\ |C_{\max}$ with a precedence relation consisting of chains of length 1. For each $j\in N$ we define two *partition* tasks J_j and K_j; J_j has processing time αa_j and K_j has processing time a_j, where $\alpha>1$. In addition, we define eight *separation* tasks, which have to create time slots of equal length for the execution of the partition tasks. The processing times of the separation tasks are given in Table 6.1. The precedence constraints are such that $J_j \rightarrow K_j$, for $j=1,\ldots,n+4$. The tasks of type J can be performed by processors M_1 and M_2; the tasks of type K have to be performed by M_3. Note that $2(\alpha b+b)$ is a lower bound on the makespan and that a schedule with $C_{\max}=2(\alpha b+b)$ contains no processor idle time.

j	$n+1$	$n+2$	$n+3$	$n+4$
J_j	0	$\alpha b+b$	$\alpha b+2b$	b
K_j	αb	αb	0	0

Table 6.1. Processing times of the separation tasks.

Suppose S is a subset of N with sum equal to b. We construct the following schedule of length $2(\alpha b+b)$. The processing of the set of J-tasks corresponding to the elements of S starts at time 0 on M_1. These tasks are executed consecutively and their execution takes αb time. The processing of the set of K-tasks corresponding to S starts at time αb; it is preceded by the execution of task K_{n+1}. Task J_{n+2} starts at time 0 and is executed without interruption by M_2. Its successor K_{n+2} is processed without any delay. The execution of the set of remaining partition tasks of type J starts at time $\alpha b+b$ and the execution of the set of their successors starts at $2\alpha b+b$ following K_{n+2}. Finally, the remaining two separation tasks are added to complete the schedule; for an illustration see Figure 6.1.

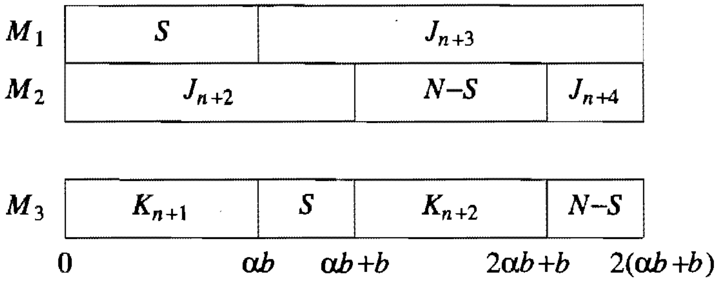


Figure 6.1. A schedule with partition sets S and $N-S$.

Conversely, suppose that a schedule σ exists with makespan $2(\alpha b+b)$. Without loss of generality, we may assume that the K-type tasks are executed without preemption in order of completion time of their predecessors. Since σ contains no processor idle time and J_{n+1} is the only task of type J with processing time equal to 0, task K_{n+1} completes at time αb. For similar reasons, tasks J_{n+3} and J_{n+4} complete at time $2(\alpha b+b)$. Let J_{n+2} complete at time $\alpha b+b+\Delta$, with $\Delta \geq 0$. Processor M_3 should perform partition tasks of type K from time αb to time $\alpha b+b+\Delta$. Their predecessors have to be performed by M_1 and M_2 before J_{n+2} and J_{n+3} start; the execution of these partition tasks takes at least time $\alpha(b+\Delta)$. The amount of time available is $\alpha b+\Delta$. From $\alpha(b+\Delta) \leq \alpha b+\Delta$ and $\alpha > 1$, it follows that $\Delta = 0$ and the total processing time of the partition tasks of type J that are processed before J_{n+3} is started is equal to αb. Hence, such a schedule σ gives a certificate of the affirmative answer to Partition.

By now, we have proven the following theorem.

Theorem 6.1. *The problem* $P3 \mid chain, fam, set, pmtn \mid C_{\max}$ *is* NP-*hard, even with a precedence relation consisting of a collection of chains of length* 1.

6.2. Strong NP-hardness

In this section, the above described problem, $P3 \mid chain, fam, set, pmtn \mid C_{\max}$, is shown to be NP-hard in the strong sense. We use the 3-Partition problem for our reduction.

3-Partition

Given an integer b and a multiset $N = \{a_1, \ldots, a_{3n}\}$ of $3n$ positive integers with $b/4 < a_j < b/2$ and $\Sigma_{j=1}^{3n} a_j = nb$, is there a partition of N into n mutually disjoint subsets $N_1, \ldots, N_n$ such that the elements in N_j add up to b, for $j = 1, \ldots, n$?

j	$6n+1$	$6n+2$	$6n+3$	$\cdots$	$7n$	$7n+1$	$7n+2$
J_j	0	$\alpha b+b$	$\alpha b+2b$	$\cdots$	$\alpha b+2b$	$\alpha b+2b$	b
K_j	αb	αb	αb	$\cdots$	αb	0	0

Table 6.2. Processing times of the separation tasks.

Given an instance of 3-partition, we construct the following instance of $P3 \mid chain, fam, set, pmtn \mid C_{\max}$ with a precedence relation consisting of chains of length 1. As in Section 6.1, for each $j \in N$ we define two *partition* tasks J_j and K_j; J_j has processing time αa_j and K_j has processing time a_j, where $\alpha > 1$. In addition, we define $2(n+2)$ *separation* tasks; their processing times are given in Table 6.2. These tasks have to create time slots. The precedence constraints are such that $J_j \rightarrow K_j$, for $j = 1, \ldots, 7n+2$. The tasks of type J can be performed by processors M_1 and M_2; the tasks of type K have to be performed by M_3. Note that $n(\alpha b + b)$ is a lower bound on the makespan and that a schedule with $C_{\max} = n(\alpha b + b)$ contains no processor idle time.

Proposition 6.1. *If 3-partition has an affirmative answer, then the instance defined above has a schedule of length* $n(\alpha b + b)$.

Proof. Let $N_1, \ldots, N_n$ constitute a yes-answer for the 3-Partition instance. Consider the schedule given in Figure 6.2. Straightforward computations show that the makespan of this schedule is equal to $n(\alpha b + b)$. □

In order to show the converse implication, that is, that 3-Partition has an affirmative answer if the instance $P3 \mid chain, fam, set, pmtn \mid C_{\max}$ has a schedule of length $n(\alpha b + b)$, we need the following propositions. The proofs follow from the same arguments as used in the proof of Theorem 6.1 and are therefore omitted.

Proposition 6.2. *There exists an optimal schedule such that* J_j *and* K_j *are completed no later than* J_{j+1} *and* K_{j+1}*, respectively, for* $j = 6n+1, \ldots, 7n+2$.
□

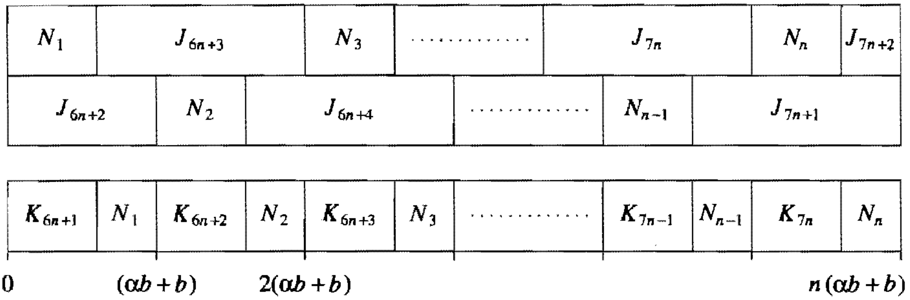


Figure 6.2. A schedule with partition sets $N_1, \ldots, N_n$.

Proposition 6.3. *A schedule of length $n(\alpha b+b)$ satisfies the following properties:*

- *it contains no processor idle time,*
- *J_j completes at time $(j-6n-1)(\alpha b+b)$, for $j=6n+1, \ldots, 7n+1$,*
- *K_j starts immediately after J_j is completed, for $j=6n+1, \ldots, 7n+2$.* □

Proposition 6.4. *Let σ be an optimal schedule of length $n(\alpha b+b)$ that satisfies Proposition 6.2. Then σ certifies that the 3-Partition instance is a yes-instance.* □

Since the above reduction requires polynomial time, we have now proven the following theorem.

Theorem 6.2. *The problem $P3 \mid chain, fam, set, pmtn \mid C_{\max}$ is* NP*-hard in the strong sense, even for a precedence relation consisting of a set of chains of length* 1. □

Corollary 6.1. *The problem $P3 \mid chain, fam, set \mid C_{\max}$ is* NP*-hard in the strong sense, even for a precedence relation consisting of a set of chains of length* 1. □

7. Tosca: methodology

From the analyses of the previous chapters, we may conclude that it is unlikely that fast algorithms exist that solve the scheduling problem in its most general form to optimality. One is confined to take an approximative approach. In the remaining chapters of this thesis we describe such an approach. We introduce Tosca, our tunable off-line scheduling algorithm, which produces approximate answers to instances of a special case of the problem type $P\,|\,prec,com,fam,set\,|\,C_{\max}$ described in Section 7.1. Tosca has been developed as a tool to support the scheduling of parallel programs on distributed memory architectures.

Tosca is an algorithm that tries to find a reasonable solution in a reasonable amount of time by *bounded enumeration*, as will be described in Section 7.2. Only a part of the solution space is taken into consideration, and the user is given the facility to determine the size of this part.

Tosca is *tunable* in the sense that it enables the user to control the speed of the solution method and the quality of the schedules produced. First of all, the size of the part of the solution space considered influences both quality and speed. Moreover, the user has to define priority rules for the selection of tasks and processors, a lower bound rule for truncating the enumeration process, as well as an evaluation rule for the evaluation of partial schedules. A number of predefined rules are incorporated in Tosca, but the user has the opportunity to define new rules, as described in Section 7.3.

Lower bounds on the makespan of an optimal schedule also provide the user with a measure of the quality of the schedules produced by Tosca. The lower bounds are based on the total amount of work that has to be done and on the structure of the precedence relation. They are described in Section 7.4.

7.1. The problem type

A collection of m identical parallel processors M_i $(i=1,\ldots,m)$ has to process a set of n tasks J_j $(j=1,\ldots,n)$. The task set is partitioned into a number of *families*. Each task belongs to a single family. Tasks that belong to the same family F have to be executed by the same processors. Each task in F can be performed by any collection of processors of a given family-dependent size s_F, unless F has a nonempty collection of processors specified as part of the problem instance. In the latter case, each task in F is fixed to this prespecified collection of processors. The processing of task J_j takes p_j time. Let F_j denote the family J_j belongs to.

With each task J_j two data sets I_j and O_j are associated, representing the data that this task requires and delivers, respectively. The data dependency between two tasks J_j and J_k is defined as follows. The tasks are independent if

$O_j \cap I_k = O_k \cap I_j = \emptyset$. If $O_j \cap I_k \neq \emptyset$ and $O_k \cap I_j = \emptyset$, then we write $J_j \to J_k$ and require that J_j has been completed before J_k can start. Conversely, if $O_k \cap I_j \neq \emptyset$ and $O_j \cap I_k = \emptyset$, then we write $J_k \to J_j$ and require that J_k has been completed before J_j can start. The case that $O_j \cap I_k \neq \emptyset$ and $O_k \cap I_j \neq \emptyset$ should not occur, since it would represent a loop in the parallel program. Thus, the data dependencies impose a precedence relation on the task set. It is denoted by an acyclic directed graph G with vertex set $\{1, \ldots, n\}$ and an arc (j,k) whenever $J_j \to J_k$. Let $P_j = \{J_k \mid J_k \to J_j\}$ and $Q_j = \{J_k \mid J_j \to J_k\}$ denote the sets of predecessors and successors of J_j, respectively.

Communication delays due to these data dependencies are modelled as follows; see also Section 2.3. Let D be the set containing all information: $D = \cup_{j=1}^{n} O_j$. Each data item $a \in D$ has a given integer *weight* w_a, and the weight of a *data set* $U \subset D$ is defined by $w(U) = \Sigma_{a \in U} w_a$. The communication cost function c specifies the time needed to transmit a data set of a given weight from one processor to another. It is assumed to be of the form $c(x) = c_1 + c_2 \lceil x/c_3 \rceil + c_4 \lceil x/c_5 \rceil^2$ and is therefore defined by the integral constants $c_1, \ldots, c_5$. Interprocessor *communication* occurs when a task J_k needs information from a predecessor J_j and makes use of at least one processor that is not used by J_j. Let M_i be such a processor and let $P(k,i)$ denote the set of tasks scheduled on M_i before and including J_k. Prior to the execution of J_k, the data set $U(i,j,k) = \cup_{l \in Q_j \cap P(k,i)} (O_j \cap I_l)$ has to be transmitted to M_i, since not only J_k but also each successor of J_j that precedes J_k on M_i requires its own data set. The time gap in between the completion of J_j (at time C_j) and the start of J_k (at time S_k) has to allow for the transmission of $U(i,j,k)$. The communication time is given by $c(w(U(i,j,k)))$. For feasibility it is required that $S_k - C_j \geq c(w(U(i,j,k)))$.

A schedule is an allocation of each task J_j to a time interval of length p_j on s_{F_j} processors such that no two of these time intervals on the same processor overlap. A schedule is *feasible* if it meets the requirements concerning the processor environment and the task characteristics as described above. A feasible schedule is *active* if no task can be moved forward in time without delaying some other task or violating the constraints. Tosca aims to minimize the length of a schedule.

