Scheduling with communication for multiprocessor computation

Scheduling met communicatie voor multiprocessor berekeningen

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van
doctor aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. dr. H.O. Voorma,
ingevolge het besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 10 juni 1998 des ochtends te 10.30 uur

door

Jacobus Hendrikus Verriet

geboren op 3 oktober 1970
te Ubbergen

Promotor:	Prof. dr. J. van Leeuwen
	Faculteit Wiskunde & Informatica
Co-promotor:	Dr. M. Veldhorst
	Faculteit Wiskunde & Informatica

ISBN 90-393-2025-X

Contents

Introduction

1 Introduction

Scheduling is concerned with the management of resources that have to be allocated to activities over time subject to a number of constraints. A (feasible) schedule is an allocation of the resources to the activities that satisfies all constraints. The objective of scheduling is finding a schedule that is optimal with respect to a certain objective function. The resources that have to be allocated and the constraints that have to be satisfied can be of various types. Hence many real-life problems can be viewed as scheduling problems.

Crew scheduling. An airline company must allocate personnel (pilots and flight attendants) to flights, such that the number of pilots and flight attendants is sufficient on each flight, each employee has a (flight-dependent) period of time off between two flights and each employee returns home regularly. An objective of crew scheduling could be minimising the number of employees and equally dividing the working hours among the personnel.

Classroom scheduling. A school has to allocate teachers and classrooms to courses, such that no teacher is in two classrooms at the same time, no course gets assigned two teachers or two classrooms, no teacher works more than seven hours on one day and no student has more than seven courses on one day. The objective of classroom scheduling could be minimising the total amount of time that the teachers and the students have to be at school.

Vehicle routing. A transport company must allocate trucks to goods that have to be transported, such that the volume of the goods on one truck does not exceed its capacity, all trucks return at their depot at the end of each day, no truck driver works more than eight hours on one day and all goods are loaded and unloaded during office hours. The objective of vehicle routing could be minimising the number of trucks.

In general, a scheduling problem assumes the existence of a set of *operations* (the activities) and a set of *machines* (the resources). The machines have to be assigned to the operations over time subject to a number of constraints.

Machine scheduling. A machine must be allocated to each operation, such that no machine is assigned to two operations at the same time and exactly one machine is assigned to each operation.

All scheduling problems are generalisations of the machine scheduling problem. For example, in crew scheduling, the personnel corresponds to the machines and the flights to the operations.

This thesis is concerned with multiprocessor scheduling, the problem of executing a computer program on a parallel computer.

Multiprocessor scheduling. The processors of a parallel computer have to be allocated to the tasks of a computer program, such that no processor executes two tasks at the same time and every task is executed exactly once.

Usually, a multiprocessor schedule has to satisfy some additional constraints.

Multiprocessor scheduling is a generalisation of machine scheduling: the processors correspond to the machines and the tasks to the operations.

1.1 Communication in parallel computers

This thesis is concerned with multiprocessor scheduling with communication. This is an essential aspect of the problem of executing a computer program on a parallel computer. A computer program can be seen as a collection of instructions. These include assignments, arithmetic instructions, conditional statements, loop statements and subroutine calls. We will assume that the instructions are combined into clusters. These clusters of instructions will be called *tasks*.

A parallel computer can be viewed as a collection of *processors* and *memories* and a *communication mechanism*; in this thesis, we will not consider the other components of a parallel computer. The processors are used to execute the tasks of a computer program. The memories are used to store data. The communication mechanism is used to transfer data between the components (processors and memories) of the parallel computer.

There are two types of parallel computer that differ in the way memory is used. In a *distributed memory computer*, each processor has a local memory. The processors of a distributed memory computer are connected by a communication network, but are not a part of this network. In *shared memory computers*, there is a global memory that is used by all processors.

The communication mechanisms of these computers are different. In both models, a data transfer can be viewed as a sequence of *communication operations*. In a shared memory computer, data is transferred from a source processor to a destination processor by writing and reading in shared memory. A data transfer consists of a *write operation* followed by a *read operation*. The source processor writes the data in a memory location, after which it can be read by the destination processor. The write operation does not interfere with the availability of the destination processor. Similarly, the source processor is not involved in the execution of the read operation. Because simultaneous access of a memory location by two processors is not allowed, the duration of the write and read operations depends on the number of processors that want to access the same memory location simultaneously.

In a distributed memory computer, data is transferred by sending messages from one processor to another through the communication network. In such computers, a data transfer consists of three communication operations: a *send operation*, a *transport operation* and a *receive operation*. The send operation is executed by the source processor; the send operation submits a message to the communication network. The transport operation is used to transport a message over the connections in the communication network from the source processor to the destination processor. No processor is involved in the execution of the transport operation. After a message has been transported, the destination processor can obtain the data from the message by executing a receive operation. The duration of the send and receive operations depends on the size of a message. The duration of a transport operation varies with the size of the message, the distance between the source and the destination processor, the capacity of the connections in the communication network and the number of messages that reside in the communication network.

1.2 Multiprocessor scheduling

During the execution of a computer program on a given input, each task has to be executed by one processor and the duration of its execution depends on the input. Some of the tasks have to be executed in a specified order, because the result of a task may be needed to execute other tasks. Such tasks will be called *data dependent*. Other tasks can be executed in an arbitrary order or simultaneously on different processors of a parallel computer. If two data-dependent tasks are executed on different processors, then the result of the first task must be transported to the processor that executes the other task using the communication mechanism.

Multiprocessor scheduling can be viewed as a generalisation of the machine scheduling problem. The machines are the processors and the components of the communication mechanism of the parallel computer. The operations are the tasks and the communication operations. Processors and components of the communication mechanism have to be allocated to each task and each communication operation for some period of time. Each task and every send and receive operation has to be assigned a processor on which it is executed. The write and read operations have to be allocated a processor and a memory location that must be accessed. A sequence of connections in the communication network has to be assigned to every transport operation: these connections form the path over which the corresponding message is sent through the communication network.

An assignment of processors and components of the communication mechanism to the tasks and the communication operations has to satisfy many constraints. Usually,

1. no processor can execute two tasks or communication operations at the same time;
2. data-dependent tasks cannot be executed at the same time;
3. if two data-dependent tasks are executed on different processors, then a data transfer must be executed between these tasks;
4. if communication is modelled by writing and reading messages in shared memory, then
   (a) no shared memory location can be accessed by two processors at the same time; and
   (b) a task cannot be executed until all data for this task is read by the processor on which it is executed; and
5. if communication between the processors is modelled by sending messages through a communication network, then
   (a) the number of messages sent over a connection of the network at the same time may not exceed the capacity of the connection; and
   (b) a task cannot be executed until all messages required for this task are received by the processor on which it is executed.

Apart from the large number of constraints that need to be satisfied, there are also many objective functions that could be minimised or maximised. The most common of these is the minimisation of the *makespan*, the duration of the execution of the computer program.

1.3 Models of parallel computation

Because of the large number of different constraints in multiprocessors scheduling and the great variety of parallel computer architectures, it is difficult to design efficient algorithms that construct good multiprocessor schedules. This is the reason to introduce an abstract model of a parallel computer, a *model of parallel computation*. In such a model, one can concentrate on those aspects in multiprocessor scheduling that have a large impact on the objective function (for instance, the makespan). A good model of parallel computation helps to understand the essence of the problem of multiprocessor scheduling with communication.

If the duration of the tasks is large compared to the duration of the communication operations, then the impact of communication on most objective functions is small. For such problems, we can use a model of parallel computation in which all communication constraints are removed. In this model, the duration of the communication operations is assumed to be negligible. A schedule for a computer program in this model is an allocation of processors over time, such that no processor executes two tasks at the same time and data-dependent tasks are executed in the right order. This is the most common scheduling model. Lawler et al. [60] give an overview of the work on scheduling without communication requirements subject to many additional constraints and several objective functions.

In a real parallel computer, sending a message through the communication network or accessing a shared memory location is a very costly operation compared to a simple arithmetic operation. So the communication-free model of parallel computation does not capture the complexity of parallel computation. Many other models have been presented that incorporate communication in some way. An overview of such models is presented in the remainder of this section. The communication constraints of the models based on shared memory parallel computers are described in Section 1.3.1 and those of the models based on distributed memory computers in Section 1.3.2. Guinand [40] and Juurlink [51] have presented more elaborate overviews of models of parallel computation.

1.3.1 Shared memory models

Most shared memory models are generalisations of the Parallel Random Access Machine introduced by Fortune and Wyllie [28]. The PRAM is the most common model of parallel computation. A PRAM consists of an infinite collection of identical processors that each have an unlimited amount of local memory. The processors execute a computer program in a synchronous manner: all processors start a task or a communication operation at the same time. The processors communicate by writing and reading in shared memory. Two processors can read the same memory location simultaneously, but a memory location cannot be written by one processor and written or read by another processor at the same time. This model of parallel computation is also called the Concurrent Read Exclusive Write PRAM. Snir [82] introduced two variants of the PRAM model: the Exclusive Read Exclusive Write PRAM in which no simultaneous access of the same memory location is allowed, and the Concurrent Read Concurrent Write PRAM in which a memory location can be read or written by several processors at the same time.

The PRAM model does not capture the complexity of communication in the execution of computer programs: a communication operation has the same duration as the execution of a computation instruction whereas in a real parallel computer, a communication operation is far more time consuming. There are several PRAM-based models of parallel computation that include other aspects of real parallel computers. Asynchronous variants of the PRAM were presented by Cole and Zajicek [15, 16] and by Gibbons [34]. In an asynchronous PRAM, the processors need not start the execution of an instruction or a communication operation simultaneously. Hence processors executing a simple arithmetic instruction do not have to wait for processors that are reading or writing in shared memory.

Most PRAM-based models of parallel computation include a more realistic representation of shared memory access than the PRAM itself. The Delay PRAM introduced by Martel and Raghunatham [65] and the Local-Memory PRAM of Aggarwal et al. [3] extend the PRAM model by including a latency for shared memory access. In these models, the duration of a communication operation is fixed and larger than the duration of an arithmetic operation. The Queue Read Queue Write PRAM presented by Gibbons et al. [35, 36] includes memory contention: it is allowed to access the same shared memory location simultaneously, but the duration of a memory access depends on the number of processors that want to read or write the same memory location. In the Block Parallel PRAM of Aggarwal et al. [2], accessing a consecutive block of shared memory locations is less time consuming than separately accessing these memory locations: the duration of a write or read operation equals the sum of a fixed latency and a function linear in the number of consecutive memory locations that must be accessed.

1.3.2 Distributed memory models

In the execution of a computer program on a distributed memory computer, each task is executed by one processor and messages are sent through the communication network. For each pair of data-dependent tasks scheduled on different processors, one needs to assign a path through the communication network that will be used to send messages. This is known as routing. In this thesis, the problem of routing will be ignored.

The simplest model of parallel computation based on a distributed memory parallel computer is a model in which the communication network is a complete graph (there is a direct connection between every pair of processors) and each connection in the communication network has an unbounded capacity. In this model, transporting a message from one processor to another takes a fixed amount of time. The communication is represented by the duration of the transport operations only; the duration of the send and receive operations is assumed to be zero. For multiprocessor scheduling, this is the most common model of parallel computation that does not neglect the communication costs. It was introduced by Rayward-Smith [79]. An overview of scheduling problems in this model is given by Chrétienne and Picouleau [13].

This basic model has been generalised in several ways. Papadimitriou and Yannakakis [75] assume that the fixed duration of the transport operations depends on the topology of the communication network. Finta and Liu [25, 26] and Picouleau [78] add an overall capacity constraint: the number of messages that can be sent through the communication network at the same time is bounded. Kalpakis and Yesha [52, 53], Cosnard and Ferreira [19] and Bampis et al. [6] consider models of parallel computation in which the communication network is not a complete graph: the duration of transport operations in such networks depends on the distance between the communicating processors.

Most models of parallel computation include only one or two aspects of real parallel computers, but some include more aspects. These models are all architecture independent and characterise the execution of computer programs on a real parallel computer by a small number of parameters. The Bulk Synchronous Parallel model was introduced by Valiant [85]. The BSP model is a synchronous model of parallel computation in which the synchronisation costs are not neglected. These costs are modelled by a communication latency. In addition, the number of messages that can be sent at the same time is bounded by the throughput of the communication network, and the duration of send or receive operations is not negligible.

The Postal model was introduced by Bar-Noy and Kipnis [7]. It includes communication overheads and communication latencies: the send and receive operations have unit length and the transport operations have a fixed duration that depends on the network topology.

The LogP model was introduced by Culler et al. [21]. The LogP model is named after its parameters: the latency L, the overhead o, the gap g and the number of processors P. The LogP model is more general than the Postal model. Like in the Postal model, the transport operations in the LogP model have a fixed duration that depends on the topology of the communication network. Sending and receiving a message of unit size takes a fixed amount of time. The bandwidth of a parallel computer is modelled as well: there is a minimum delay between two consecutive send and receive operations executed on the same processor.

1.4 An overview of the thesis

This thesis consists of four parts: an introductory part, two main parts and a concluding part. The introductory part consists of Chapters 1 and 2. In these chapters, the terminology and notation used in the main parts are presented. The two main parts (Parts I and II) are concerned with scheduling in two different models of parallel computation and subject to two different objective functions. These parts are self-contained and can therefore be read separately. The concluding part consists of Chapter 12.

Part I consists of Chapters 3, 4, 5, 6 and 7. In these chapters, we study the problem of constructing minimum-tardiness schedules in the Unit Communication Times model, the model of parallel computation in which communication is represented by a latency of unit length. The computer programs that are to be scheduled in this model consist of tasks that have been assigned a deadline. The UCT model is introduced in Chapter 3. In the remaining chapters of Part I, we present several algorithms that construct minimum-tardiness schedules (schedules in which the maximum amount of time by which a deadline is exceeded is as small as possible) for special classes of data dependencies.

Part II is concerned with the problem of constructing minimum-length schedules in the LogP model. This part consists of Chapters 8, 9, 10, and 11. Chapter 8 is used to introduce the LogP model. In the remaining chapters of Part II, the complexity of constructing minimum-length schedules in the LogP model is studied. It is proved that this problem is NP-hard even for a restricted class of data dependencies. Moreover, in Part II, we present the first approximation algorithms with a constant approximation ratio for scheduling two special classes of data dependencies in the LogP model.

2 Preliminaries

In this chapter, the general notation in multiprocessor scheduling and some preliminary results are presented. In Section 2.1, we present the terminology for precedence graphs that will be used throughout this thesis. Section 2.2 presents the general scheduling instances. The general notion of a schedule is given in Section 2.3. In Section 2.4, the notion of approximation algorithms for scheduling is presented. Special classes of precedence graphs and the properties of these classes of precedence graphs are presented in Section 2.5.

2.1 Precedence graphs

In the execution of a computer program on a parallel machine, each task of the program is executed by exactly one of the processors. The tasks can often not be executed in an arbitrary order: the result of a task may be needed by other tasks. If the result of task u_1 is needed to execute task u_2, then the execution of u_1 must be completed before the execution of u_2 can start. If the execution of u_2 does not require the result of u_1, then u_1 and u_2 can be executed in arbitrary order or at the same time on different processors.

The tasks of a computer program and their data dependencies will be represented by a precedence graph.

Definition 2.1.1. A *directed graph* is a tuple $G = (V,E)$, where V is a set of *nodes* and $E \subseteq V \times V$ is a set of *arcs* between the nodes. An arc is a pair of two nodes of V: the pair (u_1,u_2) denotes the arc from u_1 to u_2. A directed graph $G = (V,E)$ is called a *precedence graph* or *directed acyclic graph* if there is no sequence of arcs $(u_1,u_2),(u_2,u_3),\ldots,(u_k,u_1)$ in E for any $k \geq 1$.

Let $G = (V,E)$ be a precedence graph. A node from V corresponds to a task from the computer program. An arc from one node to another represents a data dependency between the corresponding tasks: if there is an arc from node u_1 to node u_2, then the result of the task corresponding to u_1 is needed to execute the task that corresponds to u_2. Since there is a one-to-one correspondence between the tasks of a computer program and the nodes in a precedence graph, we will use the term task for the nodes in a precedence graph.

Let G be a precedence graph. The set $V(G)$ denotes the set of tasks of G and $E(G)$ the set of arcs of G. Throughout this thesis, we will assume that $V(G)$ contains n tasks and $E(G)$ contains e arcs. A *path* in G is a sequence of $k \geq 2$ tasks $u_1,u_2,\ldots,u_k$ of G, such that G contains an arc from u_i to u_{i+1} for all $i \in \{1,\ldots,k-1\}$. From the definition of precedence graphs, there are no paths in G from a task to itself. The *length* of a path is the number of tasks on the path. The *height* of G is the length of a longest path in G.

Let u_1 and u_2 be two tasks of G. u_1 is called a *predecessor* of u_2 if there is a path in G from u_1 to u_2. In that case, u_2 is called a *successor* of u_1, which is denoted by $u_1 \prec_G u_2$. The sets of predecessors and successors of a task u of G are denoted by $Pred_G(u)$ and $Succ_G(u)$, respectively. Tasks without successors will be called *sinks* and tasks without predecessors will be called *sources*. u_2 is called a *child* of u_1 if (u_1,u_2) is an arc of G. If u_2 is a child of u_1, then

u_1 is called a *parent* of u_2. This is denoted by $u_1 \prec_{G,0} u_2$. The sets $Pred_{G,0}(u)$ and $Succ_{G,0}(u)$ contain the parents and children of u, respectively. The number of children of a task u is the *outdegree* of u; its *indegree* equals the number of parents of u. It is not difficult to prove that $\sum_{u \in V(G)} |Pred_{G,0}(u)| = \sum_{u \in V(G)} |Succ_{G,0}(u)| = |E(G)|$.

Two tasks u_1 and u_2 of G are called *incomparable* if neither $u_1 \prec_G u_2$, nor $u_2 \prec_G u_1$. Otherwise, they are called *comparable*. The *width* of G is the maximum number of pairwise incomparable tasks of G. Consequently, if G is a precedence graph of width w, then every subset of $V(G)$ with at least $w+1$ elements contains at least two comparable tasks. A *chain* in G is a set of pairwise comparable tasks of G. Note that the tasks on a path in G form a chain and that the size of a maximum-size chain in G equals its height. A set of pairwise incomparable tasks is called an *anti-chain* in G. So the width of G equals the size of a maximum-size anti-chain in G.

A *topological order* of a precedence graph G is a list containing all tasks of G, such that each task has a smaller index in the list than its successors. There is a topological order of all precedence graphs. A topological order of G can be constructed in $O(n+e)$ time [18].

The *transitive closure* of G is a precedence graph G^+, such that $V(G^+) = V(G)$ and $E(G^+) = \{(u_1, u_2) \mid u_1 \prec_G u_2\}$. Hence the transitive closure of G contains an arc from every task of G to each of its successors. The *transitive reduction* of G is a precedence graph G^-, such that $V(G^-) = V(G)$ and for all tasks u_1, u_2 and u_3 of G, $u_1 \prec_G u_2$ if and only if $u_1 \prec_{G^-} u_2$ and if $u_1 \prec_G u_2$ and $u_2 \prec_G u_3$, then (u_1, u_3) is not an arc of G^-. Throughout this thesis, e^- equals the number of arcs of the transitive reduction of G and e^+ the number of arcs in the transitive closure of G. A transitive closure or a transitive reduction of G can be constructed in $O(\min\{n^{2.376}, n+e+ne^-\})$ time [17, 37]. Transitive closures and transitive reductions of precedence graphs will be used to obtain more efficient implementations of algorithms.

Let U be a set of tasks of a precedence graph G. The *subgraph of G induced by U* is the precedence graph $(U, E(G) \cap (U \times U))$. This precedence graph is denoted by $G[U]$. A precedence graph H is called a *subgraph* of G, if there is a subset U of $V(G)$, such that $G[U]$ equals H. A *prefix* of a precedence graph G is a subset U of $V(G)$, such that for all tasks u_1 and u_2 of G, if $u_2 \in U$ and $u_1 \prec_G u_2$, then $u_1 \in U$.

2.2 General scheduling instances

During the execution of a computer program, the duration of the execution of a task depends on the input of the computer program. A function μ is used to specify the execution length of every task of the computer program for a given input: for each task u of G, $\mu(u)$ is the duration of the execution of u. Hence a computer program (for a given input) will be represented by a tuple (G, μ), where G is a precedence graph and $\mu : V(G) \to \mathbb{Z}^+$ is a function that assigns an *execution length* or *task length* to every task of G. We will assume that μ is also used to denote the total execution time of a precedence graph or a set of tasks. So if U is a set of tasks of G, then $\mu(U) = \sum_{u \in U} \mu(u)$. In addition, $\mu(G) = \mu(V(G)) = \sum_{u \in V(G)} \mu(u)$.

A *general scheduling instance* is represented by a tuple (G, μ, m), such that (G, μ) corresponds to a computer program and $m \in \{2, 3, \ldots, \infty\}$ equals the number of processors that is available for the execution of this computer program. If $m = \infty$, then the number of available processors is unrestricted. Since we assume that every task is executed by exactly one processor, instances (G, μ, ∞) correspond to instances (G, m, n). We will not consider instances $(G, \mu, 1)$, because the scheduling problems that will be studied in this thesis are easily solvable on one processor.

2.3 Communication-free schedules

A schedule for a computer program corresponds to the execution of the computer program on a parallel machine for a given input. A schedule assigns a starting time and a processor to all tasks.

Definition 2.3.1. A *schedule* for a scheduling instance (G, μ, m) is a pair of functions (σ, π), such that $\sigma : V(G) \to \mathbb{N}$ and $\pi : V(G) \to \{1, \ldots, m\}$.

Consider a schedule (σ, π) for an instance (G, μ, m). σ is an *assignment of starting times* and π an *assignment of processors*. $\sigma(u)$ represents the *starting time* of u and $\pi(u)$ the processor on which u is executed. A task u is said to be *scheduled* at time $\sigma(u)$ on processor $\pi(u)$. Each task has exactly one starting time. So duplication of tasks is not allowed. u starts at time $\sigma(u)$ and is completed at time $\sigma(u) + \mu(u)$, its *completion time*. Preemption is not allowed: the execution of u cannot be interrupted and resumed at a later time. u is said to be executed at time t on processor $\pi(u)$ for all times t, such that $\sigma(u) \leq t \leq \sigma(u) + \mu(u) - 1$. A processor is called *idle* at time t if no task is executed at time t on that processor.

A feasible schedule is a schedule in which no processor executes two tasks at the same time and the comparable tasks are executed in the right order.

Definition 2.3.2. A schedule (σ, π) for (G, μ, m) is called a *feasible communication-free schedule* or *feasible schedule* for (G, μ, m) if for all tasks $u_1 \neq u_2$ of G,

1. if $\pi(u_1) = \pi(u_2)$, then $\sigma(u_1) + \mu(u_1) \leq \sigma(u_2)$ or $\sigma(u_2) + \mu(u_2) \leq \sigma(u_1)$; and
2. if $u_1 \prec_G u_2$, then $\sigma(u_1) + \mu(u_1) \leq \sigma(u_2)$.

The first constraint states that no processor can execute two tasks at the same time. The second ensures that a task is scheduled after its predecessors.

Example 2.3.3. Consider the instance $(G, \mu, 2)$ shown in Figure 2.1. Every task of G is labelled with its name and its execution length. A schedule (σ, π) for $(G, \mu, 2)$ is shown in Figure 2.2: $\sigma(a_1) = 0$, $\sigma(a_2) = 0$, $\sigma(b_1) = 1$, $\sigma(b_2) = 2$, $\sigma(c_1) = 3$, $\sigma(c_2) = 3$ and $\sigma(d_1) = 6$. Moreover, $\pi(a_1) = \pi(b_1) = \pi(c_1) = \pi(d_1) = 1$ and $\pi(a_2) = \pi(b_2) = \pi(c_2) = 2$. It is not difficult to see that this is a feasible communication-free schedule for $(G, \mu, 2)$.

Let (σ, π) be a feasible (communication-free) schedule for (G, μ, m). The *length* or *makespan* of (σ, π) is the maximum completion time of a task of G; the makespan of (σ, π) equals $\max_{u \in V(G)}(\sigma(u) + \mu(u))$. (σ, π) is called a *minimum-length schedule* for (G, μ, m) if there is no feasible schedule for (G, μ, m) with a smaller length than (σ, π).

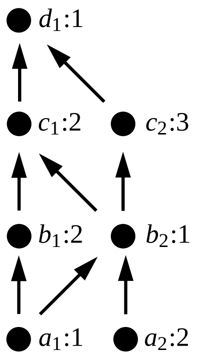

Figure 2.1. A general scheduling instance $(G,\mu,2)$

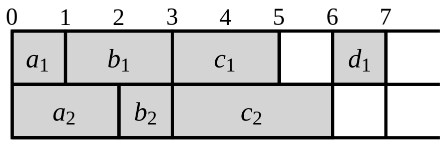

Figure 2.2. A feasible communication-free schedule for $(G,\mu,2)$

Feasible schedules in the UCT model and in the LogP model are defined in Chapters 3 and 8, respectively. Feasible schedules for these models of parallel computation can be viewed as feasible communication-free schedules. However, due to the communication requirements of the UCT model and the LogP model, a feasible communication-free schedule need not correspond to a feasible schedule in the UCT model or the LogP model.

2.4 Approximation algorithms

The goal of a scheduling problem is the construction of schedules that are optimal with respect to a certain *objective function*. For multiprocessor scheduling, the minimisation of the makespan is the most common objective. Lawler et al. [60] give an elaborate overview of scheduling problems and different objective functions.

Assume we want to minimise objective function f for a class of scheduling instances C. For each instance I in C, let $f^*(I) = \min\{f(\sigma,\pi) \mid (\sigma,\pi)$ is a feasible schedule for $I\}$. Let Algorithm A be an algorithm that constructs feasible schedules for all instances I in class C. Let $A(I)$ be the schedule for I constructed by Algorithm A. Let $\rho \in \mathbb{R}$, such that $\rho \geq 1$. Then Algorithm A is called a ρ-*approximation algorithm* if for all instances I in C, $f(A(I)) \leq \rho f^*(I)$. Algorithm A is called an *approximation algorithm with asymptotic approximation ratio* ρ if there is a positive integer N, such that for all instances I in C, if $f^*(I) \geq N$, then $f(A(I)) \leq \rho f^*(I)$. These notions of approximation algorithms correspond to those of Garey and Johnson [33]. If there is a non-negative constant $c \in \mathbb{R}$, such that $f_A(I) \leq \rho f^*(I) + c$ for all instances I in C, then Algorithm A is a $\rho + c$-approximation algorithm and an approximation algorithm with asymptotic ratio ρ.

2.5 Special precedence graphs

In this section, some properties of several special classes of precedence graphs are presented. Later in this thesis, algorithms will be presented that construct schedules (in the UCT model or in the LogP model) for precedence graphs from these classes.

2.5.1 Tree-like task systems

Tree-like task systems model divide-and-conquer computer programs, such as the evaluation of arithmetic expressions [10] and polynomial expressions [74]. We will consider two types of tree-like task systems: trees in which all tasks have at most one parent and trees in which all tasks have at most one child.

Definition 2.5.1. *Inforests* are precedence graphs in which every task has at most one child. An *intree* is an inforest that has exactly one sink. An *outforest* is an inforest in which the arcs have been reversed: an outforest is a precedence graph in which all tasks have at most one parent. An *outtree* is an outforest with exactly one source.

It is easy to see that an inforest is a collection of intrees and an outforest a collection of outtrees. The sinks of an inforest and the sources of an outforest will be called *roots*. The sources of an inforest and the sinks of an outforest will be called *leafs*. Tree-like task systems are sparse precedence graphs: a forest (an inforest or an outforest) with k roots contains exactly $n-k$ arcs.

An inforest (or intree) will be called a *d-ary inforest* (or *d-ary intree*) if all tasks have indegree at most d. Similarly, an outforest (or outtree) is called a *d-ary outforest* (or *d-ary outtree*) if all tasks have outdegree at most d.

Since in an inforest every task has at most one child, all successors of a task are comparable.

Observation 2.5.2. *Let G be an inforest. Let u_1, u_2 and u_3 be three tasks of G. If $u_1 \prec_G u_2$ and $u_1 \prec_G u_3$, then $u_2 \prec_G u_3$ or $u_3 \prec_G u_2$.*

Similarly, all predecessors of a task in an outforest are comparable.

Observation 2.5.3. *Let G be an outforest. Let u_1, u_2 and u_3 be three tasks of G. If $u_2 \prec_G u_1$ and $u_3 \prec_G u_1$, then $u_2 \prec_G u_3$ or $u_3 \prec_G u_2$.*

Let H be a subgraph of an inforest G. It is not difficult to see that H is also an inforest. H will be called a *subforest* of G. If H is an intree, then H will be called a *subtree* of G. Similarly, a subgraph of an outforest is an outforest and will also be called a subforest or a subtree.

In this thesis, we will also consider special tree-like task systems. For instance, we will consider precedence graphs that are both inforests and outforests. In such precedence graphs, every task has at most one child and at most one parent. These precedence graphs will be called *chain-like task systems*.

Moreover, in Chapter 9, send graphs are considered. A *send graph* is a precedence graph consisting of a source and its children. These children are the sinks of the precedence graph.

Receive graphs are considered in Chapter 10. A receive graph is a send graph in which the arcs have been reversed: a receive graph consists of a sink and its parents. Send and receive graphs are special instances of outtrees and intrees, respectively: a send graph is an outtree of height two and a receive graph is a intree of height two.

2.5.2 Interval orders

Unlike tree-like task systems, the class of *interval orders* or *interval-ordered tasks* is a class of precedence graphs that are not necessarily sparse.

Definition 2.5.4. A precedence graph G is called an *interval order* if for every task v of G, there is a (non-empty) closed interval $I(v) \subseteq \mathbb{R}$, such that for all tasks v_1 and v_2 of G,

$$v_1 \prec_G v_2 \quad \text{if and only if} \quad x < y \text{ for all } x \in I(v_1) \text{ and } y \in I(v_2).$$

Interval orders have a very nice property: the sets of successors of the tasks of an interval order form a total order. More precisely, if u_1 and u_2 are two tasks of an interval order G, then

$$Succ_G(u_1) \subseteq Succ_G(u_2) \qquad \text{or} \qquad Succ_G(u_2) \subseteq Succ_G(u_1).$$

This property can be generalised.

Proposition 2.5.5. *Let G be an interval order. Let U be a non-empty subset of $V(G)$. Then U contains a task u, such that*

$$Succ_G(u) = \bigcup_{v \in U} Succ_G(v).$$

Proof. By straightforward induction on the number of tasks of U. □

The transitive closure of an interval order G can be constructed more efficiently than the transitive closure of an arbitrary precedence graph. First construct a topological order $u_1, \ldots, u_n$ of G. This takes $O(n+e)$ time [18]. Using $u_1, \ldots, u_n$, the set of successors of each task can be computed inductively. Assume $Succ_G(u_{i+1}), \ldots, Succ_G(u_n)$ have been computed. Let $v_1, \ldots, v_k$ be the children of u_i. Since G is an interval order, we may assume that $Succ_G(v_1) \subseteq \cdots \subseteq Succ_G(v_k)$. Then $Succ_G(u_i) = Succ_G(v_k) \cup \{v_1, \ldots, v_k\}$. For every task v in $Succ_G(u_i)$, add an arc from u_i to v. Then the resulting precedence graph is the transitive closure of G. It is constructed in $O(n+e^+)$ time.

Lemma 2.5.6. *Let G be an interval order. Then the transitive closure of G can be constructed in $O(n+e^+)$ time.*

I Scheduling in the UCT model

3 The Unit Communication Times model

Part I is concerned with scheduling in the Unit Communication Times model of parallel computation. The UCT model is presented in this chapter. In Section 3.1, the communication requirements of the UCT model are presented. The scheduling model for tasks is extended to tasks with non-uniform deadlines. The notation concerning non-uniform deadlines are introduced in Section 3.2. The general problem instances and feasible schedules for such instances are presented in Sections 3.3 and 3.4. Section 3.5 introduces the objective functions related to scheduling with non-uniform deadlines. In Section 3.6, previous results on scheduling in the UCT model are presented. An outline of the first part of this thesis is presented in Section 3.7.

3.1 Communication requirements

In Section 2.3, feasible communication-free schedules were introduced. For the construction of feasible communication-free schedules, only two kinds of constraints have to be taken into account: the precedence constraints and the constraints due to the limited number of processors. Hence a task can be scheduled on any processor immediately after the completion of the last of its parents. The time required to transport the result of a task to another processor is neglected.

However, it turns out that communication has a great effect on the performance of parallel computers. This is the reason why there are many models of parallel computation that include a notion of communication. Some of these were mentioned in Section 1.3. Since the effect of communication is ignored in communication-free scheduling, it does not capture the true complexity of parallel programming.

The UCT model is a model of a distributed-memory computer that takes communication delays into account. In the UCT model, we will assume that the communication network is a complete graph: each processor is directly connected to all other processors. The capacities of these connections are assumed to be unbounded. From this assumption, an unbounded number of messages can be sent over any connection in the communication network at the same time. Hence the time required to send one message from one processor to another is independent of the pair of processors: the interprocessor communication delays are all equal. In the UCT model, the communication delays are assumed to be of unit length.

The unit-length communication delays add the following constraint to the scheduling problem. Consider a task u and a child v of u. If u and v are scheduled on different processors, then v cannot start immediately after u, because the result of u must be sent to another processor. There must be a delay of at least one time unit between the completion time of u and the starting time of v. If u and v are scheduled on the same processor, then the result of u need not be sent to another processor and v can be scheduled immediately after u.

3.2 Non-uniform deadlines

Apart from communication delays, non-uniform deadlines for tasks are introduced. The most common objective function for scheduling is the minimisation of the makespan. In scheduling problems with this objective, all tasks have the same priority. However, in many applications, different tasks have different priorities. Tasks with different deadlines are not equally important: tasks with a small deadline must be executed early and hence have a high priority, whereas tasks with large deadlines are less important.

A task should be completed before its deadline. If a task u finishes after its deadline, then it is called *tardy* and the *tardiness* of u is defined to be the amount of time by which the completion time of u exceeds its deadline. If u finishes before its deadline, then it is called *in time* and its tardiness equals zero. The objective of the scheduling problems considered in Part I is the minimisation of the maximum tardiness among all tasks.

The problem of constructing minimum-tardiness schedules is closely related to that of minimising the makespan: the makespan of a schedule coincides with a deadline that is met by all tasks, and if all tasks are assigned deadline zero, then the maximum tardiness of a task in a schedule equals the makespan of this schedule.

3.3 Problem instances

As shown in Chapter 2, a general scheduling instance is represented by a tuple (G,μ,m), where G is a precedence graph, μ is a function that assigns an execution length to every task of G and m is the number of processors. This scheduling problem is generalised in two ways: there are unit-length communication delays and every task has a deadline. Since the communication requirements are the same for all arcs, these are not explicitly included in the scheduling instances.

Unlike the communication delays, the deadlines are included in the instances. The new scheduling instances will be represented by tuples (G,μ,m,D), where G is a precedence graph, $\mu : V(G) \to \mathbb{Z}^+$ assigns an execution length to every task of G, $m \in \{2,3,\ldots,\infty\}$ is the number of processors, and $D : V(G) \to \mathbb{Z}$ assigns a deadline to every task of G. Note that a deadline may be non-positive and that a non-positive deadline cannot be met. If all tasks have execution length one, then the scheduling instance (G,μ,m,D) will be represented by the tuple (G,m,D).

3.4 Feasible schedules

Like for communication-free schedules, a schedule in the UCT model is represented by a pair of functions. A *schedule* for (G,μ,m,D) is a pair of functions (σ,π), such that $\sigma : V(G) \to \mathbb{N}$ and $\pi : V(G) \to \{1,\ldots,m\}$.

Definition 3.4.1. A schedule (σ,π) for (G,μ,m,D) is called a *feasible schedule* for (G,μ,m,D) if for all tasks $u_1 \neq u_2$ of G,

1. if $\pi(u_1) = \pi(u_2)$, then $\sigma(u_1) + \mu(u_1) \leq \sigma(u_2)$ or $\sigma(u_2) + \mu(u_2) \leq \sigma(u_1)$;
2. if $u_1 \prec_G u_2$, then $\sigma(u_1) + \mu(u_1) \leq \sigma(u_2)$; and
3. if $u_1 \prec_{G,0} u_2$ and $\pi(u_1) \neq \pi(u_2)$, then $\sigma(u_1) + \mu(u_1) + 1 \leq \sigma(u_2)$.

The first two constraints equal those for feasible communication-free schedules; the third one states that there must be a delay of at least one time unit between data-dependent tasks on different processors. Note that the feasibility of a schedule does not depend on the deadlines.

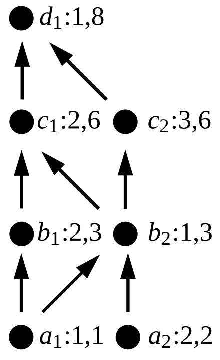

Figure 3.1. An instance $(G,\mu,2,D)$

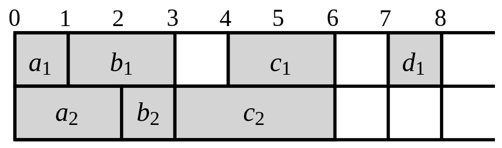

Figure 3.2. A feasible schedule for $(G,\mu,2,D)$

Example 3.4.2. Consider the instance $(G,\mu,2,D)$ shown in Figure 3.1. Each task of G is labelled with its name, execution length and deadline. Note that $(G,\mu,2,D)$ corresponds to the general scheduling instance $(G,\mu,2)$ shown in Figure 2.1. A feasible schedule (σ,π) for $(G,\mu,2,D)$ is shown in Figure 3.2. a_1 and a_2 start at time 0 on separate processors. b_1 is a successor of a_1, so it can be scheduled immediately after a_1 on the first processor. Since b_2 is a successor of a_1 and a_2, and b_2 is not scheduled on the same processor as a_1, there is a delay of one time unit between the completion time of a_1 and the starting time of b_2. c_1 and c_2 are both successors of b_2. Only one of these tasks can be executed immediately after b_2 on the second processor. The other can be scheduled after a delay of one time unit on the first processor. Similarly, d_1 cannot be executed immediately after c_1 and c_2, because c_1 and c_2 are both parents of d_1. It is easy to see that the schedule for $(G,\mu,2)$ shown in Figure 2.2 is not a feasible schedule for $(G,\mu,2,D)$.