7.2. The solution method

An active schedule can be constructed by iteratively selecting the next task to schedule, allocating a collection of processors to it, and starting it as early as possible. We can visualize the various possible choices by an enumeration tree, as shown in Figure 7.1. In Figure 7.1a we present an instance consisting of two processors, four tasks, and four families. The enumeration tree for this instance is given in Figure 7.1b, where the root represents the empty schedule, each O-node represents the selection of a task, and each □-node represents the allocation of a task to a collection of processors. Hence, each □-node corresponds to a *partial schedule* of the tasks selected so far, and the leaves of the enumeration tree correspond to all complete schedules.

The process of bounded enumeration consists of *n stages*. At each stage an available task and a collection of processors for that task is selected. A task is said to be *available* if all of its predecessors have been scheduled. In order to select a task and a processor collection, Tosca generates a subtree of the enumeration tree, as shown by the double boxes and circles in Figure 7.1b. In total, *n* subtrees have to be generated until a complete schedule has been constructed.

The subtree is determined by three *parameters d*, *t*, *u*, two *priority rules*, and a *lower bounding procedure*. The parameter *d* defines the *depth* of the subtree. At each of the *d* levels, Tosca applies the first priority rule to select *t* of the available tasks and, for each chosen task, Tosca applies the second priority rule to select *u* of the processor collections that can execute the task. For each partial schedule that is generated, Tosca computes a lower bound on the optimal makespan of schedules that are extensions of the partial schedule. If this lower bound exceeds a given upper bound, then the branch determined by the partial schedule is eliminated and will not be examined further. The parameters *d*, *t*, *u*, and the lower bounding procedure determine the size of the subtree that is rooted at the current partial schedule.

The leaves of the subtree are partial schedules, which are evaluated according to an *evaluation rule*. Each task-collection pair chosen at the first level determines a branch of the subtree; each branch contains a number of leaves, each with its own value. A task-collection pair that leads to a leaf of minimal value is chosen and a new partial schedule is constructed, as shown by the bold boxes in Figure 7.1b. A new stage has been reached.

As said before, Tosca is tunable in the sense that it allows the user to control the speed of the solution method and the quality of the schedules produced. First, by adjusting the three parameters *d*, *t*, and *u*, the user influences the size of the partial tree that is computed and chooses from a range of possibilities in between list scheduling and complete enumeration. Second, the user has to define two priority rules: one for selecting a task and another for selecting a collection of processors. And finally, the user has to specify a lower bound rule for the bounding procedure and an evaluation rule for partial schedules.

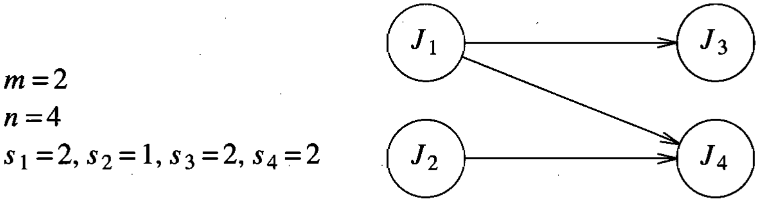


Figure 7.1a. An instance of $P \mid prec, com, fam, set \mid C_{\max}$.

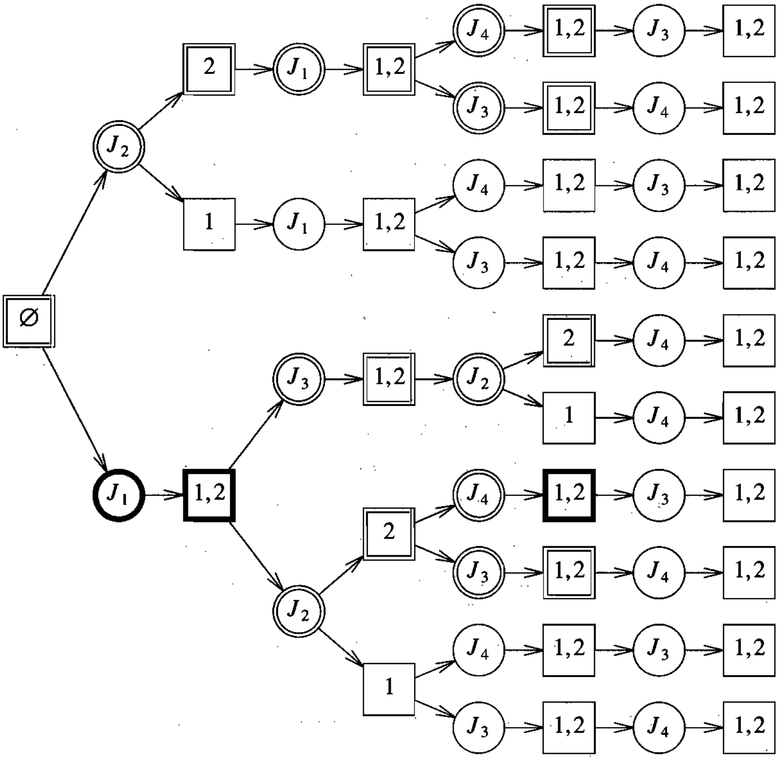


Figure 7.1b. Enumeration tree. Double boxes and circles indicate a subtree for $d=3$, $t=2$, and $u=1$. The bold circle and the bold box adjacent to it indicate a chosen task and processor allocation; the other bold box is the 'best' leaf in the partial tree. The current partial schedule is the empty schedule.

Given a problem, Tosca starts with *priority scheduling*. An evaluation rule and two priority rules are chosen and the parameters d, t and u are set to 1.

Repeatedly a task of high priority among the available tasks is selected and allocated to a processor collection of high priority. The makespan of the resulting schedule is the initial *upper bound* on the optimal makespan.

Lower bounds on the makespan of an optimal schedule are computed at the start of Tosca. These lower bounds provide the user with a measure of the quality of the schedules produced by Tosca. A description of these lower bounds is given in Section 7.4.

	attribute	name	value for J_j
static	number of processors	m	m
	number of tasks	n	n
	processing time	pj	p_j
	size	sF	s_{F_j}
	cardinality of the indata set	#in	$\lvert I_j \rvert$
	cardinality of the outdata set	#out	$\lvert O_j \rvert$
	weight of the indata set	win	$w(I_j)$
	weight of the outdata set	wout	$w(O_j)$
	number of predecessors	#pre	$\lvert P_j \rvert$
	number of successors	#suc	$\lvert Q_j \rvert$
	list index	list	j
	longest preceding path	head	
	longest succeeding path	tail	
	Papadimitriou-Yannakakis	PapYan	
dynamic	first in first out	fifo	
	minimal starting time	start	

Table 7.1. Task attributes.

7.3. Priority rules, evaluation rule, and lower bound rule

A priority rule for the selection of tasks depends on one or more *task attributes*. A task attribute is *static* if it can be computed on the basis of the data that defines a problem instance. It is *dynamic* if schedule information is required for its computation. The various task attributes are listed in Table 7.1.

Given a task J_j, the attribute head is the length of a longest path from a source node to J_j and the attribute tail is the length of a longest path from J_j to a terminal node. The attribute PapYan is a lower bound on the starting time of J_j. It is computed similarly to head, but communication delays are taken into account; cf. Section 7.4. The attribute fifo assigns to a task its rank with respect to the point in time at which it becomes available. Given a partial schedule, the earliest starting time can be computed for any available task; start assigns this value to each available task.

The predefined priority rules that are incorporated in Tosca determine an order in which the tasks are considered with respect to a given attribute. The predefined priority rules are min-sF, max-sF, max-#suc, min-list, max-head, min-tail, min-PapYan, min-fifo, and min-start. If the tasks are considered in order of nondecreasing attribute value, then the prefix min is used in the name of the priority rule. The prefix max is used if the tasks are considered in order of nonincreasing attribute value. The predefined priority rule min-sF, for example, gives high priority to the tasks of small size, whereas the priority rule max-sF gives high priority to the tasks of large size.

priority rule for tasks:
lexico(expression, . . . , expression)
expression
expression:
expression + term
expression – term
max{expression, term}
min{expression, term}
term
term:
term / primary
term * primary
primary
primary:
number
– primary
attribute
(expression)

Figure 7.2. Grammar for priority rules.

The user is given the facility to build a priority rule for the selection of tasks based upon the task attributes. This may involve the construction of new attributes on the basis of old ones. The grammar for a user defined priority rule that is accepted by Tosca is given in Figure 7.2. In words, a user defined priority rule consists of a single expression or the operator lexico applied to several expressions. The basic units of an expression are numbers, task attributes, and the operators *, /, max, min, +, and – (both unary and binary). An expression assigns a value to each task, which is then a new, user defined, task attribute. The tasks are considered in order of nondecreasing value with respect to each expression. Thus, the predefined priority rule **max-tail** leads to the same outcome as the user defined priority rule that consists of the expression –**tail**.

A priority rule for the selection of collections of processors depends on the static and dynamic *processor attributes* given in Table 7.2. As said in Section 7.2, Tosca applies a task priority rule to select a number of available tasks and, for each chosen task J_j, Tosca applies a processor priority rule to select a number of the processor collections that can execute the task. The attributes **#pre** and **data** depend on the available task (J_j) under consideration. The attribute **data** determines the point in time that I_j can be available on M_i, for $i=1,\ldots,m$. The predefined priority rules are all based on dynamic attributes; they are **min-fut**, **min-load**, **max-#pre**, and **min-data**. Again, the user is enabled to construct a new priority rule. The grammar that is accepted by Tosca is the same as the grammar for user defined task-priority rules given in Figure 7.2.

	attribute	name	value for M_i
static	number of processors	m	m
	number of tasks	n	n
dynamic	future	fut	$\max_{J_j\ on\ M_i} C_j$
	load	load	$\Sigma_{J_j\ on\ M_i} p_j$
	number of predecessors	#pre	$\lvert P_j \cap \{J_k \mid J_k\ on\ M_i\} \rvert$
	data available	data	

Table 7.2. Processor attributes.

Static attributes used in a priority rule are computed before the (bounded) generation of the enumeration tree. Dynamic attributes are computed during the generation of the enumeration tree.

An evaluation rule assigns values to partial schedules. The following evaluation rules are available:

- **makespan**: the length of the partial schedule is computed;
- **completion plus tail**: for each task J_j, the length of a longest path starting at J_j is added to its completion time and the maximum of these values is taken;
- **partial plus PapYan**: an adaptation of an algorithm of Papadimitriou and Yannakakis [1990] applied to the partial schedule yields lower bounds on

the completion times of the tasks and the maximum of these values is used; see also Section 7.4.

A lower bound rule is used to eliminate branches of the enumeration tree that will surely not lead to improvements. The user is enabled to chose from the following two rules:

- **makespan**: the length of the partial schedule is used as lower bound;
- **partial plus PapYan**: an adaptation of an algorithm of Papadimitriou and Yannakakis [1990] applied to the partial schedule yields a lower bound on the makespan.

7.4. Lower bounds on the makespan

Tosca computes three lower bounds on the makespan of an optimal schedule. For ease of use, we define $c(j,k)=c(w(O_j \cap I_k))$ and s_j as the (family dependent) number of processors J_j requires for execution. The lower bound *work load* is defined as $\Sigma_{j=1}^{n}(p_j s_j)/m$. The lower bound *longest path* is the length of a longest path with respect to the precedence relation; if $J_1 \rightarrow \cdots \rightarrow J_l$ is a path, then the length of this path is $\Sigma_{1 \le j \le l} p_j + \Sigma_{1 \le j \le l-1, s_j < s_{j+1}} c(j,j+1)$. The third lower bound is based on an algorithm of Papadimitriou and Yannakakis [1990]. A modification of this algorithm leads to a lower bound that dominates *longest path*. It is described below.

First, the procedure computes a lower bound b_j on the starting time of each task J_j. Zero lower bounds are assigned to tasks without predecessors. For any task J_k other than such a source task, consider its predecessors. For each predecessor J_j of J_k, define f_j as $f_j = b_j + p_j + c(j,k)$. In any schedule, a communication delay occurs in between any (predecessor, successor) pair of tasks J_j, J_k satisfying $s_j < s_k$. Define b_k^1 as $b_k^1 = \max\ \{ f_j \mid j \in P_k \text{ and } s_j < s_k \}$.

Sort the predecessor subset $\{ J_j \mid J_j \in P_k \text{ and } s_j \ge s_k \}$ in decreasing order of f, that is, $f_{j_1} \ge \cdots \ge f_{j_q}$. Given an integer y satisfying $f_{j_i} \ge y \ge f_{j_{i+1}}$, consider the following single-processor scheduling problem with release dates on i tasks $(L_1, \ldots, L_i)$. The release date of a task is the point in time at which it becomes available for processing. Task L_l corresponds to task J_{j_l}, that is, it has processing time $p_l = p_{j_l}$ and release date $r_l = b_{j_l}$. Let $C_{\max}(y)$ denote the minimal makespan of this single-processor scheduling problem. Define b_k^2 as the least integer y such that $y \ge C_{\max}(y)$.

The lower bound b_k on the starting time of J_k is now defined as the maximum of b_k^1 and b_k^2. The lower bound *PapYan* is given by $\max_j \{ b_j + p_j \}$.

Given a feasible schedule, let J_k be scheduled at time S_k and let all of its predecessors be scheduled at or after their lower bounds. Clearly, for feasibility it is required that $S_k \geq b_k^1$. Each predecessor J_j that satisfies $s_j \geq s_k$ and $f_j > S_k$, has to be executed by (at least) the same processors that execute J_k to prevent communication delays that would occur otherwise. Therefore, the corresponding single-processor scheduling problem mentioned above has makespan $C_{\max}^* \leq S_k$. Since b_k^2 is the least integer satisfying such constraints, we may conclude that $b_k \leq S_k$. It follows that *PapYan* is indeed a lower bound on the schedule length.