In the remaining chapters of Part I, we will use a different definition of feasible schedules. Using this definition, it is simpler to construct schedules and reason about them. In this definition, a schedule is only represented by the starting times of the tasks. A corresponding assignment of processors can be constructed using these starting times.

Definition 3.4.3. A function $S : V(G) \rightarrow \mathbb{N}$ is called a *feasible assignment of starting times* for (G,μ,m,D) if for all tasks u_1 and u_2 of G and all non-negative integers t,

1. $|\{u \in V(G) \mid S(u) \leq t < S(u)+\mu(u)\}| \leq m$;
2. if $u_1 \prec_G u_2$, then $S(u_2) \geq S(u_1)+\mu(u_1)$;

3. at most one child of u_1 starts at time $S(u_1)+\mu(u_1)$; and
4. at most one parent of u_1 finishes at time $S(u_1)$.

Note that every feasible schedule implies a feasible assignment of starting times. Conversely, given a feasible assignment of starting times S for (G,μ,m,D), we can construct an assignment of processors π, such that (S,π) is a feasible schedule for (G,μ,m,D). Such an assignment of processors is constructed by Algorithm PROCESSOR ASSIGNMENT COMPUTATION shown in Figure 3.3. For all times t starting with time 0, it assigns a processor to all tasks with starting time t. The following notations are used. At any time t, $idle(p)$ denotes the maximum completion time of a task that has been assigned to processor p and tasks $u_{i_{\min}}$ and $u_{i_{\max}}$ denote the first and last task with starting time t, respectively.

Algorithm PROCESSOR ASSIGNMENT COMPUTATION
Input. A feasible assignment of starting times S for (G,μ,m,D), such that $V(G)=\{u_1,\dots,u_n\}$ and $S(u_1)\leq\cdots\leq S(u_n)$.
Output. An assignment of processors π, such that (S,π) is a feasible schedule for (G,μ,m,D).

```
1.  for $p := 1$ to $\max\{m,n\}$
2.      do $idle(p) := 0$
3.  $i_{\max} := 0$
4.  repeat
5.      $i_{\min} := i_{\max}+1$
6.      $i_{\max} := \max\{i \geq i_{\min} \mid S(u_i) = S(u_{i_{\min}})\}$
7.      $t := S(u_{i_{\min}})$
8.      $U := \varnothing$
9.      for $i := i_{\min}$ to $i_{\max}$
10.         do if $u_i$ has a parent $v$, such that $S(v)+\mu(v)=t$
11.                then $\pi(u_i) := \pi(v)$
12.                     $idle(\pi(u_i)) := t+\mu(u_i)$
13.                else $U := U \cup \{u_i\}$
14.     for $u \in U$
15.         do determine $p$, such that $idle(p) \leq t$
16.            $\pi(u) := p$
17.            $idle(p) := t+\mu(u)$
18. until $i_{\max} = n$
```

Figure 3.3. Algorithm PROCESSOR ASSIGNMENT COMPUTATION

Now we will prove that Algorithm PROCESSOR ASSIGNMENT COMPUTATION correctly constructs feasible schedules given a feasible assignment of starting times.

Lemma 3.4.4. *Let S be a feasible assignment of starting times for (G,μ,m,D). Let π be the assignment of processors for (G,μ,m,D) constructed by Algorithm* PROCESSOR ASSIGNMENT COMPUTATION. *Then (S,π) is a feasible schedule for (G,μ,m,D).*

Proof. Because S is a feasible assignment of starting times for (G,μ,m,D), there are at most m tasks u of G, such that $S(u) \leq t < S(u)+\mu(u)$ for all times t. So for any task u of G, when the tasks of G with starting time $S(u)$ are considered by Algorithm PROCESSOR ASSIGNMENT COMPUTATION, there are sufficiently many processors p, such that $idle(p) \leq S(u)$. So every task u has been assigned a processor $\pi(u)$. Let u_1 and u_2 be two tasks of G. Since S is a feasible assignment of starting times for (G,μ,m,D), if $u_1 \prec_G u_2$, then $S(u_2) \geq S(u_1)+\mu(u_1)$. If u_2 is a child of u_1 and $\pi(u_1) \neq \pi(u_2)$, then $S(u_1)+\mu(u_1) \neq S(u_2)$. Otherwise, u_2 would have been assigned to the same processor as u_1. Assume $\pi(u_1) = \pi(u_2)$. Assume u_1 has been assigned a processor before u_2. When u_2 is assigned to a processor, $idle(\pi(u_1)) \geq S(u_1)+\mu(u_1)$. Because u_2 is assigned to processor $\pi(u_2) = \pi(u_1)$, $S(u_2) \geq idle(\pi(u_1)) \geq S(u_1)+\mu(u_1)$. So (S,π) is a feasible schedule for (G,μ,m,D). $\square$

The time complexity of Algorithm PROCESSOR ASSIGNMENT COMPUTATION can be determined as follows. Let S be a feasible assignment of starting times. Constructing a list of tasks ordered by non-decreasing starting times takes $O(n\log n)$ time. Indices $i_{\min}$ and $i_{\max}$ can be computed by one traversal of the list of tasks ordered by non-decreasing starting times. Since $i_{\min}$ and $i_{\max}$ do not decrease, updating these indices takes $O(n)$ time in total. For each task u, it has to be determined whether a parent finishes at time $S(u)$. This takes $O(|Pred_{G,0}(u)|)$ time. If there is a such a parent, then u is assigned to the same processor as this parent. Otherwise, it is added to U and assigned to an arbitrary idle processor. A task is added and removed from U at most once. If U is represented by a queue, then the operations on U take $O(n)$ time in total. If the processors are stored in a balanced search tree ordered by non-decreasing $idle(p)$-value, then each operation on this tree takes $O(\log n)$ time. So π is constructed in a total of $O(n\log n+e)$ time.

Lemma 3.4.5. *For all feasible assignments of starting times S for an instance (G,μ,m,D), Algorithm* PROCESSOR ASSIGNMENT COMPUTATION *constructs an assignment of processors π for (G,μ,m,D), such that (S,π) is a feasible schedule for (G,μ,m,D), in $O(n\log n+e)$ time.*

Because a feasible assignment of starting times for (G,μ,m,D) can be extended to a feasible schedule for (G,μ,m,D), the term feasible schedule will be used for feasible assignments of starting times as well.

Let S be a feasible schedule for an instance (G,m,D). All tasks of G have unit length. For all integers t, define $S_t = \{u \in V(G) \mid S(u) = t\}$. Then every task in S_t starts at time t and is completed at time $t+1$. S_t will be called the t^{th} *time slot* of S. S can be completely represented by a list of time slots: $S = (S_0,\ldots,S_{\ell-1})$, where ℓ is the length of S. A time slot S_t is called *idle* if it contains less than m tasks.

We conclude this section with a definition that is related to that of feasible schedules.

Definition 3.4.6. Let U be a prefix of a precedence graph G. Let S be a feasible schedule for $(G[U],\mu,m,D)$. Let u be a task in U or a source of $G[V(G)\setminus U]$. Then u is called *ready* at time t (with respect to S) if the all predecessors of u are completed at or before time t. u is called *available* at time t (with respect to S) if

1. u is ready at time t (with respect to S);
2. at most one parent of u finishes at time t; and
3. if a parent v of u finishes at time t, then no child $w \neq u$ of v starts at time t.

Let S be a feasible schedule for an instance (G,μ,m,D). It is not difficult to see that any task u is available at time $S(u)$. Note that a task can be available at time t even if m tasks are being executed at that time. Hence any unscheduled task is available one unit of time after the completion time of the last of its predecessors.

3.5 Tardiness

The objective of the scheduling problems studied in the first part of the thesis is the minimisation of the maximum tardiness of a task. Let S be a feasible schedule for an instance (G,μ,m,D). Let u be a task of G. The *tardiness* of u equals $\max\{0, S(u)+\mu(u)-D(u)\}$; its *lateness* equals $S(u)+\mu(u)-D(u)$. The *tardiness* of S is the maximum tardiness of a task of G: S has tardiness $\max\{0, \max_{u\in V(G)}(S(u)+\mu(u)-D(u))\}$. If the tardiness of S equals zero, then it is called an *in-time schedule* for (G,μ,m,D). The *lateness* of S is the maximum lateness among the tasks of G, it equals $\max_{u\in V(G)}(S(u)+\mu(u)-D(u))$.

S is called a *minimum-tardiness schedule* for (G,μ,m,D) if there is no feasible schedule for (G,μ,m,D) whose tardiness is smaller than that of S. Similarly, S is called a *minimum-lateness schedule* for (G,μ,m,D) if there is no feasible schedule for (G,μ,m,D) whose lateness is smaller than that of S. Because the tardiness of a schedule cannot be negative and an in-time schedule has tardiness zero, any in-time schedule for (G,μ,m,D) is a minimum-tardiness schedule for (G,μ,m,D). Since the lateness of a schedule can be negative, an in-time schedule for (G,μ,m,D) need not be a minimum-lateness schedule for (G,μ,m,D).

Clearly, minimising the tardiness and minimising the lateness are closely related problems. Makespan minimisation is also closely related to minimisation of the tardiness: if all deadlines equal zero, then the tardiness of a schedule equals its length. So any algorithm that constructs minimum-tardiness schedules can be used to construct minimum-length schedules.

The tardiness of a schedule can be zero. So for all $\rho \in \mathbb{R}$, such that $\rho \geq 1$, a ρ-approximation algorithm for tardiness minimisation must construct in-time schedules if such schedules exist. If all deadlines are non-positive, then the tardiness of any schedule is positive, because a non-positive deadline cannot be met. For such instances, a ρ-approximation need not construct minimum-tardiness schedules.

However, scheduling with non-positive deadlines is a bit unnatural, because a non-positive deadline cannot be met. There is a model that is equivalent to scheduling with non-positive deadlines: scheduling with *delivery times* [58, 66]. In this model, every task u has a non-negative delivery time $q(u)$. This is the amount of time that expires after the completion time of u until it is delivered. The objective in scheduling with delivery times is the minimisation of the maximum *delivery-completion time* (the sum of the completion time and the delivery time of a task). If we have an instance (G,μ,m,D) with non-positive deadlines, then we can choose $q(u) = -D(u)$

for all tasks u of G. Then minimising the maximum tardiness corresponds to minimising the maximum delivery-completion time.

3.6 Previous results

Scheduling precedence graphs subject to unit-length communication delays is a well-studied problem. Minimisation of the makespan is the most common objective of the algorithms for scheduling with unit-length communication delays. Rayward-Smith [79] was one of the first to study the problem of scheduling precedence-constrained tasks subject to unit-length communication delays. He proved that constructing minimum-length schedules for arbitrary precedence graphs with unit-length tasks is an NP-hard optimisation problem. Lenstra et al. [61] proved the same for scheduling inforests with unit-length tasks. Constructing minimum-length schedules for arbitrary precedence graphs with unit-length tasks on an unrestricted number of processors is an NP-hard optimisation problem as well [47, 77, 80].

For special classes of precedence graphs, it is possible to construct minimum-length schedules in polynomial time. Minimum-length schedules for precedence graphs with unit-length tasks on two processors can be constructed in polynomial time if the precedence constraints form an inforest or an outforest [42, 50, 61, 77, 86] or a series-parallel graph [27]. Varvarigou et al. [86] presented a dynamic-programming algorithm that constructs minimum-length schedules for outforests with unit-length tasks on m processors in $O(n^{2m-2})$ time; this algorithm constructs minimum-length schedules in polynomial time if the number of processors is a constant. For interval-ordered tasks of unit length, a minimum-length schedule on m processors can be constructed in polynomial time for any number of processors m [4, 77]. Minimum-length schedules for precedence graphs with arbitrary task lengths on an unrestricted number of processors can be constructed in polynomial time if the precedence constraints form an inforest or an outforest [12], a series-parallel graph [68, 69] or a bipartite precedence graph [77].

In addition, there are many algorithms that approximate the makespan of a minimum-length schedule. Rayward-Smith proved that a list scheduling is a $3-\frac{2}{m}$-approximation algorithm for scheduling arbitrary precedence graphs with unit-length tasks on m processors. Lawler [59] presented an algorithm that constructs schedules for outforests with unit-length tasks on m processors; Guinand et al. [41] proved that the schedules constructed by Lawler's algorithm are at most $\frac{1}{2}(m-1)$ time units longer than the length of a minimum-length schedule on m processors. Moreover, Munier and König [73] use linear programming in their $\frac{4}{3}$-approximation algorithm for scheduling arbitrary precedence graphs with unit-length tasks on an unrestricted number of processors. Munier and Hanen [72] generalised this algorithm to a $\frac{7}{3}-\frac{1}{3m}$-approximation algorithm for scheduling arbitrary precedence graphs with unit-length tasks on m processors. Schäffter [81] showed how these algorithms can be generalised to a $\frac{4}{3}$-approximation algorithm and a $\frac{7}{3}$-approximation algorithm for scheduling arbitrary precedence graphs with arbitrary task lengths on an unrestricted and a restricted number of processors, respectively.

Two of the few results concerning scheduling problems whose objective is not the minimisation of the makespan were presented by Möhring et al. [70]; they study scheduling problems whose objective is the minimisation of the weighted sum of completion times. They presented two approximation algorithms: a $\frac{10}{3}-\frac{4}{3m}$-approximation algorithm for scheduling arbitrary precedence graphs with unit-length tasks on m processors and a 6.14232-approximation algorithm for scheduling arbitrary precedence graphs with tasks of arbitrary length on m processors. In addition, there is a 3-approximation algorithm for scheduling series-parallel graphs with unit-length tasks and a 5.80899-approximation algorithm for scheduling series-parallel graphs with arbitrary task lengths [81].

3.7 Outline of the first part

Apart from this chapter, Part I consists of Chapters 4, 5, 6 and 7. These chapters are concerned with the construction of minimum-tardiness schedules in the UCT model. In Chapter 4, an algorithm for this problem is presented that consists of two parts. The first part computes smaller deadlines, that are met in all in-time schedules. These deadlines will be called consistent. The second part of the algorithm is a list scheduling algorithm that uses the consistent deadlines to construct a feasible schedule. It will be proved that this algorithm is an approximation algorithm with asymptotic approximation ratio $\max\{2, 3-\frac{3}{m}\}$ for scheduling arbitrary precedence graphs with non-positive deadlines on m processors and an approximation algorithm with asymptotic approximation ratio $2-\frac{2}{m}$ for scheduling outforests with non-positive deadlines on m processors. In addition, the algorithm constructs minimum-tardiness schedules for outforests on two processors and on an unrestricted number of processors. Moreover, it is shown that the algorithm is a 2-approximation algorithm for scheduling arbitrary precedence graphs with non-positive deadlines on an unrestricted number of processors.

The least urgent parent property is introduced in Chapter 5. It will be proved that for arbitrary precedence graphs with the least urgent parent property, minimum-tardiness schedules on an unrestricted number of processors can be constructed using a list scheduling approach. The same is proved for scheduling inforests on m processors. If an instance does not have the least urgent parent property, then its deadlines can be increased, such that the resulting instance has the least urgent parent property. The construction of instances with the least urgent parent property is used to construct schedules for arbitrary inforests. Using this construction, we obtain a 2-approximation algorithm for scheduling inforests with non-positive deadlines on m processors.

In Chapter 6, a stronger notion of consistency is introduced by considering pairs of tasks instead of individual tasks. A list scheduling algorithm uses the pairwise consistent deadlines to construct minimum-tardiness schedules for interval orders on m processors and for precedence graphs of width two on two processors. The result on scheduling interval-ordered tasks has been published in the proceedings of ISAAC'96 [89] and a final version will be published in Parallel Computing [93].

In Chapter 7, a dynamic-programming approach is used to construct minimum-tardiness schedules for arbitrary precedence graphs. For precedence graphs of bounded width with unit-length tasks, it constructs minimum-tardiness schedules on m processors in polynomial time. The same is proved for scheduling precedence graphs of bounded width with arbitrary task lengths on an unrestricted number of processors. Moreover, constructing minimum-tardiness schedules for precedence graphs of width three with arbitrary task length on two processors is shown to be an NP-hard optimisation problem.

4 Individual deadlines

The first part of this thesis is concerned with scheduling with non-uniform deadlines subject to unit-length communication delays. Most scheduling problems with precedence constraints and non-uniform deadlines neglect the communication costs. Garey and Johnson [31] were the first that studied a scheduling problem with precedence constraints and non-uniform deadlines. They presented an algorithm that constructs minimum-tardiness schedules for arbitrary precedence graphs with unit-length tasks on two processors. Hanen and Munier [44] showed that this algorithm has an asymptotic approximation ratio of $2-\frac{3}{2m}$ for scheduling arbitrary precedence graphs with unit-length tasks and non-positive deadlines on m processors. In addition, Brucker et al. [11] proved that for inforests with unit-length tasks, minimum-tardiness schedules on m processors can be constructed in polynomial time. Hall and Shmoys [43] showed that list scheduling is a 2-approximation algorithm for scheduling arbitrary precedence graphs with arbitrary task lengths with non-positive deadlines on m processors.

In this chapter, I will present an efficient algorithm that constructs schedules for precedence graphs with non-uniform deadlines subject to unit-length communication delays. The algorithm has the same overall structure as the one presented by Garey and Johnson [31]. The algorithm consists of two parts. The first part computes smaller deadlines that are met in all in-time schedules. The deadlines that are met in all in-time schedules will be called consistent. We want these deadlines to be as small as possible. Consistent deadlines will be defined in Section 4.1. The computation of the consistent deadline of a task u depends on the subgraph containing the successors of u: if u has sufficiently many successors that have to be completed at or before time d, then the deadline of u is decreased. The algorithm computing consistent deadlines is presented in Section 4.2.

The second part of the algorithm is a list scheduling algorithm that is presented in Section 4.3. This algorithm uses a list ordered by non-decreasing consistent deadlines to assign a starting time to every task. In Section 4.4, the tardiness of the schedules constructed by the list scheduling algorithm will be computed. It will be proved that the algorithm constructs minimum-tardiness schedules for outforests with unit-length tasks on two processors and for outforests with arbitrary task lengths on an unrestricted number of processors. In addition, it will be proved that this algorithm has an asymptotic approximation ratio of $2-\frac{2}{m}$ for scheduling outforests with unit-length tasks and non-positive deadlines on m processors. Its asymptotic approximation ratio for scheduling arbitrary precedence graphs with unit-length tasks and non-positive deadlines on m processors equals $\max\{2, 3-\frac{3}{m}\}$. Moreover, this algorithm is shown to be a 2-approximation algorithm for scheduling arbitrary precedence graphs with arbitrary task lengths and non-positive deadlines on an unrestricted number of processors.

4.1 Consistent deadlines

In this chapter, an algorithm is presented for scheduling precedence graphs with non-uniform deadlines subject to unit-length communication delays. The algorithm consists of two parts: the first part determines a priority of the tasks and the second part uses these priorities to assign a starting time to every task. In order to get schedules with a small tardiness, the priority of the tasks should depend on the deadlines. The priority will be defined using deadlines that are met in all in-time schedules. In order to get schedules with a small tardiness, we want these deadlines to be as small as possible. Hence the best possible deadline of a task u is the latest completion time of u in an in-time schedule. However, it is impossible to compute these completion times efficiently. Hence we will approximate these completion times by computing smaller deadlines for each task using the deadlines of its successors. These smaller deadlines will be called consistent. It will be proved that the consistent deadlines are met in all in-time schedules.

To define consistent deadlines, we need to look at the structure of in-time schedules. Let S be an in-time schedule for (G,m,D). Let u be a task of G. Assume u has $k \geq 1$ successors $v_1,\ldots,v_k$, such that $D(v_i) \leq d$ for all $i \leq k$. u is scheduled at time $S(u)$ and finishes at time $S(u)+1$. Because of the communication delays, at most one successor v_i of u can be scheduled at time $S(u)+1$. Hence the last of the other $k-1$ successors of u cannot be completed before time $S(u)+2+\lceil\frac{k-1}{m}\rceil$. Since the successors of u are all executed before time d, u must finish at or before time $d-1-\lceil\frac{k-1}{m}\rceil$.

Now we will consider the more general instance (G,μ,m,D). Let S be an in-time schedule for (G,μ,m,D). Let v be a task of G. v finishes at or before time $D(v)$. So $S(v) \leq D(v)-\mu(v)$. v can be viewed as a chain of $\mu(v)$ subtasks of unit length. Define $\mu_D(v,d)$ as the number of unit-length subtasks of v that are completed at or before time d if v finishes at time $D(v)$.

$$\mu_D(v,d) \quad = \quad \begin{cases} 0 & \text{if } d \leq D(v)-\mu(v) \\ \mu(v)-D(v)+d & \text{if } D(v)-\mu(v) < d < D(v) \\ \mu(v) & \text{if } d \geq D(v) \end{cases}$$

Note that for instances (G,m,D), $\mu_D(v,d) \in \{0,1\}$ for all tasks v of G: if $D(v) \leq d$, then $\mu_D(v,d)=1$ and if $D(v) > d$, then $\mu_D(v,d)=0$.

Let u be a task of G. Let $k=\sum_{v\in Succ_G(u)} \mu_D(v,d)$ be the total number of unit-length subtasks of the successors of u that are completed at or before time d. Then u must finish at or before time $d-1-\lceil\frac{k-1}{m}\rceil$.

Define $N_D(u,d)$ as the total number of unit-length subtasks of the successors of u that are completed at or before time d in any in-time schedule for (G,μ,m,D). More precisely,

$$N_D(u,d) \quad = \quad \sum_{v\in Succ_G(u)} \mu_D(v,d).$$

Note that for instances (G,m,D), $N_D(u,d)$ equals the number of successors of u with deadline at most d.

Example 4.1.1. Consider the instance $(G,2,D)$ shown in Figure 4.1. The following is easy to see. $N_D(d_i,9)=1$, $N_D(c_1,8)=3$, $N_D(c_1,9)=4$, $N_D(b_i,6)=1$, $N_D(b_i,8)=4$, $N_D(b_i,9)=5$, $N_D(a_1,5)=N_D(a_3,5)=2$, $N_D(a_1,6)=N_D(a_3,6)=3$, $N_D(a_1,8)=N_D(a_3,8)=6$, $N_D(a_1,9)=N_D(a_3,9)=7$, $N_D(a_2,5)=3$, $N_D(a_2,6)=4$, $N_D(a_2,8)=7$ and $N_D(a_2,9)=8$.

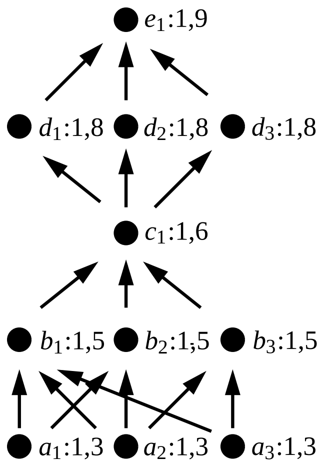

Figure 4.1. An instance $(G,2,D)$

The following observation allows the definition of consistent instances.

Observation 4.1.2. *Let S be an in-time schedule for (G,μ,m,D). Let u be a task of G. If $N_D(u,d) \geq 1$, then $S(u)+\mu(u) \leq d-1-\left\lceil \frac{1}{m}(N_D(u,d)-1)\right\rceil$.*

Observation 4.1.2 is used to define consistent deadlines. We will assume that $\left\lceil \frac{0}{\infty} \right\rceil = 0$ and $\left\lceil \frac{k}{\infty} \right\rceil = 1$ for all integers $k \geq 1$.

Definition 4.1.3. An instance (G,μ,m,D) is called *consistent* if for all tasks u of G and all integers d,

$$\text{if} \quad N_D(u,d) \;\geq\; 1, \text{then} \quad D(u) \;\leq\; d-1-\left\lceil \tfrac{1}{m}(N_D(u,d)-1)\right\rceil.$$

(G,μ,m,D) is called D_0-*consistent* if it is consistent and $D(u) \leq D_0(u)$ for all tasks u of G. A D_0-consistent instance (G,μ,m,D) is called *strongly D_0-consistent* if for all tasks u of G,

$$D(u) = D_0(u) \text{ or for some } d \in \mathbb{Z}, N_D(u,d) \geq 1 \text{ and } D(u) = d-1-\left\lceil \tfrac{1}{m}(N_D(u,d)-1)\right\rceil.$$

Example 4.1.4. Consider the instance $(G,2,D)$ shown in Figure 4.1. Assume $D_0(u) = 9$ for all tasks u of G. It is not difficult to see that $(G,2,D)$ is D_0-consistent. It is also strongly D_0-consistent, because $D(e_1) = 9 = D_0(e_1)$, $D(d_i) = 8 = 9-1-\left\lceil \frac{1}{2}(N_D(d_i,9)-1)\right\rceil$, $D(c_1) = 6 = 8-1-\left\lceil \frac{1}{2}(N_D(c_1,8)-1)\right\rceil$, $D(b_i) = 5 = 6-1-\left\lceil \frac{1}{2}(N_D(b_i,6)-1)\right\rceil$ and $D(a_i) = 3 = 5-1-\left\lceil \frac{1}{2}(N_D(a_i,5)-1)\right\rceil$.

The following observations state some properties of consistent instances. The first states that any consistent instance is strongly consistent with respect to its own deadlines.

Observation 4.1.5. *Let (G,μ,m,D) be a consistent instance. Then (G,μ,m,D) is strongly D-consistent.*

The second observation states that the deadlines of a strongly D_0-consistent instance are maximum among the D_0-consistent instances. This shows that for each instance (G,μ,m,D_0), there is exactly one strongly D_0-consistent instance (G,μ,m,D).

Observation 4.1.6. *Let (G,μ,m,D) and (G,μ,m,D') be D_0-consistent instances. If (G,μ,m,D) is strongly D_0-consistent, then $D(u) \geq D'(u)$ for all tasks u of G.*

The third observation states that if all original deadlines are increased by the same amount, then the tardiness of a minimum-tardiness schedule decreases by the same amount, unless the tardiness would become negative.

Observation 4.1.7. *Let ℓ^* be the tardiness of a minimum-tardiness schedule for (G,μ,m,D_0). If there is an integer c, such that $D(u) = D_0(u) + c$ for all tasks u of G, then the tardiness of a minimum-tardiness schedule for (G,μ,m,D) equals* $\max\{0,\ell^* - c\}$.

The following lemma proves that if all original deadlines are increased by the same amount, then so are the strongly consistent deadlines. This result will be used to compute upper bounds on the tardiness of schedules.

Lemma 4.1.8. *Let (G,μ,m,D) be the strongly D_0-consistent instance and let (G,μ,m,D') be the strongly D'_0-consistent instance. If there is an integer c, such that $D'_0(u) = D_0(u) + c$ for all tasks u of G, then $D'(u) = D(u) + c$ for all tasks u of G.*

Proof. Assume there is an integer c, such that $D'_0(u) = D_0(u) + c$ for all tasks u of G. We will prove by induction that $D'(u) = D(u) + c$ for all tasks u of G. Let u be a task of G. Assume by induction that $D'(v) = D(v) + c$ for all successors v of u. We will prove by contradiction that $D'(u) = D(u) + c$. Suppose $D'(u) \neq D(u) + c$.

Case 1. $D(u) = D_0(u)$.
Then $D'(u) < D'_0(u) = D_0(u) + c$. Because (G,μ,m,D') is strongly D'_0-consistent, there is an integer d, such that $N_{D'}(u,d) \geq 1$ and $D'(u) = d - 1 - \left\lceil \frac{1}{m}(N_{D'}(u,d) - 1) \right\rceil$. Because $N_D(u,d-c) = N_{D'}(u,d) \geq 1$ and (G,μ,m,D) is consistent, $D(u) \leq d - c - 1 - \left\lceil \frac{1}{m}(N_{D'}(u,d) - 1) \right\rceil = D'(u) - c < D_0(u)$. Contradiction. So $D'(u) = D(u) + c$.

Case 2. $D(u) \neq D_0(u)$.
Since (G,μ,m,D) is strongly D_0-consistent, there is an integer d, such that $N_D(u,d) \geq 1$ and $D(u) = d - 1 - \left\lceil \frac{1}{m}(N_D(u,d) - 1) \right\rceil$. Because $N_{D'}(u,d+c) = N_D(u,d) \geq 1$ and (G,μ,m,D') is consistent, $D'(u) \leq d + c - 1 - \left\lceil \frac{1}{m}(N_D(u,d) - 1) \right\rceil = D(u) + c$. Since $D'(u) \neq D(u) + c$, we know that $D'(u) < D(u) + c$. Hence $D'(u) \neq D'_0(u)$. Since (G,μ,m,D') is strongly D'_0-consistent, there is an integer d', such that $N_{D'}(u,d') \geq 1$ and $D'(u) = d' - 1 - \left\lceil \frac{1}{m}(N_{D'}(u,d') - 1) \right\rceil$. Since $N_D(u,d'-c) = N_{D'}(u,d') \geq 1$ and (G,μ,m,D) is consistent, $D(u) \leq d' - c - 1 - \left\lceil \frac{1}{m}(N_{D'}(u,d') - 1) \right\rceil = D'(u) - c < D(u)$. Contradiction. So $D'(u) = D(u) + c$.

In either case, $D'(u) = D(u) + c$. By induction, $D'(u) = D(u) + c$ for all tasks u of G. □

The following lemma shows that strongly consistent deadlines are met in all in-time schedules.

Lemma 4.1.9. *Let (G,μ,m,D) be the strongly D_0-consistent instance. Let S be a feasible schedule for (G,μ,m,D_0). Then S is an in-time schedule for (G,μ,m,D_0) if and only if S is an in-time schedule for (G,μ,m,D).*

Proof. Because $D(u) \leq D_0(u)$ for all tasks u of G, every in-time schedule for (G,μ,m,D) is an in-time schedule for (G,μ,m,D_0). Assume S is an in-time schedule for (G,μ,m,D_0). Define $D_S(u) = S(u) + \mu(u)$ for all tasks u of G. We will prove by contradiction that (G,μ,m,D_S) is consistent. Suppose (G,μ,m,D_S) is not consistent. Then there is a task u of G and an integer d, such that $N_{D_S}(u,d) \geq 1$ and $D_S(u) > d - 1 - \lceil \frac{1}{m}(N_{D_S}(u,d) - 1) \rceil$. Every successor v of u meets its deadline $D_S(v)$. So $N_{D_S}(u,d)$ unit-length subtasks of successors of u finish at or before time d. Hence u must be completed at or before time $d - 1 - \lceil \frac{1}{m}(N_{D_S}(u,d) - 1) \rceil$. So $D_S(u) \leq d - 1 - \lceil \frac{1}{m}(N_{D_S}(u,d) - 1) \rceil$. Contradiction. So (G,μ,m,D_S) is consistent. Because S is an in-time schedule for (G,μ,m,D_0), (G,μ,m,D_S) is also D_0-consistent. From Observation 4.1.6, $D(u) \geq D_S(u)$ for all tasks u of G. Since every deadline $D_S(u)$ is met, S is an in-time schedule for (G,μ,m,D). $\square$

The next two results will be used to construct strongly D_0-consistent instances.

Lemma 4.1.10. *Let (G,μ,m,D) be the strongly D_0-consistent instance. Let u and v be two tasks of G. If v is the only child of u, then $D(u) = \min\{D_0(u), D(v) - \mu(v)\}$.*

Proof. Assume v is the only child of u. Let $d = D(v) - \mu(v) + 1$. Then $N_D(u,d) \geq \mu_D(v,d) = 1$. So $D(u) \leq d - 1 = D(v) - \mu(v)$. We will assume that $D(u) \neq D_0(u)$. Then there is an integer d', such that $N_D(u,d') \geq 1$ and $D(u) = d' - 1 - \lceil \frac{1}{m}(N_D(u,d') - 1) \rceil$. If $N_D(u,d') \leq \mu(v)$, then $D(u_1) \geq D(v) - 1 - \lceil \frac{1}{m}(\mu(v) - 1) \rceil \geq D(v) - \mu(v)$. So we may assume that $N_D(u,d') > \mu(v)$. Since v is the only child of u and (G,μ,m,D) is consistent, $d' > D(v)$. Because v is a predecessor of all other successors of u, $N_D(v,d') = N_D(u,d') - \mu(v) \geq 1$. So

$$\begin{aligned}
D(u) &= d' - 1 - \left\lceil \tfrac{1}{m}(N_D(u,d') - 1) \right\rceil \\
&= d' - 1 - \left\lceil \tfrac{1}{m}(N_D(v,d') + \mu(v) - 1) \right\rceil \\
&\geq d' - 1 - \left\lceil \tfrac{1}{m}(N_D(v,d') - 1) \right\rceil - \mu(v) \\
&\geq D(v) - \mu(v).
\end{aligned}$$

So $D(u) = D(v) - \mu(v)$. As a result, $D(u) = \min\{D_0(u), D(v) - \mu(v)\}$. $\square$

Lemma 4.1.11. *Let (G,μ,∞,D) be the strongly D_0-consistent instance. Let u be a task of G. If u has $k \geq 2$ children $v_1,\ldots,v_k$, such that $D(v_1) - \mu(v_1) \leq \cdots \leq D(v_k) - \mu(v_k)$, then $D(u) = \min\{D_0(u), D(v_1) - \mu(v_1), D(v_2) - \mu(v_2) - 1\}$.*

Proof. Assume u has $k \geq 2$ children $v_1,\ldots,v_k$, such that $D(v_1) - \mu(v_1) \leq \cdots \leq D(v_k) - \mu(v_k)$. Let $d = D(v_1) - \mu(v_1) + 1$. Then $N_D(u,d) \geq \mu_D(v_1,d) = 1$. Since (G,μ,∞,D) is consistent, $D(u) \leq d - 1 = D(v_1) - \mu(v_1)$. Assume $D(u) \neq D_0(u)$ and $D(u) \neq D(v_1) - \mu(v_1)$. Then there is an integer

d', such that $N_D(u,d') \geq 1$ and $D(u) = d' - 1 - \lceil \frac{1}{\infty}(N_D(u,d') - 1) \rceil \leq D(v_1) - \mu(v_1) - 1$. Since $\lceil \frac{0}{\infty} \rceil = 0$ and $\lceil \frac{k}{\infty} \rceil = 1$ for all $k \geq 1$, $d' = D(v_1) - \mu(v_1) + 1$ and $N_D(u,d') \geq 2$. So $\mu_D(u_2,d') \geq 1$. Hence $D(v_2) - \mu(v_2) = D(v_1) - \mu(v_1)$. Therefore $D(u) = d' - 2 = D(v_1) - \mu(v_1) - 1 = D(v_2) - \mu(v_2) - 1$. □

4.2 Computing consistent deadlines

In this section, two algorithms will be presented that construct strongly D_0-consistent instances. The algorithm presented in Section 4.2.1 computes strongly D_0-consistent deadlines for instances (G,μ,m,D_0). For instances (G,μ,∞,D_0), strongly D_0-consistent deadlines can be computed more efficiently using the algorithm presented in Section 4.2.2.

4.2.1 A restricted number of processors

Consider the strongly D_0-consistent instance (G,μ,m,D). For each task u of G, if $N_D(u,d) \geq 1$, then $D(u) \leq d - 1 - \lceil \frac{1}{m}(N_D(u,d) - 1) \rceil$. So in order to compute the strongly D_0-consistent deadline of u, the strongly D_0-consistent deadlines of its successors must have been computed before. This is how Algorithm DEADLINE MODIFICATION shown in Figure 4.2 works: in each step of the algorithm, it computes the strongly D_0-consistent deadline of a task, such that the strongly D_0-consistent deadlines of all successors of this task have been computed before.

```
Algorithm DEADLINE MODIFICATION
Input. An instance $(G,\mu,m,D_0)$.
Output. The strongly $D_0$-consistent instance $(G,\mu,m,D)$.
1.  $D_{\min} := \min_{u \in V(G)} D_0(u)$
2.  $D_{\max} := \max_{u \in V(G)} D_0(u)$
3.  for all tasks $u$ of $G$
4.      do $D(u) := D_0(u)$
5.  $U := V(G)$
6.  while $U \neq \varnothing$
7.      do let $u$ be a sink of $G[U]$
8.          for $d := D_{\min}$ to $D_{\max}$
9.              do if $N_D(u,d) \geq 1$
10.                 then $D(u) := \min\{D(u), d - 1 - \lceil \frac{1}{m}(N_D(u,d) - 1) \rceil\}$
11.         $D_{\min} := \min\{D_{\min}, D(u)\}$
12.         $U := U \setminus \{u\}$
```

Figure 4.2. Algorithm DEADLINE MODIFICATION

Example 4.2.1. Let G be the precedence graph shown in Figure 4.1. Assume $D_0(u) = 9$ for all tasks u of G. Algorithm DEADLINE MODIFICATION computes deadlines $D(u)$ as follows. First it considers e_1. Since e_1 has no successors, $D(e_1) = D_0(e_1) = 9$. Then d_1, d_2 and d_3 are considered. These tasks have one successor with deadline 9. So $D(d_i)$ is set to $9 - 1 - \lceil \frac{0}{2} \rceil =$

8. c_1 has three successors with deadline 8 and four successors with deadline at most 9. So $D(c_1) = \min\{8-1-\lceil\frac{2}{2}\rceil, 9-1-\lceil\frac{3}{2}\rceil\} = 6$. Then the deadlines of b_1, b_2 and b_3 are computed. These tasks have one successor with deadline 6, four successors with deadline at most 8 and five successors with deadline at most 9. Hence $D(b_i)$ is set to $\min\{6-1-\lceil\frac{0}{2}\rceil, 8-1-\lceil\frac{3}{2}\rceil, 9-1-\lceil\frac{4}{2}\rceil\} = 5$. Finally, Algorithm DEADLINE MODIFICATION considers a_1, a_2 and a_3. First consider a_2. It has three successors with deadline 5, four successors with deadline at most 6, seven successors with deadline at most 8 and eight successors with deadline at most 9. So $D(a_2) = \min\{5-1-\lceil\frac{2}{2}\rceil, 6-1-\lceil\frac{3}{2}\rceil, 8-1-\lceil\frac{6}{2}\rceil, 9-1-\lceil\frac{7}{2}\rceil\} = 3$. a_1 and a_3 have two successors with deadline 5, three successors with deadline at most 6, six successors with deadline at most 8 and seven successors with deadline at most 9. So the deadlines of a_1 and a_3 computed by Algorithm DEADLINE MODIFICATION equal $\min\{5-1-\lceil\frac{1}{2}\rceil, 6-1-\lceil\frac{2}{2}\rceil, 8-1-\lceil\frac{5}{2}\rceil, 9-1-\lceil\frac{6}{2}\rceil\} = 3$. The constructed instance $(G, 2, D)$ is strongly D_0-consistent.