Due to the analyses of Papadimitriou and Yannakakis [1990] and Colin and Chrétienne [1991], one may expect the lower bound *PapYan* to be tighter for problem instances with a relatively large number of processors, and for problems where the communication delays are small with respect to the processing times.

8. Tosca: implementation

The methodology of Tosca as described in the previous chapter has been implemented using the computer language GNU C++. The overall organization of Tosca and the results of experiments with Tosca are reported in this chapter. Together, Sections 7.3, 8.1, and 8.2 form a manual for the use of Tosca.

At the start of Tosca, a problem instance is read from a file that is specified by the user. The user can also specify a file to save Tosca's results, such as a schedule and a parameter setting with a corresponding makespan. The input and output specification is given in Section 8.1.

We developed a user interface that supports the presentation of problem instances and solutions. The man-machine interaction is menu driven, in such a way that at any moment all feasible commands are visible. The user interface and the functional properties of Tosca are described in Section 8.2.

Tosca has been tested on four classes of problem instances. The first class contains instances where the precedence relation has a *layered* structure, the instances of the second class have a *series parallel* precedence relation, and the third class consists of instances with *arbitrary* precedence relations. With respect to the fourth class, two precedence relations from *practice* were at our disposal; we generated data sets, processing times, and task sizes to obtain new instances. The four problem generators are given in Section 8.3.

In Section 8.4 the results of the tests are tabulated and analyzed.

<table>
<tr><td>name</td><td>|</td><td>m</td><td>|</td><td>n</td><td>|</td><td>$w_1,\dots,w_{|D|}$</td><td>|</td></tr>
<tr><td>p_1</td><td>|</td><td>$i_1^1,\dots,i_1^{|I_1|}$</td><td>|</td><td>$o_1^1,\dots,o_1^{|O_1|}$</td><td>|</td><td>F_1</td><td>|</td></tr>
<tr><td>...</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>p_n</td><td>|</td><td>$i_n^1,\dots,i_n^{|I_n|}$</td><td>|</td><td>$o_n^1,\dots,o_n^{|O_n|}$</td><td>|</td><td>F_n</td><td>|</td></tr>
<tr><td>s_1</td><td>|</td><td>$M_1^1,\dots,M_1^{s_1}$</td><td>|</td><td></td><td></td><td></td><td></td></tr>
<tr><td>...</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>s_f</td><td>|</td><td>$M_f^1,\dots,M_f^{s_f}$</td><td>|</td><td></td><td></td><td></td><td></td></tr>
<tr><td>c_1</td><td>c_2</td><td>c_3</td><td>c_4</td><td>c_5</td><td></td><td></td><td></td></tr>
</table>

Table 8.1. Format of an input file.

8.1. Input and output specification

A problem instance is read from a file that is provided by the user. The instance has to belong to the problem class described in Section 7.1. The format of the input file is specified in more detail in Table 8.1. First of all, the problem name, the number of processors and the number of tasks are given.

Next the weights of the individual data items are specified. Tosca then requires the following data for each task J_j: processing time, the data set I_j that is required for execution, the data set O_j that is delivered after completion of the task, and the index F_j of the family J_j belongs to. Following the task characteristics Tosca requires for each family the number of processors each of its tasks has to be executed by, and the possible prespecified processor allocations for families. Finally, the integer constants that define the communication cost function have to be given. The | sign is used as a separator.

<table>
<tr><td>problem name</td><td></td><td></td><td></td></tr>
<tr><td>m</td><td>|</td><td>n</td><td>|</td></tr>
<tr><td>evaluation rule</td><td>|</td><td>d</td><td>|</td></tr>
<tr><td>task priority</td><td>|</td><td>t</td><td></td></tr>
<tr><td>processor priority</td><td>|</td><td>u</td><td>|</td></tr>
<tr><td>$C_{\max}$</td><td>|</td><td></td><td></td></tr>
<tr><td>S_1</td><td>|</td><td>$M_{F_1}^1, \dots, M_{F_1}^{s_{F_1}}$</td><td>|</td></tr>
<tr><td>...</td><td></td><td></td><td></td></tr>
<tr><td>S_n</td><td>|</td><td>$M_{F_n}^1, \dots, M_{F_n}^{s_{F_n}}$</td><td>|</td></tr>
</table>

Table 8.2a. Format of an output file: schedule.

problem name
task priority
processor priority
evaluation rule
lower bound rule
d
t
u
number of processors
cpu running time
makespan

Table 8.2b. Format of an output file: parameter setting.

As for the output, a schedule is basically an allocation of each task to a time interval on one or more processors. The user can specify a file to save the problem name, the number of processors, the number of tasks, the evaluation and priority rules, the parameter setting, the makespan, and the corresponding starting times and processors allocations of the tasks. The schedule information is written in a format as specified in Table 8.2a.

During the use of Tosca, the user generally will specify a number of distinct parameter settings. One is enabled to specify a file to save these settings. For each parameter setting this file contains the information listed in Table 8.2b.

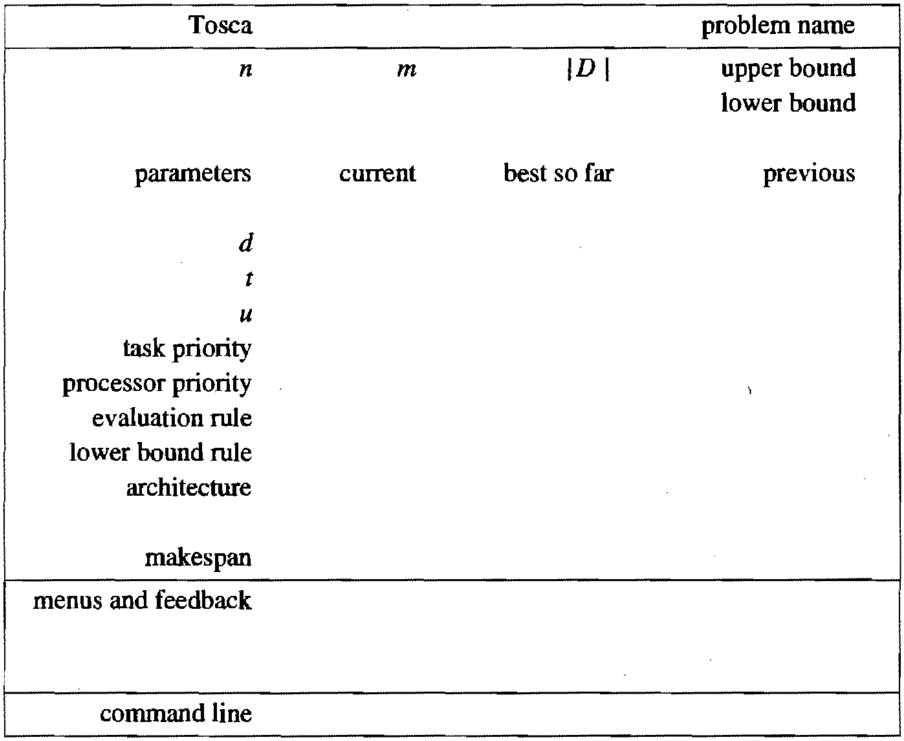


Figure 8.1. Division of the screen.

8.2. User interface and functional description

The screen is divided into three fixed regions: *problem and schedule information, menus and feedback*, and a *command line*, as in Figure 8.1. The first region specifies the problem characteristics n, m, and the cardinality of the data set D. It also gives the initial upper bound and the lower bound on the optimum. The column *current* shows the parameter values specified last. The column *best so far* specifies the best makespan so far and the parameter values used to construct the corresponding schedule. The column *previous* specifies the makespan and parameter values used to construct the one but last schedule. The second region presents menus and information about the process of bounded enumeration. It also allows for the specification of user defined priority rules, input files, and output files. Finally, the third region will show the line 'hit any key to continue' in certain situations. All information is presented in an alphanumerical manner.

The user starts by giving the command: *tosca [file name]*. The specification of a file that contains a problem instance is optional. If a file is given, then Tosca will execute the corresponding problem first. Otherwise the **main** menu appears; cf. Section 8.2.1.

The commands are divided over menus. At each point in time during the execution of Tosca at most one menu is presented to the user. The commands within a menu are numbered. Typing a number starts the execution of the corresponding command. Some commands activate a new menu. If a new menu is activated, the old menu disappears. In general, the command *continue* will activate the previous menu. The relations between the menus are given in Figure 8.2.

8.2.1. *The* **main** *menu*

The primary menu **main** appears at the start. With the aid of the command *load problem* a problem is read from an input file. The user is asked to give a file name that contains a problem instance. Immediately after loading a problem, Tosca performs priority scheduling to obtain an initial upper bound, as described in Section 7.2. It uses max-tail and min-data as task priority rule and processor priority rule, respectively. The commands *schedule*, *view*, and *save* activate new menus. By *exit* Tosca is stopped.

8.2.2. *The* **schedule** *menu*

This menu concerns the parameter setting and scheduling. The commands *task priority*, *processor priority*, *evaluation rule*, and *lower bound rule* activate new menus. At the commands *depth, tasks*, and *collections* the user is asked to specify the numbers d, t, and u, respectively. One can adjust the upper bound by choosing *upper bound*; the user is asked to type a new value. If the user is interested in scheduling an instance on a different multiprocessor architecture, then the user can specify a new number of processors by the command *architecture*. Finally, the command *run* of **schedule** starts the scheduling procedure. It uses the last defined parameters and evaluation and priority rules. These parameters and rules are displayed on the screen. During the scheduling the number of tasks scheduled so far is returned. If the sum $d+t+u$ exceeds 4, then the number of partial schedules evaluated so far is returned, too. After completion, the makespan of the schedule is presented and the menu **schedule** is activated again.

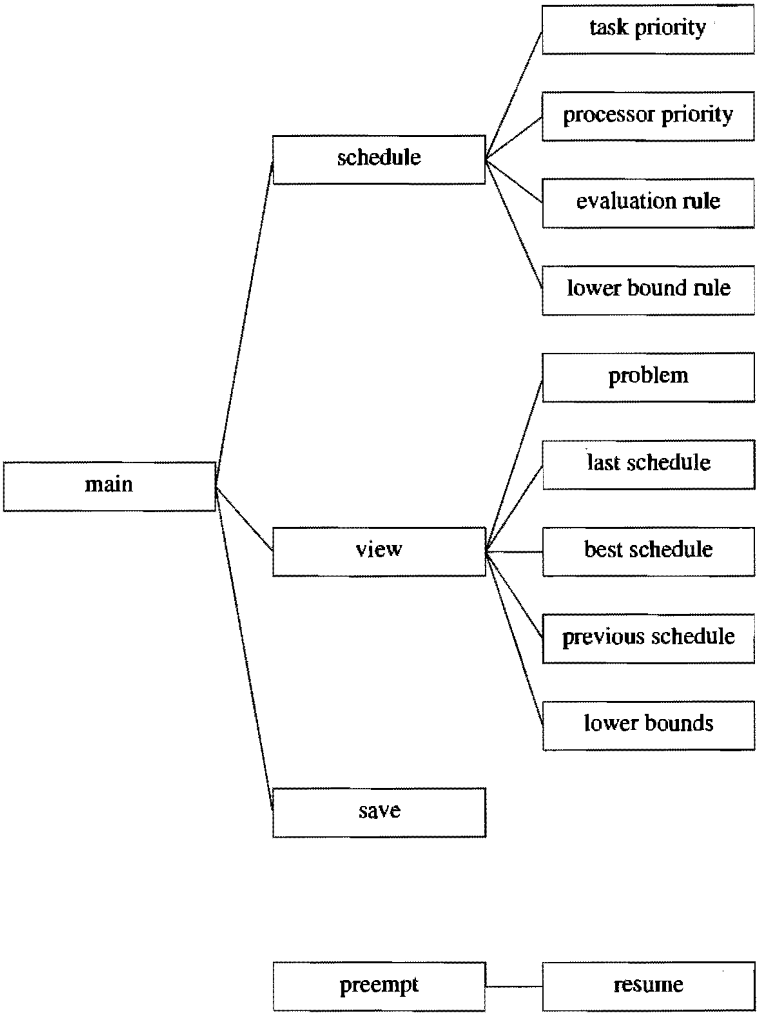


Figure 8.2. Relations between the menus.

8.2.3. *The* **task priority** *menu*

The menu **task priority** lists the predefined task priority rules and the command *user defined priority rule*. The user can either choose a task priority rule or, by *user defined priority rule*, build a new one. In order to build a new rule, numbers, attributes and operators have to be typed, according to the grammar specified in Section 7.3. The attributes and operators are listed.

8.2.4. *The* **processor priority** *menu*

The menu **processor priority** lists the predefined processor priority rules and the command *user defined priority rule*. The user can either choose a processor priority rule or, by *user defined priority rule*, build a new one. In order to build a new rule, numbers, attributes and operators have to be typed, according to the grammar specified in Section 7.3. The attributes and operators are listed.

8.2.5. *The* **evaluation rule** *menu*

The menu **evaluation rule** allows the user to select from the three evaluation procedures given in Section 7.3.

8.2.6. *The* **lower bound rule** *menu*

The menu **lower bound rule** allows the user to select from the two lower bound procedures for truncating the enumeration tree mentioned in Section 7.2.

8.2.7. *The* **view** *menu*

The menu **view** contains commands to show alphanumerical information on the lower bounds, the schedules, and the parameter settings. *Problem, last schedule, best schedule, previous schedule*, and *lower bounds* activate new menus. At the command *history* previous parameter settings and the corresponding makespans and execution times are listed.

8.2.8. *The* **problem** *menu*

In **problem**, the command *tasks* presents for each task the index, the processing time, the size, and the family the task belongs to. The command *precedence relation* gives for each task the index, the set of predecessors, and the set of successors. At *problem constants* the number of tasks, the number of processors, the cardinality of the data set D, and the constants of the communication function are given. Finally, at the command *problem* all previously mentioned data is listed and, in addition, the indata sets I_j and outdata sets O_j are presented.

8.2.9. *The* **last schedule** *menu*

After scheduling, **last schedule** allows the user to investigate the schedule. The command *tasks* shows for each task the index, the starting time, the completion time, and the collection of processors that executes the task. The command *processors* displays for each processor the index, the processor completion time, the work load, its idle time (i.e., the makespan minus its work load), and the processor completion time minus its work load.

8.2.10. *The* **best schedule** *menu*

The first schedule with the best makespan is shown, in a manner similar to the menu **last schedule**.

8.2.11. *The* **previous schedule** *menu*

The one but last schedule (if it exists) is presented, in a manner similar to the menu **last schedule**.

8.2.12. *The* **lower bounds** *menu*

In **lower bounds**, the command *makespan* shows the lower bounds *work load, longest path,* and *PapYan*, as defined in Section 7.4. The command *starting times* shows the lower bounds on the starting times of the individual tasks.

8.2.13. *The* **save** *menu*

The user of Tosca is enabled to save the last schedule, the previous schedule, the best schedule, or the current and previous parameter settings with corresponding makespans. At each of the commands listed in the menu, the user is asked to specify a file. The output is saved on the file that is specified by the user. It is written in a format as specified in Section 8.1.

8.2.14. *The* **preempt** *menu*

During the execution of the scheduling procedure within Tosca, the preempt command Ctr-C will interrupt the scheduling and activate the **preempt** menu. It contains two commands: at the command *stop* the scheduling is terminated and the menu **schedule** is reactivated, at the command *new parameters* the user is enabled to specify new parameters by use of the **resume** menu and the scheduling will be resumed with respect to this new parameter setting.

8.2.15. *The* **resume** *menu*

The **resume** menu contains a subset of the commands of **schedule**. They are: *task priority, processor priority, evaluation rule, lower bound rule, depth, tasks, collections, upper bound,* and *run*. As in **schedule**, these commands allow the user to specify (new) parameters. After the specification, the running will be resumed using the new setting.

8.3. **Four problem generators**

Four problem generators were developed to construct instances for testing Tosca. The first one generates instances with a *layered* precedence relation, that is, given an instance there exists an integer τ such that consecutively τ

tasks are mutually independent; see Figure 8.3. The second generator constructs instances with *series-parallel* precedence graphs; see Figure 8.4. A directed graph is said to be *series-parallel* if it is a single node, or it is a chain of series-parallel graphs, or it consists of a number of mutually independent series-parallel graphs preceded by a series-parallel graph and succeeded by another series-parallel graph. The third generator constructs instances with *arbitrary* precedence relations. Finally, two precedence relations from *practice* were at our disposal. We used the fourth generator to construct problem instances based on these two precedence relations.

None of the test problems has prespecified processor allocations, and each of the families within an instance consists of a single task.

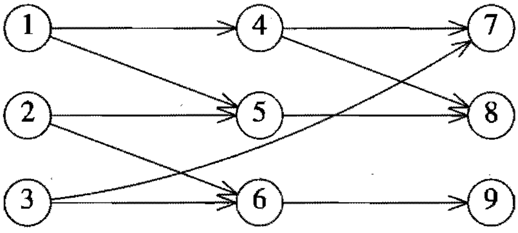


Figure 8.3. A layered precedence graph for $\tau = 3$.

8.3.1. *The generator for layered precedence relations*

The data dependencies that define the precedence graph are determined by the indata sets and outdata sets. The construction of these data sets is described below.

Since each data item is created exactly once, we may assume that the outdata sets form a partition of the data set D. The outdata set of task J_j is defined by $O_j = \{ 10(j-1)+1, \ldots, 10j \}$, for $j=1, \ldots, n$. The cardinality of the data set is $|D| = 10n$.

Given two integers υ and ρ, the indata sets are defined such that the predecessors of J_j form a subset of $\{J_{j-\upsilon-\rho}, \ldots, J_{j-\upsilon-1}\}$. Thus, at least υ tasks are unrelated with J_j and at most ρ tasks precede J_j. Formally, in order to determine the indata sets we introduce n dummy tasks $-(n-1), \ldots, 0$ with outdata sets as defined above. For each task J_j that is not a dummy, an integer δ_j is drawn from the uniform distribution $[1, 10]$. Next, δ_j data items are drawn uniformly from the outdata sets $O_{j-\upsilon-\rho}, \ldots, O_{j-\upsilon-1}$. The dummy tasks and the data items of the data sets associated with these dummies are then regarded as nonexistent. The remaining data items with respect to J_j determine the indata set I_j. Note that the cardinality $|I_j|$ is at most δ_j. The data dependency

$O_j \cap I_k \neq \emptyset$ defines the precedence constraint $J_j \rightarrow J_k$.

The weights of the data items are drawn from the uniform distribution [1,5], the processing times are drawn from the uniform distribution [1,10], and the task sizes are drawn from the uniform distribution [1,3]. The constants of the communication cost function are $c_1=1$, $c_2=1$, $c_3=3$, $c_4=0$, and $c_5=1$. Thus, the communication cost function is given by $c(x) = 1 + \lceil x/3 \rceil$.

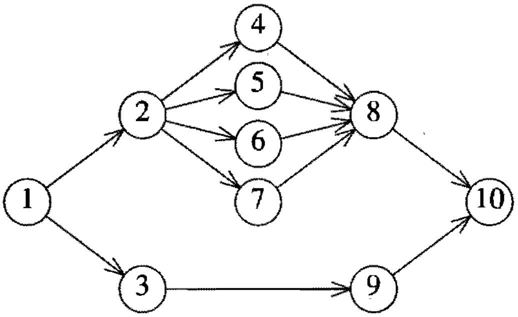


Figure 8.4. A series-parallel precedence graph.

8.3.2. *The generator for series-parallel precedence relations*

In contrast to the previous generator, the precedence relation is now constructed prior to the data sets. There are two basic operations in constructing a series-parallel graph out of a given number of series-parallel graphs. The first one is to build a chain of the series-parallel graphs; it is called the *series composition*. The second one is the *parallel composition*: one series-parallel graphs precedes and another series-parallel graph succeeds all of the remaining series-parallel graphs.

Let a list of series-parallel graphs been given. The generator randomly chooses between the two operations. If the series composition is chosen, then an integer σ is drawn from the uniform distribution [1,2] and the first σ series-parallel graphs of the list are used to construct a new one, as follows. The sink of the ith series parallel graph precedes the source of the $i+1$st graph, for $i = 1, \ldots, \sigma-1$. If the parallel composition is applied, then an integer π is drawn from the uniform distribution [4,6] and a new series-parallel graph is constructed out of the first π graphs of the list, as follows. The sink of the first graph precedes the sources of the next $\pi-2$ graphs; the sinks of the latter graphs precede the source of the last graph. The chosen graphs are removed from the beginning of the list and the new series-parallel graph is added at the end. The initial list consists of the individual tasks. The procedure is repeated until the list contains exactly one series-parallel graph.

Again, the outdata sets form a partition of the data set D. They are defined by $O_j = \{5(j-1)+1, \ldots, 5j\}$, for $j=1, \ldots, n$. Thus, the cardinality of the data set is $|D| = 5n$. For each task J_k an integer δ_k is drawn from the uniform distribution [1,10]. Next, for each predecessor J_j of J_k, δ_k data items are drawn uniformly from the outdata set O_j. These data items determine the indata set I_k. Note that the cardinality $|I_k|$ is at most $|P_k|\delta_k$.

The weights of the data items are drawn from the uniform distribution [1,4], the processing times are drawn from the uniform distribution [1,10], and the task sizes are drawn from the uniform distribution [1,3]. The constants of the communication cost function are $c_1=1$, $c_2=1$, $c_3=3$, $c_4=0$, and $c_5=1$. Thus, the communication cost function is given by $c(x) = 1 + \lceil x/3 \rceil$.

8.3.3. *The generator for arbitrary precedence relations*

Again the precedence relation is constructed prior to the data sets. Let α be a positive integer. The generator chooses α times a pair of tasks J_j, J_k from the set of not yet chosen task pairs. Each time, if $J_j \rightarrow J_k$ or $J_k \rightarrow J_j$ is part of the transitive closure of the current set of constraints, then it is added to the set of constraints and otherwise, if neither $J_j \rightarrow J_k$ nor $J_k \rightarrow J_j$ is part of the transitive closure of the current set of constraints, the generator randomly chooses between these two possibilities. The outdata sets and indata sets are constructed in a similar way as for the instances with a series-parallel precedence relation.

The weights of the data items are drawn from the uniform distribution [1,10], the processing times are drawn from the uniform distribution [1,10] for instances with $n=30$, the processing times are drawn from the uniform distribution [1,15] for instances with $n=100$, and the task sizes are drawn from the uniform distribution [1,3]. The constants of the communication cost function are $c_1=1$, $c_2=1$, $c_3=3$, $c_4=0$, and $c_5=1$. Thus, the communication cost function is given by $c(x) = 1 + \lceil x/3 \rceil$.

8.3.4. *The generator for prespecified precedence relations*

In addition to the generated problems, two precedence relations from practice were at our disposal. These originated from the program *Parasol*, which is a parallel sparse matrix system solver that has been developed by Lin and Sips [1991] within the *ParTool* project. The two precedence relations reflect the structure of the program for two different matrix systems. The graphs associated with the first and second relation consist of 74 and 388 nodes, respectively. Since only the precedence relations were made available to us, we used unit processing times and unit task sizes.

The data sets associated with the precedence relation on 74 nodes were generated in the following way. As before, the outdata sets form a partition of the data set D. They are defined by $O_j = \{10(j-1)+1, \ldots, 10j\}$, for $j=1, \ldots, n$. Thus, the cardinality of the data set is $|D| = 10n$. For each task J_k an integer δ_k is drawn from the uniform distribution $[1,3]$. Next, for each predecessor J_j of J_k, δ_k data items are drawn uniformly from the outdata set O_j. These data items determine the indata set I_k. Note that the cardinality $|I_k|$ is at most $|P_k| \delta_k$.

The data sets associated with the precedence relation on 388 nodes were generated in the following way. The outdata sets are defined by $O_j = \{j\}$, for $j=1, \ldots, n$. Thus, the cardinality of the data set is $|D| = n$. For each predecessor J_j of J_k, data item j is an element of the indata set I_k. Note that the cardinality $|I_k|$ is equal to $|P_k|$.

Three classes of communication delays were generated with respect to each of the two precedence relations: one without communication delays, one with unit time communication delays, and one with the communication cost function $c(x) = 1 + x$. For each class, the experiments were performed with m ranging from 5 to 25.

$L(\upsilon,\rho)$				min-list		min-start		min-sF		max-tail		opt
n	m	υ	ρ	fut	data	fut	data	fut	data	fut	data	
30	5	3	5	-	2	5	15	-	10	-	12	-
30	10	3	5	-	13	-	14	-	1	2	4	10
30	5	10	3	2	2	11	19	-	-	-	2	-
30	10	10	3	6	8	6	21	-	-	-	2	-
100	5	10	5	-	-	5	23	-	-	-	-	-
100	10	10	5	2	6	-	19	-	-	-	-	-
100	15	10	5	-	3	-	15	-	-	-	9	-

Table 8.3. Each cell contains the number of times (out of 25) that the corresponding combination of task and processor priority rule performed at least as well as the other given combinations. The column *opt* specifies the number of times (out of 25) that at least one of the priority combinations resulted in a schedule of length equal to the lower bound.

8.4. Computational results

Tosca has been coded in the computer language GNU C++; the experiments were conducted on a Sun Sparc Station 1. The class of problem instances with layered precedence relations, constructed as described in Section 8.3.1 with parameters υ and ρ, will be denoted by $L(\upsilon,\rho)$. The class of problem instances with series parallel precedence relations will be denoted by *SP*. We denote the class of problem instances with arbitrary precedence relations that consist of α constraints by $A(\alpha)$. Computational experiments for these three classes were performed with n ranging from 30 to 100 and m ranging from 5 to 15. Finally, the class of problem instances that have a prespecified precedence relation will be denoted by *Parasol*. Computational experiments for this class were performed with m ranging from 5 to 25.