Now we will prove that Algorithm DEADLINE MODIFICATION correctly constructs strongly D_0-consistent instances.

Lemma 4.2.2. *Let (G, μ, m, D) be the instance constructed by Algorithm* DEADLINE MODIFICATION *for an instance (G, μ, m, D_0). Then (G, μ, m, D) is strongly D_0-consistent.*

Proof. Algorithm DEADLINE MODIFICATION starts by setting $D(u) = D_0(u)$ for all tasks u of G. In each step, it computes a deadline for a task of G. Let $u_1, \dots, u_n$ be the order in which the tasks are considered. For all $i \leq n$, let G_i the subgraph of G induced by $\{u_1, \dots, u_i\}$. For all $i \leq n$ and all tasks u of G, let $D_i(u)$ be the deadline of u after the i^{th} step. Clearly, $D_i(u_j) = \cdots = D_n(u_j)$ for all $j \leq i$. Let $D_{\min,i}$ and $D_{\max,i}$ be the values of $D_{\min}$ and $D_{\max}$ after step i.

It will be proved by induction that the instances (G_i, μ, m, D_i) are strongly D_0-consistent. It is not difficult to see that (G_1, μ, m, D_1) is strongly D_0-consistent. Assume by induction that (G_i, μ, m, D_i) is strongly D_0-consistent. Consider $(G_{i+1}, \mu, m, D_{i+1})$. For all $j \leq i$, $D_{i+1}(u_j) = D_i(u_j)$. So (G_i, μ, m, D_{i+1}) is strongly D_0-consistent. Now consider u_{i+1}. Clearly, $D_{i+1}(u_{i+1}) \leq D_0(u_{i+1})$. Assume $N_{D_{i+1}}(u_{i+1}, d) \geq 1$ for some integer d. Then $D_{\min,i} \leq d \leq D_{\max,i}$. Hence $D_{i+1}(u_{i+1}) \leq d - 1 - \lceil\frac{1}{m}(N_{D_{i+1}}(u_{i+1}, d) - 1)\rceil$. So $(G_{i+1}, \mu, m, D_{i+1})$ is D_0-consistent. It is easy to see that if $D_{i+1}(u_{i+1}) \neq D_0(u_{i+1})$, then there is an integer d, such that $N_{D_{i+1}}(u_{i+1}, d) \geq 1$ and $D_{i+1}(u_{i+1}) = d - 1 - \lceil\frac{1}{m}(N_{D_{i+1}}(u_{i+1}, d) - 1)\rceil$. So $(G_{i+1}, \mu, m, D_{i+1})$ is strongly D_0-consistent. By induction, (G_n, μ, m, D_n) is strongly D_0-consistent. Since $G = G_n$ and $D(u) = D_n(u)$ for all tasks u of G, (G, μ, m, D) is strongly D_0-consistent. □

The time complexity of Algorithm DEADLINE MODIFICATION can be determined as follows. Consider an instance (G, μ, m, D_0). Algorithm DEADLINE MODIFICATION starts by computing $D_{\min}$ and $D_{\max}$ and setting $D(u) = D_0(u)$ for all tasks u of G. This takes $O(n)$ time. In each step, the algorithm computes a deadline of a task. This can be done using a reversed topological order of G. Such an order can be constructed in $O(n + e)$ time [18]. In order to bound the time complexity, we have to fill in a few details of Algorithm DEADLINE MODIFICATION. We distinguish two cases: whether or not G is known to be a transitive closure. If it is unknown whether G is a transitive closure, then Algorithm DEADLINE MODIFICATION should first compute the transitive closure of G. Coppersmith and Winograd [17] proved that the transitive closure of a precedence graph can be computed in $O(n^{2.376})$ time. Goralčíkova and Koubek [37] showed that it can be computed in $O(n+e+ne^-)$ time. In the remainder of the analysis of the time complexity of Algorithm DEADLINE MODIFICATION, we assume that G is a transitive closure.

For the computation of the strongly D_0-consistent deadline of a task u, we need to compute $N_D(u,d)$ for all d. These values can be computed by traversing the children v of u in G^+ and determining $\mu_D(v,d)$. This takes $O(|Succ_G(u)|)$ time for each d.

We can prove that Algorithm DEADLINE MODIFICATION needs to consider only $O(n)$ values of d for each task u. These are the values $D(v)$ and $D(v)-\mu(v)+1$ for some task v of G. Assume $d \neq D(v)$ and $d \neq D(v)-\mu(v)+1$ for all tasks v of G. Assume $N_D(u,d) \geq 1$. Then after Algorithm DEADLINE MODIFICATION has considered d, $D(u) \leq d-1-\lceil \frac{1}{m}(N_D(u,d)-1)\rceil$. Let k be the number of successors v of u, such that $D(v)-\mu(v)+1<d<D(v)$. We consider three cases.

Case 1. $k=0$.
Let $d' = \max\{D(w) \mid w \in V(G) \wedge D(w) < d\}$. Then $N_D(u,d') = N_D(u,d)$. After d' is considered by Algorithm DEADLINE MODIFICATION, $D(u) \leq d'-1-\lceil \frac{1}{m}(N_D(u,d')-1)\rceil \leq d-1-\lceil \frac{1}{m}(N_D(u,d)-1)\rceil$. In that case, d need not be considered by Algorithm DEADLINE MODIFICATION.

Case 2. $1 \leq k \leq m-1$.
Let $d' = \max\{D(w)-\mu(w)+1 \mid w \in Succ_G(u) \wedge D(w)-\mu(w)+1 < d\}$. Let v be a successor of u, such that $D(v)-\mu(v)+1 < d < D(v)$. Then $D(v)-\mu(v)+1 \leq d' < d < D(v)$. So $\mu_D(v,d') = \mu(v)-D(v)+d' = \mu(v)-D(v)+d-(d-d') = \mu_D(v,d)-(d-d')$. Hence $N_D(u,d') \geq N_D(u,d)-k(d-d') \geq N_D(u,d)-m(d-d')$. Moreover, $\mu_D(v,d') \geq 1$. So $N_D(u,d') \geq 1$. After d' was taken into account, $D(u) \leq d'-1-\lceil \frac{1}{m}(N_D(u,d')-1)\rceil \leq d'-1-\lceil \frac{1}{m}(N_D(u,d)-1-m(d'-d))\rceil = d-1-\lceil \frac{1}{m}(N_D(u,d)-1)\rceil$. So d need not be considered by Algorithm DEADLINE MODIFICATION.

Case 3. $k \geq m$.
Let $d' = \min\{D(w) \mid w \in Succ_G(u) \wedge D(w) > d\}$. Let v be a successor of u, such that $D(v)-\mu(v)+1 < d < D(v)$. Then $D(v) \geq d' \geq D(v)-\mu(v)+1$. So $\mu_D(v,d') = \mu(v)-D(v)+d' = \mu(v)-D(v)+d+(d'-d) = \mu_D(v,d)+(d'-d)$. Hence $N_D(u,d') \geq N_D(u,d)+k(d'-d) \geq N_D(u,d)+m(d'-d)$. After d' has been considered by Algorithm DEADLINE MODIFICATION, $D(u) \leq d'-1-\lceil \frac{1}{m}(N_D(u,d')-1)\rceil \leq d'-1-\lceil \frac{1}{m}(N_D(u,d)-1+m(d'-d))\rceil = d-1-\lceil \frac{1}{m}(N_D(u,d)-1)\rceil$. So d need not be considered by Algorithm DEADLINE MODIFICATION.

So the computation of the strongly D_0-consistent deadline of u takes $O(n|Succ_G(u)|)$ time. Since the outdegree of u in G^+ equals $|Succ_G(u)|$, this takes $O(n^2+ne^+)$ time in total. Hence we have proved the following result.

Lemma 4.2.3. *For all instances (G,μ,m,D_0), Algorithm* DEADLINE MODIFICATION *constructs the strongly D_0-consistent instance (G,μ,m,D) in $O(n^2+ne^+)$ time.*

A strongly D_0-consistent instance (G,m,D) can be computed more efficiently. The transitive closure G^+ of G can be constructed in $O(\min\{n^{2.376}, n+e+ne^-\})$ time. The values $N_D(u,d)$ can be computed by determining the number of successors v of u with deadline d for all d. These numbers are stored in an array and a prefix sum operation is applied on this array. Then we find $N_D(u,d)$ for all d in $O(|Succ_G(u)| + (D_{\max} - D_{\min}))$ time. Since there is a feasible schedule for (G,m,D) of length at most n, we may assume that $D_{\max} - D_{\min}$ is at most n. Consequently, the strongly D_0-consistent deadline of u can be computed in $O(n)$ time. Hence the strongly D_0-consistent instance (G,m,D) can be computed in $O(n^2 + \min\{n^{2.376}, n+e+ne^-\})$ time.

Lemma 4.2.4. *For all instances (G,m,D_0), Algorithm* DEADLINE MODIFICATION *constructs the strongly D_0-consistent instance (G,m,D) in $O(\min\{n^{2.376}, n^2+ne^-\})$ time.*

4.2.2 An unrestricted number of processors

Constructing strongly D_0-consistent instances (G,μ,∞,D) is less complicated than computing strongly D_0-consistent instances (G,μ,m,D). Let (G,μ,∞,D) be the strongly D_0-consistent instance. Let u be a task of G. Lemma 4.1.10 shows that if u has only one child v, then $D(u) = \min\{D_0(u), D(v) - \mu(v)\}$. Moreover, Lemma 4.1.11 states that if u has $k \geq 2$ children $v_1, \ldots, v_k$, such that $D(v_1) - \mu(v_1) \leq D(v_2) - \mu(v_2)$ and $D(v_2) - \mu(v_2) \leq D(v_i) - \mu(v_i)$ for all $i \geq 3$, then $D(u) = \min\{D_0(u), D(v_1) - \mu(v_1), D(v_2) - \mu(v_2) - 1\}$.

This can be used to construct strongly D_0-consistent instances (G,μ,∞,D). Consider an instance (G,μ,∞,D_0). Let $u_1, \ldots, u_n$ be a topological order of G. Assume that the strongly D_0-consistent deadlines of the tasks $u_{i+1}, \ldots, u_n$ have been computed. Consider task u_i. If u_i is a sink of G, then let $D(u_i) = D_0(u_i)$. If u_i has exactly one child v, then let $D(u_i) = \min\{D_0(u_i), D(v) - \mu(v)\}$. Otherwise, let $v_1, \ldots, v_k$ be the children of u_i, such that $D(v_1) - \mu(v_1) \leq D(v_2) - \mu(v_2)$ and $D(v_2) - \mu(v_2) \leq D(v_i) - \mu(v_i)$ for all $i \geq 3$. Then let $D(u_i) = \min\{D_0(u_i), D(v_1) - \mu(v_1), D(v_2) - \mu(v_2) - 1\}$. Clearly, the resulting instance (G,μ,∞,D) is strongly D_0-consistent.

Computing a topological order of a precedence graph G takes $O(n+e)$ time [18]. For each task u of G, $O(|Succ_{G,0}(u)|)$ time is required to find two children v_1 and v_2 of u, such that $D(v_1) - \mu(v_1) \leq D(v_2) - \mu(v_2)$ and $D(v_2) - \mu(v_2) \leq D(v_i) - \mu(v_i)$ for all $i \geq 3$. So $O(|Succ_{G,0}(u)|)$ time is used to compute the deadline of u. Consequently, the strongly D_0-consistent instance (G,μ,∞,D) can be computed in $O(n+e)$ time. Hence we have proved the following result.

Lemma 4.2.5. *For all instances (G,μ,∞,D_0), the strongly D_0-consistent instance (G,μ,∞,D) can be constructed in $O(n+e)$ time.*

4.3 List scheduling

The second step in the construction of feasible schedules uses a list scheduling approach. List scheduling is a common approach to multiprocessor scheduling that was introduced by Graham [38, 39] for scheduling without communication delays. His list scheduling algorithm has been generalised to many other scheduling problems. Rayward-Smith [79] was the first to use a list scheduling approach for scheduling precedence-constrained tasks subject to unit-length communication delays.

Basically, list scheduling works as follows. A list containing all tasks defines the priority among the tasks: the first tasks are more important than the last and should be scheduled at an earlier time. At each time, a list scheduling algorithm determines all tasks that are available at that time and schedules the available tasks with the smallest index in the priority list.

A schedule constructed by a list scheduling algorithm is determined by the priority list. This makes list scheduling a useful tool for constructing schedules: many scheduling algorithms consist of an algorithm that constructs a priority list and a list scheduling algorithm that uses this list to construct a schedule [4, 31, 32, 73, 76]. The same approach is used here: the list scheduling algorithm presented in this section uses a list of tasks ordered by non-decreasing strongly D_0-consistent deadlines to construct a schedule for an instance (G,μ,m,D_0).

Algorithm LIST SCHEDULING is shown in Figure 4.3. Using any list containing all tasks of G, it constructs feasible schedules for instances (G,μ,m,D). The following notation is used. t is the current time and N equals the number of tasks that are being executed at time t.

```
Algorithm LIST SCHEDULING
Input. An instance $(G,\mu,m,D)$ and a list $L$ containing all tasks of $G$.
Output. A feasible schedule $S$ for $(G,\mu,m,D)$.
1.  $t := 0$
2.  $N := 0$
3.  while there are unscheduled tasks
4.    do while there are unscheduled tasks available at time $t$ and $N < m$
5.         do let $u$ be the unscheduled available task with the smallest index in $L$
6.            $S(u) := t$
7.            $N := N+1$
8.       if $N = m$ or no unscheduled task is available at time $t$ or at time $t+1$
9.         then $t := \min\{S(u)+\mu(u) \mid S(u)+\mu(u) \geq t+1\}$
10.        else $t := t+1$
11.      $N := |\{v \in V(G) \mid S(v) \leq t < S(v)+\mu(v)\}|$
```

Figure 4.3. Algorithm LIST SCHEDULING

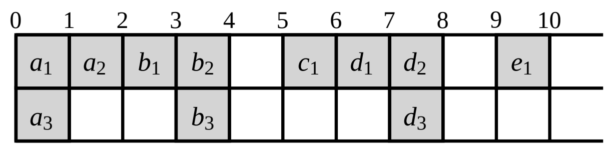

Figure 4.4. The schedule for $(G,2,D)$ constructed by Algorithm LIST SCHEDULING

Example 4.3.1. Let $(G,2,D)$ be the instance shown in Figure 4.1. Using priority list $L = (a_1,a_3,a_2,b_1,b_2,b_3,c_1,d_1,d_2,d_3,e_1)$, Algorithm LIST SCHEDULING constructs a schedule for $(G,2,D)$ as follows. a_1 and a_3 are sources of G with the smallest index in L. So a_1 and a_3 are scheduled at time 0. a_2 is the only task that is available at time 1. So it is scheduled at time 1. b_1, b_2 and b_3 are available at time 2. Since these tasks are all successors of a_2 and b_1 has the smallest index in L, only b_1 is scheduled at time 2. b_2 and b_3 are scheduled at time 3. c_1 becomes available at time 5. So it is scheduled at time 5. Only one successor of c_1 can be scheduled at time 6. Because d_1 is the child of c_1 with the smallest index in L, d_1 is the only task scheduled at time 6. d_2 and d_3 are scheduled at time 7. e_1 is scheduled at time 9, because that is the first time it becomes available. So Algorithm LIST SCHEDULING constructs the schedule shown in Figure 4.4.

Now we will prove that Algorithm LIST SCHEDULING correctly constructs feasible schedules.

Lemma 4.3.2. *Let S be the schedule for an instance (G,μ,m,D) constructed by Algorithm* LIST SCHEDULING *using a list containing all tasks of G. Then S is a feasible schedule for (G,μ,m,D).*

Proof. For all $i \leq n$, let u_i be the i^{th} task of G to be assigned a starting time by Algorithm LIST SCHEDULING. Then $S(u_1) \leq \cdots \leq S(u_n)$. For all $i \leq n$, let G_i be the subgraph of G induced by $\{u_1,\ldots,u_n\}$ and S^i the restriction of S to $\{u_1,\ldots,u_n\}$. It will be proved by induction that S^i is a feasible schedule for (G_i,μ,m,D) for all $i \leq n$. Clearly, S^1 is a feasible schedule for (G_1,μ,m,D). Assume by induction that S^i is a feasible schedule for (G_i,μ,m,D). $S^{i+1}(u) = S^i(u)$ for all tasks of G_i. Hence to determine the feasibility of S^{i+1} for (G_{i+1},μ,m,D), we only need to consider u_{i+1}. Since u_{i+1} is scheduled at time $S^{i+1}(u_{i+1})$, at most m tasks are being executed at time $S^{i+1}(u_{i+1})$. Since $S^{i+1}(u_1) \leq \cdots \leq S^{i+1}(u_{i+1})$, at most m tasks are being executed at each time $t \geq S^{i+1}(u_{i+1})$. Moreover, u_{i+1} is available at time $S^{i+1}(u_{i+1})$. So all predecessors of u_{i+1} are completed at or before time $S^{i+1}(u_{i+1})$, at most one parent of u_{i+1} finishes at time $S^{i+1}(u_{i+1})$ and if a parent of u_{i+1} finishes at time $S^{i+1}(u_{i+1})$, then no other child of this parent is scheduled at time $S^{i+1}(u_{i+1})$. So S^{i+1} is a feasible schedule for (G_{i+1},μ,m,D). By induction, S^n is a feasible schedule for (G_n,μ,m,D). Because $G = G_n$ and $S(u) = S^n(u)$ for all tasks u of G, S is a feasible schedule for (G,μ,m,D). □

Before we determine the time complexity of Algorithm LIST SCHEDULING, it is shown how Algorithm LIST SCHEDULING can be implemented. Consider an instance (G,μ,m,D). For all tasks u of G, let $par(u)$ be the number of parents of u that are not completed at or before time t. Let Av be the set of ready tasks that are available at time t and $Av1$ the set of ready tasks that become available at time $t+1$. The set $Active$ contains all tasks that are being executed at time t. At time 0, the sets Av, $Av1$ and $Active$ are empty, N equals zero and $par(u)$ equals the indegree of u for all tasks u of G.

Algorithm LIST SCHEDULING considers times t until all tasks have been assigned a starting time. At each time t, if at most $m-1$ tasks are being executed at time t, then the unscheduled available task with the smallest index in L is chosen. Let u be this task. u is scheduled at time t, removed from Av and added to $Active$. Moreover, N is increased by one. If a parent v of u finishes at time t, then the children of v in Av are no longer available at time t. These are moved from Av to $Av1$.

This is repeated until m tasks are executed at time t or there are no unscheduled tasks left that are available at time t. Then t is increased. If $N = m$, then the new time t is the next time at which a processor is idle. If there are no tasks that are available at time t or time $t+1$, then the new time t is the next time that a task finishes. Otherwise, $t+1$ is the new time. The tasks in $Av1$ are available at the new time t, so these are moved from $Av1$ to Av. Then we determine all tasks in $Active$ that finish at the new time t. These are removed from $Active$. For each of these tasks u, N is decreased by one and $par(v)$ is decreased by one for all children v of u. If $par(v)$ becomes zero, then it is added to Av or $Av1$. If exactly one parent of v finishes at time t, then v is added to Av. Otherwise, it is added to $Av1$.

The time complexity of Algorithm LIST SCHEDULING can be determined as follows. Obviously, a task is added to Av at most twice. Moreover, a task is added to $Active$ exactly once. Assume Av is represented by a balanced search tree (for instance, a red-black tree [18]) ordered by non-decreasing index in L and $Active$ by a balanced search tree ordered by non-decreasing completion time. Then adding and removing a task in Av or $Active$ takes $O(\log n)$ time. Moreover, the minimum element in Av or $Active$ can be found in $O(\log n)$ time. Since a task is added and removed at most three times, these operations take $O(n \log n)$ time in total. Because all tasks in $Av1$ are moved to Av simultaneously, $Av1$ can be represented by a queue. Then adding and removing tasks in $Av1$ takes $O(n)$ time in total.

If a task u finishes at time t, then $par(v)$ is decreased for all children v of u. This takes $O(|Succ_{G,0}(u)|)$ time, so $O(n+e)$ time in total. If $par(v)$ becomes zero, then v is added to Av or $Av1$ depending on the number of parents of v that finish at time t. This number can be found in $O(|Pred_{G,0}(v)|)$ time. Hence this requires $O(n+e)$ time in total.

If a task u is scheduled at time t and a parent v of u finishes at time t, then the available children of v are moved from Av to $Av1$. Since there is at most one such parent v, this takes $O(|Pred_{G,0}(u)| + |Succ_{G,0}(v)|)$ time apart from the time needed to move the tasks from Av to $Av1$. So this takes $O(n+e)$ time in total.

It is easy to see that assigning a starting time to all tasks takes $O(n)$ time. Moreover, at each time t considered by Algorithm LIST SCHEDULING, either a task starts or a task finishes. Therefore Algorithm LIST SCHEDULING considers at most $2n$ different times. Hence we have proved the following result.

Lemma 4.3.3. *For all instances (G, μ, m, D) and all lists L containing all tasks of G, Algorithm* LIST SCHEDULING *constructs a feasible schedule for (G, μ, m, D) in $O(n \log n + e)$ time using priority list L.*

Stadtherr [84] proved that using Union-Find operations [30], a list schedule for precedence graphs with unit-length tasks can be constructed in linear time. This method cannot easily be generalised for precedence graphs with tasks of arbitrary length.

Lemma 4.3.4. *For all instances (G, m, D) and all lists L containing all tasks of G, the schedule for (G, m, D) constructed by Algorithm* LIST SCHEDULING *using priority list L can be constructed in $O(n+e)$ time.*

The following observations state two important properties of schedules constructed by Algorithm LIST SCHEDULING. The first states that the schedules constructed by Algorithm LIST SCHEDULING are independent of the deadlines.

Observation 4.3.5. *Let L be a list containing all tasks of a precedence graph G. Let S and S' be the schedules for (G,μ,m,D) and (G,μ,m,D') constructed by Algorithm* List scheduling *using priority list L. Then $S(u) = S'(u)$ for all tasks u of G.*

The second observation states that if a task u is available at a time t and is scheduled at a later time, then no processor is idle at time t and all tasks with starting time t have a higher priority than u.

Observation 4.3.6. *Let L be a list containing all tasks of a precedence graph G. Let S be the schedule for (G,μ,m,D) constructed by Algorithm* List scheduling *using L. Let u_1 and u_2 be two tasks of G. If $S(u_1) < S(u_2)$ and u_2 is available at time $S(u_1)$, then u_1 has a smaller index in L than u_2 and there are m tasks v of G, such that $S(v) \leq S(u_1) < S(v) + \mu(v)$.*

4.4 Constructing feasible schedules

For strongly D_0-consistent instances (G,μ,m,D), we will consider the schedules for (G,μ,m,D_0) constructed by Algorithm List scheduling using a priority list L that is ordered by the latest possible starting time in an in-time schedule for (G,μ,m,D). Such a list will be called a *latest starting time list* or *lst-list* of (G,μ,m,D). More precisely, $L = (u_1,\ldots,u_n)$ is called an lst-list of (G,μ,m,D) if

$$D(u_1) - \mu(u_1) \leq D(u_2) - \mu(u_2) \leq \ldots \leq D(u_n) - \mu(u_n).$$

It is not difficult to see that an lst-list of the strongly D_0-consistent instance (G,μ,m,D) can be constructed in $O(n \log n)$ time. For instances (G,m,D), an lst-list is ordered by non-decreasing deadlines. For such instances, we may assume that the maximum deadline differs at most $n-1$ from the minimum deadline. Using bucket sort [18], an lst-list of (G,m,D) can be constructed in $O(n)$ time.

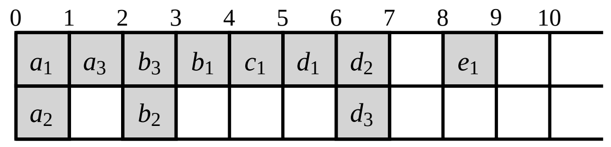

Figure 4.5. An in-time schedule for $(G,2,D_0)$

Example 4.4.1. Let $(G,2,D)$ be the instance shown in Figure 4.1. Let $D_0(u) = 9$ for all tasks u of G. Then $(G,2,D)$ is strongly D_0-consistent and $L = (a_1,a_3,a_2,b_1,b_2,b_3,c_1,d_1,d_2,d_3,e_1)$ is an lst-list of $(G,2,D)$. Using this list, Algorithm List scheduling constructs the schedule shown in Figure 4.4. This is not an in-time schedule for $(G,2,D_0)$: e_1 violates its deadline. An in-time schedule for $(G,2,D_0)$ is shown in Figure 4.5. This schedule can be constructed by Algorithm List scheduling using lst-list $(a_1,a_2,a_3,b_1,b_2,b_3,c_1,d_1,d_2,d_3,e_1)$ of $(G,2,D)$.

Example 4.4.1 shows that Algorithm LIST SCHEDULING does not necessarily construct minimum-tardiness schedules for an instance (G,m,D_0) using an lst-list of the strongly D_0-consistent instance (G,m,D). In this section, upper bounds on the tardiness of the schedules constructed by Algorithm LIST SCHEDULING are derived. Sections 4.4.1 and 4.4.2 consider schedules for arbitrary precedence graphs on a restricted and an unrestricted number of processors, respectively. Sections 4.4.3 and 4.4.4 are concerned with schedules for outforests on a restricted and an unrestricted number of processors, respectively.

4.4.1 Arbitrary graphs on a restricted number of processors

In this section, upper bounds on the tardiness of schedules for instances (G,m,D_0) constructed by Algorithm LIST SCHEDULING are derived. Hanen and Munier [44] considered precedence graphs that have two sources that are predecessors of all other tasks to compute an upper bound on the tardiness for instances (G,m,D_0) for which there is an in-time schedule. The following lemma was proved by Hanen and Munier [44]. We include a more detailed proof.

Lemma 4.4.2. *Let G be a precedence graph with two sources that are predecessors of all other tasks of G. Let (G,m,D) be the strongly D_0-consistent instance. Let S be a schedule for (G,m,D_0) constructed by Algorithm* LIST SCHEDULING *using an lst-list of (G,m,D). If there is an in-time schedule for (G,m,D_0), then for all tasks u of G, if $m=2$, then $S(u)+1 \leq 2D(u)-1$ and if $m \geq 3$, then $S(u)+1 \leq (3-\frac{3}{m})D(u)-(2-\frac{3}{m})$.*

Proof. Assume there is an in-time schedule for (G,m,D_0). From Lemma 4.1.9, there is an in-time schedule for (G,m,D). Let $\rho_2 = 2$ and $\rho_m = 3-\frac{3}{m}$ for all $m \geq 3$. It will be proved by contradiction that $S(u)+1 \leq \rho_m D(u)+(\rho_m-1)$ for all tasks u of G. Suppose there is a task u of G, such that $S(u)+1 > \rho_m D(u)-(\rho_m-1)$. Since there is an in-time schedule for (G,m,D), $D(v) \geq 1$ for all tasks v of G. Hence $\rho_m D(v)+(\rho_m-1) \geq 1$ for all tasks v of G. Because both sources of G are scheduled at time 0, u cannot be a source of G. Assume there is no task u', such that $S(u') < S(u)$ and $S(u')+1 > \rho_m D(u')-(\rho_m-1)$. Let $t=S(u)$. Let $S_{t'}$ be the last time slot before S_t, such that $S_{t'-1} \cup S_{t'}$ contains at most two tasks with deadline at most $D(u)$ and $S_{t'}$ contains at most one task with deadline at most $D(u)$. There is such a time t', because $S_0 \cup S_1$ only contains the two sources of G and S_1 does not contain any tasks.

Let H be the subgraph of G induced by $\{v \in \bigcup_{i=t'}^{t} S_i \mid D(v) \leq D(u)\}$. Since (G,m,D) is consistent, every predecessor of a task of H has a smaller deadline than u. We will prove that there is a task v scheduled at time $t'-1$ that is a predecessor of all tasks of H. We will consider two possibilities.

Case 1. $S_{t'}$ contains a task w with a smaller deadline than u.

Case 1.1. $S_{t'-1}$ contains a parent v of w.
From the choice of t', v is the only task in $S_{t'-1}$ with a smaller deadline than u. Let x be a source of $H[V(H) \setminus \{w\}]$. At most one task with a deadline smaller than that of x is scheduled at time t'. From Observation 4.3.6, x cannot be available at time t'. Since no two parents of x are scheduled at time $t'-1$, x must be a child of v or a child of w. In either case, x is a successor of v. So v is a predecessor of all tasks of H.

Case 1.2. $S_{t'-1}$ does not contain a parent of w.
Let x be a source of $H[V(H) \setminus \{w\}]$. From the choice of t', w is the only task with deadline at most $D(u)$ scheduled at time t'. From Observation 4.3.6, x cannot be available at time t'. From the choice of t', at most one parent of x is scheduled at time $t'-1$. Because no parent of w is scheduled at time $t'-1$ and x is not available at time t', x must be a child of w. Hence w is a predecessor of all tasks of $H[V(H) \setminus \{w\}]$. Because of communication delays, at most one successor of w can be executed at time $t'+1$. So $t' = t-1$, otherwise, t' would have been chosen differently. Since $D(w) \leq D(u)-1$, $S(w)+1 = t'+1 = (t+1)-1 > \rho_m D(u) - (\rho_m - 1) - 1 \geq \rho_m(D(w)+1) - \rho_m = \rho_m D(w) \geq \rho_m D(w) - (\rho_m - 1)$. Contradiction.

Case 2. $S_{t'}$ does not contain a task with a smaller deadline than u.
Let x be a source of H. From Observation 4.3.6, x cannot be available at time t'. Since $S_{t'}$ does not contain a parent of x, two parents of x must be executed at time $t'-1$. So $S_{t'-1}$ contains at least two tasks that are predecessors of all tasks of H. Let v be one of these tasks.

In either case, v is scheduled at time $t'-1$ and is a predecessor of all tasks of H. Now we will inductively construct a set of clusters. C_0 contains the tasks of H that are executed at time t. Assume C_i has been defined before. Let t_i be the smallest starting time of a task of C_i. Let t'_i be the largest time t'', such that $t'' < t_i$, $t'' \geq t'-1$ and at most $m-1$ tasks of H are executed at time t''. Then C_{i+1} is defined as follows.

1. If $t'_i = t'-1$, or no task of H is scheduled at time $t'_i - 1$, then let C_{i+1} be the set of tasks of H executed at time t'_i. Then C_{i+1} is said to be a cluster of Type 1.
2. Otherwise, C_{i+1} contains all tasks of H that are scheduled at time t'_i or $t'_i - 1$. Then C_{i+1} is said to be a cluster of Type 2.

Assume C_k is the last cluster that can be defined this way. Then v is an element of C_k. Let α_1 be the number of clusters of Type 1 and α_2 the number of clusters of Type 2. Note that cluster C_0 has no type. The clusters contain all tasks of H that are contained in a time slot that contains at most $m-1$ tasks of H. Between two consecutive clusters, only tasks of H are scheduled.

Consider two consecutive clusters C_i and C_{i+1}. It will be proved by contradiction that every task in C_i has a predecessor in C_{i+1}. Let x be a task in C_i. Suppose x does not have a predecessor in C_{i+1}. Then $C_{i+1} \neq C_k$, because C_k contains v and v is a predecessor of all tasks of H. At time t'_i, at most $m-1$ tasks with deadline at most $D(x)$ are scheduled. No predecessor of x is scheduled at time t'_i. From Observation 4.3.6, x is not available at time t'_i. So at least two predecessors of x must be scheduled at time $t'_i - 1$. Since (G, m, D) is consistent, these must be tasks of H. In that case, C_{i+1} is of Type 2 and these predecessors of x are elements of C_{i+1}. Contradiction. So every task in C_i has a predecessor in C_{i+1}. Since v is a predecessor of all tasks of H, there is a path from v to u, that contains a task in every cluster. Because u is an element of C_0, this path contains at least $\alpha_1 + \alpha_2 + 1$ tasks. Since (G, m, D) is consistent, $D(u) - D(v) \geq \alpha_1 + \alpha_2$.

From the choice of t', every cluster C_i of Type 2 contains at least three tasks and each cluster C_i of Type 1 contains at least two tasks, unless $i = k$. Now consider the same cases as before.

Case 1. $S_{t'}$ contains a task w with a smaller deadline than u.

v is a parent of w that is scheduled at time $t'-1$. If the last cluster is of Type 1, then it only contains v. Hence

$$\begin{aligned} N_D(v,D(u))-1 &\geq m(t-t')-(\alpha_1-1)(m-2)-\alpha_2(2m-3) \\ &= m(t-t')-\alpha_1(m-2)-\alpha_2(2m-3)+(m-2) \\ &\geq m(t-t')-(\alpha_1+\alpha_2)(2m-3)+(m-2). \end{aligned}$$

Otherwise, the last cluster is of Type 2 and

$$\begin{aligned} N_D(v,D(u))-1 &\geq m(t-t')-\alpha_1(m-2)-(\alpha_2-1)(2m-3)-(m-1) \\ &= m(t-t')-\alpha_1(m-2)-\alpha_2(2m-3)-(m-1)+(2m-3) \\ &\geq m(t-t')-(\alpha_1+\alpha_2)(2m-3)+(m-2). \end{aligned}$$

Case 2. $S_{t'}$ does not contain a task with a smaller deadline than u.

At time $t'-1$, two tasks with a smaller deadline than u are scheduled. One of these tasks is v. Since no task of H is scheduled at time t', the last cluster can only be of Type 2. Because no task of H is scheduled at time t',

$$\begin{aligned} N_D(v,D(u))-1 &\geq m(t-t')-\alpha_1(m-2)-(\alpha_2-1)(2m-3)-m \\ &= m(t-t')-\alpha_1(m-2)-\alpha_2(2m-3)-m+(2m-3) \\ &\geq m(t-t')-(\alpha_1+\alpha_2)(2m-3)+(m-3). \end{aligned}$$

In either case, $N_D(v,D(u))-1 \geq m(t-t')-(\alpha_1+\alpha_2)(2m-3)+(m-3)$. Because (G,m,D) is consistent, $D(v) \leq D(u)-1-\lceil \frac{1}{m}(N_D(v,D(u))-1)\rceil$. So

$$\begin{aligned} D(u)-D(v) &\geq 1+\lceil \tfrac{1}{m}(N_D(v,D(u))-1)\rceil \\ &\geq 1+\tfrac{1}{m}(m(t-t')-(\alpha_1+\alpha_2)(2m-3)+(m-3)) \\ &\geq t-t'-(\alpha_1+\alpha_2)(2-\tfrac{3}{m})+(2-\tfrac{3}{m}) \\ &\geq (S(u)+1)-(S(v)+1)-(D(u)-D(v))(2-\tfrac{3}{m})+(1-\tfrac{3}{m}). \end{aligned}$$

Since $S(u)+1 > \rho_m D(u)-(\rho_m-1)$, we obtain $S(v)+1 > \rho_m D(u)-(\rho_m-1)-(3-\frac{3}{m})(D(u)-D(v))+(1-\frac{3}{m})$. If $m \geq 3$, then

$$\begin{aligned} S(v)+1 &> (3-\tfrac{3}{m})D(u)-(2-\tfrac{3}{m})-(3-\tfrac{3}{m})(D(u)-D(v))+(1-\tfrac{3}{m}) \\ &\geq (3-\tfrac{3}{m})D(v)-(2-\tfrac{3}{m}). \end{aligned}$$

Contradiction. If $m=2$, then

$$\begin{aligned} S(v)+1 &> 2D(u)-1-(\tfrac{3}{2}D(u)-\tfrac{3}{2}D(v))-\tfrac{1}{2} \\ &= \tfrac{1}{2}D(u)+\tfrac{3}{2}D(v)-\tfrac{3}{2} \\ &\geq \tfrac{1}{2}(D(v)+1)+\tfrac{3}{2}D(v)-\tfrac{3}{2} \\ &= 2D(v)-1. \end{aligned}$$

Contradiction. □

By adding two dummy sources, any precedence graph can be transformed into a precedence graph with two sources that are predecessors of all other tasks. Using this construction, we can prove an upper bound on the tardiness of schedules for all instances (G,m,D_0).

Lemma 4.4.3. *Let (G,m,D) be the strongly D_0-consistent instance. Let S be a schedule for (G,m,D_0) constructed by Algorithm* LIST SCHEDULING *using an lst-list of (G,m,D). If there is an in-time schedule for (G,m,D_0), then for all tasks u of G, if $m=2$, then $S(u)+1 \leq 2D(u)+1$ and if $m \geq 3$, then $S(u)+1 \leq (3-\frac{3}{m})D(u)+(2-\frac{3}{m})$.*

Proof. Assume there is an in-time schedule for (G,m,D_0). Assume S is constructed by Algorithm LIST SCHEDULING using lst-list $L=(u_1,\ldots,u_n)$ of (G,m,D). Construct an instance (G',m,D') as follows. G' is constructed from G by adding two tasks r_1 and r_2 and arcs from r_1 and r_2 to all sources of G. For all tasks u of G, let $D_0'(u)=D_0(u)+2$ and $D'(u)=D(u)+2$. In addition, let $D_0'(r_1)=D_0'(r_2)=D'(r_1)=D'(r_2)=1$. From Observation 4.1.5, (G',m,D') is strongly D'-consistent. Because there is an in-time schedule for (G,m,D_0), there is also an in-time schedule for (G',m,D_0'). Let S' be the schedule for (G',m,D_0') constructed by Algorithm LIST SCHEDULING using the lst-list $L'=(r_1,r_2,u_1,\ldots,u_n)$ of (G',m,D'). From Lemma 4.4.2, if $m=2$, then for all tasks u of G', $S'(u) \leq 2D'(u)-1$ and if $m \geq 3$, then $S'(u) \leq (3-\frac{3}{m})D'(u)-(2-\frac{3}{m})$ for all tasks u of G'. It is easy to see that $S'(u)=S(u)+2$ for all tasks u of G. So if $m=2$, then for all tasks u of G, $S(u)+1=(S'(u)+1)-2 \leq 2D'(u)-3=2(D(u)+2)-3=2D(u)+1$. And if $m \geq 3$, then $S(u)+1=(S'(u)+1)-2 \leq (3-\frac{3}{m})D'(u)-(4-\frac{3}{m})=(3-\frac{3}{m})(D(u)+2)-(4-\frac{3}{m})=(3-\frac{3}{m})D(u)+(2-\frac{3}{m})$ for all tasks u of G. □

Using Lemma 4.1.8, we can bound the tardiness of the schedules for arbitrary instances (G,m,D_0) constructed using Algorithms DEADLINE MODIFICATION and LIST SCHEDULING.