$L(\upsilon,\rho)$				min-	min-	min-	max-	min-	min-	max-	opt
n	m	υ	ρ	list	start	sF	tail	PapYan	fifo	succ	
100	5	5	25	1	24	-	-	-	-	-	-
100	10	5	25	4	13	-	7	-	1	-	-
100	15	5	25	2	5	1	22	3	-	-	-
100	5	25	25	-	25	-	-	-	-	-	-
100	10	25	25	-	25	-	-	-	-	-	-
100	15	25	25	2	22	-	-	3	4	-	-

$A(\alpha)$			min-	min-	min-	max-	min-	min-	max-	opt
n	m	α	list	start	sF	tail	PapYan	fifo	succ	
30	5	60	-	8	-	14	2	2	-	-
30	10	60	2	3	2	24	1	1	-	12
30	5	90	-	9	-	13	7	3	2	4
30	10	90	5	7	6	21	6	4	6	16
100	5	300	-	7	-	17	1	-	-	-
100	10	300	-	1	-	24	-	-	-	5
100	15	300	-	1	-	24	-	-	-	7
100	5	1000	1	4	-	18	1	2	-	-
100	10	1000	-	1	1	22	2	-	-	3
100	15	1000	-	1	2	22	1	-	-	3

SP		min-	min-	min-	max-	min-	min-	max-	opt
n	m	list	start	sF	tail	PapYan	fifo	succ	
30	5	-	10	4	8	8	1	-	-
30	10	-	7	11	13	6	2	-	3
100	5	-	13	-	9	5	-	-	-
100	10	-	12	-	13	3	-	-	-
100	15	-	5	7	12	4	-	-	-

Table 8.4. Each cell contains the number of times (out of 25) that the corresponding task priority rule performed at least as well as the other priority rules. The column *opt* states the number of times (out of 25) that at least one of the task priorities resulted in a schedule with makespan equal to the lower bound.

8.4.1. *List scheduling*

For a start, the parameters *d*, *t*, and *u*, as described in Section 7.2, were set to 1 and list scheduling was performed for a number of different combinations of task and processor priority rules. A description of the priority rules can be found in Section 7.3. The results are given in Table 8.3 throughout Table 8.7.

L(υ,ρ)				min-	min-	min-	max-	min-	min-	max-	*lb*
n	*m*	υ	ρ	list	start	sF	tail	PapYan	fifo	succ	
100	5	5	25	267	**251**	314	273	273	270	327	224
100	10	5	25	139	**136**	153	138	142	141	176	115
100	15	5	25	120	120	122	**116**	121	122	133	112
100	5	25	25	267	**243**	303	270	267	268	312	224
100	10	25	25	124	**117**	135	124	124	123	156	112
100	15	25	25	82	**79**	90	83	82	82	107	75
A(α)				min-	min-	min-	max-	min-	min-	max-	*lb*
n	*m*	α		list	start	sF	tail	PapYan	fifo	succ	
30	5	60		159	127	137	**116**	159	127	137	96
30	10	60		130	103	127	**101**	131	103	126	96
30	5	90		141	136	146	**135**	137	138	144	124
30	10	90		129	127	128	**126**	128	129	130	124
100	5	300		622	523	622	**517**	544	559	614	418
100	10	300		473	448	456	**426**	453	466	474	417
100	15	300		455	442	442	**423**	444	457	462	417
100	5	1000		784	749	775	**740**	749	757	786	701
100	10	1000		735	725	724	**710**	725	734	736	701
100	15	1000		735	725	724	**710**	725	734	736	701
SP				min-	min-	min-	max-	min-	min-	max-	*lb*
n	*m*			list	start	sF	tail	PapYan	fifo	succ	
30	5			136	**123**	130	124	124	127	137	105
30	10			115	111	111	**110**	111	113	117	105
100	5			451	**372**	434	374	379	383	441	281
100	10			345	**301**	330	302	307	311	339	280
100	15			314	294	304	**293**	297	301	313	280

Table 8.5. Each cell contains the average makespan over 25 instances. The column *lb* states the average lower bounds. Bold figures indicate the lowest average makespans.

In Table 8.3 we report on the results obtained with four task priority rules and two processor priority rules on problem instances of the class *L*(υ,ρ). The task priority rules are list, min-start, min-sF, and max-tail. The processor priority rules are min-fut and min-data. For each combination of *n*, *m*, υ, and ρ we considered 25 instances. The processor priority rule min-data clearly outperforms the processor priority rule min-fut, which is probably explained by the fact that it gives way to processors that enable a task to start as early as possible. A similar difference in behavior can be reported for min-data with respect to the other processor priority rules described in Section 7.3. Given this information, the processor priority rule was fixed to min-data for all of the remaining experiments, although this priority rule is more time consuming than the other ones.

$L(\upsilon,\rho)$				min-list	min-start	min-sF	max-tail	min-PapYan	min-fifo	max-succ
n	m	υ	ρ							
100	5	5	25	7	13	7	8	8	8	8
100	10	5	25	8	20	8	8	8	9	8
100	15	5	25	9	27	9	9	9	9	9
100	5	25	25	7	17	7	7	7	7	7
100	10	25	25	8	29	7	8	8	8	8
100	15	25	25	8	45	8	8	8	9	8
$A(\alpha)$				min-list	min-start	min-sF	max-tail	min-PapYan	min-fifo	max-succ
n	m	α								
30	5	60		<1	1	1	1	1	1	<1
30	10	60		<1	1	1	1	1	1	1
30	5	90		1	1	1	1	1	1	1
30	10	90		1	2	1	1	2	1	2
100	5	300		8	11	8	8	8	8	8
100	10	300		9	15	9	9	10	10	10
100	15	300		10	19	10	10	11	11	11
100	5	1000		1:03	1:39	1:05	1:05	1:07	1:07	1:08
100	10	1000		1:14	2:01	1:14	1:15	1:18	1:18	1:20
100	15	1000		1:20	2:17	1:22	1:24	1:26	1:26	1:29
SP				min-list	min-start	min-sF	max-tail	min-PapYan	min-fifo	max-succ
n	m									
30	5			<1	1	<1	<1	<1	<1	<1
30	10			<1	1	<1	<1	<1	<1	<1
100	5			4	4	4	4	4	4	4
100	10			4	6	4	4	4	4	4
100	15			4	7	4	4	4	4	4

Table 8.6. Each cell contains the average cpu-time in minutes and seconds over 25 instances.

In Table 8.4 we investigated seven task priority rules on problem instances of the classes $L(\upsilon,\rho)$, $A(\alpha)$, and SP. For each problem specification, we considered 25 instances.

The priority rule min-start gives good results when the precedence relation is regular and the number of machines is severely restrictive, i.e., for problem instances of the class $L(\upsilon,\rho)$ and $m=5$. For a sufficiently large number of machines and less regular precedence relations the priority rule max-tail gives the best results.

A measure of the quality of the schedules produced by the seven task priority rules can be found in Table 8.5. Again, we conclude that the priority rules min-start and max-tail outperform the other predefined task priority rules.

The time requirements of the priority rules are mutually comparable, with the exception of the more time consuming priority rule min-start. Table 8.6 lists the average cpu-times in minutes and seconds on a Sun Sparc Station 1.

The results for the instances associated with the two precedence relations that originated from the program *Parasol*, as described in Section 8.3.4, are given in Table 8.7. Each specification of the parameters n, m, and $c(x)$ corresponds to a unique instance. Each time, the best upper bound is printed in bold. Again, the priority rules min-start and max-tail are the best.

Parasol			min-	min-	min-	max-	min-	min-	max-	*lb*
n	m	$c(x)$	list	start	sF	tail	PapYan	fifo	succ	
74	5	0	25	23	25	**21**	24	23	26	19
74	10	0	20	20	20	**19**	20	20	20	19
74	25	0	**19**	**19**	**19**	**19**	**19**	**19**	**19**	19
74	5	1	29	27	29	**26**	31	32	29	24
74	10	1	26	26	26	**25**	28	31	26	24
74	25	1	**25**	26	**25**	**25**	27	29	26	24
74	5	1+x	62	**54**	62	71	69	70	72	27
74	10	1+x	62	**54**	62	68	69	70	72	27
74	25	1+x	62	**54**	62	68	69	70	72	27
388	5	0	89	90	89	**84**	96	89	104	78
388	10	0	64	59	64	**53**	62	59	67	46
388	25	0	48	48	48	**46**	48	48	48	46
388	5	1	101	102	101	**91**	106	105	123	78
388	10	1	80	73	80	**68**	85	83	85	59
388	25	1	**62**	65	**62**	65	75	75	66	59
388	5	1+x	122	111	122	**105**	121	119	141	78
388	10	1+x	96	**83**	96	84	102	105	107	59
388	25	1+x	80	81	80	**75**	97	101	90	59

Table 8.7. Each cell gives the *length* of the schedule with respect to the task priority rule. The column *lb* specifies the value of the lower bound.

L(υ,ρ)	*n*	*m*	υ	ρ	*lb*	*C*(1)	*C*(2)	*C*(3)
1	30	15	10	3	29	30	29	29
2	30	15	10	3	33	34	33	33
3	30	15	10	3	29	30	29	-
4	30	15	10	3	32	33	-	32
5	30	15	10	3	24	25	24	-
6	30	15	10	3	29	30	29	29
7	100	15	5	25	114	118	117	-
8	100	15	5	25	101	104	101	102
9	100	15	5	25	120	121	120	120
A(α)	*n*	*m*	α		*lb*	*C*(1)	*C*(2)	*C*(3)
10	30	5	60		103	114	109	111
11	30	5	60		77	95	93	90
12	30	5	60		92	119	-	110
13	30	5	60		89	107	104	-
14	30	5	60		84	104	99	100
15	30	10	60		103	107	103	-
16	30	10	60		74	78	-	77
17	30	5	90		107	116	115	-
18	30	5	90		112	128	125	127
19	30	5	90		117	133	131	122
20	30	5	90		133	146	145	-
21	100	5	1000		614	692	686	670
22	100	5	1000		584	617	-	614
23	100	5	1000		757	799	-	796
24	100	5	1000		706	731	-	721
25	100	5	1000		729	776	764	754
26	100	10	1000		784	796	792	792
27	100	10	1000		614	638	633	633
28	100	10	1000		757	769	-	767
29	100	10	1000		706	722	-	717
30	100	10	1000		729	741	-	738
31	100	15	1000		784	796	792	792
32	100	15	1000		614	638	633	633
33	100	15	1000		757	769	-	767
34	100	15	1000		706	722	-	717
35	100	15	1000		729	741	-	738
SP	*n*	*m*			*lb*	*C*(1)	*C*(2)	*C*(3)
36	30	5			94	124	121	-
37	30	5			99	124	123	123
38	30	5			99	115	-	114
39	30	5			109	120	-	119
40	30	10			100	103	100	-
41	30	10			99	100	99	-
42	100	5			309	391	-	388
43	100	10			309	327	-	324
44	100	15			309	317	316	-
45	100	15			312	329	-	326
Parasol	*n*	*m*	*c*(*x*)		*lb*	*C*(1)	*C*(2)	*C*(3)
46	388	25	1		59	62	61	61

Table 8.8. Improvements.

8.4.2. *Searching with* $d = t = u = 2$

From the instances mentioned in the previous section, we chose 225 instances for bounded enumeration. From the class $L(\upsilon,\rho)$ we chose seven problem types, from the class $A(\alpha)$ we chose nine problem types and from the class *SP* we chose five problem types. For each problem type we selected ten instances for bounded enumeration. The remaining fifteen instances were those of the class *Parasol* for which no optimal solutions were found by list scheduling. The parameters d, t, and u were set to 2 and the same seven task priority rules as in the previous section were tested. The processor priority rule remained fixed to min-data. For each instance, the upper bound was specified as the length of the best schedule found during list scheduling. Tosca searched for an improvement.

$L(\upsilon,\rho)$				min-list	min-start	min-sF	max-tail	min-PapYan	min-fifo	max-succ
n	m	υ	ρ							
30	15	10	3	9	20	5	6	5	4	4
100	5	5	25	1:52	2:16	1:43	1:52	1:53	1:52	1:43
100	10	5	25	1:01	1:38	58	1:12	1:14	1:10	60
100	15	5	25	56	1:46	36	1:31	41	43	45
100	5	25	25	1:51	2:17	1:43	1:53	1:54	1:52	1:41
100	10	25	25	1:44	2:57	1:39	1:51	1:52	1:50	1:36
100	15	25	25	1:35	3:19	1:31	1:45	1:42	1:45	1:30
$A(\alpha)$				min-list	min-start	min-sF	max-tail	min-PapYan	min-fifo	max-succ
n	m	α								
30	5	60		5	7	5	5	5	5	5
30	10	60		7	7	5	7	6	6	6
30	5	90		8	9	8	8	8	7	7
100	5	300		1:02	1:37	1:16	1:24	1:22	1:24	1:16
100	10	300		28	41	41	54	41	32	31
100	15	300		48	57	57	1:10	52	53	48
100	5	1000		5:57	7:41	6:52	6:05	7:06	6:01	6:34
100	10	1000		5:09	6:58	5:39	5:09	6:27	4:54	4:54
100	15	1000		5:30	6:58	5:30	5:09	6:17	5:11	5:14
SP				min-list	min-start	min-sF	max-tail	min-PapYan	min-fifo	max-succ
n	m									
30	5			3	4	3	3	3	3	3
30	10			3	3	2	2	2	2	2
100	5			34	44	37	41	42	40	37
100	10			28	45	37	35	35	33	32
100	15			31	51	37	37	40	31	35

Table 8.9. Each cell gives the average cpu-time in seconds over 10 instances.

In total, bounded enumeration with parameters *d*, *t*, and *u* equal to 2 yielded a better makespan than list scheduling for 29 instances out of 210. In Table 8.8 these makespans are reported in column $C(2)$; they are the best over all priority rules. The column *lb* gives the lower bound for the corresponding problem instance and the column $C(1)$ specifies the makespan of the best schedule generated by list scheduling.

The average cpu-times are listed in Table 8.9.

$L(\upsilon,\rho)$				min-start	max-tail
n	*m*	υ	ρ		
30	15	10	3	8:18	3:20
100	5	5	25	1:22:44	1:12:08
100	10	5	25	55:25	43:27
100	15	5	25	53:52	50:15
100	5	25	25	1:27:45	1:17:01
100	10	25	25	1:32:04	1:11:53
100	15	25	25	1:31:23	1:06:30

$A(\alpha)$			min-start	max-tail
n	*m*	α		
30	5	60	2:22	2:22
30	10	60	2:15	2:43
30	5	90	2:06	1:59
100	5	300	44:59	41:16
100	10	300	18:59	21:11
100	15	300	30:8	31:12
100	5	1000	2:10:03	2:03:55
100	10	1000	1:26:33	1:13:22
100	15	1000	1:27:27	1:12:43

SP		min-start	max-tail
n	*m*		
30	5	1:12	1:07
30	10	1:05	56
100	5	18:29	18:10
100	10	16:31	14:23
100	15	16:52	15:55

Table 8.10. Each cell gives the average cpu-time in seconds over 10 instances.

8.4.3. *Searching with* $d=t=u=3$

The 225 instances mentioned in Section 8.4.2 were also subjected to bounded enumeration with parameters *d*, *t*, and *u* set to 3. For each instance, the two task priority rules min-start and max-tail were used to improve the best schedule generated by list scheduling. Thus the initial upper bound was specified as the length of the best schedule found during list scheduling. As before, the processor priority rule remained fixed to min-data. In total, bounded enumeration with parameters *d*, *t*, and *u* equal to 3 yielded a better makespan than list scheduling for 35 instances. In Table 8.8 these makespans are reported in column $C(3)$.

The average cpu-times are listed in Table 8.10.

List scheduling turns out to be rather effective. It generates good schedules for problem instances with a sufficiently large number of processors; cf. Table 8.5. The process of bounded enumeration is time consuming and for only 46 out of 225 instances a better schedule was constructed. Most of the improvements are minor. This may be due to the choice of test problems, although we tried to reduce this factor by constructing instances with the aid of four different types of generators.

As mentioned in Section 7.4, the lower bound is tighter for problem instances with a relatively large number of processors, and for problems where the communication delays are small with respect to the processing times; cf. Table 8.5 and 8.7. The lower bound remains valid if task duplication is allowed. Task duplication may be of little help in practice, since the lower bound is often tight even if duplication is not allowed.

9. Tosca: an example

As an illustration of the models and methodology described in the previous chapters, we will now give an example of the application of Tosca to a problem instance of the type $P \mid prec, com, fam, set \mid C_{\max}$. Its data is given in Table 9.1 in the format of an input file required by Tosca; cf. Table 8.1.

```
example  |   2   |   9   | 3, 3, 1, 2, 2, 4 |

   2     |       |   1   |        1         |
   2     |       |   2   |        2         |
   1     |  1,2  |   3   |        3         |
   2     |   3   |       |        3         |
   1     |   3   |   4   |        4         |
   2     |   4   |   5   |        5         |
   1     |   5   |   6   |        6         |
   3     |   5   |       |        7         |
   4     |   6   |       |        8         |

   1     |       |
   1     |       |
   1     |   2   |
   2     |       |
   1     |       |
   2     |       |
   1     |       |
   2     |       |

   0     1   1   0   1
```

Table 9.1. The input file.

In total, nine tasks have to be executed by two processors. The data set D consists of six data items with integer weights 3, 3, 1, 2, 2, and 4, respectively. For each task J_j, the processing time p_j and the number of processors s_j it requires are given in Table 9.2. The tasks J_3 and J_4 belong to the same family F_3. The tasks of this family have to be executed by processor M_2. The remaining tasks still have to be allocated. Note that for each of the tasks J_5, J_7, and J_9 there is a unique feasible processor allocation; each one has to be executed by both processors simultaneously.

The data dependencies impose a precedence relation on the task set. It is represented by means of the directed graph given in Figure 9.1; the nodes of the graph correspond to the tasks and the arcs represent the constraints. The constants of the communication cost function are $c_1 = 0$, $c_2 = 1$, $c_3 = 1$, $c_4 = 0$, and $c_5 = 1$. These constants define the communication cost function $c(x) = x$.

J_j	1	2	3	4	5	6	7	8	9
p_j	2	2	1	2	1	2	1	3	4
s_j	1	1	1	1	2	1	2	1	2

Table 9.2. Task characteristics.

The total amount of work is $\Sigma_{j=1}^{9} p_j s_j = 24$ time units. Since there are two processors , we have that $C^*_{\max} \geq 12$.

The length of a longest path with respect to the precedence relation yields another lower bound. For the precedence relation of the example we have that $C^*_{\max} \geq p_1 + p_3 + c(w(3)) + p_5 + p_6 + c(w(5)) + p_7 + p_9 = 14$. Here, the communication delay in between J_3 and J_5 is taken into account; this delay cannot be avoided since $s_3 < s_5$. The data set that has to be transmitted is $\{3\}$. The weight of this set is equal to $w_3 = 1$. Thus, the delay in between J_3 and J_5 takes $c(1) = 1$ unit of time. A similar computation can be made for the communication delay in between J_6 and J_7. This approach also generates lower bounds on the starting times of the individual tasks. These lower bounds are given in Table 9.3 by the name *longest path.*

If the tasks J_1 and J_2 are allocated to distinct processors, then the starting time of task J_3 is at least equal to $S_3 \geq 5$. However, if the predecessors of J_3 are assigned to a single processor, then $S_3 \geq 4$. Since these are the only possible allocations of J_1 and J_2, it follows that $S_3 \geq 4$. This value is computed by the lower bound *PapYan* as described in Section 7.4. The lower bounds for the individual tasks are listed in Table 9.3 by the name *PapYan.*

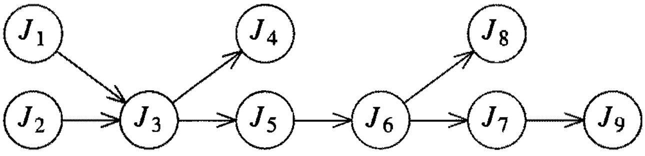
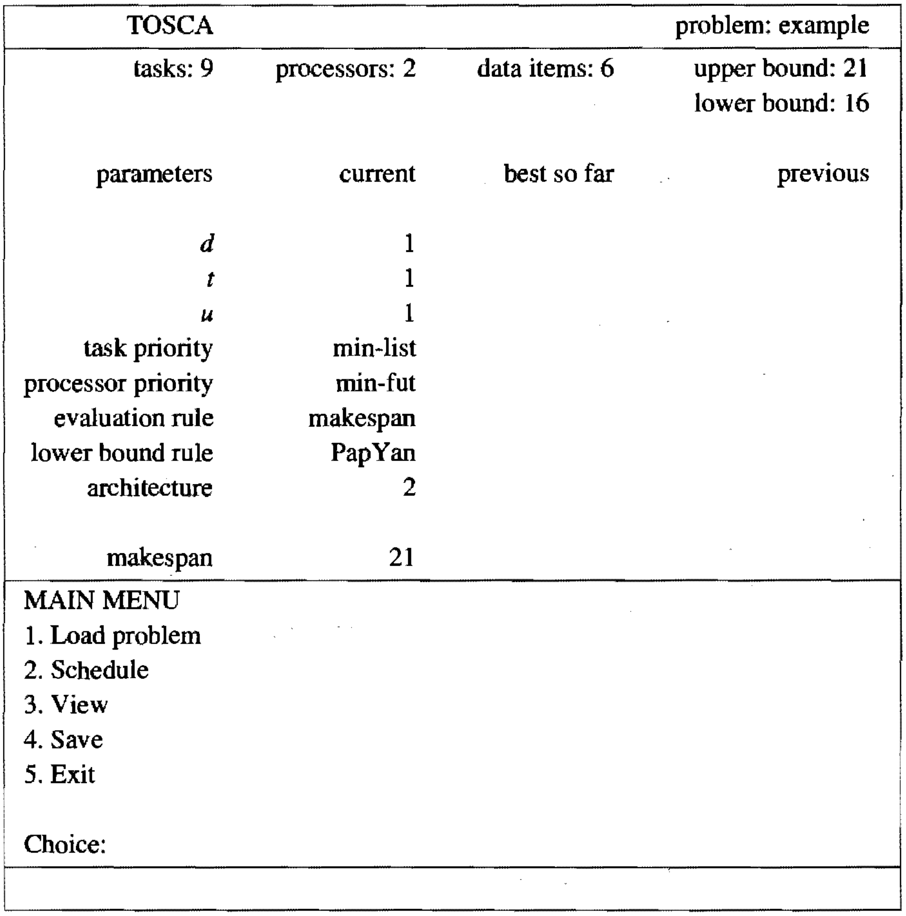


Figure 9.1. The precedence relation.

The lower bounds and an initial upper bound are computed at the start of Tosca. Tosca is started by giving the command *tosca example.* The upper bound is the length of the initial schedule. The first screen that is presented will look like Figure 9.2. This figure also specifies the priority rules and the parameters that were used to construct the initial schedule. The schedule itself is given in Figure 9.3a; it is of length 21. The dotted line represents the lower bound PapYan on the makespan.

	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9	$C_{\max}$
longest path	0	0	2	3	4	5	9	7	10	14
PapYan	0	0	4	5	6	7	11	9	12	16

Table 9.3. Lower bounds.



```
          TOSCA                                          problem: example
------------------------------------------------------------------------
         tasks: 9      processors: 2      data items: 6     upper bound: 21
                                                           lower bound: 16

       parameters           current       best so far           previous

                d                 1
                t                 1
                u                 1
    task priority          min-list
processor priority          min-fut
  evaluation rule          makespan
 lower bound rule            PapYan
     architecture                 2

         makespan                21
------------------------------------------------------------------------
MAIN MENU
1. Load problem
2. Schedule
3. View
4. Save
5. Exit

Choice:
------------------------------------------------------------------------
```

Figure 9.2. The initial screen.

The initial schedule is suboptimal in three ways. In order to start J_3 as early as possible, both predecessors of J_3 have to be executed by processor M_2, which executes J_3. Secondly, in order to start J_5 as early as possible, it should be executed before J_4. Finally, task J_8 has to start before J_7.

Given the **main** menu, the command *schedule* (select 2) activates the menu that allows for new choices of priority rules and evaluation rules. Changing the processor priority rule into, for instance, min-data and the evaluation rule into, for instance, PapYan will not help to construct a better schedule.

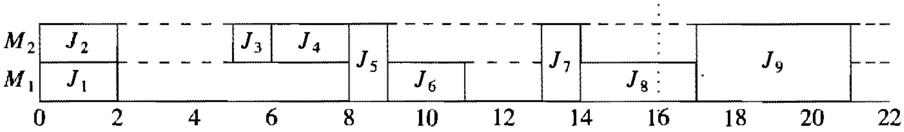


Figure 9.3a. List scheduling: initial schedule.

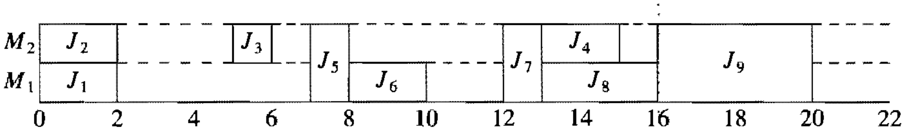


Figure 9.3b. List scheduling: max-tail, min-data.

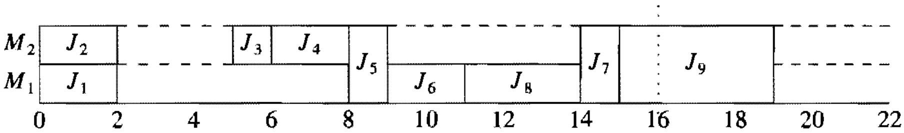


Figure 9.3c. List scheduling: min-start, min-data.

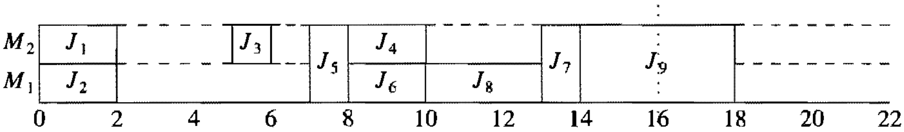


Figure 9.3d. List scheduling: user defined priority rule.

Since the amount of work that has to be done after completion of J_5 is more than the work following J_4, the choice of task priority rule max-tail will generate a schedule of length 20, as given in Figure 9.3b. Note that J_5 is executed before J_4.

The choice of task priority rule min-start will execute J_8 before J_7, because J_8 can start immediately following the completion of task J_6. However, tasks J_4 and J_5 are placed in the wrong order again. The resulting schedule has makespan 19 and is given in Figure 9.3c.

If one searches for a better task priority rule, then one has to construct such a rule manually; the predefined priority rules will not help anymore. Such a user priority rule has to choose J_5 before J_4 and J_8 before J_7. The user rule –(list–4)*(list–16/3)*(list–7) is such a priority rule. It considers the tasks in the order 2, 1, 3, 5, 4, 6, 8, 7, 9 and generates the schedule of length 18, given in Figure 9.3d.

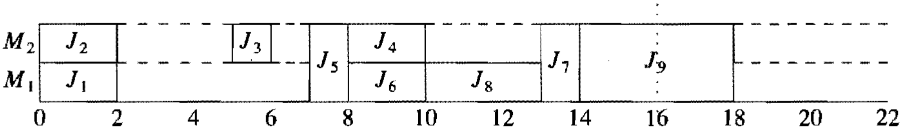


Figure 9.3e. Bounded enumeration: d=3, t=2, and u=1.