Theorem 4.4.4. *There is an algorithm with an $O(\min\{n^2+ne^-,n^{2.376}\})$ time complexity that constructs feasible schedules S for instances (G,m,D_0), such that*

1. *if $m=2$, then the tardiness of S is at most $2\ell^*+\max_{u\in V(G)} D_0(u)+1$, and*
2. *if $m \geq 3$, then the tardiness of S is at most $(3-\frac{2}{m})\ell^*+(2-\frac{2}{m})\max_{u\in V(G)} D_0(u)+(2-\frac{2}{m})$,*

where ℓ^* is the tardiness of a minimum-tardiness schedule for (G,m,D_0).

Proof. Consider an instance (G,m,D_0). Define $\rho_2=2$ and $\rho_m=3-\frac{2}{m}$ for all $m \geq 3$. Let (G,m,D) be the strongly D_0-consistent instance. Let S be the schedule for (G,m,D_0) constructed by Algorithm LIST SCHEDULING using lst-list L of (G,m,D). Let ℓ^* be the tardiness of a minimum-tardiness schedule for (G,m,D_0). We will prove that the tardiness of S is at most $\rho_m\ell^*+(\rho_m-1)\max_{u\in V(G)} D_0(u)+(\rho_m-1)$. Define $D_0'(u)=D_0(u)+\ell^*$ for all tasks u of G. From Observation 4.1.7, there is an in-time schedule for (G,m,D_0'). Let (G,m,D') be the strongly D_0'-consistent instance. From Lemma 4.1.8, $D'(u)=D(u)+\ell^*$ for all tasks u of G. So L is an lst-list of (G,m,D'). From Lemma 4.4.3, $S(u)+1 \leq \rho_m D'(u)+(\rho_m -$

$1) \leq \rho_m(D_0(u)+\ell^*)+(\rho_m-1)$ for all tasks u of G. So the tardiness of S as schedule for (G,m,D_0) is at most $\rho_m\ell^*+(\rho_m-1)\max_{u\in V(G)}D_0(u)+(\rho_m-1)$. If $m=2$, then S has tardiness at most $2\ell^*+\max_{u\in V(G)}D_0(u)+1$. Otherwise, $m\geq 3$ and S has tardiness at most $(3-\frac{2}{m})\ell^*+(2-\frac{2}{m})\max_{u\in V(G)}D_0(u)+(2-\frac{2}{m})$. From Lemmas 4.2.4 and 4.3.4, S can be constructed in $O(\min\{n^2+ne^-,n^{2.376}\})$ time. □

Theorem 4.4.4 shows that there is a polynomial-time approximation algorithm for scheduling arbitrary precedence graphs with non-positive deadlines on m processors. The asymptotic approximation ratio of this algorithm equals 2 if $m=2$ and $3-\frac{3}{m}$ if $m\geq 3$.

Corollary 4.4.5. *There is an algorithm with an $O(\min\{n^2+ne^-,n^{2.376}\})$ time complexity that constructs feasible schedules S for instances (G,m,D_0) with non-positive deadlines, such that*

1. *if $m=2$, then the tardiness of S is at most $2\ell^*+1$, and*
2. *if $m\geq 3$, then the tardiness of S is at most $(3-\frac{2}{m})\ell^*+(2-\frac{2}{m})$,*

where ℓ^* is the tardiness of a minimum-tardiness schedule for (G,m,D_0).

Proof. Obvious from Theorem 4.4.4. □

4.4.2 Arbitrary graphs on an unrestricted number of processors

Bounding the tardiness of schedules constructed by Algorithm LIST SCHEDULING for instances (G,μ,∞,D_0) is less complicated. The following lemma proves an upper bound for instances (G,μ,∞,D_0) for which there is an in-time schedule.

Lemma 4.4.6. *Let (G,μ,∞,D) be the strongly D_0-consistent instance. Let S be a schedule for (G,μ,∞,D_0) constructed by Algorithm* LIST SCHEDULING *using an lst-list of (G,μ,∞,D). If there is an in-time schedule for (G,μ,∞,D_0), then for all tasks u of G, $S(u)+\mu(u)\leq 2D(u)-1$.*

Proof. Assume there is an in-time schedule for (G,μ,∞,D_0). From Lemma 4.1.9, there is an in-time schedule for (G,μ,∞,D). It will be proved by contradiction that $S(u)+\mu(u)\leq 2D(u)-1$ for all tasks u of G. Suppose there is a task u of G, such that $S(u)+\mu(u)>2D(u)-1$. We may assume that there is no task w, such that $S(w)<S(u)$ and $S(w)+\mu(w)>2D(w)-1$. Since there is an in-time schedule for (G,μ,∞,D) and all sources of G are scheduled at time zero, u cannot be a source of G. Let v be a parent of u with maximum completion time among the parents of u. Since (G,μ,∞,D) is consistent, $D(v)\leq D(u)-\mu(u)$. Since v is a parent of u with the largest completion time, u is available at time $S(v)+\mu(v)+1$. Hence u starts at time $S(v)+\mu(v)$ or at time $S(v)+\mu(v)+1$. Therefore $S(v)+\mu(v)\geq(S(u)+\mu(u))-(\mu(u)+1)>2D(u)-1-2\mu(u)\geq 2D(v)-1$. Contradiction. □

Lemma 4.4.6 is used to bound the tardiness of the schedule constructed for all instances (G,μ,∞,D_0).

Theorem 4.4.7. *There is an algorithm with an $O(n\log n+e)$ time complexity that constructs feasible schedules for instances (G,μ,∞,D_0) with tardiness at most $2\ell^*+\max_{u\in V(G)}D_0(u)-1$, where ℓ^* is the tardiness of a minimum-tardiness schedule for (G,μ,∞,D_0).*

Proof. Consider an instance (G,μ,∞,D_0). Let (G,μ,∞,D) be the strongly D_0-consistent instance. Let S be the schedule for (G,μ,∞,D_0) constructed by Algorithm LIST SCHEDULING using lst-list L of (G,μ,∞,D). Let ℓ^* be the tardiness of a minimum-tardiness schedule for (G,μ,∞,D_0). We will prove that the tardiness of S is at most $2\ell^* + \max_{u\in V(G)} D_0(u) - 1$. Define $D_0'(u) = D_0(u) + \ell^*$ for all tasks u of G. From Observation 4.1.7, there is an in-time schedule for (G,μ,∞,D_0'). Let (G,μ,∞,D') be the strongly D_0'-consistent instance. From Lemma 4.1.8, $D'(u) = D(u) + \ell^*$ for all tasks u of G. So L is an lst-list of (G,μ,∞,D'). From Lemma 4.4.6, $S(u) + \mu(u) \leq 2D'(u) - 1 \leq 2(D_0(u) + \ell^*) - 1$ for all tasks u of G. So the tardiness of S as schedule for (G,m,D_0) is at most $2\ell^* + \max_{u\in V(G)} D_0(u) - 1$. From Lemmas 4.2.5 and 4.3.3, S can be constructed in $O(n\log n + e)$ time. $\square$

Theorem 4.4.7 shows that there is a polynomial-time 2-approximation algorithm for scheduling arbitrary precedence graphs with non-positive deadlines on an unrestricted number of processors.

Corollary 4.4.8. *There is an algorithm with an $O(n\log n + e)$ time complexity that constructs feasible schedules for instances (G,μ,∞,D_0) with non-positive deadlines with tardiness at most $2\ell^* - 1$, where ℓ^* is the tardiness of a minimum-tardiness schedule for (G,μ,∞,D_0).*

Proof. Obvious from Theorem 4.4.7. $\square$

4.4.3 Outforests on a restricted number of processors

In this section, we consider schedules constructed by Algorithm LIST SCHEDULING for instances (G,m,D), such that G is an outforest. The bounds on the tardiness for these schedules are better than those for arbitrary precedence graphs proved in Section 4.4.1.

It will be proved that minimum-tardiness schedules for instances $(G,2,D_0)$, such that G is an outforest, can be constructed in polynomial time. In order to prove this, we need to bound the number of idle time slots in any schedule for the strongly D_0-consistent instance (G,m,D) constructed by Algorithm LIST SCHEDULING using an lst-list of (G,m,D).

Lemma 4.4.9. *Let G be an outforest. Let (G,m,D) be a consistent instance. Let S be a schedule for (G,m,D) constructed by Algorithm* LIST SCHEDULING *using an lst-list of (G,m,D). Then the number of idle time slots in S is at most $\max_{u\in V(G)} D(u) - \min_{u\in V(G)} D(u) + 1$.*

Proof. We inductively define a list of tasks $u_1,\ldots,u_k$ as follows. Let u_1 be a task with maximum completion time. If u_i is not a source of G, then let u_{i+1} be the parent of u_i. Assume u_k is the last task obtained this way. Then u_k is a source of G. Define $t_i = S(u_i)$ for all $i \in \{1,\ldots,k\}$. Define $I(t)$ as the number of idle slots in S from time t onward. It will be proved by induction that $I(t_i) \leq \max_{u\in V(G)} D(u) - D(u_i) + 1$ for all $i \in \{1,\ldots,k\}$. Clearly, $I(t_1) \leq 1 \leq \max_{u\in V(G)} D(u) - D(u_1) + 1$. Let $i \geq 1$. Assume by induction that $I(t_i) \leq \max_{u\in V(G)} D(u) - D(u_i) + 1$. Consider time t_{i+1}. We consider two cases.

Case 1. $I(t_{i+1}) - I(t_i) \leq 1$.
Since (G,m,D) is consistent, $D(u_{i+1}) \leq D(u_i) - 1$. So $I(t_{i+1}) \leq I(t_i) + 1 \leq \max_{u\in V(G)} D(u) - D(u_i) + 2 \leq \max_{u\in V(G)} D(u) - D(u_{i+1}) + 1$.

Case 2. $I(t_{i+1}) - I(t_i) \geq 2$.
Since G is an outforest, u_i is available at time $t_{i+1}+2$. From Observation 4.3.6, the time slots $S_{t_{i+1}+2}, \ldots, S_{t_i - 1}$ cannot be idle. So the time slots $S_{t_{i+1}}$ and $S_{t_{i+1}+1}$ must be idle. From Observation 4.3.6, u_i is not available at time $t_{i+1}+1$. Hence another child of u_{i+1} is executed at time $t_{i+1}+1$. Let v be this child. Since v is scheduled instead of u_i, $D(v) \leq D(u_i)$. Hence $N_D(u_{i+1}, D(u_i)) \geq 2$. Since (G,m,D) is consistent, $D(u_{i+1}) \leq D(u_i) - 2$. Consequently, $I(t_{i+1}) = I(t_i) + 2 \leq \max_{u \in V(G)} D(u) - D(u_i) + 3 \leq \max_{u \in V(G)} D(u) - D(u_{i+1}) + 1$.

In either case, $I(t_{i+1}) \leq \max_{u \in V(G)} D(u) - D(u_{i+1}) + 1$. By induction, $I(t_k) \leq \max_{u \in V(G)} D(u) - D(u_k) + 1$. Since u_k is a source of G, u_k is available at times $0, \ldots, S(u_k) - 1$. From Observation 4.3.6, no processor is idle before time $S(u_k)$. Hence $I(0) = I(t_k) \leq \max_{u \in V(G)} D(u) - D(u_k) + 1 \leq \max_{u \in V(G)} D(u) - \min_{u \in V(G)} D(u) + 1$. □

Lemma 4.4.9 is used to compute an upper bound on the tardiness of the schedules constructed by Algorithm LIST SCHEDULING for instances (G,m,D_0), such that G is an outtree.

Lemma 4.4.10. *Let G be an outtree. Let (G,m,D) be the strongly D_0-consistent instance. Let S be a schedule for (G,m,D_0) constructed by Algorithm* LIST SCHEDULING *using an lst-list L of (G,m,D). If there is an in-time schedule for (G,m,D_0), then for all tasks u of G, $S(u) + 1 \leq (2 - \frac{2}{m})D(u) - (1 - \frac{2}{m})$.*

Proof. Assume there is an in-time schedule for (G,m,D_0). From Lemma 4.1.9, there is an in-time schedule for (G,m,D). It will be proved by contradiction that $S(u)+1 \leq (2-\frac{2}{m})D(u) - (1-\frac{2}{m})$ for all tasks u of G. Suppose there is a task u, such that $S(u)+1 > (2-\frac{2}{m})D(u) - (1-\frac{2}{m})$. Because there is an in-time schedule for (G,m,D), $D(v) \geq 1$ for all tasks v of G. Since the root of G is scheduled at time 0, u cannot be the root of G. Assume $S(u) = t$ and there is no task v, such that $S(v) < t$ and $S(v)+1 > (2-\frac{2}{m})D(v) - (1-\frac{2}{m})$. Let t' be the last time before time t, such that at most one task with deadline at most $D(u)$ is scheduled at time t'. Such a time exists, because at time 0, only the root of G is executed. Because G is an outtree and (G,m,D) is consistent, a task v with deadline at most $D(u)$ is scheduled at time t'. Let H be the subgraph of G induced by $\{w \in \bigcup_{i=t'+1}^{t-1} S_i \mid D(w) \leq D(u)\} \cup \{u\}$.

Case 1. v is a predecessor of all tasks of H.
Because of communication delays, at most one successor of v can be scheduled immediately after v. Hence $t' = t-1$ and u is a child of v. Since (G,m,D) is consistent, $D(v) \leq D(u) - 1$ and $S(v)+1 = t = (S(u)+1) - 1 > (2-\frac{2}{m})D(u) - (2-\frac{2}{m}) - (1-\frac{2}{m}) = (2-\frac{2}{m})D(v) - (1-\frac{2}{m})$. Contradiction.

Case 2. Not every task of H is a successor of v.
Let x be a source of H that is not a successor of v. From Observation 4.3.6, x cannot be available at time t'. Because v is not a predecessor of x, a parent w of x must be scheduled at time $t'-1$ and another child of w is executed at time t'. Since this child is scheduled instead of x, it must have a deadline at most $D(x)$. Because v is the only task with deadline at most $D(u)$ scheduled at time $t'-1$, w is the parent of v as well. So all tasks of H are successors of

w. Let k be the number of time slots among time slots $S_{t'}, \ldots, S_{t-1}$ that contain at most $m-1$ tasks from H. Then $N_D(w, D(u)) \geq m(t-t') + 1 - k(m-2)$. Since (G, m, D) is consistent,

$$D(w) \;\leq\; D(u) - 1 - (t-t') + k(1 - \frac{2}{m}).$$

Let S' be the schedule for $(G[V(H) \cup \{w\}], m, D)$ constructed by Algorithm LIST SCHEDULING using the sublist of L containing all tasks in $V(H) \cup \{w\}$. From Lemma 4.4.9, the number of idle slots in S' is at most $D(u) - D(w) + 1$. It is not difficult to see that $S(x) = S'(x) + S(w) = S'(x) + t' - 1$ for all tasks x in $V(H) \cup \{w\}$. So the number of time slots in $S_{t'}, \ldots, S_{t-1}$ that contain at most $m-1$ tasks of H is at most $D(u) - D(w) - 1$. Hence

$$\begin{array}{rcl} D(u) - D(w) & \geq & (t-t') + 1 - k(1 - \frac{2}{m}) \\ & \geq & (t+1) - t' - (D(u) - D(w) - 1)(1 - \frac{2}{m}) \\ & \geq & (S(u)+1) - (S(w)+1) - (D(u) - D(w))(1 - \frac{2}{m}). \end{array}$$

As a result,

$$\begin{array}{rcl} S(w) + 1 & \geq & S(u) + 1 - (2 - \frac{2}{m})(D(u) - D(w)) \\ & > & (2 - \frac{2}{m})D(u) - (1 - \frac{2}{m}) - (2 - \frac{2}{m})(D(u) - D(w)) \\ & = & (2 - \frac{2}{m})D(w) - (1 - \frac{2}{m}). \end{array}$$

Contradiction.

□

An outforest can be transformed into an outtree by adding two tasks. This construction is used to compute upper bounds of the tardiness of the schedules constructed by Algorithm LIST SCHEDULING for instances (G, m, D_0), such that G is an outforest.

Lemma 4.4.11. *Let G be an outforest. Let (G, m, D) be the strongly D_0-consistent instance. Let S be a schedule for (G, m, D_0) constructed by Algorithm* LIST SCHEDULING *using an lst-list of (G, m, D). If there is an in-time schedule for (G, m, D_0), then for all tasks u of G, $S(u) + 1 \leq (2 - \frac{2}{m})D(u) + (1 - \frac{2}{m})$.*

Proof. Assume there is an in-time schedule for (G, m, D_0). Assume S is constructed by Algorithm LIST SCHEDULING using lst-list $L = (u_1, \ldots, u_n)$ of (G, m, D). If G has only one source, then G is an outtree. In that case, from Lemma 4.4.10, $S(u) + 1 \leq (2 - \frac{2}{m})D(u) - (1 - \frac{2}{m})$ for all tasks u of G. So we may assume that G has at least two sources. Construct an instance (G', m, D') as follows. G' is constructed from G by adding two tasks r and s and arcs from r to s, from s to u_1 (this is a source of G) and from r to all other sources of G. Then G' is an outtree. For all tasks u of G, let $D'(u) = D(u) + 2$. In addition, let $D'_0(r) = D'(r) = 1$ and $D'_0(s) = D'(s) = 2$. Then (G', m, D') is strongly D'-consistent. Because there is an in-time schedule for (G, m, D_0), there is also an in-time schedule for (G', m, D'_0). Let S' be the schedule for

(G',m,D'_0) constructed by Algorithm LIST SCHEDULING using lst-list $L'=(r,s,u_1,\dots,u_n)$ of (G',m,D'). From Lemma 4.4.10, $S'(u) \leq (2-\frac{2}{m})D'(u)-(1-\frac{2}{m})$ for all tasks u of G'. It is easy to see that $S'(u)=S(u)+2$ for all tasks u of G. So for all tasks u of G, $S(u)+1=(S'(u)+1)-2 \leq (2-\frac{2}{m})D'(u)-(1-\frac{2}{m})-2=(2-\frac{2}{m})(D(u)+2)-(3-\frac{2}{m})=(2-\frac{2}{m})D(u)+(1-\frac{2}{m})$. □

Lemma 4.4.11 can be used to bound the tardiness of the constructed schedules for all instances (G,m,D_0), such that G is an outforest.

Theorem 4.4.12. *There is an algorithm with an $O(n^2)$ time complexity that constructs feasible schedules for instances (G,m,D_0), such that G is an outforest, with tardiness at most $(2-\frac{2}{m})\ell^*+(1-\frac{2}{m})\max_{u\in V(G)} D_0(u)-(1-\frac{2}{m})$, where ℓ^* is the tardiness of a minimum-tardiness schedule for (G,m,D_0).*

Proof. Consider an instance (G,m,D_0), such that G is an outforest. Let (G,m,D) be the strongly D_0-consistent instance. Let S be the schedule for (G,m,D_0) constructed by Algorithm LIST SCHEDULING using lst-list L of (G,m,D). Let ℓ^* be the tardiness of a minimum-tardiness schedule for (G,m,D_0). We will prove that the tardiness of S is at most $(2-\frac{2}{m})\ell^*+(1-\frac{2}{m})\max_{u\in V(G)} D_0(u)+(1-\frac{2}{m})$. Define $D'_0(u)=D_0(u)+\ell^*$ for all tasks u of G. From Observation 4.1.7, there is an in-time schedule for (G,m,D'_0). Let (G,m,D') be the strongly D'_0-consistent instance. From Lemma 4.1.8, $D'(u)=D(u)+\ell^*$ for all tasks u of G. So L is an lst-list of (G,m,D'). From Lemma 4.4.11, $S(u)+1 \leq (2-\frac{2}{m})D'(u)+(1-\frac{2}{m}) \leq (2-\frac{2}{m})(D_0(u)+\ell^*)+(1-\frac{2}{m})$ for all tasks u of G. So the tardiness of S as schedule for (G,m,D_0) is at most $(2-\frac{2}{m})\ell^*+(1-\frac{2}{m})\max_{u\in V(G)} D_0(u)+(1-\frac{2}{m})$. From Lemmas 4.2.4 and 4.3.4, S can be constructed in $O(n^2)$ time. □

Theorem 4.4.12 shows that a minimum-tardiness schedule for an outforest on two processors can be constructed in polynomial time.

Theorem 4.4.13. *There is an algorithm with an $O(n^2)$ time complexity that constructs minimum-tardiness schedules for instances $(G,2,D_0)$, such that G is an outforest.*

Proof. Obvious from Theorem 4.4.12. □

Moreover, for all scheduling instances (G,m,D_0) with non-positive deadlines, such that G is an outforest, there is a polynomial-time approximation algorithm with an asymptotic approximation ratio of $2-\frac{2}{m}$.

Corollary 4.4.14. *There is an algorithm with an $O(n^2)$ time complexity that constructs feasible schedules for instances (G,m,D_0) with non-positive deadlines, such that G is an outforest, with tardiness at most $(2-\frac{2}{m})\ell^*+(1-\frac{2}{m})$, where ℓ^* is the tardiness of a minimum-tardiness schedule for (G,m,D_0).*

Proof. Obvious from Theorem 4.4.12. □

4.4.4 Outforests on an unrestricted number of processors

In this section, we will derive an upper bound on the tardiness of the constructed schedules for instances (G,μ,∞,D), such that G is an outforest, that is smaller than the upper bound for arbitrary instances (G,μ,∞,D) proved in Section 4.4.2: it will be proved that for all outforests G, minimum-tardiness schedules for instances (G,μ,∞,D_0) can be constructed in polynomial time. The basis of the proof is the following lemma.

Lemma 4.4.15. *Let G be an outforest. Let (G,μ,∞,D) be the strongly D_0-consistent instance. Let S be a schedule for (G,μ,∞,D_0) constructed by Algorithm* LIST SCHEDULING *using an lst-list of (G,μ,∞,D). If there is an in-time schedule for (G,μ,∞,D_0), then S is an in-time schedule for (G,μ,∞,D_0).*

Proof. Assume there is an in-time schedule for (G,μ,∞,D_0). From Lemma 4.1.9, there is an in-time schedule for (G,μ,∞,D). It will be proved by contradiction that S is an in-time schedule for (G,μ,∞,D_0). Suppose S is not an in-time schedule for (G,μ,∞,D_0). From Lemma 4.1.9, S is not an in-time schedule for (G,μ,∞,D). Assume task u does not finish at or before time $D(u)$ and there is no task that starts before u and violates its deadline. Since there is an in-time schedule for (G,μ,∞,D) and the sources of G are scheduled at time zero, u cannot be a source of G. Let v be the parent of u. Clearly, u is available at time $S(v)+\mu(v)+1$. So u starts at time $S(v)+\mu(v)$ or at time $S(v)+\mu(v)+1$.

Case 1. u starts at time $S(v)+\mu(v)$.

Let $d = D(u)-\mu(u)+1$. Then $N_D(v,d) \geq \mu_D(u,d) = 1$. Because (G,μ,∞,D) is consistent, $D(v) \leq d-1 = D(u)-\mu(u)$. Since u violates its deadline, $S(v)+\mu(v) = S(u) \geq D(u)-\mu(u)+1 \geq D(v)+1$. Contradiction.

Case 2. u starts at time $S(v)+\mu(v)+1$.

From Observation 4.3.6, u cannot be available at time $S(v)+\mu(v)$. So another child w of v starts at time $S(v)+\mu(v)$. Since Algorithm LIST SCHEDULING scheduled w instead of u, $D(w)-\mu(w) \leq D(u)-\mu(u)$. Let $d = D(u)-\mu(u)+1$. Then $N_D(v,d) \geq \mu_D(u,d)+\mu_D(w,d) \geq 2$. Because (G,μ,∞,D) is consistent, $D(v) \leq d-2 = D(u)-\mu(u)-1$. Because u is not completed at or before time $D(u)$, $S(u) \geq D(u)-\mu(u)+1$. So $S(v)+\mu(v) = S(u)-1 \geq D(u)-\mu(u) \geq D(v)+1$. Contradiction.

□

Using this result, we can prove that minimum-tardiness schedules for outforests on an unrestricted number of processors can be constructed in polynomial time.

Theorem 4.4.16. *There is an algorithm with an $O(n\log n)$ time complexity that constructs minimum-tardiness schedules for instances (G,μ,∞,D_0), such that G is an outforest.*

Proof. Consider an instance (G,μ,∞,D_0), such that G is an outforest. Let (G,μ,∞,D) be the strongly D_0-consistent instance. Let S be the schedule for (G,μ,∞,D_0) constructed by Algorithm LIST SCHEDULING using lst-list L of (G,μ,∞,D). We will prove that S is a minimum-tardiness schedule for (G,μ,∞,D_0). Let ℓ^* be the tardiness of a minimum-tardiness schedule for (G,μ,∞,D_0). Define $D'_0(u) = D_0(u) + \ell^*$ for all tasks u of G. From Observation 4.1.7, there is an in-time schedule for (G,μ,∞,D'_0). Let (G,μ,∞,D') be the strongly D'_0-consistent instance. From Lemma 4.1.8, $D'(u) = D(u) + \ell^*$ for all tasks u of G. So L is an lst-list of (G,μ,∞,D'). From Lemma 4.4.15, S is an in-time schedule for (G,μ,∞,D'_0). Hence $S(u) + \mu(u) \leq D'_0(u) \leq D_0(u) + \ell^*$ for all tasks u of G. So the tardiness of S as schedule for (G,μ,∞,D_0) is at most ℓ^*. So S is a minimum-tardiness schedule for (G,μ,∞,D_0). From Lemmas 4.2.5 and 4.3.3, S can be constructed in $O(n\log n)$ time. $\square$

4.5 Concluding remarks

In this chapter, an algorithm was presented for scheduling precedence-constrained tasks with non-uniform deadlines subject to unit-length communication delays. It is the first polynomial-time algorithm that constructs minimum-tardiness schedules (for outforests) subject to non-zero communication delays.

Most results presented in this chapter can be generalised in two ways. First, if we consider scheduling with release dates (a task cannot start before its release date) and deadlines, then minimum-tardiness schedules for outforests on two processors [88] and on an unrestricted number of processors can be constructed in polynomial time.

Second, if we consider $\{0,1\}$-communication delays instead of unit-length communication delays, then an algorithm similar to the one presented in this chapter constructs minimum-tardiness schedules for outforests on two processors or on an unrestricted number of processors. With $\{0,1\}$-communication delays, every arc has communication delay zero or one. If a task u_1 is a parent of u_2 and the arc from u_1 to u_2 has communication delay zero, then u_2 can be scheduled immediately after u_1 on any processor. If the delay of this arc equals one and u_2 is scheduled immediately after u_1, then it must be executed on the same processor as u_1.

5 The least urgent parent property

In Chapter 4, an algorithm was presented for scheduling precedence graphs with non-uniform deadlines subject to unit-length communication delays. This algorithm has the same overall structure as the one presented by Garey and Johnson [31] for scheduling without communication delays. In the first step, consistent deadlines are computed. In the second, the tasks are scheduled by a list scheduling algorithm.

The exact deadline modification for a task u depends on the subgraph of its successors: if u has sufficiently many successors that have to be completed at or before time d, then the deadline of u is decreased. For the case of scheduling on two processors without communication delays [31], this turns out to be sufficient: the algorithm of Garey and Johnson constructs minimum-tardiness schedules for arbitrary precedence graphs on two processors.

For scheduling subject to unit-length communication delays, we are only able to construct minimum-tardiness schedules for outforests on two processors or an unrestricted number of processors. In Chapter 4, Algorithm DEADLINE MODIFICATION was presented. This algorithm uses the knowledge that for every task u, at most one child of u can be scheduled immediately after u. However, it does not use the knowledge that at most one predecessor of u can be scheduled immediately before u.

In this chapter, we will consider instances that satisfy a special constraint, called the least urgent parent property. For instances with the least urgent parent property, every task u that is not a source has a parent that is the best candidate to be scheduled immediately before u. We can construct minimum-tardiness schedules for arbitrary precedence graphs with the least urgent parent property on an unrestricted number of processors and for inforests with the least urgent parent property on m processors. By transforming arbitrary instances into instances with the least urgent parent property and constructing schedules for these instances, we obtain a 2-approximation algorithm for scheduling inforests with non-positive deadlines on m processors.

5.1 The least urgent parent property

The least urgent parent property entails that every task that is not a source has a parent that is the best candidate to be executed immediately before this task. This least urgent parent has a deadline that exceeds the deadlines of all other parents.

Definition 5.1.1. An instance (G, μ, m, D) has the *least urgent parent property* if for all tasks u of G, if u is not a source, then u has a parent whose deadline exceeds the deadlines of the other parents of u. This parent is called the *least urgent parent* of u.

In a schedule with the least urgent parent property, the completion time of the least urgent parent of a task exceeds the completion times of the other parents.

Definition 5.1.2. Let (G, μ, m, D) be an instance with the least urgent parent property. Let S be a feasible schedule for (G, μ, m, D). S is a schedule for (G, μ, m, D) with the *least urgent parent property* if for all tasks u of G, if u is not a source of G, then the least urgent parent of u finishes after the other parents of u.

The least urgent parent property is closely related to the favoured child property that was introduced by Lawler [59]. A schedule S for an instance (G,m,D) has the favoured child property if for each task u of G, a child of u is scheduled before all other children of u. This child is the favoured child of u.

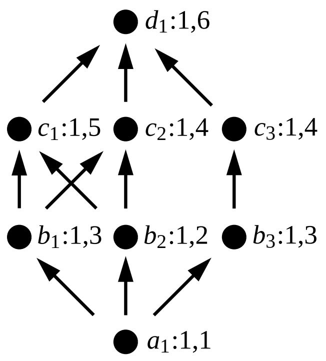

Figure 5.1. An instance $(G,2,D)$ with the least urgent parent property

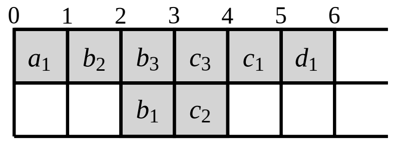

Figure 5.2. A schedule for $(G,2,D)$ with the least urgent parent property

Example 5.1.3. Figure 5.1 shows an instance $(G,2,D)$ with the least urgent parent property. a_1 is the least urgent parent of b_1, b_2 and b_3, b_1 is the least urgent parent of c_1 and c_2, b_3 is the least urgent parent of c_3 and c_1 is the least urgent parent of d_1. Figure 5.2 shows a feasible schedule for $(G,2,D)$ with the least urgent parent property.

5.2 Using the least urgent parent property

In this section, it will be proved that for all consistent instances (G,μ,∞,D) with the least urgent parent property, Algorithm LIST SCHEDULING, that was presented in Chapter 4, constructs in-time schedules if such schedules exist. In fact, this is proved for all instances (G,μ,∞,D), such that each task u of G has at most one parent with deadline $D(u)-\mu(u)$. Obviously, all consistent instances with the least urgent parent property satisfy this constraint.

Lemma 5.2.1. *Let (G,μ,∞,D) be the strongly D_0-consistent instance. Let S be a schedule for (G,μ,∞,D_0) constructed by Algorithm* LIST SCHEDULING *using an lst-list of (G,μ,∞,D). If every task u of G has at most one parent with deadline $D(u)-\mu(u)$ and there is an in-time schedule for (G,μ,∞,D_0), then S is an in-time schedule for (G,μ,∞,D_0).*

Proof. Assume there is an in-time schedule for (G,μ,∞,D_0) and every task u of G has at most one parent with deadline $D(u)-\mu(u)$. It will be proved by contradiction that S is an in-time schedule for (G,μ,∞,D_0). Suppose S is not an in-time schedule for (G,μ,∞,D_0). From Lemma 4.1.9, S is not an in-time schedule for (G,μ,∞,D). Let u be a task with an earliest starting time that violates its deadline. Then $S(u)+\mu(u) > D(u)$ and there is no task v, such that $S(v) < S(u)$ and $S(v)+\mu(v) > D(v)$. Because there is an in-time schedule for (G,μ,∞,D) and the sources of G are scheduled at time 0, u cannot be a source of G. Let v_1 be a parent of u with the largest completion time among the parents of u. Since u is available at time $S(v_1)+\mu(v_1)+1$, u is scheduled at time $S(v_1)+\mu(v_1)$ or at time $S(v_1)+\mu(v_1)+1$.

Case 1. $S(u) = S(v_1)+\mu(v_1)$.
Since (G,μ,∞,D) is consistent, $D(v_1) \le D(u)-\mu(u)$. Hence $S(v_1)+\mu(v_1) = S(u) > D(u)-\mu(u) \ge D(v_1)$. Contradiction.

Case 2. $S(u) = S(v_1)+\mu(v_1)+1$.

Case 2.1. v_1 is the only parent of u that finishes at time $S(v_1)+\mu(v_1)$.
From Observation 4.3.6, u is not available at time $S(v_1)+\mu(v_1)$. So another child w of v_1 starts at time $S(v_1)+\mu(v_1)$. Since Algorithm LIST SCHEDULING scheduled w instead of u, $D(w)-\mu(w) \le D(u)-\mu(u)$. From Lemma 4.1.11, $D(v_1) \le D(u)-\mu(u)-1$. So $S(v_1)+\mu(v_1) = S(u)-1 > D(u)-\mu(u)-1 \ge D(v_1)$. Contradiction.

Case 2.2. At least two parents of u finish at time $S(v_1)+\mu(v_1)$.
Let v_2 be another parent of u that finishes at time $S(v_1)+\mu(v_1)$. Assume $D(v_1) \le D(v_2)$. Because at most one parent of u has deadline $D(u)-\mu(u)$, $D(v_1) \le D(u)-\mu(u)-1$. Hence $S(v_1)+\mu(v_1) = S(u)-1 > D(u)-\mu(u)-1 \ge D(v_1)$. Contradiction.

□

This shows that for instances with the least urgent parent property, minimum-tardiness schedules can be constructed in polynomial time.

Theorem 5.2.2. *There is an algorithm with an $O(n\log n+e)$ time complexity that constructs minimum-tardiness schedules for instances (G,μ,∞,D_0), such that the strongly D_0-consistent instance (G,μ,∞,D) has the least urgent parent property.*

Proof. Consider an instance (G,μ,∞,D_0). Let (G,μ,∞,D) be the strongly D_0-consistent instance. Assume (G,μ,∞,D) has the least urgent parent property. Then every task u of G has at most one parent with deadline $D(u)-\mu(u)$. Let S be the schedule for (G,μ,∞,D_0) constructed by Algorithm LIST SCHEDULING using lst-list L of (G,μ,∞,D). We will prove that S is a minimum-tardiness schedule for (G,μ,∞,D_0). Let ℓ^* be the tardiness of a minimum-tardiness schedule for (G,μ,∞,D_0). Define $D_0'(u) = D_0(u)+\ell^*$ for all tasks u of G. From Observation 4.1.7, there is an in-time schedule for (G,μ,∞,D_0'). Let (G,μ,∞,D') be the strongly D_0'-consistent instance. From Lemma 4.1.8, $D'(u) = D(u)+\ell^*$ for all tasks u of G. So L is an lst-list of (G,μ,∞,D') and every task u of G has at most one parent with deadline $D'(u)-\mu(u)$. From Lemma 5.2.1, S is an in-time schedule for (G,μ,∞,D_0'). Hence $S(u)+\mu(u) \le D_0'(u) = D_0(u)+\ell^*$ for all tasks u of

G. So the tardiness of S as schedule for (G,μ,∞,D_0) is at most ℓ^*. So S is a minimum-tardiness schedule for (G,μ,∞,D_0). From Lemmas 4.2.5 and 4.3.3, S can be constructed in $O(n\log n+e)$ time. □

5.3 List scheduling with the least urgent parent property

In this section, we present an algorithm that constructs schedules with the least urgent parent property on a restricted number of processors for precedence graphs with unit-length tasks. We will use an algorithm that is similar to Algorithm LIST SCHEDULING. Algorithm LEAST URGENT PARENT LIST SCHEDULING is presented in Figure 5.3. The starting time of the least urgent parent of a task u is determined after all other parents u are completed. Unfortunately, for instances (G,μ,m,D) with the least urgent parent property, the least urgent parent of a task u of G could start before and finish after another parent of u in a schedule for (G,μ,m,D) with the least urgent parent property. Since Algorithm LIST SCHEDULING does not schedule a task at an earlier time than a task that was already scheduled, Algorithm LEAST URGENT PARENT LIST SCHEDULING will only be used for instances (G,m,D) with the least urgent parent property.

We use the same notation as for Algorithm LIST SCHEDULING. t is the current time. N is the number of tasks scheduled at time t. Moreover, an available task u will be called *lup-available* at time t if it is available at time t and if u is the least urgent parent of a task v, then all other parents of v finish at or before time t.

Algorithm LEAST URGENT PARENT LIST SCHEDULING
Input. An instance (G,m,D) with the least urgent parent property and a list L containing all tasks of G.
Output. A feasible schedule S for (G,m,D) with the least urgent parent property.

```
1. $t := 0$
2. $N := 0$
3. while there are unscheduled tasks
4.   do while there are unscheduled tasks lup-available at time $t$ and $N < m$
5.        do let $u$ be the unscheduled lup-available task with the smallest index in $L$
6.           $S(u) := t$
7.           $N := N+1$
8.      $t := t+1$
9.      $N := 0$
```

Figure 5.3. Algorithm LEAST URGENT PARENT LIST SCHEDULING

Example 5.3.1. Consider the instance $(G,2,D)$ shown in Figure 5.1. $(G,2,D)$ has the least urgent parent property. Using priority list $L=(a_1,b_2,b_1,b_3,c_3,c_2,c_1,d_1)$, Algorithm LEAST URGENT PARENT LIST SCHEDULING constructs a schedule for $(G,2,D)$ as follows. At time 0, a_1 is scheduled, because a_1 is not the least urgent parent of a task with at least two unscheduled parents. b_2 and b_3 become lup-available at time 1; b_1 does not, because it is the least urgent parent of c_1 and c_2, and b_2 is another unscheduled parent of c_1 and c_2. At time 1, b_2 is scheduled, because it has a smaller index in L than b_3. After b_2 has been scheduled, b_1 is the only unscheduled parent of c_1 and c_2. Hence b_1 becomes lup-available at time 2. Tasks b_1 and b_3 are scheduled at time 2. Then c_2 and c_3 become lup-available at time 3. Since c_1 is the least urgent parent of d_1, it is not lup-available at time 3. Both c_2 and c_3 are scheduled at time 3. Thereafter, c_1 is scheduled at time 4 and d_1 at time 5. Hence we obtain the schedule shown in Figure 5.2. This schedule has the least urgent parent property.

Now we will prove that Algorithm LEAST URGENT PARENT LIST SCHEDULING correctly constructs feasible schedules with the least urgent parent property.

Lemma 5.3.2. *Let (G,m,D) be an instance with the least urgent parent property. Let S be the schedule for (G,m,D) constructed by Algorithm* LEAST URGENT PARENT LIST SCHEDULING *using a list containing all tasks of G. Then S is a feasible schedule for (G,m,D) with the least urgent parent property.*

Proof. For all $i \leq n$, let u_i be the i^{th} task of G to be assigned a starting time by Algorithm LEAST URGENT PARENT LIST SCHEDULING. Then $S(u_1) \leq \cdots \leq S(u_n)$. For all $i \leq n$, let G_i be the subgraph of G induced by $\{u_1,\ldots,u_i\}$ and S^i the restriction of S to $\{u_1,\ldots,u_i\}$. Then the instances (G_i,m,D) all have the least urgent parent property. It will be proved by induction that S^i is a feasible schedule for (G_i,m,D) with the least urgent parent property for all $i \in \{1,\ldots,n\}$. Clearly, S^1 is a feasible schedule for (G_1,m,D) with the least urgent parent property. Assume by induction that S^i is a feasible schedule for (G_i,m,D) with the least urgent parent property. Because $S^{i+1}(u) = S^i(u)$ for all tasks u of G_i, we only need to consider u_{i+1} to determine the feasibility of S^{i+1} for (G_{i+1},m,D). Since u_{i+1} is scheduled at time $S^{i+1}(u_{i+1})$, at most m tasks are scheduled at time $S^{i+1}(u_{i+1})$. Moreover, u_{i+1} is available at time $S^{i+1}(u_{i+1})$, because it is lup-available at time $S^{i+1}(u_{i+1})$. So all predecessors of u_{i+1} are completed at or before time $S^{i+1}(u_{i+1})$, at most one parent of u_{i+1} finishes at time $S^{i+1}(u_{i+1})$, and if a parent v of u_{i+1} finishes at time $S^{i+1}(u_{i+1})$, then no other child of v is scheduled at time $S^{i+1}(u_{i+1})$. So S^{i+1} is a feasible schedule for (G_{i+1},m,D). In addition, if u_{i+1} is the least urgent parent of a task v, then it is scheduled after all other parents of v, since u_{i+1} is lup-available at time $S^{i+1}(u_{i+1})$. So S^{i+1} is a feasible schedule for (G_{i+1},m,D) with the least urgent parent property. By induction, S^n is a feasible schedule for (G_n,m,D) with the least urgent parent property. Because $G_n = G$ and $S^n(u) = S(u)$ for all tasks u of G, S is a feasible schedule for (G,m,D) with the least urgent parent property. □

Algorithm LEAST URGENT PARENT LIST SCHEDULING can be implemented as follows. Consider an instance (G,m,D) with the least urgent parent property. For all tasks u of G, let $par(u)$ be the number of parents of u that are not completed at or before time t and $lup(u)$ the number of children v of u, such that u is the least urgent parent of v and the number of unscheduled parents of v is at least two. Then an available task u is lup-available if $lup(u) = 0$. A task u will be called *lup-ready* if $par(u) = 0$ and $lup(u) = 0$. Av is the set of lup-ready tasks that are lup-available at time t, and $Av1$ the set of lup-ready tasks that become lup-available at time $t+1$. At time 0, the sets Av and $Av1$ are empty, N equals zero, and for all tasks u of G, $par(u)$ equals the indegree of u and $lup(u)$ the number of children v of u with indegree at least two, such that u is the least urgent parent of v.

Algorithm LEAST URGENT PARENT LIST SCHEDULING considers times t until all tasks have been assigned a starting time. At each time t, the unscheduled lup-available task with the smallest index in L is chosen. Assume u is this task. u is scheduled at time t and removed from Av. Moreover, N is increased by one. If a parent v of u finishes at time t, then the children of v in Av are no longer lup-available at time t, because u is scheduled at time t. So the children of v are moved from Av to $Av1$.

This is repeated until m tasks are scheduled at time t or there are no unscheduled lup-available tasks. Then t is increased by one. Because the tasks in $Av1$ becomes available at the new time t, the tasks of $Av1$ are moved to Av. Then all tasks that finish at the new time t are considered. For each of these tasks u, $par(v)$ is decreased by one for all children v of u. If $par(v)$ and $lup(v)$ both equal zero, then v is lup-ready at time t. Then v is added to Av or $Av1$. If exactly one parent of v finishes at time t, then v is lup-available at time t and it is added to Av. Otherwise, it is added to $Av1$, because it becomes lup-available at time $t+1$. In addition, if $par(v)$ becomes one, then $lup(w)$ can be decreased for the least urgent parent w of v. If $par(w)$ and $lup(w)$ both equal zero, then w is lup-ready at time t. If at most one parent of w is scheduled at time $t-1$, then w is added to Av. Otherwise, it is added to $Av1$, because it becomes lup-available at time $t+1$.

The time complexity of Algorithm LIST SCHEDULING can be determined as follows. Obviously, a task is added to Av at most twice. Assume Av is represented by a balanced search tree ordered by non-decreasing index in L. Then adding and removing a task in Av takes $O(\log n)$ time. In addition, the smallest element of Av can be found in $O(\log n)$ time. Because a task is added and removed at most twice, these operations take $O(n \log n)$ time in total. $Av1$ can be represented by a queue. Because all tasks in $Av1$ are moved to Av simultaneously, adding and removing tasks in $Av1$ takes $O(n)$ time in total.

If a task u finishes at time t, then $par(v)$ is decreased for all children v of u. This takes $O(|Succ_{G,0}(u)|)$ time, so $O(n+e)$ time in total. If $par(v)$ becomes zero and $lup(v)$ equals zero, then v is added to Av or $Av1$ depending on the number of parents of v that finish at time t. This number can be found in $O(|Pred_{G,0}(u)|)$ time. Hence this requires $O(n+e)$ time in total. If $par(v)$ becomes one, then $lup(w)$ is decreased by one for the least urgent parent w of v. If $lup(w)$ and $par(w)$ both equal zero, then w is added to Av or $Av1$. Because every task has exactly one least urgent parent, this requires $O(n+e)$ time in total.

If a task u is scheduled at time t and a parent v of u finishes at time t, then the lup-available children of v are moved from Av to $Av1$. Since there is at most one such parent v, this takes $O(|Pred_{G,0}(u)| + |Succ_{G,0}(v)|)$ time apart from the time needed to move the tasks from Av to $Av1$. So this takes $O(n+e)$ time in total.

It is easy to see that assigning a starting time to every task of G takes $O(n)$ time. Moreover, it is not difficult to see that the length of the schedule constructed by Algorithm LEAST URGENT PARENT LIST SCHEDULING is at most n. Hence we have proved the following result.

Lemma 5.3.3. *For all instances (G,m,D) with the least urgent parent property and all lists L containing all tasks of G, Algorithm* LEAST URGENT PARENT LIST SCHEDULING *constructs a feasible schedule for (G,m,D) with the least urgent parent property in $O(n \log n + e)$ time using priority list L.*

Because any consistent instance (G,m,D), such that G is an outforest, has the least urgent parent property, Algorithms LIST SCHEDULING and LEAST URGENT PARENT LIST SCHEDULING construct the same schedule for instances (G,m,D), such that G is an outforest.

Observation 5.3.4. *Let G be an outforest. Let L be a list containing all tasks of G. Let S be the schedule for (G,m,D) constructed by Algorithm* LIST SCHEDULING *using L and S' the schedule for (G,m,D) constructed by Algorithm* LEAST URGENT PARENT LIST SCHEDULING *using L. Then $S(u) = S'(u)$ for all tasks u of G.*

The following observation states an important property of schedules constructed by Algorithm LEAST URGENT PARENT LIST SCHEDULING. It is similar to Observation 4.3.6 that states a property of schedules constructed by Algorithm LIST SCHEDULING: it states that if a task u is lup-available at time t and u is scheduled at a later time, then no processor is idle at time t and all tasks scheduled at time t have a higher priority than u.

Observation 5.3.5. *Let (G,m,D) be an instance with the least urgent parent property. Let S be the schedule for (G,m,D) constructed by Algorithm* LEAST URGENT PARENT LIST SCHEDULING *using list L containing all tasks of G. Let u_1 and u_2 be two tasks of G. If $S(u_1) < S(u_2)$ and u_2 is lup-available at time $S(u_1)$, then u_1 has a smaller index in L than u_2 and there are m tasks v of G, such that $S(v) = S(u_1)$.*

5.4 Inforests

In this section, I will present an approximation algorithm for scheduling inforests. It will be proved in Section 5.4.1 that Algorithm LEAST URGENT PARENT LIST SCHEDULING can be used to construct minimum-tardiness schedules for inforests with the least urgent parent property. In Section 5.4.2, this result is used to present a 2-approximation algorithm for scheduling arbitrary inforests. This algorithm transforms an arbitrary instance into an instance with the least urgent parent property and uses Algorithm LEAST URGENT PARENT LIST SCHEDULING to construct a schedule whose tardiness is at most twice the tardiness of a minimum-tardiness schedule.

5.4.1 Constructing minimum-tardiness schedules

In this section, we will consider the schedules for instances with the least urgent parent property constructed by Algorithm LEAST URGENT PARENT LIST SCHEDULING. This algorithm does not construct minimum-tardiness schedules for all instances with the least urgent parent property.

Example 5.4.1. Consider the instance $(G,2,D)$ shown in Figure 5.4. This instance has the least urgent parent property. In any in-time schedule for $(G,2,D)$, a_1 and a_2 are scheduled at time 0. In fact, there is only one in-time schedule for $(G,2,D)$ and it is shown in Figure 5.5. So there is no in-time schedule for $(G,2,D)$ with the least urgent parent property.

Example 5.4.1 shows that Algorithm LEAST URGENT PARENT LIST SCHEDULING does not construct minimum-tardiness schedules for arbitrary precedence graphs with the least urgent parent property. However, we will show that it does construct such schedules for inforests with the least urgent parent property.

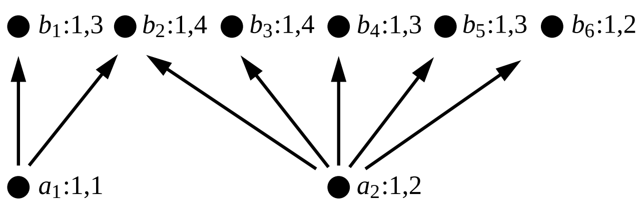

Figure 5.4. An instance $(G,2,D)$ with the least urgent parent property

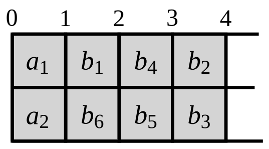

Figure 5.5. The only in-time schedule for $(G,2,D)$

Lemma 5.4.2. *Let G be an inforest. Let (G,m,D) be the strongly D_0-consistent instance. If (G,m,D) has the least urgent parent property and there is an in-time schedule for (G,m,D_0), then any schedule for (G,m,D_0) constructed by Algorithm* LEAST URGENT PARENT LIST SCHEDULING *using an lst-list of (G,m,D) is an in-time schedule for (G,m,D_0).*

Proof. Assume there is an in-time schedule for (G,m,D_0) and (G,m,D) has the least urgent parent property. From Lemma 4.1.9, there is an in-time schedule for (G,m,D). Let S be a schedule for (G,m,D_0) constructed by Algorithm LEAST URGENT PARENT LIST SCHEDULING using an lst-list of (G,m,D). It will be proved by contradiction that S is an in-time schedule for (G,m,D_0). Suppose S is not an in-time schedule for (G,m,D_0). From Lemma 4.1.9, S is not an in-time schedule for (G,m,D). Let S_t be the earliest time slot that contains a task u, such that $D(u) \leq t$. Since there is an in-time schedule for (G,m,D), there are at most mt tasks with deadline at most t. Let $S_{t'-1}$ be the last time slot before S_t that contains at most $m-1$ tasks with deadline at most t. Let H be the subgraph of G induced by $\bigcup_{i=t'}^{t-1} S_i \cup \{u\}$. Then H contains $m(t-t')+1$ tasks with deadline at most t. Define $Q = \{v \in S_{t'-1} \mid D(v) \leq t\}$.

Case 1. $t = t'$.

From Observation 5.3.5, u cannot be lup-available at time $t'-1$.

Case 1.1. u is available at time $t'-1$.

Then u is the least urgent parent of a task v, such that at least two parents of v are not scheduled before time $t'-1$. Since u is scheduled at time t, another parent w of v must be scheduled at time $t'-1$. Since u is the least urgent parent of v, $D(w) \leq D(u)-1 \leq t-1$. So w violates its deadline. Contradiction.

Case 1.2. u is not available at time $t'-1$.

Q cannot contain a parent of u, because it would violate its deadline. Because every task of G has outdegree at most one, two parents of u must be scheduled at time $t'-2$. Since S

has the least urgent parent property, the least urgent parent of u must be executed at time $t'-1$. Then Q contains a parent of u. Contradiction.

Case 2. $t \neq t'$.

For each task v in Q, at most one child of v can be scheduled at time t'. Since m tasks with deadline at most t are scheduled at time t', some tasks of H have no predecessor in Q. Let V_0 be the set containing the tasks in $S_{t'}$ that have a parent in Q. Define V_1 as the set of tasks in $S_{t'} \setminus V_0$ that are the least urgent parent of some task w that has another parent in Q. Let $V = V_0 \cup V_1$. Since every task has at most one child, $|V| \leq |Q| \leq m-1$. So $S_{t'} \setminus V$ is not empty. Let v be a task in $S_{t'} \setminus V$. From Observation 5.3.5, v is not lup-available at time $t'-1$.

Case 2.1. v is available at time $t'-1$.

Then v is the least urgent parent of a task w, such that at least two parents of w are not scheduled before time $t'-1$. Because v is scheduled at time t', another parent w' of w must be scheduled at time $t'-1$. Since v is the least urgent parent of w, $D(w') \leq D(v)-1 \leq t$. So w' is a task of Q and v must be an element of V_1. Contradiction.

Case 2.2. v is not available at time $t'-1$.

No parent of v is scheduled at time $t'-1$ and no task has more than one child, so two parents of v must be executed at time $t'-2$. Since S has the least urgent parent property, the least urgent parent of v must be scheduled at time $t'-1$. So v must be an element of V_0. Contradiction.

□

Using Lemma 5.4.2, the next theorem proves that minimum-tardiness schedules for inforests with the least urgent parent property can be constructed in polynomial time.

Theorem 5.4.3. *There is an algorithm with an $O(n \log n)$ time complexity that constructs minimum-tardiness schedules for instances (G,m,D_0), such that G is an inforest and the strongly D_0-consistent instance (G,m,D) has the least urgent parent property.*

Proof. Consider an instance (G,m,D_0), such that G is an inforest. Let (G,m,D) be the strongly D_0-consistent instance. Assume (G,m,D) has the least urgent parent property. Let S be the schedule for (G,m,D_0) constructed by Algorithm LEAST URGENT PARENT LIST SCHEDULING using lst-list L of (G,m,D). We will prove that S is a minimum-tardiness schedule for (G,m,D_0). Let ℓ^* be the tardiness of a minimum-tardiness schedule for (G,m,D_0). Define $D'_0(u) = D_0(u) + \ell^*$ for all tasks u of G. From Observation 4.1.7, there is an in-time schedule for (G,m,D'_0). Let (G,m,D') be the strongly D'_0-consistent instance. From Lemma 4.1.8, $D'(u) = D(u) + \ell^*$ for all tasks u of G. So L is an lst-list of (G,m,D') and (G,m,D') has the least urgent parent property. From Lemma 5.4.2, S is an in-time schedule for (G,m,D'_0). Hence $S(u)+1 \leq D'_0(u) = D_0(u) + \ell^*$ for all tasks u of G. So the tardiness of S as schedule for (G,m,D_0) is at most ℓ^*. Hence S is a minimum-tardiness schedule for (G,m,D_0). From Lemmas 4.1.10 and 5.3.3, S can be constructed in $O(n \log n)$ time. □

Let G be a chain-like task system. Because a chain-like task system is an outforest, every strongly D_0-consistent instance (G,m,D) has the least urgent parent property. Since every chain-like task system is an inforest, a minimum-tardiness schedule for a chain-like task system can be constructed in polynomial time.

Theorem 5.4.4. *There is an algorithm with an $O(n\log n)$ time complexity that constructs minimum-tardiness schedules for instances (G,m,D_0), such that G is a chain-like task system.*

Proof. Obvious from Theorem 5.4.3. □

5.4.2 Using the least urgent parent property for approximation

Algorithm LEAST URGENT PARENT LIST SCHEDULING can be used to schedules for all instances (G,m,D_0) if the strongly D_0-consistent instance (G,m,D) is transformed into an instance (G,m,D') with the least urgent parent property. This is the basis of the approximation algorithm for scheduling inforests presented in this section. This algorithm works as follows. First the strongly D_0-consistent instance (G,m,D) is transformed into a consistent instance (G,m,D') with the least urgent parent property. Second Algorithm LEAST URGENT PARENT LIST SCHEDULING constructs a schedule for (G,m,D').

The following lemma shows how to construct an instance with the least urgent parent property from a consistent instance (G,m,D), such that G is an inforest.

Lemma 5.4.5. *Let G be an inforest. Let (G,m,D) be a consistent instance. If $D(u) \geq 1$ for all tasks u of G, then there is a consistent instance (G,m,D') with the least urgent parent property, such that for all tasks u of G, $D(u) \leq D'(u) \leq 2D(u)$.*

Proof. Assume $D(u) \geq 1$ for all tasks u of G. Let u be a task of G that is not a source of G. Let v be a parent of u with maximum deadline among the parents of u. Let $D'(v) = 2D(v)$ and let $D'(w) = 2D(w) - 1$ for all other parents w of u. For all sources u of G, let $D'(u) = 2D(u) - 1$. Then $D(u) \leq D'(u) \leq 2D(u)$ for all tasks u of G. Let u_1 and u_2 be two tasks of G, such that u_1 is a parent of u_2. Since (G,m,D) is consistent, $D'(u_1) \leq 2D(u_1) \leq 2D(u_2) - 2 \leq D'(u_2) - 1$. Hence (G,m,D') is consistent and has the least urgent parent property. □

From the proof of Lemma 5.4.5, it is easy to see that instances with the least urgent parent property can be constructed in linear time. Moreover, the same construction can be used for precedence graphs in which every pair of tasks with a common child have the same children. However, Lemma 5.4.5 is not true for arbitrary precedence graphs.

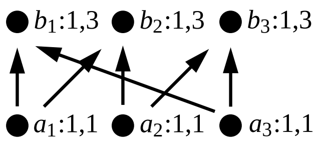

Figure 5.6. A consistent instance (G,m,D)

Example 5.4.6. Consider the consistent instance (G,m,D) shown in Figure 5.6. Let (G,m,D') be a consistent instance with the least urgent parent property, such that $D'(u) \geq D(u)$ for all tasks u of G. b_1 is a child of a_1 and a_3. Since (G,m,D') has the least urgent parent property, $D'(a_1) \neq D'(a_3)$. Similarly, $D'(a_1) \neq D'(a_2)$ and $D'(a_2) \neq D'(a_3)$. So the deadlines $D'(a_1)$, $D'(a_2)$ and $D'(a_3)$ are all different. Then for some $i \in \{1,2,3\}$, $D'(a_i) \geq 3 > 2D(a_i)$.

Example 5.4.6 shows that Lemma 5.4.5 is not true for arbitrary precedence graphs. The reason is the fact that a task can be the least urgent parent of more than one task. In fact, there are consistent instances (G,m,D) with positive deadlines, in which a deadline must be increased by at least $\frac{1}{2}n-1$ to obtain a consistent instance (G,m,D') with the least urgent parent property, such that $D'(u) \geq D(u)$ for all tasks u of G.

Lemma 5.4.5 can be used to construct schedules for all strongly D_0-consistent instances (G,m,D), such that G is an inforest. Lemma 4.1.10 shows that the strongly D_0-consistent instances for inforests can be constructed in $O(n)$ time. This allows us to prove the following result.

Theorem 5.4.7. *There is an algorithm with an $O(n \log n)$ time complexity that constructs feasible schedules for instances (G,m,D_0), such that G is an inforest, with tardiness at most $2\ell^* + \max_{v \in V(G)} D_0(v)$, where ℓ^* is the tardiness of a minimum-tardiness schedule for (G,m,D_0).*

Proof. Consider an instance (G,m,D_0), such that G is an inforest. Let (G,m,D) be the strongly D_0-consistent instance. For all tasks u of G, define

$$D'_0(u) \;=\; D_0(u) - \min_{v \in V(G)} D(v) + 1 \qquad \text{and} \qquad D'(u) \;=\; D(u) - \min_{v \in V(G)} D(v) + 1.$$

Then $D'(u) \geq 1$ for all tasks u of G and (G,m,D') is strongly D'_0-consistent. Let (G,m,D'') be a consistent instance with the least urgent parent property, such that $D'(u) \leq D''(u) \leq 2D'(u)$ for all tasks u of G. From the proof of Lemma 5.4.5, we may assume that $D''(u) = 2D'(u) - 1$ or $D''(u) = 2D'(u)$ for all tasks u of G. Let S be the schedule for (G,m,D_0) constructed by Algorithm LEAST URGENT PARENT LIST SCHEDULING using lst-list L of (G,m,D''). Let ℓ^* be the tardiness of a minimum-tardiness schedule for (G,m,D_0). We will prove that the tardiness of S is at most $2\ell^* + \max_{v \in V(G)} D_0(v)$. Define $D_1(u) = D_0(u) + \ell^*$ for all tasks u of G. From Observation 4.1.7, there is an in-time schedule for (G,m,D_1). Let (G,m,D'_1) be the strongly D_1-consistent instance. From Lemma 4.1.8, for all tasks u of G,

$$D'_1(u) \;=\; D(u) + \ell^* \;=\; D'(u) + (\ell^* + \min_{v \in V(G)} D(v) - 1).$$

From Lemma 4.1.9, there is an in-time schedule for (G,m,D'_1). Hence $D'_1(u) \geq 1$ for all tasks u of G. For all tasks u of G, define $D''_1(u)$ as follows.

$$D''_1(u) \;=\; \begin{cases} 2D'_1(u) - 1 & \text{if } D''(u) = 2D'(u) - 1 \\ 2D'_1(u) & \text{if } D''(u) = 2D'(u) \end{cases}$$

Because (G,m,D_1') is consistent, so is (G,m,D_1''). It is not difficult to see that (G,m,D_1'') has the least urgent parent property. Let u be a task of G. If $D''(u) = 2D'(u)-1$, then $D_1''(u) = 2D_1'(u) - 1 = 2D'(u) - 1 + 2(\ell^* + \min_{v\in V(G)} D(v) - 1) = D''(u) + 2(\ell^* + \min_{v\in V(G)} D(v) - 1)$. Otherwise, $D''(u) = 2D'(u)$ and $D_1''(u) = 2D_1'(u) = 2D'(u) + 2(\ell^* + \min_{v\in V(G)} D(v) - 1) = D''(u) + 2(\ell^* + \min_{v\in V(G)} D(v) - 1)$. Hence $D_1''(u) = D''(u) + 2(\ell^* + \min_{v\in V(G)} D(v) - 1)$ for all tasks u of G. So L is an lst-list of (G,m,D_1''). From Lemma 5.4.2, S is an in-time schedule for (G,m,D_1''). Hence for all tasks u of G,

$$\begin{aligned}
S(u)+1 &\leq D_1''(u) \\
&= D''(u) + 2(\ell^* + \min_{v\in V(G)} D(v) - 1) \\
&\leq 2D'(u) + 2(\ell^* + \min_{v\in V(G)} D(v) - 1) \\
&= 2(D(u) - \min_{v\in V(G)} D(v) + 1) + 2(\ell^* + \min_{v\in V(G)} D(v) - 1) \\
&\leq 2D_0(u) + 2\ell^*.
\end{aligned}$$

So the tardiness of S as schedule for (G,μ,m,D_0) is at most $2\ell^* + \max_{v\in V(G)} D_0(v)$. From Lemmas 4.1.10, 5.4.5 and 5.3.3, S can be constructed in $O(n\log n)$ time. □

Consequently, there is a polynomial-time 2-approximation algorithm for inforests with non-positive deadlines.

Corollary 5.4.8. *There is an algorithm with an $O(n\log n)$ time complexity that constructs feasible schedules for instances (G,m,D_0) with non-positive deadlines, such that G is an inforest, with tardiness at most $2\ell^*$, where ℓ^* is the tardiness of a minimum-tardiness schedule for (G,m,D_0).*

Proof. Obvious from Theorem 5.4.7. □

5.5 Concluding remarks

In this chapter, it was shown that the least urgent property allows the construction of minimum-tardiness schedules for a larger class of precedence graphs. Because constructing minimum-length schedules for arbitrary precedence graphs on an unrestricted number of processors is NP-hard [47, 77, 80] as well as for inforests on m processors [61], we have identified two special cases of NP-hard optimisation problems that are solvable in polynomial time.

Like for the problems presented in Chapter 4, some generalisations are possible. Introducing release dates makes that the existence of in-time schedules with the least urgent parent property for inforests with the least urgent parent property is not guaranteed. Hence this approach cannot be generalised to scheduling with release dates and deadlines.

With $\{0,1\}$-communication delays, the definition of the least urgent parent property needs to be changed. With the altered least urgent parent property, minimum-tardiness schedules for arbitrary precedence graphs on an unrestricted number of processors and for inforests on m processors can also be constructed in polynomial time.

6 Pairwise deadlines

In Chapter 4, an algorithm was presented for scheduling precedence-constrained tasks with the objective of minimising the maximum tardiness. This algorithm constructs minimum-tardiness schedules for a small class of precedence graphs. This is due to the fact that Algorithm DEADLINE MODIFICATION does not use the knowledge that a task cannot be scheduled immediately after two of its parents. In Chapter 5, the least urgent parent property was introduced. For each task, this property allows the choice of a parent that has to finish after the other parents. Using the least urgent parent property, minimum-tardiness schedules can be constructed for a larger class of precedence graphs.

In this chapter, we will use the knowledge that a task cannot be scheduled after two of its parents in a different way. Like Bartusch et al. [8] for scheduling without communication delays, we will compute deadlines for sets of tasks: a deadline will be computed for every pair of tasks instead of for individual tasks. In order to meet the deadline $D(u_1,u_2)$ of a pair (u_1,u_2), u_1 or u_2 has to be completed at or before time $D(u_1,u_2)$. Like the individual deadlines, the deadline of a pair of tasks (u_1,u_2) depends on the successors of u_1 and u_2: if u_1 and u_2 have sufficiently many common successors that have to be scheduled before time d, then the deadline of (u_1,u_2) is decreased. Using these pairwise deadlines, minimum-tardiness schedules can be constructed for interval orders on m processors and for precedence graphs of width two on two processors.

6.1 Pairwise consistent deadlines

In this section, we will define pairwise deadlines that are met in all in-time schedules. To define these pairwise consistent deadlines, we need to look at the structure of in-time schedules. Let S be an in-time schedule for (G,m,D). Let u be a task of G. Assume u has $k \geq 1$ successors $v_1,\ldots,v_k$ with deadlines at most d. u starts at time $S(u)$ and finishes at time $S(u)+1$. Because of communication delays, at most one task v_i can be scheduled at time $S(u)+1$. Hence the last of the $k-1$ remaining successors of u cannot be completed before time $S(u)+2+\lceil\frac{k-1}{m}\rceil$. Since the successors of u are completed at or before time d, u must be completed at or before time $d-1-\lceil\frac{k-1}{m}\rceil$. This observation led to the notion of consistent deadlines in Chapter 4.

Let u_1 and u_2 be two tasks of G that have $k \geq 1$ common successors with deadline at most d. Because the successors of u_1 and u_2 meet their deadlines, the first must be scheduled at or before time $d-\lceil\frac{k}{m}\rceil$. Because of the communication delays, u_1 and u_2 cannot both be executed immediately before a common successor of u_1 and u_2. So u_1 or u_2 must be completed at or before time $d-1-\lceil\frac{k}{m}\rceil$. Using this observation, we might be able to determine upper bounds on the completion time of common predecessors v of u_1 and u_2 in each in-time schedule that are smaller than the consistent deadline of v as defined in Chapter 4.

To use this knowledge, we will introduce pairwise deadlines. A pair of (not necessarily different) tasks (u_1,u_2) will be assigned a deadline $D(u_1,u_2)$. We will consider instances (G,m,D), such that $D: V(G)\times V(G)\to\mathbb{Z}$ is a function that assigns a deadline to every pair of tasks of G. We will assume that $D(u_1,u_2)=D(u_2,u_1)$ for all pairs of tasks (u_1,u_2) of G. In addition, we will use $D(u)$ instead of $D(u,u)$ for all tasks u of G.

Let S be a feasible schedule for an instance (G,m,D) with pairwise deadlines. The pair (u_1,u_2) *meets its deadline* if the completion time of u_1 or u_2 is at most $D(u_1,u_2)$. If no deadline $D(u_1,u_2)$ is violated, S will be called an *in-time schedule* for (G,m,D).

Now we will define pairwise consistent deadlines that are met in all in-time schedules for an instance (G,m,D_0). To define such deadlines, we need the following definitions. Let (u_1,u_2) be a pair of tasks of G and let d be an integer. $N_D(u_1,u_2,d)$ equals the number of common successors of u_1 and u_2 with individual deadline at most d. $P_D(u_1,u_2,d)$ equals $\max\{|U|-1,0\}$, where U is a maximum-size subset of the common successors of u_1 and u_2 with individual deadline at least $d+1$ and pairwise deadline at most d. More precisely, for all pairs of tasks (u_1,u_2) of G and all integers d,

$$N_D(u_1,u_2,d) \quad = \quad |\{v \in Succ_G(u_1) \cap Succ_G(u_2) \mid D(v) \leq d\}|$$

and

$$\begin{aligned} P_D(u_1,u_2,d) \quad = \quad \max\{0, \max\{|U|-1 \mid\ & U \subseteq Succ_G(u_1) \cap Succ_G(u_2) \wedge \\ & D(v) \geq d+1 \text{ for all tasks } v \text{ in } U \wedge \\ & D(v_1,v_2) \leq d \text{ for all tasks } v_1 \neq v_2 \text{ in } U\}\}. \end{aligned}$$

$T_D(u_1,u_2,d)$ denotes the total number of common successors of u_1 and u_2 that must be completed at or before time d in an in-time schedule for (G,m,D). For all pairs of tasks (u_1,u_2) of G and all integers d, define

$$T_D(u_1,u_2,d) \quad = \quad N_D(u_1,u_2,d) + P_D(u_1,u_2,d).$$

In addition, for all tasks u of G, define $T_D(u,d) = N_D(u,d) + P_D(u,d)$, where $N_D(u,d) = N_D(u,u,d)$ and $P_D(u,d) = P_D(u,u,d)$. Hence $T_D(u,d) = T_D(u,u,d)$ for all tasks u of G.

Note that for all pairs of tasks (u_1,u_2) of G and all integers d, $N_D(u_1,u_2,d) = N_D(u_2,u_1,d)$, $P_D(u_1,u_2,d) = P_D(u_2,u_1,d)$, $N_D(u_1,u_2,d) \leq N_D(u_1,d)$ and $P_D(u_1,u_2,d) \leq P_D(u_1,d)$.

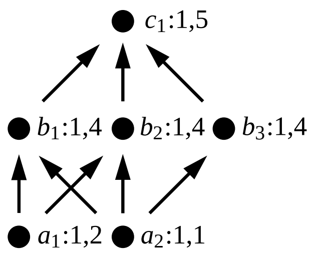

Figure 6.1. An instance $(G,2,D)$ with pairwise deadlines

Example 6.1.1. Consider the instance $(G,2,D)$ shown in Figure 6.1. Assume $D(b_1,b_2) = D(b_1,b_3) = D(b_2,b_3) = 3$ and $D(u_1,u_2) = \min\{D(u_1),D(u_2)\}$ for all other pairs of tasks (u_1,u_2)

of G. Since c_1 has no successors, $T_D(c_1,d) = 0$ for all d. Tasks b_1, b_2 and b_3 have one successor with deadline 5 and no other successors, so $T_D(b_i,5) = N_D(b_i,5) = 1$ and $T_D(b_i,b_j,5) = N_D(b_i,b_j,5) = 1$. a_1 has two successors with individual deadline 4 and pairwise deadline 3. So $T_D(a_1,4) = N_D(a_1,4) = 2$ and $T_D(a_1,3) = P_D(a_1,3) = 1$. Moreover, $T_D(a_1,5) = 3$. Similarly, $T_D(a_2,3) = 2$, $T_D(a_2,4) = 3$ and $T_D(a_2,5) = 4$.

To define pairwise consistent deadlines, we need to look at the structure of in-time schedules. Consider an instance (G,m,D) with pairwise deadlines. Let u_1 and u_2 be two tasks of G. Let U be a non-empty subset of $Succ_G(u_1) \cap Succ_G(u_2)$, such that every task in U has a deadline at least $d+1$ and every pair of different tasks in U has a deadline at most d. Then in every in-time schedule for (G,m,D), at most one task in U can be scheduled at time d or later. Obviously, every common successor of u_1 and u_2 with deadline at most d must be scheduled before time d. Consequently, in each in-time schedule for (G,m,D), at least $T_D(u_1,u_2,d) = N_D(u_1,u_2,d) + P_D(u_1,u_2,d)$ common successors of u_1 and u_2 are completed at or before time d.

Let (G,m,D) be an instance with pairwise deadlines. Let u be a task of G, such that $T_D(u,d) \geq 1$. In an in-time schedule for (G,m,D), $T_D(u,d)$ successors of u are completed at or before time d. Because at most one successor of u can be executed immediately after u, u must be completed at or before time $d-1-\left\lceil \frac{1}{m}(T_D(u,d)-1) \right\rceil$.

Observation 6.1.2. *Let (G,m,D) be an instance with pairwise deadlines. Let S be an in-time schedule for (G,m,D). Let u be a task of G. If $T_D(u,d) \geq 1$, then $S(u)+1 \leq d-1-\left\lceil \frac{1}{m}(T_D(u,d)-1) \right\rceil$.*

Consider an instance (G,m,D) with pairwise deadlines. Let u_1 and u_2 be two tasks of G, such that $T_D(u_1,u_2,d) \geq 1$. In an in-time schedule for (G,m,D), $T_D(u_1,u_2,d)$ common successors of u_1 and u_2 are completed at or before time d. The first of these starts at or before time $d - \left\lceil \frac{1}{m} T_D(u_1,u_2,d) \right\rceil$. Because u_1 and u_2 cannot both be executed immediately before a common successor, u_1 or u_2 is completed at or before time $d-1-\left\lceil \frac{1}{m} T_D(u_1,u_2,d) \right\rceil$.

Observation 6.1.3. *Let (G,m,D) be an instance with pairwise deadlines. Let S be an in-time schedule for (G,m,D). Let $u_1 \neq u_2$ be two tasks of G. If $T_D(u_1,u_2,d) \geq 1$, then* $\min\{S(u_1)+1, S(u_2)+1\} \leq d-1-\left\lceil \frac{1}{m} T_D(u_1,u_2,d) \right\rceil$.

Observations 6.1.2 and 6.1.3 are used to define pairwise consistent instances.

Definition 6.1.4. Let (G,m,D) be an instance with pairwise deadlines. (G,m,D) is called *pairwise consistent* if for all tasks $u_1 \neq u_2$ of G and all integers d,

1. $D(u_1,u_2) \leq \min\{D(u_1), D(u_2)\}$;
2. if $T_D(u_1,d) \geq 1$, then $D(u_1) \leq d-1-\left\lceil \frac{1}{m}(T_D(u_1,d)-1) \right\rceil$; and
3. if $T_D(u_1,u_2,d) \geq 1$, then $D(u_1,u_2) \leq d-1-\left\lceil \frac{1}{m} T_D(u_1,u_2,d) \right\rceil$.

(G,m,D) is called *pairwise D_0-consistent* if it is pairwise consistent and $D(u) \leq D_0(u)$ for all tasks u of G. It is called *pairwise strongly D_0-consistent* if it is pairwise D_0-consistent and for all tasks $u_1 \neq u_2$ of G,

1. $D(u_1) = D_0(u_1)$, or there is an integer d, such that $T_D(u_1,d) \geq 1$ and $D(u_1) = d - 1 - \lceil \frac{1}{m}(T_D(u_1,d) - 1) \rceil$; and
2. $D(u_1,u_2) = \min\{D(u_1), D(u_2)\}$, or there is an integer d, such that $T_D(u_1,u_2,d) \geq 1$ and $D(u_1,u_2) = d - 1 - \lceil \frac{1}{m} T_D(u_1,u_2,d) \rceil$.