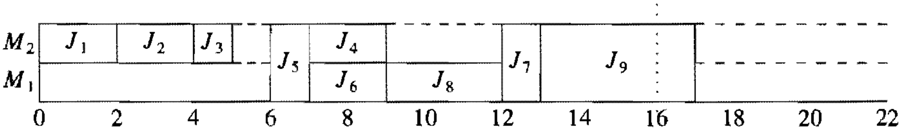


Figure 9.3f. Bounded enumeration: d=3, t=2, and u=2.

A schedule of length 18 can also be constructed by bounded enumeration with parameters d=3, t=2, and u=1, and task priority rule max-tail, processor priority min-data, and evaluation rule PapYan; it is given in Figure 9.3e.

Finally, bounded enumeration with parameters d=3, t=2, and u=2 will yield a schedule of length 17; see Figure 9.3f. It is optimal since J_7 and J_8 cannot be scheduled in parallel. An optimal schedule cannot be constructed by use of list scheduling. It is impossible to construct a processor priority rule that schedules tasks J_1 and J_2 on processor M_2.

References

A.V. Aho, J.E. Hopcroft, J.D. Ullman (1988). *Data structures and algorithms*, Addison Wesley, Reading, MA.

M. Ahuja, Y. Zhu (1990). An $O(n\log n)$ feasibility algorithm for preemptive scheduling of n jobs on a hypercube. *Inform. Process. Lett. 35*, 7-11.

F.D. Anger, J. Hwang, Y. Chow (1990). Scheduling with sufficiently loosely coupled processors. *J. Parallel Distributed Comput. 9*, 87-92.

T.S. Arthanri (1974). *On some problems of sequencing and grouping*, Dissertation, Indian Statistical Institute, Calcutta.

L. Bianco, P. Dell'Olmo, J. Blazewicz (1992). Scheduling multiprocessor independent tasks with dedicated processors and additional resources. Presentation, *EURO XII - TIMS XXXI*, Helsinki, 1992.

L. Bianco, P. Dell'Olmo, M.G. Speranza (1991). On scheduling independent tasks with dedicated resources. Presentation, *14th Int. Symp. Math. Programming*, Amsterdam, August 5-9, 1991.

J. Blazewicz, P. Dell'Olmo, M. Drozdowski, M.G. Speranza (1992). Scheduling multiprocessor tasks on three dedicated processors. *Inform. Process. Lett. 41*, 275-280.

J. Blazewicz, M. Drabowski, J. Weglarz (1986). Scheduling multiprocessor tasks to minimize schedule length. *IEEE Trans. Comput. C-35*, 389-393.

J. Blazewicz, M. Drozdowski, G. Schmidt, D. de Werra (1990). Scheduling independent two processor tasks on a uniform duo-processor system. *Discrete Appl. Math. 28*, 11-20.

J. Blazewicz, M. Drozdowski, G. Schmidt, D. de Werra (1992). *Scheduling independent multiprocessor tasks on a uniform k-processor system*, Report R30-92, Instytut Informatyki, Politechnika Poznańska, Poznań.

J. Blazewicz, G. Finke, R. Haupt, G Schmidt (1988). New trends in machine scheduling. *European J. Oper Res. 37*, 303-317.

J. Blazewicz, J.K. Lenstra, A.H.G. Rinnooy Kan (1983). Scheduling subject to resource constraints: classification and complexity. *Discrete Appl. Math. 5*, 11-24.

J. Blazewicz, J. Weglarz, M. Drabowski (1984). Scheduling independent 2-processor tasks to minimize schedule length. *Inform. Process. Lett. 18*, 267-273.

S.H. Bokhari (1981). On the mapping problem. *IEEE Trans. Comput. C-30*, 207-214.

G. Bozoki, J.P. Richard (1970). A branch-and-bound algorithm for the continuous-process task shop scheduling problem. *AIIE Trans. 2*, 246-252.

R.E. Buten, V.Y. Shen (1973). A scheduling model for computer systems with two classes of processors, *Proc. 1973 Sagamore Comput. Conference*

Parallel Processing, 130-138.

R.S. Chang, R.C.T. Lee (1988). On a scheduling problem where a job can be executed only by a limited number of processors. *Comput. Oper. Res. 15*, 471-478.

Y.L. Chen, Y.H. Chin (1989). Scheduling unit-time jobs on processors with different capabilities. *Comput. Oper. Res. 16*, 409-417.

G.I. Chen, T.H. Lai (1988A). Scheduling independent jobs on hypercubes. *Proc. Conf. Theoretical Aspects of Computer Science*, 273-280.

G.I. Chen, T.H. Lai (1988B). Preemptive scheduling of independent jobs on a hypercube. *Inform. Process. Lett. 28*, 201-206.

G.I. Chen, T.H. Lai (1991). Scheduling independent jobs on partitionable hypercubes. *J. Parallel Distributed Comput. 12*, 74-78.

P. Chrétienne (1989). A polynomial algorithm to optimally schedule tasks on a virtual distributed system under tree-like precedence constraints. *European J. Oper. Res. 43*, 225-230.

P. Chrétienne (1992). Task scheduling with interprocessor communication delays. *European J. Oper. Res. 57*, 348-354.

P. Chrétienne, C. Picouleau (1991). *The basic scheduling problem with interprocessor communication delays*. RP91/6, MASI, Institut Blaise Pascal, Université Paris VI, Paris.

J.Y. Colin, P. Chrétienne (1993). C.P.M. scheduling with small communication delays and task duplication. *Oper. Res.*, to appear.

R.W. Conway, W.L. Maxwell, L.W. Miller (1967). *Theory of scheduling*, Addison-Wesley, Reading, MA.

G. Dobson, U.S. Karmarkar (1989). Simultaneous resource scheduling to minimize weighted flow times. *Oper. Res. 37*, 592-600.

J. Du, J. Y-T. Leung (1989). Complexity of scheduling parallel task systems. *SIAM J. Discrete Math. 2*, 473-487.

H. El-Rewini, T.G. Lewis (1990). Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distributed Comput. 9*, 138-153.

M.J. Flynn (1966). Very high-speed computing systems. *Proc. IEEE 54*, 1901-1909.

M. Fujii, T. Kasami, K. Ninomiya (1969, 1971). Optimal sequencing of two equivalent processors. *SIAM J. Appl. Math. 17*, 784-789; Erratum. *SIAM J. Appl. Math. 20*, 141.

M.R. Garey, D.S. Johnson (1979). *Computers and intractability: a guide to the theory of NP-completeness*, Freeman, San Francisco.

A. Gerasoulis, T. Yang (1990, revision 1992). *On the granularity and clustering of directed acyclic task graphs*. Report TR-153, Dept. Comput. Sci., Rutgers University.

M.C. Golumbic (1980). *Algorithmic graph theory and perfect graphs*, Academic Press, New York.

R.L. Graham (1966). Bounds for certain multiprocessing anomalies. *Bell System Tech. J. 45*, 1563-1581.

R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann. Discrete Math. 5*, 287-326.

J.A. Hoogeveen, J.K. Lenstra, B. Veltman (1992). *Three, four, five, six, or the complexity of scheduling with communication delays*, Report BS-R9229, CWI, Amsterdam.

J.A. Hoogeveen, J.K. Lenstra, B. Veltman (1993). *Minimizing makespan in a multiprocessor flow shop is strongly* NP-*hard*, in preparation.

J.A. Hoogeveen, S.L. van de Velde, B. Veltman (1993). Complexity of scheduling multiprocessor tasks with prespecified processor allocations. *Discrete Appl. Math.*, to appear.

T.C. Hu (1961). Parallel sequencing and assembly line problems. *Oper. Res. 9*, 841-848.

C.A.J. Hurkens (1992). Private communication.

J.J. Hwang, Y.C. Chow, F.D. Anger, C.Y. Lee (1989). Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput. 18*, 244-257.

A. Jakoby, R. Reischuk (1992). The complexity of scheduling problems with communication delays for trees. *Proc. Skandinavian Workshop on Algorithmic Theory 3*, 165-177.

S.M. Johnson (1954). Optimal two- and three-stage production schedules with set-up times included. *Nav. Res. Logist. Quart. 1*, 61-68.

H. Jung, L. Kirousis, P. Spirakis (1989). Lower bounds and efficient algorithms for multiprocessor scheduling of dags with communication delays. *Proc. ACM Symp. Parallel Algorithms and Architectures*, 254-264.

H. Kellerer, G.J. Woeginger (1992). UET-Scheduling with constrained processor allocations. *Comput. Oper. Res. 19*, 1-8.

S.J. Kim (1988). *A general approach to multiprocessor scheduling*, Report TR-88-04, Dept. Comput. Sci., University of Texas, Austin.

G.A.P. Kindervater, J.K. Lenstra (1988). Parallel computing in combinatorial optimization. *Ann. Oper. Res. 14*, 245-289.

H. Krawczyk, M. Kubale (1985). An approximation algorithm for diagnostic test scheduling in multiprocessor systems. *IEEE Trans. Comp. C-34*, 869-872.

B. Kruatrachue, T. Lewis (1988). Grain size determination for parallel processing. *IEEE Software*, 23-32.

M. Kubale (1987). The complexity of scheduling independent two-processor tasks on dedicated processors. *Inform. Process. Lett. 24*, 141-147.

E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (1993). Sequencing and scheduling: algorithms and complexity. *Handbooks in Operations Research and Management Science, Volume 4: Logistics of Production and Inventory,* edited by S.C. Graves, A.H.G. Rinnooy Kan and P. Zipkin, North-Holland, Amsterdam.

C.Y. Lee, J.J. Hwang, Y.C. Chow, F.D. Anger (1988). Multiprocessor scheduling with interprocessor communication delays. *Oper. Res. Lett. 7*, 141-147.

H.W. Lenstra, Jr. (1983). Integer programming with a fixed number of variables. *Math. Oper. Res. 8*, 538-548.

J.K. Lenstra, A.H.G. Rinnooy Kan, P. Brucker (1977). Complexity of machine scheduling problems. *Ann. Discrete Math. 1*, 343-362.

J.K. Lenstra, M. Veldhorst, B. Veltman (1993). *The complexity of scheduling trees with communication delays*, in preparation.

K. Li, K.H. Cheng (1990). Static job scheduling in partitionable mesh connected systems. *J. Parallel Distributed Comput. 10*, 152-159.

H.X. Lin, H.J. Sips (1991). *A distributed direct solver for the Diana finite elements system*, Report WP 1.4, Genesis, Esprit project.

E.L. Lloyd (1981). Concurrent task systems. *Oper. Res. 29*, 189-201.

S. Martello, P.Toth (1990). *Knapsack problems: algorithms and computer implementations*, Wiley, Chichester.

R. McNaughton (1959). Scheduling with deadlines and loss functions. *Management Sci. 6*, 1-12.

R.H. Möhring (1989). Computationally tractable classes of ordered sets. I. Rival (ed.) (1989). *Algorithms and Order*, Kluwer Academic, Dordrecht, 105-193.

C.H. Papadimitriou, J.D. Ullman (1987). A communication-time tradeoff. *SIAM J. Comput. 16*, 639-646.

C.H. Papadimitriou, M. Yannakakis (1990). Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput. 19*, 322-328.

C. Picouleau (1991A). *Ordonnancement de taches de durée unitaire avec des delais de communication unitaires sur m processeurs.* Report RP91/64, MASI, Institut Blaise Pascal, Université Paris VI, Paris.

C. Picouleau (1991B). *Two new NP-complete scheduling problems with communication delays and unlimited number of processors.* Report RP91/24, MASI, Institut Blaise Pascal, Université Paris VI, Paris.

C. Picouleau (1992). *Etude de problèmes d'optimisation dans les systèmes distribués.* Thèse de doctorat, Université Paris VI.

V.J. Rayward-Smith (1987A). UET scheduling with unit interprocessor communication delays. *Discrete Appl. Math. 18*, 55-71.

V.J. Rayward-Smith (1987B). The complexity of preemptive scheduling given interprocessor communication delays. *Inform. Process. Lett. 25*, 123-125.

M.S. Salvador (1970). A solution to a special class of flow shop scheduling problems, Proc. Symp. Theory of Scheduling and its Applications. In S.E. Elmaghraby (ed.) (1973). *Lecture Notes in Economics and Mathematical Systems 86*, Springer, Berlin, 83-91.

V. Sarkar (1989). *Partitioning and scheduling parallel programs for multiprocessors*, Pitman, London.

A. Schrijver (1986). *Theory of linear and integer programming*, Wiley, Chichester.

J.T. Schwartz (1980). Ultracomputers. *ACM Trans. Programming Languages and Systems 2*, 484-521.

X. Shen, E.M. Reingold (1991). Scheduling on a hypercube. *Inform. Process. Lett. 40*, 323-328.

P.C. Treleaven, D.R. Brownbridge, R.P. Hopkins (1982). Data-driven and demand-driven computer architecture. *Comput. Surveys 14*, 93-143.

T.A. Varvarigou, V.P. Roychowdhury, T. Kailath (1992). Unpublished manuscript.

B. Veltman, B.J. Lageweg, J.K. Lenstra (1990). Multiprocessor scheduling with communication delays. *Parallel Comput. 16*, 173-182.

B. Veltman, B.J. Lageweg, J.K. Lenstra (1993). *Tosca: a tunable off-line scheduling algorithm*, in preparation.

Y. Zhu, M Ahuja (1990A). Preemptive job scheduling on a hypercube. *Proc. Int. Conf. Parallel Process.*, 301-304.

Y. Zhu, M Ahuja (1990B). Job scheduling on a hypercube. *Proc. Int. Conf. Distributed Comput. Systems*, 510-517.

Samenvatting

Vele processoren maken licht werk moet het motto geweest zijn om parallelle computers te gaan ontwerpen en bouwen. Een *processor* is een rekenmachine die slechts één operatie tegelijkertijd kan uitvoeren. Een *parallelle computer* of *multiprocessor* is opgebouwd uit verscheidene processoren en kan dus een aantal taken tegelijkertijd verwerken.