Example 6.1.5. Consider the instance $(G,2,D)$ shown in Figure 6.1. Assume $D(b_1,b_2) = D(b_1,b_3) = D(b_2,b_3) = 3$ and $D(u_1,u_2) = \min\{D(u_1),D(u_2)\}$ for all other pairs of tasks (u_1,u_2) of G. Assume $D_0(u) = 5$ for all tasks u of G. It is not difficult to see that $(G,2,D)$ is pairwise D_0-consistent. $(G,2,D)$ is also pairwise strongly D_0-consistent, because $D(c) = 5 = D_0(c)$, $D(b_i) = 4 = 5 - 1 - \lceil \frac{1}{2}(T_D(b_i,5) - 1) \rceil$, $D(b_i,b_j) = 3 = 5 - 1 - \lceil \frac{1}{2} T_D(b_i,b_j,5) \rceil$, $D(a_1) = 2 = 3 - 1 - \lceil \frac{1}{2}(T_D(a_1,3) - 1) \rceil$ and $D(a_2) = 1 = 3 - 1 - \lceil \frac{1}{2}(T_D(a_2,3) - 1) \rceil$. The pairwise strongly D_0-consistent deadlines are smaller than the strongly D_0-consistent deadlines: if $(G,2,D')$ is strongly D_0-consistent, then $D'(a_2) = 2$, whereas $D(a_2) = 1$.

Example 6.1.5 shows that pairwise consistent deadlines can be smaller than the consistent deadlines, that were defined in Chapter 4. The following lemma shows that the pairwise consistent deadlines cannot be larger.

Lemma 6.1.6. *Let (G,m,D_1) be the strongly D_0-consistent instance and (G,m,D_2) the pairwise strongly D_0-consistent instance. Then $D_2(u) \leq D_1(u)$ for all tasks u of G.*

Proof. It will be proved by induction that $D_2(u) \leq D_1(u)$ for all tasks u of G. Let u be a task of G. Assume by induction that $D_2(v) \leq D_1(v)$ for all successors v of u. It is proved by contradiction that $D_2(u) \leq D_1(u)$. Suppose $D_1(u) < D_2(u)$. Then $D_1(u) \neq D_0(u)$. Hence there is an integer d, such that $N_{D_1}(u,d) \geq 1$ and $D_1(u) = d - 1 - \lceil \frac{1}{m}(N_{D_1}(u,d) - 1) \rceil$. Since $D_2(v) \leq D_1(v)$ for all successors v of u, $T_{D_2}(u,d) \geq N_{D_2}(u,d) \geq N_{D_1}(u,d)$. Because (G,m,D_2) is pairwise consistent, $D_2(u) \leq d - 1 - \lceil \frac{1}{m}(T_{D_2}(u,d) - 1) \rceil \leq d - 1 - \lceil \frac{1}{m}(N_{D_1}(u,d) - 1) \rceil = D_1(u)$. Contradiction. By induction, $D_2(u) \leq D_1(u)$ for all tasks u of G. □

It is not difficult to see that the deadlines of a pairwise D_0-consistent instance do not exceed those of a pairwise strongly D_0-consistent instance.

Observation 6.1.7. *Let (G,m,D_1) and (G,m,D_2) be two pairwise D_0-consistent instances. If (G,m,D_1) is pairwise strongly D_0-consistent, then $D_1(u_1,u_2) \geq D_2(u_1,u_2)$ for all pairs of tasks (u_1,u_2) of G.*

This shows that for each instance (G,m,D_0), there is exactly one pairwise strongly D_0-consistent instance (G,m,D).

Like for strongly D_0-consistent instances, if all original deadlines are increased by the same amount, then the strongly pairwise D_0-consistent deadlines are increased by the same amount.

Lemma 6.1.8. *Let (G,m,D) be the pairwise strongly D_0-consistent instance and (G,m,D') the pairwise strongly D'_0-consistent instance. If there is an integer c, such that $D'_0(u) = D_0(u) + c$ for all tasks u of G, then $D'(u_1,u_2) = D(u_1,u_2) + c$ for all pairs of tasks (u_1,u_2) of G.*

Proof. Assume there is an integer c, such that $D'_0(u) = D_0(u) + c$ for all tasks u of G. It is proved by induction that $D'(u_1, u_2) = D(u_1, u_2) + c$ for all pairs of tasks (u_1, u_2) of G. Let u be a task of G. Assume by induction that $D'(v_1, v_2) = D(v_1, v_2) + c$ for all successors v_1 and v_2 of u. It will be proved by contradiction that $D'(u) = D(u) + c$. Suppose $D'(u) \neq D(u) + c$.

Case 1. $D(u) = D_0(u)$.
Then $D'(u) \neq D'_0(u)$. Because (G, m, D') is pairwise strongly D'_0-consistent, there is an integer d, such that $T_{D'}(u, d) \geq 1$ and $D'(u) = d - 1 - \left\lceil \frac{1}{m}(T_{D'}(u, d) - 1) \right\rceil$. Because $T_D(u, d - c) = T_{D'}(u, d) \geq 1$ and (G, m, D) is pairwise consistent, $D(u) \leq d - c - 1 - \left\lceil \frac{1}{m}(T_{D'}(u, d) - 1) \right\rceil = D'(u) - c < D_0(u)$. Contradiction. So $D'(u) = D(u) + c$.

Case 2. $D(u) \neq D_0(u)$.
Because (G, m, D) is pairwise strongly D_0-consistent, there is an integer d, such that $T_D(u, d) \geq 1$ and $D(u) = d - 1 - \left\lceil \frac{1}{m}(T_D(u, d) - 1) \right\rceil$. Since $T_{D'}(u, d + c) = T_D(u, d) \geq 1$ and (G, m, D') is pairwise consistent, $D'(u) \leq d + c - 1 - \left\lceil \frac{1}{m}(T_D(u, d) - 1) \right\rceil = D(u) + c$. Because $D'(u) \neq D(u) + c$, we obtain $D'(u) < D(u) + c \neq D_0(u) + c = D'_0(u)$. Since (G, m, D') is pairwise strongly D'_0-consistent, there is an integer d', such that $T_{D'}(u, d') \geq 1$ and $D'(u) = d' - 1 - \left\lceil \frac{1}{m}(T_{D'}(u, d') - 1) \right\rceil$. Since $T_D(u, d' - c) = T_{D'}(u, d') \geq 1$ and (G, m, D) is pairwise consistent, $D(u) \leq d' - c - 1 - \left\lceil \frac{1}{m}(T_{D'}(u, d') - 1) \right\rceil = D'(u) - c < D(u)$. Contradiction. So $D'(u) = D(u) + c$.

In either case, $D'(u) = D(u) + c$. Let $u_1 \neq u_2$ be two tasks of G. Assume by induction that $D'(u_1) = D(u_1) + c$, $D'(u_2) = D(u_2) + c$ and $D'(v_1, v_2) = D(v_1, v_2) + c$ for all successors v_1 and v_2 of u_1 and u_2. It will be proved by contradiction that $D'(u_1, u_2) = D(u_1, u_2) + c$. Suppose $D'(u_1, u_2) \neq D(u_1, u_2) + c$.

Case 1. $D(u_1, u_2) = \min\{D(u_1), D(u_2)\}$.
Then $D'(u_1, u_2) \neq \min\{D'(u_1), D'(u_2)\}$. Since (G, m, D') is pairwise strongly D'_0-consistent, there is an integer d, such that $T_{D'}(u_1, u_2, d) \geq 1$ and $D'(u) = d - 1 - \left\lceil \frac{1}{m} T_{D'}(u_1, u_2, d) \right\rceil$. Because $T_D(u_1, u_2, d - c) = T_{D'}(u_1, u_2, d) \geq 1$ and (G, m, D) is pairwise consistent, $D(u_1, u_2) \leq d - c - 1 - \left\lceil \frac{1}{m} T_{D'}(u_1, u_2, d) \right\rceil = D'(u_1, u_2) - c < \min\{D(u_1), D(u_2)\}$. Contradiction. So $D'(u_1, u_2) = D(u_1, u_2) + c$.

Case 2. $D(u_1, u_2) \neq \min\{D(u_1), D(u_2)\}$.
Because (G, m, D) is pairwise strongly D_0-consistent, there is an integer d, such that $T_D(u_1, u_2, d) \geq 1$ and $D(u_1, u_2) = d - 1 - \left\lceil \frac{1}{m} T_D(u_1, u_2, d) \right\rceil$. Since $T_{D'}(u_1, u_2, d + c) = T_D(u_1, u_2, d) \geq 1$ and (G, m, D') is pairwise consistent, $D'(u_1, u_2) \leq d + c - 1 - \left\lceil \frac{1}{m} T_D(u_1, u_2, d) \right\rceil = D(u_1, u_2) + c$. Since $D'(u_1, u_2) \neq D(u_1, u_2) + c$, we obtain $D'(u_1, u_2) < D(u_1, u_2) + c \neq \min\{D'(u_1), D'(u_2)\}$. Because (G, m, D') is pairwise strongly D'_0-consistent, there is an integer d', such that $T_{D'}(u_1, u_2, d') \geq 1$ and $D'(u) = d' - 1 - \left\lceil \frac{1}{m} T_{D'}(u_1, u_2, d') \right\rceil$. Since $T_D(u_1, u_2, d' - c) = T_{D'}(u_1, u_2, d') \geq 1$ and (G, m, D) is pairwise consistent, $D(u_1, u_2) \leq d' - c - 1 - \left\lceil \frac{1}{m} T_{D'}(u_1, u_2, d') \right\rceil = D'(u_1, u_2) - c < D(u_1, u_2)$. Contradiction. So $D'(u_1, u_2) = D(u_1, u_2) + c$.

In either case, $D'(u_1, u_2) = D(u_1, u_2) + c$. By induction, $D'(u_1, u_2) = D(u_1, u_2) + c$ for all pairs of tasks (u_1, u_2) of G. □

Like for strongly D_0-consistent instances, an in-time schedule for (G,m,D_0) is also an in-time schedule for the pairwise strongly D_0-consistent instance (G,m,D).

Lemma 6.1.9. *Let (G,m,D) be the pairwise strongly D_0-consistent instance. Let S be a feasible schedule for (G,m,D_0). Then S is an in-time schedule for (G,m,D_0) if and only if S is an in-time schedule for (G,m,D).*

Proof. Because $D(u) \leq D_0(u)$ for all tasks u of G, every in-time schedule for (G,m,D) is an in-time schedule for (G,m,D_0). Assume S is an in-time schedule for (G,m,D_0). Define $D_S(u_1,u_2) = \min\{S(u_1)+1, S(u_2)+1\}$ for all tasks u_1 and u_2 of G. We will prove by contradiction that (G,m,D_S) is pairwise consistent. Suppose (G,m,D_S) is not pairwise consistent.

Case 1. $T_{D_S}(u,d) \geq 1$ and $D_S(u) > d-1-\left\lceil \frac{1}{m}(T_{D_S}(u,d)-1)\right\rceil$ for some u and d.
Every pair of successors of u meets its deadline. So $T_{D_S}(u,d)$ successors of u finish at or before time d. Hence u must be completed at or before time $d-1-\left\lceil \frac{1}{m}(T_{D_S}(u,d)-1)\right\rceil$. So $D_S(u) \leq d-1-\left\lceil \frac{1}{m}(T_{D_S}(u,d)-1)\right\rceil$. Contradiction.

Case 2. $T_{D_S}(u_1,u_2,d) \geq 1$ and $D_S(u_1,u_2) > d-1-\left\lceil \frac{1}{m}T_{D_S}(u_1,u_2,d)\right\rceil$ for some $u_1 \neq u_2$ and d.
Since every pair of successors of u_1 and u_2 meets its deadline, $T_{D_S}(u_1,u_2,d)$ common successors of u_1 and u_2 finish at or before time d. Then u_1 or u_2 must be completed at or before time $d-1-\left\lceil \frac{1}{m}T_{D_S}(u_1,u_2,d)\right\rceil$. So $D_S(u_1,u_2) \leq d-1-\left\lceil \frac{1}{m}T_{D_S}(u_1,u_2,d)\right\rceil$. Contradiction.

So (G,m,D_S) is pairwise consistent. Since S is an in-time schedule for (G,m,D_0), $D_S(u) \leq D_0(u)$ for all tasks u of G. Hence (G,m,D_S) is pairwise D_0-consistent. From Observation 6.1.7, $D(u_1,u_2) \geq D_S(u_1,u_2)$ for all pairs of tasks (u_1,u_2) of G. Since every deadline $D_S(u_1,u_2)$ is met, S is an in-time schedule for (G,m,D). □

In the remainder of this section, we prove some properties of pairwise strongly D_0-consistent instances. These will be used to compute such instances.

Lemma 6.1.10. *Let (G,m,D) be the pairwise strongly D_0-consistent instance. Let u_1 and u_2 be two tasks of G. If $D(u_1,u_2) < \min\{D(u_1),D(u_2)\}$, then there are integers d and k, such that $T_D(u_1,u_2,d) = km+1$ and $D(u_1,u_2) = d-2-k$.*

Proof. Assume $D(u_1,u_2) < \min\{D(u_1),D(u_2)\}$. Because (G,m,D) is pairwise strongly D_0-consistent, there is an integer d, such that $T_D(u_1,u_2,d) \geq 1$ and $D(u_1,u_2) = d-1-\left\lceil \frac{1}{m}T_D(u_1,u_2,d)\right\rceil$. There is an integer $k \geq 0$, such that $(k+1)m \geq T_D(u_1,u_2,d) \geq km+1$. Then $D(u_1,u_2) = d-2-k$. Suppose $T_D(u_1,u_2,d) \geq km+2$. Then $T_D(u_1,d) \geq T_D(u_1,u_2,d) \geq km+2$ and $D(u_1) \leq d-2-k = D(u_1,u_2)$. Contradiction. Hence $T_D(u_1,u_2,d) = km+1$. □

The next lemma shows that the deadline of a pair of tasks differs at most one from the minimum of the individual deadlines. This will allow us to redefine $P_D(u_1,u_2,d)$.

Lemma 6.1.11. *Let (G,m,D) be the pairwise strongly D_0-consistent instance. Let u_1 and u_2 be two tasks of G. If $D(u_1,u_2) < \min\{D(u_1),D(u_2)\}$, then $D(u_1) = D(u_2) = D(u_1,u_2)+1$ and there is an integer d, such that $T_D(u_1,d) = T_D(u_2,d) = T_D(u_1,u_2,d) = (d-D(u_1)-1)m+1$.*

Proof. Assume $D(u_1,u_2) < \min\{D(u_1),D(u_2)\}$. From Lemma 6.1.10, there are integers d and k, such that $T_D(u_1,u_2,d) = km+1$ and $D(u_1,u_2) = d-2-k$. Suppose $T_D(u_i,d) \neq T_D(u_1,u_2,d)$ for some $i \in \{1,2\}$. Then $T_D(u_i,d) \geq T_D(u_1,u_2,d)+1 \geq km+2$. Since (G,m,D) is pairwise consistent, $D(u_i) \leq d-2-k = D(u_1,u_2)$. Contradiction. So $T_D(u_1,d) = T_D(u_2,d) = T_D(u_1,u_2,d) = km+1$. Because (G,m,D) is pairwise consistent, $D(u_i) \leq d-1-k = D(u_1,u_2)+1$. Since $D(u_1,u_2) < D(u_i)$, $D(u_1) = D(u_2) = D(u_1,u_2)+1$. So $D(u_1) = d-1-k$ and $k = d-D(u_1)-1$. As a result, $T_D(u_1,d) = T_D(u_2,d) = T_D(u_1,u_2,d) = (d-D(u_1)-1)m+1$. □

Lemma 6.1.11 shows that for the computation of the pairwise strongly D_0-consistent instance (G,m,D), we only need to consider pairs of tasks (u_1,u_2) of G, such that $D(u_1) = D(u_2)$ and $T_D(u_1,d) = T_D(u_2,d) = T_D(u_1,u_2,d) = (d-D(u_1)-1)m+1$ for some integer d. The deadlines of the other pairs can be set to the minimum of the individual deadlines. Moreover, it shows that $P_D(u_1,u_2,d)$ can be redefined. For all pairs of tasks (u_1,u_2) of G and all integers d,

$$\begin{aligned} P_D(u_1,u_2,d) \quad = \quad \max\{0,\max\{|U|-1 \mid\ & U \subseteq Succ_G(u_1) \cap Succ_G(u_2) \wedge \\ & D(v) = d+1 \text{ for all tasks } v \text{ in } U \wedge \\ & D(v_1,v_2) = d \text{ for all tasks } v_1 \neq v_2 \text{ in } U\}\}. \end{aligned}$$

The result proved in the following lemma will be used for the computation of pairwise strongly D_0-consistent instances for interval-ordered tasks.

Lemma 6.1.12. *Let (G,m,D) be the pairwise strongly D_0-consistent instance. Let u_1 and u_2 be two tasks of G, such that $D(u_1,u_2) < \min\{D(u_1),D(u_2)\}$. If there is a task $v \neq u_1,u_2$ of G, such that $Succ_G(u_1) \cap Succ_G(u_2) \subseteq Succ_G(v)$ and $D(v) = D(u_1)$, then $D(u_1,v) = D(u_2,v) = D(u_1,u_2)$.*

Proof. Assume there is a task $v \neq u_1,u_2$ of G, such that $Succ_G(u_1) \cap Succ_G(u_2) \subseteq Succ_G(v)$ and $D(v) = D(u_1)$. From Lemma 6.1.11, $D(u_1) = D(u_2) = D(u_1,u_2)+1$ and $T_D(u_1,u_2,d) = (d-D(u_1)-1)m+1$ for some integer d. Let $i \in \{1,2\}$. Since $Succ_G(u_1) \cap Succ_G(u_2) \subseteq Succ_G(v)$, $T_D(u_i,v,d) \geq T_D(u_1,u_2,d)$. So $T_D(u_i,v,d) \geq (d-D(u_1)-1)m+1$. Because (G,m,D) is pairwise consistent, $D(u_i,v) \leq d-1-(d-D(u_1)-1+1) = D(u_1)-1 = D(v)-1$. From Lemma 6.1.11, $D(u_1,v) = D(u_2,v) = D(v)-1 = D(u_1,u_2)$. □

6.2 Computing pairwise consistent deadlines

In this section, two algorithms are presented that compute pairwise strongly D_0-consistent instances. The first is presented in Section 6.2.1. The time complexity of this algorithm is exponential in the width of the precedence graphs; it constructs pairwise strongly D_0-consistent instances for precedence graphs of bounded width in polynomial time. The second algorithm is presented in Section 6.2.2. It constructs pairwise strongly D_0-consistent instances for interval orders in polynomial time.

6.2.1 Arbitrary precedence graphs

Algorithm PAIRWISE DEADLINE MODIFICATION shown in Figure 6.2 is used to construct pairwise strongly D_0-consistent instances (G,m,D) for instances (G,m,D_0). Its structure is similar to that of Algorithm DEADLINE MODIFICATION. In each step, it computes the pairwise strongly D_0-consistent deadline of a task u of G, such that the pairwise strongly consistent deadlines of all successors and all pairs of successors of u have been computed before, and for all pairs of tasks (u,v), such that the pairwise strongly consistent deadline of v has been computed in an earlier step.

The following notation is used. L_d denotes the set of tasks of G with pairwise strongly D_0-consistent deadline d. Since a pairwise strongly D_0-consistent deadline of a task can be smaller than its original deadline, we have to consider sets L_d, such that d is smaller than the smallest original deadline. Since a pairwise strongly D_0-consistent deadline differs at most $n-1$ from the corresponding original deadline, we need sets L_d, such that $\min_{u \in V(G)} D_0(u) - n + 1 \leq d \leq \max_{u \in V(G)} D_0(u)$. The sets L_d are used to compute the pairwise strongly D_0-consistent deadlines of pairs of tasks: from Lemma 6.1.11, we only need to compute pairwise deadlines for pairs of tasks with equal pairwise strongly D_0-consistent deadlines, the other pairwise deadlines can be set to the minimum of the individual deadlines.

```
Algorithm PAIRWISE DEADLINE MODIFICATION
Input. An instance $(G,m,D_0)$ with individual deadlines.
Output. The pairwise strongly $D_0$-consistent instance $(G,m,D)$.
1.  $D_{\max} := \max_{u \in V(G)} D_0(u)$
2.  $D_{\min} := \min_{u \in V(G)} D_0(u)$
3.  for $d := D_{\min} - n + 1$ to $D_{\max}$
4.      do $L_d := \varnothing$
5.  for all tasks $u$ of $G$
6.      do $D(u) := D_0(u)$
7.  $U := V(G)$
8.  while $U \neq \varnothing$
9.      do let $u$ be a sink of $G[U]$
10.         for $d := D_{\min}$ to $D_{\max}$
11.             do if $T_D(u,d) \geq 1$
12.                 then $D(u) := \min\left\{D(u), d - 1 - \left\lceil \frac{1}{m}(T_D(u,d) - 1) \right\rceil\right\}$
13.         $L_{D(u)} := L_{D(u)} \cup \{u\}$
14.         for $v \in V(G) \setminus U$
15.             do $D(u,v) := \min\{D(u), D(v)\}$
16.                $D(v,u) := \min\{D(u), D(v)\}$
17.         for $v \in L_{D(u)} \setminus \{u\}$
18.             do for $d := D_{\min}$ to $D_{\max}$
19.                 do if $T_D(u,v,d) \geq 1$
20.                     then $D(u,v) := \min\{D(u,v), d - 1 - \left\lceil \frac{1}{m} T_D(u,v,d) \right\rceil\}$
21.                $D(v,u) := D(u,v)$
22.         $D_{\min} := \min\{D_{\min}, \min_{v \in V(G) \setminus U} D(u,v)\}$
23.         $U := U \setminus \{u\}$
```

Figure 6.2. Algorithm PAIRWISE DEADLINE MODIFICATION

Example 6.2.1. Let G be the precedence graph shown in Figure 6.1. Assume $D_0(u) = 5$ for all tasks u of G. In the beginning, all deadlines are set to 5. Algorithm PAIRWISE DEADLINE MODIFICATION computes deadlines $D(u_1,u_2)$ as follows. First c_1 is considered. Since c_1 has no successors, $D(c_1) = D_0(c_1) = 5$. Next b_1, b_2 and b_3 are considered. These have one successor with deadline 5 and no pairs of successors with deadline 5. So $D(b_i)$ is set to $5-1-\lceil\frac{0}{2}\rceil = 4$. Moreover, b_i and b_j have a common successor with deadline 5. So $D(b_i,b_j)$ is set to $5-1-\lceil\frac{1}{2}\rceil = 3$. a_1 has two successors with deadline 4. These successors have pairwise deadline 3. Moreover, a_1 has three successors with deadline at most 5. So $D(a_1) = \min\{5-1-\lceil\frac{2}{2}\rceil, 4-1-\lceil\frac{1}{2}\rceil, 3-1-\lceil\frac{0}{2}\rceil\} = 2$. Similarly, $T_D(a_2,3) = 2$, $T_D(a_2,4) = 3$ and $T_D(a_2,5) = 4$. Consequently, $D(a_2) = \min\{5-1-\lceil\frac{3}{2}\rceil, 4-1-\lceil\frac{2}{2}\rceil, 3-1-\lceil\frac{1}{2}\rceil\} = 1$. The resulting instance $(G,2,D)$ is pairwise strongly D_0-consistent.

Now we will prove that Algorithm PAIRWISE DEADLINE MODIFICATION correctly constructs pairwise strongly D_0-consistent instances.

Lemma 6.2.2. *Let (G,m,D_0) be an instance with individual deadlines. Let (G,m,D) be the instance constructed by Algorithm* PAIRWISE DEADLINE MODIFICATION *for instance (G,m,D_0). Then (G,m,D) is pairwise strongly D_0-consistent.*

Proof. Algorithm PAIRWISE DEADLINE MODIFICATION executes n steps. In each step, it computes a deadline for a task of G and for pairs containing this task. Assume the tasks are chosen in the order $u_1,\ldots,u_n$. For all $i \le n$ and all pairs of tasks (v_1,v_2) of G, let $D_i(v_1,v_2)$ be the deadline of (v_1,v_2) after step i and let G_i the subgraph of G induced by $\{u_1,\ldots,u_i\}$. For all $i \le n$, the sets $L_{d,i}$ coincide with the sets L_d after step i and $D_{\min,i}$ and $D_{\max,i}$ with the values of $D_{\min}$ and $D_{\max}$ after step i.

We will prove by induction that all instances (G_i,m,D_i) are pairwise strongly D_0-consistent. It is easy to see that (G_1,m,D_1) is pairwise strongly D_0-consistent. Assume by induction that (G_i,m,D_i) is pairwise strongly D_0-consistent. For all $j_1, j_2 \le i$, $D_{i+1}(u_{j_1},u_{j_2}) = D_i(u_{j_1},u_{j_2})$. So (G_i,m,D_{i+1}) is pairwise strongly D_0-consistent. Now consider u_{i+1}. Clearly, $D_{i+1}(u_{i+1}) \le D_0(u_{i+1})$. It is not difficult to see that if $T_{D_{i+1}}(u_{i+1},d) \ge 1$, then $D_{\min,i} \le d \le D_{\max,i}$. Then $D_{i+1}(u_{i+1}) \le d-1-\lceil\frac{1}{m}(T_{D_{i+1}}(u_{i+1},d)-1)\rceil$. Moreover, if $D_{i+1}(u_{i+1}) \ne D_0(u_{i+1})$, then there is an integer d, such that $D_{\min,i} \le d \le D_{\max,i}$, $T_{D_{i+1}}(u_{i+1},d) \ge 1$ and $D_{i+1}(u_{i+1}) = d-1-\lceil\frac{1}{m}(T_{D_{i+1}}(u_{i+1},d)-1)\rceil$.

Consider a pair (u_{i+1},u_j), such that $j \le i$. It is not difficult to see that $D_{i+1}(u_{i+1},u_j) \le \min\{D_{i+1}(u_{i+1}),D_{i+1}(u_j)\}$. Assume $D_{i+1}(u_{i+1}) = D_{i+1}(u_j)$ and $T_{D_{i+1}}(u_{i+1},u_j,d) \ge 1$. Then $D_{\min,i} \le d \le D_{\max,i}$. So $D_{i+1}(u_{i+1},u_j) \le d-1-\lceil\frac{1}{m}T_{D_{i+1}}(u_{i+1},u_j,d)\rceil$. If $D_{i+1}(u_{i+1},u_j) \ne \min\{D_{i+1}(u_{i+1}),D_{i+1}(u_j)\}$, then there must be an integer d, such that $D_{\min,i} \le d \le D_{\max,i}$, $T_{D_{i+1}}(u_{i+1},u_j) \ge 1$ and $D_{i+1}(u_{i+1},u_j) = d-1-\lceil\frac{1}{m}T_{D_{i+1}}(u_{i+1},u_j,d)\rceil$. Hence (G_{i+1},m,D_{i+1}) is pairwise strongly D_0-consistent. By induction, (G_n,m,D_n) is pairwise strongly D_0-consistent. Because $G_n = G$ and $D_n(u_1,u_2) = D(u_1,u_2)$ for all pairs of tasks (u_1,u_2) of G, (G,m,D) is pairwise strongly D_0-consistent. □

The following results will be used to determine the time complexity of Algorithm PAIRWISE DEADLINE MODIFICATION.

Lemma 6.2.3. *Let G be a precedence graph of width w. Let (G,m,D) be a pairwise consistent instance. Then G contains at most w tasks u, such that $D(u) = d$ for all integers d.*

Proof. It is proved by contradiction that G contains at most w tasks with deadline d. Suppose G contains at least $w+1$ tasks with deadline d. Let $u_1,\dots,u_{w+1}$ be $w+1$ tasks of G with deadline d. Since G has width w, we may assume that $u_1 \prec_G u_2$. Then $N_D(u_1,D(u_2)) \geq 1$. Because (G,m,D) is pairwise consistent, $D(u_1) \leq D(u_2) - 1 = d - 1$. Contradiction. So G contains at most w tasks with deadline d. □

Corollary 6.2.4. *Let G be a precedence graph of width w. Let (G,m,D) be a pairwise consistent instance. Then for all tasks u_1 and u_2 of G and all integers d, $P_D(u_1,u_2,d) \leq w-1$.*

Proof. Let u_1 and u_2 be two tasks of G. Let U be a maximum-size subset U' of $Succ_G(u_1) \cap Succ_G(u_2)$, such that $D(v) \geq d+1$ for all tasks v in U' and $D(v_1,v_2) \leq d$ for all tasks $v_1 \neq v_2$ in U'. Then $P_D(u_1,u_2,d) = \max\{0,|U|-1\}$. From Lemma 6.1.11, $D(v) = d+1$ for all tasks v in U. Lemma 6.2.3 shows that G contains at most w tasks with deadline $d+1$. Hence $|U| \leq w$. Consequently, $P_D(u_1,u_2,d) \leq w-1$. □

The time complexity of Algorithm PAIRWISE DEADLINE MODIFICATION can be determined as follows. Consider an instance (G,m,D_0), such that G is a precedence graph of width w. Because there is a minimum-tardiness schedule for (G,m,D_0) of length at most n, we may assume that the smallest and largest deadline differ at most n. Moreover, no deadline is decreased by more than n. Hence the initialisation part of Algorithm PAIRWISE DEADLINE MODIFICATION takes $O(n^2)$ time.

To obtain a better time complexity, we will consider two cases depending on whether G is known to be a transitive closure or not. If it is unknown whether G is a transitive closure, then Algorithm PAIRWISE DEADLINE MODIFICATION should first compute the transitive closure G^+ of G. This takes $O(n+e+ne^-)$ time [37]. In the transitive reduction of G, every task has at most w children. Hence $e^- \leq wn$. So G^+ can be computed in $O(wn^2)$ time. In the remainder of the analysis of the time complexity of G, we will assume that G is a transitive closure.

For each pair of tasks (u_1,u_2) of G, Algorithm PAIRWISE DEADLINE MODIFICATION has to compute $T_D(u_1,u_2,d)$ for all integers d, such that $D_{\min} \leq d \leq D_{\max}$. Since there are schedules for (G,m,D) of length at most n, we may assume that $D_{\max} - D_{\min} \leq n$. $N_D(u_1,u_2,d)$ can be computed by determining the number of common successors of u_1 and u_2 with deadline d and storing these numbers in an array. By applying a prefix sum operation on this array, we obtain the values $N_D(u_1,u_2,d)$ for all d in $O(n)$ time.

Computing $P_D(u_1,u_2,d)$ is more complicated. In order to compute $P_D(u_1,u_2,d)$, we need to consider every subset of $L_{d+1} \cap Succ_G(u_1) \cap Succ_G(u_2)$. Lemma 6.2.3 shows that L_{d+1} contains at most w tasks. So at most 2^w subsets V of $L_{d+1} \cap Succ_G(u_1) \cap Succ_G(u_2)$ have to be taken into account. For each subset V, $O(|V|^2)$ time is used to check if all pairs of different tasks of V have deadline d. So the values $P_D(u_1,u_2,d)$ can be computed in a total of $O(w^2 2^w n)$ time.

$T_D(u_1,d)$ must be computed for every task u_1 and every d. For each task u_1, $T_D(u_1,u_2,d)$ needs to be computed for at most $w-1$ pairs (u_1,u_2) and all integers d. So the computation of $T_D(u_1,u_2,d)$ takes $O(w^3 2^w n^2)$ time in total.

Assigning a deadline $D(u_1,u_2)$ to a pair of tasks (u_1,u_2) of G takes constant time for each pair (u_1,u_2). Hence this takes $O(n^2)$ time in total. The other operations take linear time. Consequently, the pairwise strongly D_0-consistent instance is constructed in $O(w^3 2^w n^2)$ time.

Lemma 6.2.5. *For all instances (G,m,D_0), such that G is a precedence graph of width w, Algorithm* PAIRWISE DEADLINE MODIFICATION *constructs the pairwise strongly D_0-consistent instance (G,m,D) in $O(w^3 2^w n^2)$ time.*

Lemma 6.2.5 shows that if G is a precedence graph of bounded width, then the pairwise D_0-consistent instance (G,m,D) can be constructed in polynomial time.

Lemma 6.2.6. *For all instances (G,m,D_0), such that G is a precedence graph of constant width w, Algorithm* PAIRWISE DEADLINE MODIFICATION *constructs the pairwise strongly D_0-consistent instance (G,m,D) in $O(n^2)$ time.*

6.2.2 Interval-ordered tasks

Lemma 6.2.6 shows that for precedence graphs of constant width w, the pairwise strongly D_0-consistent deadlines can be computed in polynomial time. Interval orders can have an arbitrarily large width, so Algorithm PAIRWISE DEADLINE MODIFICATION cannot be used to compute pairwise consistent deadlines for interval orders in polynomial time. However, using the properties of interval orders presented in Section 2.5.2, it is possible to construct the pairwise strongly D_0-consistent deadlines in polynomial time. The algorithm computing such deadlines is presented in this section.

Consider an instance (G,m,D_0) with individual deadlines. The main difficulty in the computation of pairwise strongly D_0-consistent deadlines is the computation of $P_D(u_1,u_2,d)$. For arbitrary instances (G,m,D), computing $P_D(u_1,u_2,d)$ corresponds to finding a maximum-size clique in an undirected graph containing the common successors v of u_1 and u_2 with deadline $d+1$ and edges between the common successors v_1 and v_2 with pairwise deadline d. Since finding a maximum-size clique in an arbitrary undirected graph is a strongly NP-hard optimisation problem [33], this definition does not give an efficient way of computing $P_D(u_1,u_2,d)$. For interval orders, an alternative definition of $P_D(u_1,u_2,d)$ can be derived. This definition will allow us to compute $P_D(u_1,u_2,d)$ in linear time.

Let (G,m,D) be an instance with pairwise deadlines. For all tasks u_1 of G, define

$$d_{\min}(u_1) \;=\; \min\{D(u_1,u_2) \mid u_2 \in V(G) \wedge D(u_2) = D(u_1)\}.$$

From Lemma 6.1.11, if (G,m,D) is pairwise strongly D_0-consistent, then $D(u_1)-1 \leq d_{\min}(u_1) \leq D(u_1)$ for all tasks u_1 of G. Moreover, $d_{\min}(u_1) = D(u_1)-1$ if and only if there is a task u_2 of G, such that $D(u_2) = D(u_1)$ and $D(u_1,u_2) = D(u_1)-1$.

Lemma 6.2.7. *Let G be an interval order. Let (G,m,D) be the pairwise strongly D_0-consistent instance. Let u_1 and u_2 be two tasks of G. Then for all integers d,*

$$P_D(u_1,u_2,d) \;=\; \max\{0, |\{v \in Succ_G(u_1) \cap Succ_G(u_2) \mid D(v) = d+1 \wedge d_{\min}(v) = d\}| - 1\}.$$

Proof. Define $U = \{v \in Succ_G(u_1) \cap Succ_G(u_2) \mid D(v) = d+1 \wedge d_{\min}(v) = d\}$. Let U_P be a maximum-size subset U' of $Succ_G(u_1) \cap Succ_G(u_2)$, such that each task in U' has deadline $d+1$ and each pair of different tasks in U' has deadline d. From Lemma 6.1.11, $P_D(u_1,u_2,d) = \max\{0, |U_P|-1\}$.

Case 1. $P_D(u_1,u_2,d) = 0$.
Then for every pair of common successors (v_1,v_2) of u_1 and u_2, if $D(v_1) = D(v_2) = d+1$, then $D(v_1,v_2) = d+1$. So $d_{\min}(v) = d+1$ for all common successors v of u_1 and u_2, such that $D(v) = d+1$. Hence $U = \varnothing$ and $P_D(u_1,u_2,d) = \max\{0,|U|-1\}$.

Case 2. $P_D(u_1,u_2,d) \geq 1$.
Then U_P contains at least two tasks. So for every task v_1 in U_P, there is a task v_2, such that $D(v_1,v_2) = d$ and $D(v_2) = d+1$. So U_P is a subset of U. Since G is an interval order, we may assume that $U_P = \{v_1,\ldots,v_k\}$, such that $Succ_G(v_1) \subseteq \cdots \subseteq Succ_G(v_k)$. We will prove by contradiction that $U = U_P$. Suppose U is not a subset of U_P. Let v be a task in $U \setminus U_P$.

Case 2.1. $Succ_G(v_1) \subseteq Succ_G(v)$.
Then $Succ_G(v_1) \cap Succ_G(v_i) \subseteq Succ_G(v)$ for all $i \in \{2,\ldots,k\}$. From Lemma 6.1.12, $D(v_i,v) = d$ for every $i \in \{1,\ldots,k\}$. Since U_P is of maximum size, v must be an element of U_P. Contradiction.

Case 2.2. $Succ_G(v) \subseteq Succ_G(v_1)$.
v is a task in U, so there is a task w, such that $D(w) = d+1$ and $D(v,w) = d$. If $w = v_1$, then $D(v_1,v) = d$. Otherwise, $Succ_G(v) \cap Succ_G(w) \subseteq Succ_G(v_1)$ and from Lemma 6.1.12, $D(v_1,v) = d$. In either case, $D(v_1,v) = d$. Hence $Succ_G(v_1) \cap Succ_G(v) \subseteq Succ_G(v_i)$ for all $i \in \{2,\ldots,k\}$. From Lemma 6.1.12, $D(v_i,v) = d$ for all $i \leq k$. Because U_P is of maximum size, v must be a task in U_P. Contradiction.

So $U = U_P$ and $P_D(u_1,u_2,d) = \max\{0,|U|-1\}$.

□

This result allows the computation of pairwise strongly D_0-consistent instances without actually computing a deadline for each pair of tasks. The following lemma shows how the pairwise deadlines can be computed from the individual deadlines.

Lemma 6.2.8. *Let G be an interval order. Let (G,m,D) be the pairwise strongly D_0-consistent instance. Let u_1 and u_2 be two different tasks of G. If $D(u_1) = D(u_2)$ and for some integer d, $T_D(u_1,d) = T_D(u_2,d) = (d - D(u_1) - 1)m + 1$, then $D(u_1,u_2) = D(u_1) - 1$.*

Proof. Assume $D(u_1) = D(u_2)$ and $T_D(u_1,d) = T_D(u_2,d) = (d-D(u_1)-1)m+1$ for some integer d. Since G is an interval order, $Succ_G(u_1) \subseteq Succ_G(u_2)$ or $Succ_G(u_2) \subseteq Succ_G(u_1)$. In either case, $T_D(u_1,u_2,d) = (d-D(u_1)-1)m+1$. Because (G,m,D) is pairwise consistent, $D(u_1,u_2) \leq d-1-(d-D(u_1)-1+1) = D(u_1)-1$. From Lemma 6.1.11, $D(u_1,u_2) = D(u_1)-1$. □

These results will be used in the algorithm that computes pairwise strongly D_0-consistent instances (G,m,D), such that G is an interval order. The algorithm starts by setting $D(u)=D_0(u)$ for all tasks u of G. Next it executes n steps. In each step of the algorithm, the pairwise strongly D_0-consistent consistent deadline of a task of G is computed. Algorithm INTERVAL ORDER DEADLINE MODIFICATION is shown in Figure 6.3.

The following notation is used. L_d denotes the set of tasks u of G with pairwise strongly D_0-consistent deadline d. $L_{d,d'}$ is the subset of L_d containing the tasks u, such that $T_D(u,d')=(d'-d-1)m+1$. Like for Algorithm PAIRWISE DEADLINE MODIFICATION, we need to consider sets L_d and $L_{d,d'}$, such that $\min_{u\in V(G)} D_0(u)-n+1 \leq d,d' \leq \max_{u\in V(G)} D_0(u)$. U denotes the set of tasks that have not been considered. $d_{\max}$ denotes the maximum d, such that $d_{\min}(u)$ has not been computed for the tasks u of G with pairwise strongly D_0-consistent deadline d.

Algorithm INTERVAL ORDER DEADLINE MODIFICATION does not compute deadlines for the pairs of tasks of G. These can be computed using the sets $L_{d,d'}$. Using Lemma 6.2.8, every pair of different tasks of a set $L_{d,d'}$ gets deadline $d-1$. The deadlines of the remaining pairs equal the minimum of the individual deadlines.

Example 6.2.9. Let G be the precedence graph shown in Figure 6.1. Note that G is an interval order. Assume $D_0(u)=5$ for all tasks u of G. Algorithm INTERVAL ORDER DEADLINE MODIFICATION computes the pairwise strongly D_0-consistent instance as follows. First, a deadline is computed for c_1. Since c_1 has no successors, its deadline is not decreased. c_1 is added to L_5 and the deadlines $D(b_i)$ are set to 4. When b_1, b_2 and b_3 are considered, their deadlines are not decreased, because c_1 is their only successor. These tasks are added to $L_{4,5}$, since $T_D(b_i,5)=1$. The deadlines of a_1 and a_2 are set to 3. In the next step, a_1 is considered. First $d_{\min}(b_i)$ is set to 3, because $L_{4,5}$ contains b_1, b_2 and b_3. Since $T_D(a_1,4)=2$, the pairwise strongly D_0-consistent deadline of a_1 equals 2. Finally, Algorithm INTERVAL ORDER DEADLINE MODIFICATION considers a_2. $T_D(a_2,3)=2$, so $D(a_2)$ is set to 1. The resulting instance is pairwise strongly D_0-consistent.

Now we will prove that Algorithm INTERVAL ORDER DEADLINE MODIFICATION correctly computes pairwise strongly D_0-consistent instances for interval orders.

Lemma 6.2.10. *Let G be an interval order. Let (G,m,D_0) be an instance with individual deadlines. Let (G,m,D) be the instance constructed by Algorithm* INTERVAL ORDER DEADLINE MODIFICATION *for instance (G,m,D_0). Then (G,m,D) is pairwise strongly D_0-consistent.*

Proof. Algorithm INTERVAL ORDER DEADLINE MODIFICATION executes n steps. In each step, it computes a deadline of a task of G. Assume the tasks are chosen in the order $u_1,\ldots,u_n$. For all $i\leq n$ and all tasks u of G, let $D_i(u)$ be the deadline of u after step i. The sets L^i_d and $L^i_{d,d'}$ coincide with the sets L_d and $L_{d,d'}$ after step i for all $i\leq n$. For all $i\leq n$, let G_i be the subgraph of G induced by $\{u_1,\ldots,u_i\}$. Then all subgraphs G_i are interval orders. We will consider instances (G_i,m,D_i), where $D_i(v_1,v_2)$ is defined as follows. If v_1 and v_2 are two different elements of $L^i_{d,d'}$ for some integers d and d', then $D_i(v_1,v_2)=d-1$. Otherwise, $D_i(v_1,v_2)=\min\{D_i(v_1),D_i(v_2)\}$.

We will prove by induction that the instances (G_i,m,D_i) are pairwise strongly D_0-consistent. It is not difficult to see that (G_1,m,D_1) is pairwise strongly D_0-consistent. Assume by induction

```
Algorithm INTERVAL ORDER DEADLINE MODIFICATION
Input. An instance $(G,m,D_0)$ with individual deadlines, such that $G$ is an interval order.
Output. The pairwise strongly $D_0$-consistent instance $(G,m,D)$.
1.  $D_{\min} := \min_{u \in V(G)} D_0(u)$
2.  $D_{\max} := \max_{u \in V(G)} D_0(u)$
3.  for $d := D_{\min} - n + 1$ to $D_{\max}$
4.      do $L_d := \varnothing$
5.         for $d' := d+1$ to $D_{\max}$
6.             do $L_{d,d'} := \varnothing$
7.  $d_{\max} := D_{\max}$
8.  for all tasks $u$ of $G$
9.      do $D(u) := D_0(u)$
10. $U := V(G)$
11. while $U \neq \varnothing$
12.     do let $u$ be a sink of $G[U]$ with maximum $D(u)$
13.        for $d := d_{\max}$ downto $D(u)+1$
14.            do for $v \in L_d$
15.                   do $d_{\min}(v) := d$
16.               for $d' := d+1$ to $D_{\max}$
17.                   do if $|L_{d,d'}| \geq 2$
18.                         then for $v \in L_{d,d'}$
19.                                  do $d_{\min}(v) := d-1$
20.        $d_{\max} := D(u)$
21.        for $d := D(u)$ to $D_{\max}$
22.            do if $T_D(u,d) \geq 1$
23.                  then $D(u) := \min\left\{D(u), d-1-\left\lceil \frac{1}{m}(T_D(u,d)-1)\right\rceil\right\}$
24.        $L_{D(u)} := L_{D(u)} \cup \{u\}$
25.        for $d := D(u)+1$ to $D_{\max}$
26.            do if $T_D(u,d) = (d-D(u)-1)m+1$
27.                  then $L_{D(u),d} := L_{D(u),d} \cup \{u\}$
28.        for all parents $v$ of $u$
29.            do $D(v) := \min\{D(v), D(u)-1\}$
30.        $U := U \setminus \{u\}$
```

Figure 6.3. Algorithm INTERVAL ORDER DEADLINE MODIFICATION

(G_i,m,D_i) is pairwise strongly D_0-consistent. Now consider (G_{i+1},m,D_{i+1}). It is easy to see that $D_{i+1}(u_{j_1},u_{j_2}) = D_i(u_{j_1},u_{j_2})$ and $T_{D_{i+1}}(u_{j_1},u_{j_2},d) = T_{D_i}(u_{j_1},u_{j_2},d)$ for all $j_1, j_2 \leq i$ and all integers d. So (G_i,m,D_{i+1}) is pairwise strongly D_0-consistent.

Consider u_{i+1}. Clearly, $D_i(u_{i+1}) = D_0(u_{i+1})$ or $D_i(u_{i+1}) = D_i(v)-1$ for some child v of u_{i+1}. From Lemma 6.2.8, $d_{\min}(v)$ is computed correctly for all successors v of u_{i+1}. These values are used to compute $D_{i+1}(u_{i+1})$. Suppose $T_{D_{i+1}}(u_{i+1},d) \geq 1$ for some integer d. Then $d \geq D_i(u_{i+1})$, because $D_i(v) > D_i(u_{i+1})$ for all successors v of u_{i+1}. Hence $D_{i+1}(u_{i+1}) \leq d -$

$1 - \lceil \frac{1}{m}(T_{D_{i+1}}(u_{i+1},d)-1)\rceil$. It is not difficult to verify that $D_{i+1}(u_{i+1}) = D_0(u_{i+1})$, or there is an integer d, such that $T_{D_{i+1}}(u_{i+1},d) \geq 1$ and $D_{i+1}(u_{i+1}) = d - 1 - \lceil \frac{1}{m}(T_{D_{i+1}}(u_{i+1},d)-1)\rceil$.

Let (v_1,v_2) be a pair of tasks of G_{i+1}. Assume $D_{i+1}(v_1,v_2) \neq \min\{D_{i+1}(v_1), D_{i+1}(v_2)\}$. Then $D_{i+1}(v_1,v_2) = D_{i+1}(v_1) - 1 = D_{i+1}(v_2) - 1$ and for some integer d, $T_{D_{i+1}}(v_1,d) = T_{D_{i+1}}(v_2,d) = (d - D_{i+1}(v_1) - 1)m + 1$. Then $T_{D_{i+1}}(v_1,v_2,d) = (d - D_{i+1}(v_1) - 1)m + 1$ and $D_{i+1}(v_1,v_2) = D_{i+1}(v_1) - 1 = d - 1 - \lceil \frac{1}{m} T_{D_{i+1}}(v_1,v_2,d)\rceil$. So (G_{i+1},m,D_{i+1}) is pairwise strongly D_0-consistent. By induction, (G_n,m,D_n) is pairwise strongly D_0-consistent. Since $G_n = G$ and $D_n(u_1,u_2) = D(u_1,u_2)$ for all pairs of tasks (u_1,u_2) of G, (G,m,D) is pairwise strongly D_0-consistent. □

Now we will determine the time complexity of Algorithm INTERVAL ORDER DEADLINE MODIFICATION. Let G be an interval order. Consider an instance (G,m,D_0) with individual deadlines. Like in the analysis of the time complexity of Algorithm PAIRWISE DEADLINE MODIFICATION, we start by computing the transitive closure of G if it is unknown whether G is a transitive closure. From Lemma 2.5.6, G^+ can be constructed in $O(n+e^+)$ time. In the remainder of the analysis of the time complexity of Algorithm INTERVAL ORDER DEADLINE MODIFICATION, we will assume that G is a transitive closure.

The fact that G is a transitive closure allows us to compute $N_D(u,d)$ in an efficient way. For each integer d, determine the number of successors v of u, such that $D(v) = d$. By applying a prefix sum operation on these numbers, we find $N_D(u,d)$ for all integers d. Since we may assume that the largest deadline differs at most n from the smallest deadline, the traversal of the successors of u and the prefix sum operation both take $O(n)$ time. $P_D(u,d)$ can also be computed using a traversal of the successors of u. From Lemma 6.2.7, $P_D(u,d)$ equals the number of successors v of u, such that $D(v) = d+1$ and $d_{\min}(v) = d$. Hence $T_D(u,d)$ can be computed in $O(n)$ time for all integers d simultaneously.

Because we may assume that the smallest and largest deadlines differ at most n, the initialisation part of Algorithm INTERVAL ORDER DEADLINE MODIFICATION requires $O(n^2)$ time.

The first for-loop (Lines 13–19) of Algorithm INTERVAL ORDER DEADLINE MODIFICATION is executed for every d in $D_{\min} - n + 1, \ldots, D_{\max}$. For every task v in $L_{d,d'}$, $d_{\min}(v)$ is determined. This takes $O(|L_d|)$ time for each d'. So $O(|L_d|n)$ time is used to compute $d_{\min}(v)$ for every task v in L_d. Since every task is added to exactly one set L_d, Algorithm INTERVAL ORDER DEADLINE MODIFICATION uses $O(\sum_{d=D_{\min}-n+1}^{D_{\max}} |L_d|n) = O(n^2)$ time for executing its first for-loop.

The main loop (Lines 11–30) is executed for each task u of G. In every iteration, the values $T_D(u,d)$ are computed in linear time. Hence the pairwise strongly D_0-consistent deadline of u is computed in $O(n)$ time. Adding u to a set L_d takes constant time and adding u to sets $L_{d,d'}$ takes $O(n)$ time. The deadline of a parent of u is decreased if it is not smaller than the deadline of u. This requires constant time for every parent of u, so $O(|Pred_{G,0}(u)|)$ time in total. Consequently, $O(n^2)$ time is used in the main loop.

Hence we have proved the following result.

Lemma 6.2.11. *For all instances (G,m,D_0), such that G is an interval order, Algorithm* INTERVAL ORDER DEADLINE MODIFICATION *constructs the pairwise strongly D_0-consistent instance (G,m,D) in $O(n^2)$ time.*

6.3 Constructing minimum-tardiness schedules

For pairwise strongly D_0-consistent instances (G,m,D), Algorithm LIST SCHEDULING is used to construct schedules for instances (G,m,D_0). It will be proved that these schedules are minimum-tardiness schedules if G is a precedence graph of width two or an interval order. The pairwise deadlines are not used by Algorithm LIST SCHEDULING; these deadlines were only used to construct a better priority list than the lst-lists based on the strongly D_0-consistent deadlines.

6.3.1 Precedence graphs of width two

In this section, it is proved that minimum-tardiness schedules for instances $(G,2,D_0)$, such that G is a precedence graph of width two, can be constructed in polynomial time. Such schedules are constructed by Algorithm LIST SCHEDULING using an lst-list of the pairwise strongly D_0-consistent instance $(G,2,D)$.

Lemma 6.3.1. *Let G be a precedence graph of width two. Let $(G,2,D)$ be the pairwise strongly D_0-consistent instance. Let S be a schedule for $(G,2,D_0)$ constructed by Algorithm* LIST SCHEDULING *using an lst-list of $(G,2,D)$. If there is an in-time schedule for $(G,2,D_0)$, then S is an in-time schedule for $(G,2,D_0)$.*

Proof. Assume there is an in-time schedule for $(G,2,D_0)$. From Lemma 6.1.9, there is an in-time schedule for $(G,2,D)$. It will be proved by contradiction that S is an in-time schedule for $(G,2,D)$. Suppose S is not an in-time schedule for $(G,2,D)$. Assume S_t is the first time slot that contains a task u_1 of G in a pair of tasks (u_1,u_2) whose deadline $D(u_1,u_2)$ is violated. Then both u_1 and u_2 finish after time $D(u_1,u_2)$. Hence $D(u_1,u_2) \leq t$. From Lemma 6.1.11, there are two possibilities: $\min\{D(u_1),D(u_2)\} \leq t$, or $D(u_1,u_2) = t$ and $D(u_1) = D(u_2) = t+1$.

Case 1. $\min\{D(u_1),D(u_2)\} \leq t$.

Let u be one of the tasks u_1 and u_2, such that $D(u) \leq t$. Because there is an in-time schedule for $(G,2,D)$, there are at most $2t$ tasks with deadline at most t. Hence there is a time slot before S_t that contains at most one task with deadline at most t. Let $t'-1$ be the latest time before time t at which at most one task with deadline at most t is scheduled. Let H_1 be the subgraph of G induced by $\bigcup_{i=t'}^{t-1} S_i \cup \{v \in \bigcup_{i\geq t} S_i \mid v \prec_G u\} \cup \{u\}$. Then H_1 contains at least $2(t-t')+1$ tasks with deadline at most t. From Observation 4.3.6, no task of H_1 is available at time $t'-1$. Hence every task of H_1 has a predecessor that is scheduled at time $t'-2$ or $t'-1$.

Case 1.1. Every task of H_1 has a predecessor in $S_{t'-1}$.

Define $Q = \{v \in S_{t'-1} \mid D(v) \leq t\}$. Then Q contains exactly one task w. Because of communication delays, at most one successor of w is scheduled at time t'. Hence $t = t'$. As a result, w is a predecessor of u. So $T_D(w,t) \geq 1$. Since $(G,2,D)$ is pairwise consistent, $D(w) \leq t-1 = t'-1$. Hence w is not completed at or before time $D(w)$. Contradiction.

Case 1.2. Not every task of H_1 has a predecessor in $S_{t'-1}$.

Let v be a source of H_1 without a predecessor in $S_{t'-1}$. Then a predecessor w_1 of v starts at time $t'-2$.

Case 1.2.1. $S_{t'-2}$ contains exactly one task with a successor in H_1.

v is not available at time $t'-1$. Because at most one predecessor of v is scheduled at time $t'-2$, a child $x \neq v$ of w_1 starts at time $t'-1$. Since Algorithm LIST SCHEDULING scheduled x instead of v, $D(x) \leq D(v)$. Because every task of H_1 has a predecessor that is scheduled at time $t'-2$ or $t'-1$ and x is a child of w_1, all tasks of H_1 are successors of w_1. Hence $T_D(w_1,t) \geq 2(t-t')+2$. Because $(G,2,D)$ is pairwise consistent, $D(w_1) \leq t'-2$. So w_1 is not completed at or before time $D(w_1)$. Contradiction.

Case 1.2.2. $S_{t'-2}$ contains two tasks with a successor in H_1.

Let w_2 be the other task executed at time $t'-2$. Then w_2 is a predecessor of a task of H_1. Because G is a precedence graph of width two and w_1 and w_2 are incomparable tasks, every task of H_1 is a successor of w_1 or w_2.

Case 1.2.2.1. Every task of H_1 is a successor of w_1 and w_2.

Then w_1 and w_2 have at least $2(t-t')+1$ common successors with deadline at most t. Hence $N_D(w_1,w_2,t) \geq 2(t-t')+1$. Since $(G,2,D)$ is pairwise consistent, $D(w_1,w_2) \leq t'-2$. So (w_1,w_2) violates its deadline $D(w_1,w_2)$. Contradiction.

Case 1.2.2.2. H_1 contains a task of $Succ_G(w_1) \setminus Succ_G(w_2)$.

Let x_1 be such a task. Assume x_1 is a source of H_1. x_1 is not available at time $t'-1$. Because w_2 is not a parent of x, a child y_1 of w_1 must be executed at time $t'-1$. Since y_1 is scheduled by Algorithm LIST SCHEDULING instead of x_1, $D(y_1) \leq D(x_1) \leq t$. We will prove by contradiction that all successors of w_2 in H_1 are successors of w_1. Suppose H_1 contains a task x_2 that is a successor of w_2, but not a successor of w_1. Then $S_{t'-1}$ contains a child y_2 of w_2, such that $D(y_2) \leq D(x_2) \leq t$. At time $t'-1$, at most one task with deadline at most t is executed. So $y_1 = y_2$ and $w_1 = w_2$. Contradiction. So every task of H_1 is a successor of w_1. Hence w_1 has at least $2(t-t')+2$ successors with deadline at most t. Therefore $T_D(w_1,t) \geq 2(t-t')+2$. Because $(G,2,D)$ is pairwise consistent, $D(w_1) \leq t'-2$. So w_1 does not finish at or before time $D(w_1)$. Contradiction.

Case 1.2.2.3. H_1 contains a task of $Succ_G(w_2) \setminus Succ_G(w_1)$.

Similar to Case 1.2.2.2.

Case 2. $D(u_1) = D(u_2) = t+1$ and $D(u_1,u_2) = t$.

In any in-time schedule for $(G,2,D)$, u_1 or u_2 is completed at or before time t. Since there is an in-time schedule for $(G,2,D)$, there are at most $2t-1$ tasks with deadline at most t. Let $S_{t'-1}$ be the last time slot before time slot S_t that contains at most one task with deadline at most t. Let H_2 be the subgraph of G induced by $\bigcup_{i=t'}^{t-1} S_i \cup \{u_1,u_2\} \cup \{v \in \bigcup_{i \geq t} S_i \mid v \prec_G u_2\}$. Then H_2 contains at least $2(t-t')+2$ tasks. From Observation 4.3.6, no task of H_2 is available at time $t'-1$. Hence every task of H_2 has a predecessor that starts at time $t'-2$ or $t'-1$.

Case 2.1. Every task of H_2 has a predecessor in $S_{t'-1}$.

Define $Q = \{v \in S_{t'-1} \mid D(v) \leq t\}$. Clearly, Q contains exactly one task. Let w be this task. Since H_2 contains at least $2(t-t')$ tasks with deadline at most t, $N_D(w,t) \geq 2(t-t')$. Furthermore, u_1 and u_2 are successors of w. Hence $P_D(w,t) = 1$. Consequently, $T_D(w,t) \geq$

$2(t-t')+1$. Since $(G,2,D)$ is pairwise consistent, $D(w) \leq t'-1$. So w does not finish at or before time $D(w)$. Contradiction.

Case 2.2. Not every task of H_2 has a predecessor in $S_{t'-1}$.
Let v be a task of H_2 that has no predecessor in $S_{t'-1}$. Assume v is a source of H_2. Then a parent w_1 of v is executed at time $t'-2$.

Case 2.2.1. $S_{t'-2}$ contains exactly one task with a successor in H_2.
v is not available at time $t'-1$. Since only one parent of x is scheduled at time $t'-2$, a child $x \neq v$ of w_1 is executed at time $t'-1$. Because all tasks of H_2 have a predecessor scheduled at time $t'-2$ or $t'-1$ and x is a parent of w_1, w_1 is a predecessor of all tasks of H_2. Because x is scheduled by Algorithm LIST SCHEDULING instead of v, $D(x) \leq D(v)$. So w_1 has at least $2(t-t')+1$ successors with deadline at most t. Since u_1 and u_2 are successors of w_1, $P_D(w_1,t)=1$. Hence $T_D(w_1,t) \geq 2(t-t')+2$. Because $(G,2,D)$ is pairwise consistent, $D(w_1) \leq t'-2$. So w_1 is not completed at or before time $D(w_1)$. Contradiction.

Case 2.2.2. $S_{t'-2}$ contains two tasks with a successor in H_2.
Let w_2 be the other task scheduled at time $t'-2$. Because G is a precedence graph of width two and w_1 and w_2 are incomparable tasks, every task of H_2 is a successor of w_1 or w_2.

Case 2.2.2.1. Every task of H_2 is a successor of w_1 and w_2.
Clearly, $N_D(w_1,w_2,t) \geq 2(t-t')$ and $P_D(w_1,w_2,t) \geq 1$. Because $(G,2,D)$ is pairwise consistent, $D(w_1,w_2) \leq t'-2$. So (w_1,w_2) violates its deadline $D(w_1,w_2)$. Contradiction.

Case 2.2.2.2. H_2 contains a task of $Succ_G(w_1) \setminus Succ_G(w_2)$.
Let x_1 be such a task. We may assume that x_1 is a source of H_2. x_1 is not available at time $t'-1$. Because only one parent of x_1 is scheduled at time $t'-2$, a child y_1 of w_1 is executed at time $t'-1$. y_1 is scheduled instead of x_1, so $D(y_1) \leq D(x_1) \leq t$. Since y_1 is executed at time $t'-1$, y_1 is not a child of w_2. We will prove by contradiction that all successors of w_2 in H_2 are successors of w_1. Suppose H_2 contains a task x_2 that is a successor of w_2, but not a successor of w_1. In that case, $S_{t'-1}$ contains a child y_2 of w_2, such that $D(y_2) \leq D(x_2) \leq t$. y_2 is not a successor of w_1, so $y_1 \neq y_2$. Consequently, two tasks with deadline at most t are executed at time $t'-1$. Contradiction. Therefore every task of H_2 is a successor of w_1. Hence $T_D(w_1,t) = N_D(w_1,t)+P_D(w_1,t) \geq 2(t-t')+2$. Because $(G,2,D)$ is pairwise consistent $D(w_1) \leq t'-2$. So w_1 does not finish at or before time $D(w_1)$. Contradiction.

Case 2.2.2.3. H_2 contains a task of $Succ_G(w_2) \setminus Succ_G(w_1)$.
Similar to Case 2.2.2.2.

□

This allows us to prove that minimum-tardiness schedules for precedence graphs of width two on two processors can be constructed in polynomial time.

Theorem 6.3.2. *There is an algorithm with an $O(n^2)$ time complexity that constructs minimum-tardiness schedules for instances $(G,2,D_0)$, such that G is a precedence graph of width two.*

Proof. Consider an instance $(G,2,D_0)$, such that G is a precedence graph of width two. Let $(G,2,D)$ be the pairwise strongly D_0-consistent instance. Let S be the schedule for $(G,2,D_0)$ constructed by Algorithm LIST SCHEDULING using lst-list L of $(G,2,D)$. We will prove that S is a minimum-tardiness schedule for $(G,2,D_0)$. Let ℓ^* be the tardiness of a minimum-tardiness schedule for $(G,2,D_0)$. Define $D'_0(u) = D_0(u) + \ell^*$ for all tasks u of G. From Observation 4.1.7, there is an in-time schedule for $(G,2,D'_0)$. Let $(G,2,D')$ be the pairwise strongly D'_0-consistent instance. From Lemma 6.1.8, $D'(u_1,u_2) = D(u_1,u_2) + \ell^*$ for all pairs of tasks (u_1,u_2) of G. So L is an lst-list of $(G,2,D')$. From Lemma 6.3.1, S is an in-time schedule for (G,m,D'_0). Hence $S(u)+1 \leq D'_0(u) = D_0(u) + \ell^*$ for all tasks u of G. So the tardiness of S as schedule for $(G,2,D_0)$ is at most ℓ^*. Hence S is a minimum-tardiness schedule for $(G,2,D_0)$. From Lemmas 6.2.6 and 4.3.4, S can be constructed in $O(n^2)$ time. $\square$

6.3.2 Interval-ordered tasks

For scheduling interval orders on m processors, we will use a special kind of lst-list. Let G be an interval order and (G,m,D) the pairwise strongly D_0-consistent instance. Let u_1 and u_2 be two tasks of G. Then u_1 has a higher priority than u_2 if

$$\text{either} \quad D(u_1) < D(u_2), \quad \text{or} \quad D(u_1) = D(u_2) \text{ and } Succ_G(u_1) \supsetneq Succ_G(u_2).$$

A list of tasks ordered by non-increasing priority will be called an *interval order lst-list* or *ilst-list* of (G,m,D). Using an ilst-list, Algorithm LIST SCHEDULING constructs in-time schedules, if such schedules exist. The proof is similar to that of Lemma 6.3.1.

Lemma 6.3.3. *Let G be an interval order. Let (G,m,D) be the pairwise strongly D_0-consistent instance. Let S be a schedule for (G,m,D_0) constructed by Algorithm* LIST SCHEDULING *using an ilst-list of (G,m,D). If there is an in-time schedule for (G,m,D_0), then S is an in-time schedule for (G,m,D_0).*

Proof. Assume there is an in-time schedule for (G,m,D_0). From Lemma 6.1.9, there is an in-time schedule for (G,m,D). Assume S is constructed by Algorithm LIST SCHEDULING using ilst-list L of (G,m,D). It will be proved by contradiction that S is an in-time schedule for (G,m,D_0). Suppose S is not an in-time schedule for (G,m,D_0). From Lemma 6.1.9, S is not an in-time schedule for (G,m,D). Assume S_t is the first time slot that contains a task u_1 of G in a pair of tasks (u_1,u_2) whose deadline $D(u_1,u_2)$ is violated. Then u_1 and u_2 are completed after time $D(u_1,u_2)$. Hence $D(u_1,u_2) \leq t$. From Lemma 6.1.11, there are two possibilities: $\min\{D(u_1),D(u_2)\} \leq t$, or $D(u_1,u_2) = t$ and $D(u_1) = D(u_2) = t+1$.

Case 1. $\min\{D(u_1),D(u_2)\} \leq t$.

Let u be one of the tasks u_1 and u_2, such that $D(u) \leq t$. Because there is an in-time schedule for (G,m,D), G contains at most mt tasks with deadline at most t. So there is a time slot $S_{t'-1}$ before S_t that contains less than m tasks with deadline at most t. Assume $S_{t'-1}$ is the latest time slot before S_t that contains at most $m-1$ tasks with deadline at most t. Let H_1 be the subgraph of G induced by $\bigcup_{i=t'}^{t-1} S_i \cup \{v \in \bigcup_{i \geq t} S_i \mid v \prec_G u\} \cup \{u\}$. Then H_1 contains at least $m(t-t')+1$ tasks with deadline at most t. From Observation 4.3.6, no task of H_1 is available at time $t'-1$. Hence every task of H_1 has a predecessor in $S_{t'-2} \cup S_{t'-1}$.

Case 1.1. Every task of H_1 has a predecessor in $S_{t'-1}$.
Define $Q = \{v \in S_{t'-1} \mid D(v) \leq t\}$. Since each task of H_1 has a deadline at most t, each task of H_1 has a predecessor in Q. From Proposition 2.5.5, Q contains a task w that is a predecessor of all tasks of H_1. Because of the communication delays, at most one successor of w can be scheduled at time t'. Consequently, $t = t'$ and u is a successor of w. So $T_D(w,t) \geq 1$. Since (G,m,D) is pairwise consistent, $D(w) \leq t-1 = t'-1$. So w is not completed at or before time $D(w)$. Contradiction.

Case 1.2. Not every task of H_1 has a predecessor in $S_{t'-1}$.
Define $W = \{v \in S_{t'-2} \cup S_{t'-1} \mid v \text{ is a parent of a source of } H_1\}$. From Proposition 2.5.5, W contains a task w_1 that is a predecessor of every task of H_1. Define $W' = W \setminus \{w_1\}$.

Case 1.2.1. Every task of H_1 has a predecessor in W'.
From Proposition 2.5.5, W' contains a task w_2 that is a predecessor of every task of H_1. Then w_1 and w_2 have at least $m(t-t')+1$ common successors with deadline at most t. So $T_D(w_1,w_2,t) \geq m(t-t')+1$. Because (G,m,D) is pairwise consistent, $D(w_1,w_2) \leq t'-2$. So (w_1,w_2) violates deadline $D(w_1,w_2)$. Contradiction.

Case 1.2.2. Not every task of H_1 has a predecessor in W'.
Let v be a task of H_1 that does not have a predecessor in W'. Assume v is a source of H_1. W contains a parent of v, but W' does not. So w_1 is a parent of v. Not every task of H_1 has a predecessor in $S_{t'-1}$, so w_1 is scheduled at time $t'-2$. Because $S_{t'-2}$ does not contain another parent of v and v is not available at time $t'-1$, $S_{t'-1}$ contains a child x of w_1. Algorithm LIST SCHEDULING scheduled x at time $t'-1$ instead of v, so x has a smallest index in L than v. Thus $D(x) \leq D(v)$. As a result, $T_D(w_1,t) \geq m(t-t')+2$. Since (G,m,D) is pairwise consistent, $D(w_1) \leq t'-2$. So w_1 does not finish at or before time $D(w_1)$. Contradiction.

Case 2. $D(u_1,u_2) = t$ and $D(u_1) = D(u_2) = t+1$.
Let u be the task from u_1 and u_2 with the smallest priority (highest index in L). Let U be the set of tasks of G whose priority is at least as high as that of u. Let v_1 and v_2 be two tasks in U. Clearly, $D(v_1), D(v_2) \leq D(u) = t+1$. If $D(v_1) \leq t$ or $D(v_2) \leq t$, then $D(v_1,v_2) \leq t$. Assume $D(v_1) = D(v_2) = t+1$. Since the priority of v_1 and v_2 is at least as high as that of u, $Succ_G(u) = Succ_G(u_1) \cap Succ_G(u_2) \subseteq Succ_G(v_1), Succ_G(v_2)$. By applying Lemma 6.1.12 twice, we obtain $D(v_1,v_2) = t$. In an in-time schedule for (G,m,D), at most one task in U is scheduled after time $t-1$. Since there is an in-time schedule for (G,m,D), U contains at most $mt+1$ tasks. Therefore there is a time slot $S_{t'-1}$ before S_t that contains at most $m-1$ tasks with priority at least as high as u. Assume $S_{t'-1}$ is the last such time slot. Let H_2 be the subgraph of G induced by $\bigcup_{i=t'}^{t-1} S_i \cup \{u_1,u_2\} \cup \{v \in \bigcup_{i \geq t} S_i \mid v \prec_G u_2\}$. Then H_2 contains at least $m(t-t')+2$ tasks and $D(x_1,x_2) \leq t$ for all tasks $x_1 \neq x_2$ of H_2. From Observation 4.3.6,

no task of H_2 is available at time $t'-1$. Hence every task of H_2 has a predecessor that is scheduled at time $t'-2$ or $t'-1$.

Case 2.1. Every task of H_2 has a predecessor in $S_{t'-1}$.
Define $Q = \{v \in S_{t'-1} \mid D(v) \leq t\}$. Since all tasks of H_2 have a deadline at most $t+1$ and (G,m,D) is pairwise consistent, each task of H_2 is a successor of a task in Q. From Proposition 2.5.5, Q contains a task w that is a predecessor of all tasks of H_2. Due to communication delays, at most one successor of w can be scheduled at time t'. As a result, $t=t'$. Then $T_D(w,t+1) \geq 2$. Since (G,m,D) is pairwise consistent, $D(w) \leq t-1$. So w finishes after time $D(w)$. Contradiction.

Case 2.2. Not every task of H_2 has a predecessor in $S_{t'-1}$.
Define $W = \{v \in S_{t'-2} \cup S_{t'-1} \mid v \text{ is a parent of a task of } H_2\}$. From Proposition 2.5.5, W contains a task w_1 that is a predecessor of every task of H_2. Obviously, w_1 is scheduled at time $t'-2$. Let $W' = W \setminus \{w_1\}$.

Case 2.2.1. Every task of H_2 has a predecessor in W'.
From Proposition 2.5.5, W' contains a task w_2 that is a predecessor of all tasks of H_2. Then every task of H_2 is a common successor of w_1 and w_2. Let $V_1 = \{v \in V(H_2) \mid D(v) \leq t\}$ and $V_2 = \{v \in V(H_2) \mid D(v) = t+1\}$. It is easy to see that $N_D(w_1,w_2,t) \geq |V_1|$. All tasks of H_2 have a priority at least as high as u. From Lemma 6.1.12, $P_D(w_1,w_2,t) \geq |V_2|-1$. So $T_D(w_1,w_2,t) \geq m(t-t')+1$. Because (G,m,D) is pairwise consistent, $D(w_1,w_2) \leq t'-2$. So deadline $D(w_1,w_2)$ is violated. Contradiction.

Case 2.2.2. Not every task of H_2 has a predecessor in W'.
Let v be a task of H_2 that has no predecessor in W'. Assume v is a source of H_2. W' does not contain a parent of v. So v is a child of w_1. Since v is not available at time $t'-1$ and $S_{t'-2}$ contains only one parent of v, $S_{t'-1}$ contains another child x of w_1. Algorithm LIST SCHEDULING scheduled x instead of v, so x has a smaller index in L than v. So x has a priority at least as high as u. Using Lemma 6.1.12, $D(x_1,x_2) \leq t$ for all tasks $x_1 \neq x_2$ in $V(H_2) \cup \{x\}$. Let $V_1 = \{v \in V(H_2) \cup \{x\} \mid D(v) \leq t\}$ and $V_2 = \{v \in V(H_2) \cup \{x\} \mid D(v) = t+1\}$. Then $N_D(w_1,t) \geq |V_1|$ and $P_D(w_1,t) \geq |V_2|-1$. Therefore $T_D(w_1,t) \geq m(t-t')+2$. Since (G,m,D) is pairwise consistent, $D(w_1) \leq t'-2$. Hence w_1 is not completed at or before time $D(w_1)$. Contradiction.

□

Lemma 6.3.3 shows that minimum-tardiness schedules for interval-ordered tasks can be constructed in polynomial time.

Theorem 6.3.4. *There is an algorithm with an $O(n^2)$ time complexity that constructs minimum-tardiness schedules for instances $(G,2,D_0)$, such that G is an interval order.*

Proof. Consider an instance $(G,2,D_0)$, such that G is an interval order. Let (G,m,D) be the pairwise strongly D_0-consistent instance. Let S be the schedule for (G,m,D_0) constructed by

Algorithm LIST SCHEDULING using ilst-list L of (G, m, D). We will prove that S is a minimum-tardiness schedule for (G, m, D_0). Let ℓ^* be the tardiness of a minimum-tardiness schedule for (G, m, D_0). Define $D'_0(u) = D_0(u) + \ell^*$ for all tasks u of G. From Observation 4.1.7, there is an in-time schedule for (G, m, D'_0). Let (G, m, D') be the pairwise strongly D'_0-consistent instance. From Lemma 6.1.8, $D'(u_1, u_2) = D(u_1, u_2) + \ell^*$ for all pairs of tasks (u_1, u_2) of G. So L is an ilst-list of (G, m, D'). From Lemma 6.3.3, S is an in-time schedule for (G, m, D'_0). Hence $S(u) + 1 \leq D'_0(u) = D_0(u) + \ell^*$ for all tasks u of G. So the tardiness of S as schedule for (G, m, D_0) is at most ℓ^*. Hence S is a minimum-tardiness schedule for (G, m, D_0). From Lemmas 6.2.11 and 4.3.4, S can be constructed in $O(n^2)$ time. $\square$

6.4 Concluding remarks

In this chapter, it was shown that minimum-tardiness schedules for precedence graphs of width two on two processors and for interval orders on *m* processors can be constructed in polynomial time. For scheduling with release dates and deadlines, a similar approach as the one presented in this chapter can be applied: minimum-tardiness schedules for interval orders and precedence graphs of width two with release dates and deadlines can be constructed in polynomial time [90]. In addition, minimum-tardiness schedule for precedence graphs of width two with arbitrary task lengths can also be constructed in polynomial time using an approach similar to that presented in this chapter [91]. This approach is not suited for interval orders with arbitrary task lengths, because if in an interval order, every task is replaced by a chain of tasks, then the resulting precedence graph is not an interval order.

Like for outforests, a similar approach as the one presented in this chapter can be used to construct minimum-tardiness schedules for precedence graphs of width two on two processors subject to $\{0,1\}$-communication delays in polynomial time. This is not true for interval orders: using a generalisation of a proof of Hoogeveen et al. [47], Schäffter [81] proved that constructing minimum-length schedules for interval orders on *m* processors subject to $\{0,1\}$-communication delays is an NP-hard optimisation problem. Hence it is unlikely that minimum-tardiness schedules for interval orders on *m* processors subject to $\{0,1\}$-communication delays can be constructed in polynomial time.