De *taken* die een parallelle computer te verwerken krijgt zijn de modules waarin een parallel programma gepartitioneerd is. Tussen zulke taken bestaan in het algemeen *informatie-afhankelijkheden*, zodat zij niet zonder meer in iedere willekeurige volgorde verwerkt kunnen worden. Een *schedule* bepaalt voor elke taak het tijdstip waarop en de processoren waardoor deze uitgevoerd zal worden. Het streven is een snelle verwerking van de taken te garanderen.

In principe kan men een optimaal schedule vinden door te kiezen uit de aftelbare en vaak eindige verzameling van alternatieven. Dit suggereert dat botweg *compleet aftellen* effectief zou zijn: genereer alle mogelijke oplossingen, bepaal hun kosten en kies een beste. Helaas is het aantal oplossingen vaak zo groot dat deze methode in de praktijk te veel tijd vergt. Men is gedwongen te zoeken naar snellere algoritmen. De fundamentele vraag is of er een algoritme bestaat dat een gegeven schedulingprobleem optimaal oplost in *polynomiale tijd*. Als dit het geval is dan beschouwen we zo'n algoritme als 'snel' en is het probleem 'goed oplosbaar'. Voor andere problemen kan men aantonen dat het zeer onwaarschijnlijk is dat zo'n algoritme bestaat; dit zijn de NP-*lastige* problemen. In de hoofdstukken 3-6 van dit proefschrift behandelen we de complexiteit van een aantal schedulingproblemen die optreden bij het gebruik maken van een multiprocessor. Deze problemen verschillen van hun klassieke varianten op een aantal punten. Wij hebben theoretisch onderzoek gedaan naar deze verschillen en hun consequenties.

Ten eerste dient men bij multiprocessorscheduling rekening te houden met *communicatievertragingen*. De informatie-afhankelijkheden leiden op natuurlijke wijze tot een precedentiestructuur op de taakverzameling; de verwerking van een taak kan pas beginnen als zijn voorgangers verwerkt zijn en als alle informatie aanwezig is op de processoren die de desbetreffende taak uit zullen voeren. In Hoofdstuk 3 bestuderen we het meest eenvoudige model met communicatievertragingen: elke taak vergt slechts één processor voor executie en zowel verwerkingen als communicatievertragingen nemen één tijdseenheid in beslag. In het algemeen geldt dat zelfs dit eenvoudige model met communicatievertragingen al NP-lastig is.

Door middel van *taakduplicatie* kan men communicatievertragingen verkleinen of zelfs vermijden. Hoofdstuk 4 laat zien dat duplicatie een optimaal schedule kan verkorten met een factor gelijk aan ten hoogste het aantal processoren waaruit de multiprocessor bestaat. In het voornoemde geval van verwerkingstijden en communicatievertragingen van één tijdseenheid lengte kan duplicatie slechts een factor twee helpen.

Een derde aspect betreft problemen waarbij het mogelijk is dat een taak verscheidene processoren vergt voor executie. Zulke taken noemen we *multiprocessortaken*. Hoofdstuk 5 behandelt de complexiteit van het toewijzen van starttijden aan multiprocessortaken, waarbij voor elke taak een *van tevoren vastgestelde verzameling processoren* is bepaald. Communicatie wordt nu buiten beschouwing gelaten. In het algemeen geldt dat zelfs het vinden van alleen starttijden een NP-lastig probleem is.

In Hoofdstuk 6 beschouwen we, evenals in de hoofdstukken 3-4, taken die slechts één processor voor executie vergen. De informatie-afhankelijkheden zijn zodanig dat de precedentiestructuur uit losse ketens van elk twee taken bestaat. De taken zonder voorgangers kunnen door twee processoren verwerkt worden, maar de taken met voorgangers worden allen door een en dezelfde processor uitgevoerd. Het vinden van een schedule dat de maximale voltooiingstijd minimaliseert is een NP-lastig probleem, ook als men *preemptie* toestaat. Preemptie is het onderbreken van de executie van een taak om deze op een eventueel later tijdstip op dezelfde of een andere processor voort te zetten. Dit probleem is een variant van het klassieke flow shop schedulingprobleem.

Uit de analyses van hoofdstukken 3-6 concluderen we dat het onwaarschijnlijk is dat er een snel algoritme bestaat om het algemene schedulingprobleem, met communicatievertragingen en multiprocessortaken, op te lossen. Men is genoodzaakt een benaderend algoritme te gebruiken. *Tosca*, een ‘tunable off-line scheduling algorithm’, belichaamt zo’n methode. Tosca is ontworpen om het verwerken van parallelle programma’s op gedistribueerde systemen te ondersteunen. Tosca kan gebruikt worden om prestaties van een programma in ontwikkeling te voorspellen; de kwaliteit van een schedule is een maat voor de kwaliteit van een gegeven decompositie van zo’n programma.

Hoofdstuk 7 geeft een beschrijving van de methodiek van Tosca. Tosca tracht een goed schedule te construeren binnen een aanvaardbaar tijdsbestek door middel van *begrensde aftelling*. In principe kan men een schedule construeren door de taken een voor een te voorzien van een starttijd en een processortoewijzing. De verschillende keuzemogelijkheden kunnen gerepresenteerd worden door middel van een aftellingsboom. Het proces van begrensde aftelling beschouwt, in tegenstelling tot complete aftelling, slechts een deel van deze boom. Het proces bestaat uit een aantal stappen. Per stap worden een taak en een processortoewijzing bepaald. Om deze te kunnen bepalen wordt een deelboom berekend. Drie parameters, twee prioriteitsregels en één ondergrensregel bepalen de vorm van deze deelboom. De bladeren van de deelboom worden geëvalueerd met behulp van een evaluatieregel. Een taak-processortoewijzing combinatie die een tak vastlegt waarin een blad van minimale waarde zit, wordt gekozen. Met behulp van de drie parameters geeft men een bovengrens aan de diepte en de breedte van de deelboom. Eén van de twee prioriteitsregels betreft de keuze van taken, de ander betreft de keuze van processortoewijzingen. Deze regels kunnen gekozen worden uit een gegeven verzameling of zijn door de gebruiker gemaakt. De gebruiker moet daarnaast een ondergrensregel en een evaluatieregel kiezen. Tosca is *regelbaar* daar het de gebruiker zeggenschap geeft over de snelheid van de oplossingsmethode en de kwaliteit van de geproduceerde schedules.

Tosca is voorzien van een eenvoudige gebruikersinterface. Informatie wordt gepresenteerd op alfanumerieke wijze en de mens-machine interactie verloopt met behulp van menu's. We hebben Tosca getest op vier typen probleeminstanties: gelaagde precedentierelaties, serie-parallelle precedentierelaties, willekeurige precedentierelaties en twee precedentierelaties uit de praktijk. Bij de precedentierelaties genereerden we verwerkingstijden en informatie-afhankelijkheden. Hoofdstuk 8 beschrijft de gebruikersinterface, de probleemgeneratoren en de testresultaten.

Uit de testresultaten blijkt dat in het algemeen *list scheduling*, waarbij in elke stap precies één taak en één processor toewijzing bekeken wordt, tamelijk goed werkt. Geavanceerdere vormen van begrensde aftelling nemen al vlug veel tijd in beslag en vinden slechts marginale verbeteringen. Het vinden van een optimaal schedule is, behalve voor kleine probleeminstanties, een hopeloze zaak.

Ofwel, het is niet eenvoudig om vele processoren licht werk te laten maken.

Curriculum vitae

Bart Veltman werd geboren op 14 december 1963 te Reinheim (BRD). In 1982 behaalde hij het VWO-diploma aan het Elzendaalcollege te Boxmeer. Hij studeerde vanaf 1982 wiskunde aan de Katholieke Universiteit Nijmegen en voltooide deze studie met het doctoraal diploma in 1987. Hij trad aansluitend als onderzoeker in opleiding in dienst van de Stichting Mathematisch Centrum en was tot december 1992 werkzaam bij het Centrum voor Wiskunde en Informatica te Amsterdam. Hij was in de periode 1989-1991 tevens verbonden aan de Nederlandse Organisatie voor Toegepast Natuurwetenschappelijk Onderzoek te Delft en betrokken bij het ParTool project aldaar. Sinds december 1992 is Bart als universitair docent in dienst van de Technische Universiteit Eindhoven.

STELLINGEN

behorende bij het proefschrift

Multiprocessor scheduling with communication delays

van

Bart Veltman

I

Beschouw het coöperatieve spel (N,v), met spelersverzameling $N=\{1,\dots,n\}$ en waardefunctie $v:2^N\rightarrow\mathbb{R}$, dat voldoet aan

(1) $v(\{j\})=0$ voor alle $j\in N$,
(2) v is superadditief, en
(3) $v(S)=\Sigma_{T\in S/\pi}\, v(T)$ voor alle $S\in 2^N$.

Hier is π een permutatie van de spelersverzameling N en S/π de verzameling van componenten van S onder deze permutatie. Voor zo'n coöperatief spel kan men de uitbetalingsregel $\beta:N\rightarrow R$ definiëren door

$$\beta(j)=\tfrac{1}{2}(v(P_j\cup\{j\})-v(P_j)+v(F_j\cup\{j\})-v(F_j)) \text{ voor } j\in N.$$

Hier zijn P_j en F_j respectievelijk de voorgangers en opvolgers van j onder π. Deze uitbetalingsregel garandeert de stabiliteit van de grote coalitie N, zodat het spel (N,v) gebalanceerd is.

I.J. Curiel, J.A.M. Potters, V. Rajendra Prasad, S.H. Tijs, B. Veltman (1993). Sequencing and coöperation. *Oper. Res.*, te verschijnen.

II

Picouleau [1992] presenteert reducties om de NP-lastigheid aan te tonen van de volgende problemen:

(1) $P\,|prec,c=1,p_j=1\,|\,C_{\max}$: een verzameling van n taken moet worden verwerkt door m processoren rekening houdend met informatie-afhankelijkheden en zowel verwerkingstijden als communicatievertragingen van één tijdseenheid lengte; bepaal een rooster zodanig dat de maximale voltooiingstijd wordt geminimaliseerd;
(2) $P\,|prec,c=1,dup,p_j=1\,|\,C_{\max}$: de variant waarbij duplicatie is toegestaan;
(3) $P2\,|prec,c=1,fix,p_j=1\,|\,C_{\max}$: de variant waarbij de taken verdeeld zijn over twee processoren;
(4) $\bar{P}\,|\,tree,c\,|\,C_{\max}$: de variant waarbij informatie-afhankelijkheden leiden tot een precedentierelatie in de vorm van een boom, met een onbeperkt aantal processoren, constante communicatievertragingen en willekeurige verwerkingstijden.

Zijn eerste drie reducties zijn niet polynomiaal; zijn vierde reductie is incorrect om een andere reden.

C. Picouleau (1992). *Etude de problèmes d'optimisation dans les systèmes distribués*. Thèse de doctorat, Université Paris VI, Paris.

III

De in stelling II genoemde problemen zijn NP-lastig.

J.A. Hoogeveen, J.K. Lenstra, B. Veltman (1992). *Three, four, five, six, or the complexity of scheduling with communication delays*, Report BS-R9229, CWI, Amsterdam.

J.A. Hoogeveen, S.L. van de Velde, B. Veltman (1993). Complexity of scheduling multiprocessor tasks with prespecified processor allocations. *Discrete Appl. Math.*, te verschijnen.

A. Jakoby, R. Reischuk (1992). The complexity of scheduling problems with communication delays for trees. *Proc. Skandinavian Workshop on Algorithmic Theory 3*, 165-177.

IV

Het flow shop schedulingprobleem met twee fasen en in de tweede fase twee identieke parallelle machines, $F2(1,P2)\,||\,C_{\max}$, en de variant waarbij preemptie is toegestaan, $F2(1,P2)\,|pmtn\,|\,C_{\max}$, zijn sterk NP-lastig.

J.A. Hoogeveen, J.K. Lenstra, B. Veltman (1993). *Minimizing makespan in a multiprocessor flow shop is strongly* NP-*hard*, in voorbereiding.

B. Veltman (1993). Dit proefschrift, hoofdstuk 6.

V

Schedulingtheorie levert een goed raamwerk voor het analyseren van een parallel programma, mits het model waarmee zo'n programma beschreven wordt in hoge mate architectuuronafhankelijk is.

VI

Het dupliceren van taken kan in theorie een schedule sterk verkorten maar zal in praktijk meestal weinig effect hebben.

VII

Het is onmogelijk om met behulp van een rondbreinaald een naadloze Möbiusband te breien. Wel kunnen er draaiingen in het breiwerk optreden, maar door goed op te passen is dit te voorkomen.

VIII

Cultuur is het geheel aan vormen waarin een object zich tot de omgeving, de natuur, verhoudt. Natuur is de omgeving waarbinnen cultuurvormen aanwezig zijn of gecultiveerd worden. Natuur en cultuur bestaan niet los van elkaar. Wat voor de één cultuur is, is voor de ander natuur. Cultuur is niet typisch menselijk.

IX

Veel motorrijders zullen de eerste keer dat zij met een leeg zijspan een bocht naar rechts maken verrast worden door het omhoogkomende bakje.