7 Dynamic programming

In this chapter, we will present two dynamic-programming algorithms for scheduling arbitrary precedence graphs with non-uniform deadlines subject to unit-length communication delays. Using these algorithms, we can construct minimum-tardiness schedules for arbitrary precedence graphs. In Section 7.1, an algorithm of Fulkerson [29] is presented that decomposes precedence graphs of width w into w disjoint chains. Such chain decompositions are used by the dynamic-programming algorithms that are presented in Sections 7.2 and 7.4. The first algorithm is presented in Section 7.2. This dynamic-programming algorithm constructs minimum-tardiness schedules for instances (G,m,D_0). It is similar to the dynamic-programming algorithm presented by Möhring [67] that constructs minimum-length communication-free schedule for precedence graphs with unit-length tasks and the dynamic-programming algorithm of Veltman [87] that constructs minimum-length schedules for precedence graphs with unit-length tasks subject to unit-length communication delays. Like the algorithms of Möhring [67] and Veltman [87], the time complexity of the algorithm presented in Section 7.2 is exponential in the width of the precedence graph. Hence it constructs minimum-tardiness schedules in polynomial time for precedence graphs of bounded width.

Sections 7.3 and 7.4 are concerned with scheduling precedence graphs with arbitrary task lengths. In Section 7.3, it is proved that constructing a minimum-tardiness schedule for a precedence graph of width w on less than w processors is an NP-hard optimisation problem. In Section 7.4, a second dynamic-programming algorithm is presented. This algorithm constructs minimum-tardiness schedules for precedence graphs of width w on at least w processors. Like the algorithm presented in Section 7.2, the time complexity of this algorithm is exponential is the width of the precedence graph, but it constructs minimum-tardiness schedules for precedence graphs of bounded width.

7.1 Decompositions into chains

In this section, we will show how a precedence graph can be decomposed into disjoint chains. Every precedence graph can be viewed as a collection of disjoint chains with precedence constraints between tasks in different chains: every precedence graph with n tasks can be considered as the disjoint union of n chains consisting of one task. Obviously, precedence graphs that do not consist of n pairwise incomparable tasks can be decomposed into a smaller number of chains.

Definition 7.1.1. Let G be a precedence graph. A *chain decomposition* of G is a collection of disjoint chains $C_1,\dots,C_k$ in G, such that $C_1 \cup \cdots \cup C_k = V(G)$.

Let $C_1,\dots,C_k$ be a chain decomposition of a precedence graph G. Then $C_1,\dots,C_k$ will be called a chain decomposition of G into k chains.

Example 7.1.2. Let G be the precedence graph shown in Figure 7.1. It is easy to see that G is a precedence graph of width two. Figure 7.1 shows a chain decomposition of G into two disjoint chains $C_1 = \{c_{1,1},c_{1,2},c_{1,3},c_{1,4},c_{1,5},c_{1,6}\}$ and $C_2 = \{c_{2,1},c_{2,2},c_{2,3},c_{2,4}\}$. A chain decomposition of G into two disjoint chains is not unique: other chain decompositions of G consisting of two

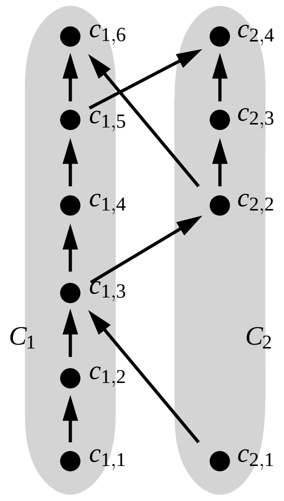

Figure 7.1. A chain decomposition of a precedence graph of width two into two chains

chains are formed by the chains $C'_1 = \{c_{1,1}, c_{1,2}, c_{2,2}, c_{2,3}, c_{2,4}\}$ and $C'_2 = \{c_{2,1}, c_{1,3}, c_{1,4}, c_{1,5}, c_{1,6}\}$ and by the chains $C''_1 = \{c_{1,1}, c_{1,2}, c_{1,3}, c_{2,2}, c_{2,3}, c_{2,4}\}$ and $C''_2 = \{c_{2,1}, c_{1,4}, c_{1,5}, c_{1,6}\}$.

Because a precedence graph of width w contains w pairwise incomparable tasks and incomparable tasks cannot be elements of one chain, a precedence graph of width w cannot be decomposed into less than w chains. Dilworth [22] proved that a precedence graph of width w can be viewed as the disjoint union of exactly w chains.

Theorem 7.1.3. *Let G be a precedence graph of width w. There is a chain decomposition of G into w disjoint chains.*

A chain decomposition of a precedence graph of width w into w chains will be used by the dynamic-programming algorithms presented in Sections 7.2 and 7.4. Dilworth's proof [22] of Theorem 7.1.3 is not constructive, but the proof by Fulkerson [29] is. In his proof of Dilworth's decomposition theorem, Fulkerson presented Algorithm CHAIN DECOMPOSITION shown in Figure 7.2 and proved that it constructs chain decompositions of precedence graphs of width w into w chains.

Algorithm CHAIN DECOMPOSITION works as follows. For a precedence graph G, it constructs an undirected bipartite graph H that contains an edge for every pair of comparable tasks of G and computes a maximum matching of H. This matching is used to construct a chain decomposition of G into disjoint chains.

The time complexity of Algorithm CHAIN DECOMPOSITION can be determined as follows. Let G be a precedence graph of width w. To obtain a better time complexity, we will distinguish two cases depending on whether G is known to be a transitive closure or not. If it is unknown whether G is a transitive closure, then Algorithm CHAIN DECOMPOSITION should start by com-

```
Algorithm CHAIN DECOMPOSITION
Input. A precedence graph $G$ of width $w$, such that $V(G) = \{u_1,\ldots,u_n\}$.
Output. A chain decomposition $C_1,\ldots,C_w$ of $G$.
1.  $V := \{a_1,\ldots,a_n\} \cup \{b_1,\ldots,b_n\}$
2.  $E := \{(a_i,b_j) \mid u_i \prec_G u_j\}$
3.  let $H$ be the undirected bipartite graph $(V,E)$
4.  let $M$ be a maximum matching of $H$
5.  $E' := \{(u_i,u_j) \mid (a_i,b_j) \in M\}$
6.  let $G'$ be the precedence graph $(V(G),E')$
7.  $i := 1$
8.  while $G'$ contains unmarked tasks
9.     do let $u$ be an unmarked source of $G'$
10.        $C_i := \{v \in V(G) \mid$ there is a path from $u$ to $v$ in $G'\}$
11.        mark all tasks in $C_i$
12.        $i := i+1$
```

Figure 7.2. Algorithm CHAIN DECOMPOSITION

puting the transitive closure of G. This takes $O(n+e+ne^-)$ time [37]. In the transitive reduction of a precedence graph of width w, every task has at most w children. Hence $e^- \leq wn$. So the transitive closure of G can be constructed in $O(wn^2)$ time. In the remainder of the analysis of the time complexity of Algorithm CHAIN DECOMPOSITION, we will assume that G is a transitive closure.

Since G is a transitive closure, the bipartite graph H can be constructed in $O(n+e^+)$ time. Since $e^+ \leq n^2$, H is constructed in $O(wn^2)$ time. Hopcroft and Karp [48] presented an algorithm that computes a maximum matching in $O(e\sqrt{n})$ time for bipartite graphs with n nodes and e edges. Alt et al. [5] presented an algorithm whose time complexity is better for dense graphs: their algorithm constructs a maximum matching of a bipartite graph in $O(n\sqrt{ne/\log n})$ time.

The number of edges of H equals e^+. As a result, a maximum matching M of H can be constructed in $O(\min\{e^+\sqrt{n}, n\sqrt{ne^+/\log n}\})$ time. Because the maximum matching of H contains at most n edges, constructing the precedence graph G' takes $O(n)$ time. G' is a chain-like task system. Since every task in G' has indegree and outdegree at most one, constructing the chains in G from G' takes $O(n)$ time. So constructing a chain decomposition of G into w disjoint chains takes $O(wn^2 + \min\{e^+\sqrt{n}, n\sqrt{ne^+/\log n}\})$ time.

Lemma 7.1.4. *For all precedence graphs G of width w, Algorithm* CHAIN DECOMPOSITION *constructs a chain decomposition of G into w chains in $O(wn^2 + \min\{e^+\sqrt{n}, n\sqrt{ne^+/\log n}\})$ time.*

Let G be a precedence graph of width w. Since G can be decomposed into w disjoint chains, G contains a chain that contains at least $\frac{n}{w}$ tasks. The transitive closure of a chain containing at least $\frac{n}{w}$ tasks contains at least $\frac{n(n-w)}{2w^2}$ arcs. So G^+ contains at least $\frac{n(n-w)}{2w^2}$ arcs. If w is a constant, then G^+ contains $\Theta(n^2)$ arcs. Hence using the algorithm of Alt et al. [5], a chain decomposition of a precedence graph of bounded width can be constructed in $O(n^2\sqrt{n/\log n})$ time.

Lemma 7.1.5. *For all precedence graphs G of constant width w, Algorithm* CHAIN DECOMPOSITION *constructs a chain decomposition of G into w chains in $O(n^2\sqrt{n/\log n})$ time.*

7.2 A dynamic-programming algorithm

In this section, a dynamic-programming algorithm will be presented that constructs minimum-tardiness schedules for instances (G,m,D_0). For precedence graphs of width w, it constructs a minimum-tardiness schedule in $O(n^{w+3})$ time. Hence minimum-tardiness schedules for precedence graphs of bounded width can be constructed in polynomial time. The same approach can be used to construct schedules that are optimal with respect to other objective functions (including the minimisation of the makespan) without increasing the time complexity [91]. This leads to an improvement over a result presented by Veltman [87], who showed that minimum-length schedules for precedence graphs of width w can be constructed in $O(n^{2w})$ time.

The time complexity of the dynamic-programming algorithm is exponential in the width of the precedence graph. It is unlikely that there is an algorithm that constructs minimum-length schedules in $O(n^c)$ time, where c is a constant independent of the width of the precedence graph: Bodlaender and Fellows [9] proved that constructing a minimum-length communication-free schedule for arbitrary precedence graphs on k processors is $W[2]$-hard, where $W[2]$ is the second class of the W-hierarchy for parametrised problems introduced by Downey and Fellows [23]. This implies that it is unlikely that for all fixed positive integers k, a minimum-length schedule for a precedence graph on k processors can be constructed in $O(n^c)$ time for some constant c. In fact, Bodlaender and Fellows [9] proved that constructing minimum-length communication-free schedules for precedence graphs of width $k+1$ on k processors is $W[2]$-hard. Their result can be easily generalised for scheduling subject to unit-length communication delays with the objective of minimising the maximum tardiness.

Dynamic programming is a method of constructing an optimal solution of a problem by extending or combining optimal solutions of subproblems. In dynamic programming, the optimal solutions of the subproblems are stored in a table that has an entry for every (relevant) subproblem. The table is then used to construct the best extension or combination of the optimal solutions of the subproblems.

A feasible schedule S for an instance (G,m,D_0) is a list of time slots $(S_0,\dots,S_{\ell-1})$. For each time t, $\bigcup_{i=0}^{t-1} S_i$ is a prefix of G and $(S_0,\dots,S_{t-1})$ is a feasible schedule for $(G[\bigcup_{i=0}^{t-1} S_i],m,D_0)$. $(S_0,\dots,S_{t-1})$ will be called a *partial schedule* for (G,m,D_0). Any schedule S_U for $(G[U],m,D_0)$, such that U is a prefix of G, can be extended to a feasible schedule for (G,m,D_0) by scheduling the remaining tasks after the completion time of the last task of U. So a (minimum-tardiness) schedule for (G,m,D_0) can be constructed by starting with an empty schedule and repeatedly adding the next time slot.

This is the basis of the dynamic-programming algorithm presented in this section: a table containing information about the structure and tardiness of minimum-tardiness partial schedules of (G,m,D_0) is constructed and used to construct a minimum-tardiness schedule for (G,m,D_0).

Let $S=(S_0,\dots,S_{\ell-1})$ be a minimum-tardiness schedule for (G,m,D_0). Then for all times

$t \in \{0,\dots,\ell-1\}$, $(S_0,\dots,S_{t-1})$ is a feasible schedule for $(G[\bigcup_{i=0}^{t-1} S_i], m, D_0)$ and S_t is a set of sources of $G[V(G) \setminus \bigcup_{i=0}^{t-1} S_i]$. So for each task u in S_t, at most one parent of u is an element of S_{t-1} and for each task u in S_{t-1}, at most one child of u is an element of S_t.

The basic idea of extending partial schedules is the following. Let U be a prefix of G and let $(S_0,\dots,S_{t-1})$ be a feasible schedule for $(G[U], m, D_0)$. Then a set of sources V of $G[V(G) \setminus U]$ is called *available* with respect to S if

1. $|V| \leq m$;
2. for all tasks u in V, at most one parent of u finishes at time t; and
3. for all tasks u in U, if u finishes at time t, then V contains at most one child of u.

Note that the availability of V only depends on the size of V and the tasks in U that finish at time t. Hence V will also be called available with respect to (U, S_{t-1}).

If V is available with respect to (U, S_{t-1}), then the schedule $(S_0,\dots,S_{t-1},V)$ is a feasible schedule for $(G[U \cup V], m, D_0)$. Moreover, it is easy to see that for any feasible schedule $S = (S_0,\dots,S_{\ell-1})$ for (G,m,D_0), the time slot S_t is available with respect to $(\bigcup_{i=0}^{t-1} S_i, S_{t-1})$ for all $t \in \{0,\dots,\ell-1\}$.

We will represent a partial schedule S for (G,m,D_0) by a tuple (U,V,t,ℓ): U is the prefix of G, such that S is a feasible schedule for $(G[U],m,D_0)$, t is a starting time that exceeds the starting times of all tasks in U, V is the set of sinks of $G[U]$ that finish at time t and ℓ is the maximum tardiness of a task in U. Note that V may be empty. The time t is used to denote the next time at which the remaining tasks of G can be scheduled.

A tuple (U,V,t,ℓ) will be called a *feasible tuple* of (G,m,D_0) if U is a prefix of G, V is a set of sinks of $G[U]$, and there is a feasible schedule S for $(G[U],m,D_0)$ with tardiness ℓ, such that $S(u) \leq t-1$ for all tasks u in U and $S(u) = t-1$ for all tasks u in V. Since there are minimum-tardiness schedules for (G,m,D_0) of length at most n, we will only consider feasible tuples (U,V,t,ℓ) of (G,m,D_0), such that $0 \leq t \leq n-1$.

Let $S = (S_0,\dots,S_{\ell-1})$ be a feasible schedule for (G,m,D_0). For each time $t \in \{0,\dots,\ell-1\}$, the partial schedule $(S_0,\dots,S_{t-1})$ can be represented by the feasible tuple $(\bigcup_{i=0}^{t-1} S_i, S_{t-1}, t, \ell_t)$ of (G,m,D_0), where $\ell_t = \max\{0, \max\{S(u)+1-D_0(u) \mid S(u) \leq t-1\}\}$.

Note that a feasible tuple (U,V,t,ℓ) of (G,m,D_0) may represent more than one partial schedule. For all partial schedules S represented by (U,V,t,ℓ), the availability of a set of sources of $G[V(G) \setminus U]$ at time t only depends on U and V. So all partial schedules represented by (U,V,t,ℓ) can be extended in the same way. Because the tardiness of such an extension only depends on ℓ and the starting times of the tasks of $G[V(G) \setminus U]$, the minimum-tardiness extensions of the schedules represented by (U,V,t,ℓ) all have the same tardiness. So to construct a minimum-tardiness schedule for (G,m,D_0), we only need to consider feasible tuples of (G,m,D_0).

Partial schedules for (G,m,D_0) can be extended by adding a time slot. The notion of extensions is used for feasible tuples as well. Let (U,V,t,ℓ) and (U',V',t',ℓ') be two feasible tuples of (G,m,D_0). Then (U',V',t',ℓ') is called *available* with respect to (U,V,t,ℓ) if

1. $U' = U \cup V'$;

2. $t' = t+1$; and
3. $\ell' = \max\{\ell, \max_{u \in V'}(t+1-D_0(u))\}$.

The set $Av(U,V,t,\ell)$ contains all feasible tuples of (G,m,D_0) that are available with respect to (U,V,t,ℓ). Note that $Av(U,V,t,\ell)$ cannot be empty, because $(U,\varnothing,t+1,\ell)$ is an element of $Av(U,V,t,\ell)$ for all feasible tuples (U,V,t,ℓ) of (G,m,D_0).

Let $S = (S_0,\dots,S_{\ell-1})$ be a feasible tuple of (G,m,D_0). Then the feasible tuple $(\bigcup_{i=0}^{t} S_i, S_t, t+1, \max\{0, \max\{S(u)+1-D_0(u) \mid S(u) \leq t\}\})$ of (G,m,D_0) is available with respect to the feasible tuple $(\bigcup_{i=0}^{t-1} S_i, S_{t-1}, t, \max\{0, \max\{S(u)+1-D_0(u) \mid S(u) \leq t-1\}\})$ of (G,m,D_0) for all $t \in \{0,\dots,\ell-1\}$.

Let (U,V,t,ℓ) be a feasible tuple of (G,m,D_0). Assume S is a partial schedule for (G,m,D_0) corresponding to (U,V,t,ℓ). Define $T(U,V,t,\ell)$ as the smallest tardiness of a feasible schedule for (G,m,D_0) that extends S. More precisely, if $U \neq V(G)$, then

$$T(U,V,t,\ell) \;=\; \min\{T(U',V',t',\ell') \mid (U',V',t',\ell') \in Av(U,V,t,\ell)\},$$

and if $U = V(G)$, then

$$T(U,V,t,\ell) \;=\; \ell.$$

Then $T(\varnothing,\varnothing,0,0)$ equals the tardiness of a minimum-tardiness schedule for (G,m,D_0). Note that $T(U,V,t,\ell)$ is independent of the partial schedule corresponding to (U,V,t,ℓ): each schedule S for $(G[U],m,D_0)$ with tardiness ℓ, such that $S(u) = t-1$ for all tasks u in V and $S(u) \leq t-1$ for all tasks u in U, can be extended to a feasible schedule for (G,m,D_0) with tardiness $T(U,V,t,\ell)$.

A minimum-tardiness schedule for (G,m,D_0) is computed by Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING presented in Figure 7.3. First, it computes a table *Tab*, such that $Tab[U,V,t,\ell]$ equals $T(U,V,t,\ell)$ for all feasible tuples (U,V,t,ℓ) of (G,m,D_0). Second, it uses this table to construct a minimum-tardiness schedule for (G,m,D_0).

Now we will prove that Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING correctly constructs minimum-tardiness schedules.

Lemma 7.2.1. *Let S be the schedule for (G,m,D_0) constructed by Algorithm* UNIT EXECUTION TIMES DYNAMIC PROGRAMMING. *Then S is a minimum-tardiness schedule for (G,m,D_0).*

Proof. Let *Tab* be the table constructed by Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING. We will prove by induction that $Tab[U,V,t,\ell] = T(U,V,t,\ell)$ for all feasible tuples (U,V,t,ℓ) of (G,m,D_0). Let (U,V,t,ℓ) be a feasible tuple of (G,m,D_0). Assume by induction that $Tab[U',V',t',\ell'] = T(U',V',t',\ell')$ for all feasible tuples (U',V',t',ℓ') in $Av(U,V,t,\ell)$.

If $U = V(G)$, then $T(U,V,t,\ell) = \ell$ for all feasible tuples (U,V,t,ℓ) of (G,m,D_0). In that case, $Tab[U,V,t,\ell] = T(U,V,t,\ell)$. So we may assume that $U \neq V(G)$. Because $T(U,V,t,\ell)$ equals $\min\{T(U',V',t',\ell') \mid (U',V',t',\ell') \in Av(U,V,t,\ell)\}$ and $Tab[U',V',t',\ell'] = T(U',V',t',\ell')$ for all feasible tuples (U',V',t',ℓ') in $Av(U,V,t,\ell)$, $Tab[U,V,t,\ell]$ equals $\min\{T(U',V',t',\ell') \mid$

```
Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING
Input. An instance $(G,m,D_0)$.
Output. A minimum-tardiness schedule for $(G,m,D_0)$.
1.  for all feasible tuples $(U,V,t,\ell)$ of $(G,m,D_0)$
2.      do $Tab[U,V,t,\ell] := \infty$
3.  CONSTRUCT$(\varnothing,\varnothing,0,0)$
4.  $(U,V,t,\ell) := (\varnothing,\varnothing,0,0)$
5.  while $U \neq V(G)$
6.     do let $(U',V',t',\ell') = succ(U,V,t,\ell)$
7.        for $u \in V'$
8.           do $S(u) := t$
9.        $(U,V,t,\ell) := (U',V',t',\ell')$
10.
11. Procedure CONSTRUCT$(U,V,t,\ell)$
12. if $Tab[U,V,t,\ell] = \infty$
13.   then if $U = V(G)$
14.          then $Tab[U,V,t,\ell] := \ell$
15.          else $T := \infty$
16.               for $(U',V',t',\ell') \in Av(U,V,t,\ell)$
17.                  do CONSTRUCT$(U',V',t',\ell')$
18.                     if $Tab[U',V',t',\ell'] < T$
19.                       then $T := Tab[U',V',t',\ell']$
20.                            $succ(U,V,t,\ell) := (U',V',t',\ell')$
21.               $Tab[U,V,t,\ell] := T$
```

Figure 7.3. Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING

$(U',V',t',\ell') \in Av(U,V,t,\ell)\} = T(U,V,t,\ell)$. By induction, $Tab[U,V,t,\ell] = T(U,V,t,\ell)$ for all feasible tuples (U',V',t',ℓ') of (G,m,D_0).

In addition, it is not difficult to see that for all feasible tuples (U,V,t,ℓ) of (G,m,D_0), if $U \neq V(G)$, then $succ(U,V,t,\ell)$ is a feasible tuple in $Av(U,V,t,\ell)$, such that $Tab[succ(U,V,t,\ell)] = Tab[U,V,t,\ell]$. Consequently, for all feasible tuples (U,V,t,ℓ) of (G,m,D_0), if $U \neq V(G)$, then $T(succ(U,V,t,\ell))$ equals $T(U,V,t,\ell)$.

Because $Tab[U,V,t,\ell]$ equals $T(U,V,t,\ell)$ for all feasible tuples (U,V,t,ℓ) of (G,m,D_0), $Tab[\varnothing,\varnothing,0,0]$ equals the tardiness of a minimum-tardiness schedule for (G,m,D_0). This is used to construct a schedule for (G,m,D_0). We inductively define feasible tuples (U_i,V_i,t_i,ℓ_i) of (G,m,D_0). Let $(U_0,V_0,t_0,\ell_0) = (\varnothing,\varnothing,0,0)$. If $U_i \neq V(G)$, then let $(U_{i+1},V_{i+1},t_{i+1},\ell_{i+1}) = succ(U_i,V_i,t_i,\ell_i)$. Assume (U_k,V_k,t_k,ℓ_k) is the last feasible tuple of (G,m,D_0) that can be constructed this way. Then $U_k = V(G)$. It is not difficult to prove that $T(U_i,V_i,t_i,\ell_i) = T(U_0,V_0,t_0,\ell_0)$ for all $i \in \{0,\dots,k\}$. So each feasible tuple (U_i,V_i,t_i,ℓ_i) of (G,m,D_0) represents a partial schedule for (G,m,D_0) that can be extended to a minimum-tardiness schedule for (G,m,D_0). It is easy to prove by induction that the feasible tuple (U_i,V_i,t_i,ℓ_i) of (G,m,D_0) represents the partial schedule $(V_1,\dots,V_i)$ for all $i \in \{0,\dots,k\}$. So $(V_1,\dots,V_k)$ is a minimum-tardiness schedule for (G,m,D_0). This is the schedule constructed by Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING. □

The time complexity of Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING can be determined as follows. Consider an instance (G,m,D_0), such that G is a precedence graph of width w. In order to obtain a better time complexity, we need to consider two possibilities depending on whether G is known to be a transitive reduction or not. If it is unknown whether G is a transitive reduction, then Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING should start by computing the transitive reduction of G. This takes $O(n^{2.376})$ time [17]. In the remainder of the analysis of the time complexity of Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING, we will assume that G is a transitive reduction.

Assume $C_1,\ldots,C_w$ is a chain decomposition of G, such that $C_i = \{c_{i,1},\ldots,c_{i,k_i}\}$ for all $i \in \{1,\ldots,w\}$. From Lemma 7.1.4, such a chain decomposition can be constructed in $O(wn^2 + e^+\sqrt{n})$ time.

Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING first computes $T(U,V,t,\ell)$ for all feasible tuples (U,V,t,ℓ) of (G,m,D_0). Since there is a minimum-tardiness schedule for (G,m,D_0) of length at most n, we may assume that $t \in \{0,\ldots,n-1\}$. In addition, because every task has at most n starting times, at most n^2 different values of ℓ need to be taken into account. A prefix U of G is a set $\bigcup_{i=1}^{w}\{c_{i,1},\ldots,c_{i,b_i}\}$, such that $0 \leq b_i \leq k_i$ for all $i \in \{1,\ldots,w\}$. A set of sinks V of $G[U]$ is a subset of the set $\{c_{1,b_1},\ldots,c_{w,b_w}\}$. A subset V of $\{c_{1,b_1},\ldots,c_{w,b_w}\}$ can be represented by a tuple $(a_1,\ldots,a_w)$, such that $a_i \in \{0,1\}$ for all $i \in \{1,\ldots,w\}$: $a_i = 1$ if $c_{i,b_i} \in V$ and $a_i = 0$ if $c_{i,b_i} \notin V$. So a feasible tuple of (G,m,D_0) can be represented by a tuple $(b_1,\ldots,b_w,a_1,\ldots,a_w,t,\ell)$, such that $0 \leq b_i \leq k_i$ and $a_i \in \{0,1\}$ for all $i \in \{1,\ldots,w\}$, $t \in \{0,\ldots,n-1\}$ and $\ell \in \bigcup_{u\in V(G)}\{1-D_0(u),\ldots,n-D_0(u)\}$. So the number of feasible tuples of (G,m,D_0) is at most

$$n^3 2^w \prod_{i=1}^{w}(k_i+1) \;\leq\; n^3 2^w \prod_{i=1}^{w} 2k_i \;\leq\; n^3 2^{2w} \prod_{i=1}^{w} \frac{n}{w} \;\leq\; 2^w n^{w+3}.$$

For every feasible tuple (U,V,t,ℓ) of (G,m,D_0), Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING computes the set $Av(U,V,t,\ell)$. There is a one-to-one correspondence between the elements of $Av(U,V,t,\ell)$ and the sets of sources of $G[V(G)\setminus U]$. Because G is a precedence graph of width w and the sources of a precedence graph are incomparable, $G[V(G)\setminus U]$ has at most w sources. As a result, $Av(U,V,t,\ell)$ contains at most 2^w elements. Checking the availability of a tuple (U',V',t',ℓ') of (G,m,D_0) with respect to (U,V,t,ℓ) can be done as follows. U' must be the set $U \cup V'$, V' must be a set containing at most m sources of $G[V(G)\setminus U]$, every task in V may have at most one child in V' and every task in V' may have at most one parent in V. Because G is a transitive reduction, every task of G has indegree and outdegree at most w. So the availability of a set of sources of $G[V(G)\setminus U]$ can be checked in $O(w^2)$ time. Hence for each feasible tuple (U,V,t,ℓ) of (G,m,D_0), Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING uses $O(w^2 2^w)$ time. So Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING constructs the table *Tab* in $O(w^2 2^{2w} n^{w+3})$ time.

It is not difficult to see that the construction of the minimum-tardiness schedule for (G,m,D_0) does not require as much time as the construction of the table. So Algorithm UNIT EXECUTION

TIMES DYNAMIC PROGRAMMING constructs a minimum-tardiness schedule for (G,m,D_0) in $O(w^2 2^{2w} n^{w+3})$ time. Hence we have proved the following result.

Theorem 7.2.2. *There is an algorithm with an $O(w^2 2^{2w} n^{w+3})$ time complexity that constructs minimum-tardiness schedules for instances (G,m,D_0), such that G is a precedence graph of width w.*

Consequently, for constant w, a minimum-tardiness schedule for a precedence graph of width w can be constructed in polynomial time.

Theorem 7.2.3. *There is an algorithm with an $O(n^{w+3})$ time complexity that constructs minimum-tardiness schedules for instances (G,m,D_0), such that G is a precedence graph of constant width w.*

Proof. Obvious from Theorem 7.2.2. □

7.3 An NP-completeness result

In the previous section, it was proved that there is a polynomial-time algorithm that constructs minimum-tardiness schedules for precedence graphs of bounded width with unit-length tasks on m processors. Moreover, using a generalisation of the algorithm presented in Chapter 6, a minimum-tardiness schedule for precedence graphs of width two with arbitrary task lengths can be constructed in polynomial time [91].

In this section, it will be shown that constructing a minimum-tardiness schedule for precedence graphs of width w on less than w processors is an NP-hard optimisation problem. This is proved using a polynomial reduction from PARTITION [33].

Problem. PARTITION
Instance. A set of positive integers $A = \{a_1, \ldots, a_n\}$.
Question. Is there a subset A' of A, such that $\sum_{a \in A'} a = \sum_{a \in A \setminus A'} a$?

PARTITION is a well-known NP-complete decision problem [33]. Let WIDTH3ON2 be the following decision problem.

Problem. WIDTH3ON2
Instance. An instance $(G,\mu,2,D_0)$, such that G is a precedence graph of width three.
Question. Is there an in-time schedule for $(G,\mu,2,D_0)$?

Using a polynomial reduction from PARTITION, it will be shown that WIDTH3ON2 is an NP-complete decision problem.

Lemma 7.3.1. *There is a polynomial reduction from* PARTITION *to* WIDTH3ON2.

Proof. Let $A = \{a_1, \ldots, a_n\}$ be an instance of PARTITION. Define $N = \sum_{a \in A} a$ and $M = N + 1$. Construct an instance $(G,\mu,2,D_0)$ as follows. G is a precedence graph consisting of three chains. The first two chains, C_1 and C_2, each consist of $n+1$ tasks $c_{1,i}$ and $c_{2,i}$ of length $\mu(c_{j,i}) = M$, such that $c_{j,0} \prec_{G,0} \cdots \prec_{G,0} c_{j,n}$. The third chain, C_3, consists of n tasks $u_1,\dots,u_n$ with lengths $\mu(u_i) = a_i$ for all $i \in \{1,\dots,n\}$ and precedence constraints $u_1 \prec_{G,0} \cdots \prec_{G,0} u_n$. Let $D_0(u) = \frac{1}{2}N + (n+1)M$ for all tasks u of G. Now we can prove that there is a subset A_1 of A, such that $\sum_{a\in A_1} a = \sum_{a\in A\setminus A_1} a$ if and only if there is an in-time schedule for $(G,\mu,2,D)$.

($\Rightarrow$) Assume there is a subset A_1 of A, such that $\sum_{a\in A_1} a = \sum_{a\in A\setminus A_1} a$. Define $A_2 = A \setminus A_1$. A feasible in-time schedule S for $(G,\mu,2,D_0)$ can be constructed as follows. For each $i \in \{1,\dots,n\}$ and $p \in \{1,2\}$, if $a_i \in A_p$, then let

$$S(u_i) \;=\; iM + \sum_{j<i:a_j\in A_p} a_j.$$

Furthermore, for all $i \in \{0,\dots,n\}$, let

$$S(c_{1,i}) \;=\; iM + \sum_{j\le i:a_j\in A_1} a_j \qquad \text{and} \qquad S(c_{2,i}) \;=\; iM + \sum_{j\le i:a_j\in A_2} a_j.$$

Clearly, $S(c_{p,i+1}) \ge S(c_{p,i}) + M$ for all $i \in \{0,\dots,n\}$ and $p \in \{1,2\}$. In addition, for all $i \in \{1,\dots,n\}$ and $p \in \{1,2\}$, if $a_i \in A_p$, then

$$S(u_i) \;=\; S(c_{p,i-1}) + M \qquad \text{and} \qquad S(u_i) + \mu(u_i) \;=\; S(c_{p,i}).$$

So at most two tasks are executed at the same time. Furthermore, for all $i \in \{0,\dots,n-1\}$ and $p \in \{1,2\}$, if $u_{i+1} \in A_p$, then

$$\begin{aligned} S(u_{i+1}) &= (i+1)M + \textstyle\sum_{j<i+1:a_j\in A_p} a_j \\ &\ge iM + M \\ &> iM + a_i + \textstyle\sum_{j<i} a_j \\ &\ge S(u_i) + \mu(u_i). \end{aligned}$$

So S is a feasible schedule for $(G,\mu,2,D_0)$. Every task of G finishes at or before time $\max\{S(c_{1,n}) + \mu(c_{1,n}), S(c_{2,n}) + \mu(c_{2,n})\} = nM + \frac{1}{2}N + M = (n+1)M + \frac{1}{2}N$. So S is an in-time schedule for $(G,\mu,2,D_0)$.

($\Leftarrow$) Assume S is an in-time schedule for $(G,\mu,2,D_0)$. Then all tasks of G are completed at or before time $(n+1)M + \frac{1}{2}N$. Let π be the processor assignment for S constructed by Algorithm PROCESSOR ASSIGNMENT COMPUTATION. Each processor can execute at most $n+1$ tasks in C_1 or C_2, otherwise, S has length at least $(n+2)M > (n+1)M + \sum_{a\in A} a > (n+1)M + \frac{1}{2}N$. So both processors execute exactly $n+1$ tasks of length M. The sum of the execution lengths of all tasks of G equals $2(n+1)M + N$. So no processor is idle before time $(n+1)M + \frac{1}{2}N$. Define

$$A_1 \;=\; \{a_i \mid \pi(u_i) = 1\} \qquad \text{and} \qquad A_2 \;=\; \{a_i \mid \pi(u_i) = 2\}.$$

Since no processor is idle before time $(n+1)M + \frac{1}{2}N$, $\sum_{a\in A_1} a = (n+1)M + \frac{1}{2}N - (n+1)M = \frac{1}{2}\sum_{a\in A} a$.

□

Lemma 7.3.1 shows that constructing minimum-tardiness schedules for precedence graph of width three on two processors is an NP-hard optimisation problem. It is easy to see that a similar proof can be used to show that constructing minimum-tardiness schedules for precedence graphs of width w on less than w processors is NP-hard as well.

Theorem 7.3.2. *Constructing minimum-tardiness schedules for instances* (G,μ,m,D_0)*, such that* G *is a precedence graph of constant width* w *and* $2 \leq m < w$*, is an* NP*-hard optimisation problem.*

7.4 Another dynamic programming algorithm

In Section 7.2, it was proved that minimum-tardiness schedules for precedence graphs of bounded width can be constructed in polynomial time if all tasks have unit length. In Section 7.3, it is shown that constructing minimum-tardiness schedules for precedence graphs of width w with tasks of arbitrary length on less than w processors is an NP-hard optimisation problem. The complexity of constructing minimum-tardiness schedules for precedence graphs of width w with arbitrary task lengths on at least w processors remains open. Without communication delays, minimum-tardiness schedules for precedence graphs of width w on w processors can be constructed by a list scheduling algorithm (using any priority list). This is not true for scheduling subject to unit-length communication delays.

Example 7.4.1. Consider the instance $(G,3,D_0)$ shown in Figure 7.4. Note that G is a precedence graph of width three. It is not difficult to see that $(G,3,D_0)$ is consistent. Moreover, $(G,3,D_0)$ can be converted into a pairwise consistent instance without decreasing any individual deadlines. Using the lst-list $(a_1,b_3,b_1,b_2,c_3,c_1,c_2,d_1)$, Algorithm LIST SCHEDULING constructs the schedule shown in Figure 7.5. This is not an in-time schedule for $(G,3,D_0)$, because d_1 violates its deadline. In Figure 7.6, an in-time schedule for $(G,3,D_0)$ is shown. This schedule can be constructed by Algorithm LIST SCHEDULING using lst-list $(a_1,b_1,b_2,b_3,c_3,c_1,c_2,d_1)$.

Example 7.4.1 shows that list scheduling does not construct minimum-tardiness schedules for precedence graphs of width w on w processors. In this section, it will be shown that a minimum-tardiness schedule for precedence graphs of width w with arbitrary task lengths on at least w processors can be constructed in polynomial time for each constant w. Like in Section 7.2, we will use a dynamic-programming approach that can be generalised to scheduling problems with other objective functions [91].

Let G be a precedence graph of width w. Consider an instance (G,μ,m,D_0), such that $m \geq w$. In a feasible schedule S for (G,μ,m,D_0), at most w tasks can be executed simultaneously. Hence any feasible schedule for (G,μ,∞,D_0) is a feasible schedule for (G,μ,m,D_0) as well. On the other hand, any feasible schedule for (G,μ,m,D_0) is also a feasible schedule for (G,μ,∞,D_0). Therefore we will consider instances (G,μ,∞,D_0).

A schedule S for (G,μ,∞,D_0) is called *greedy* if for all tasks u of G, there is no feasible schedule S' for (G,μ,∞,D_0), such that $S'(u) < S(u)$ and $S'(v) = S(v)$ for all tasks $v \neq u$ of G.

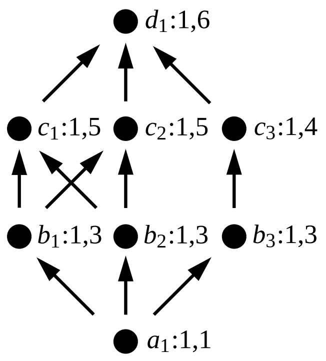

Figure 7.4. A consistent instance $(G, 3, D)$

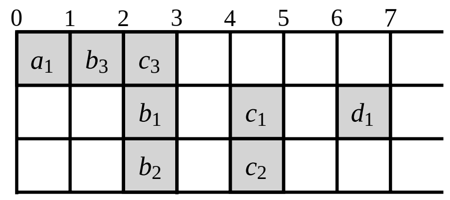

Figure 7.5. The schedule for $(G, 3, D)$ constructed by Algorithm LIST SCHEDULING

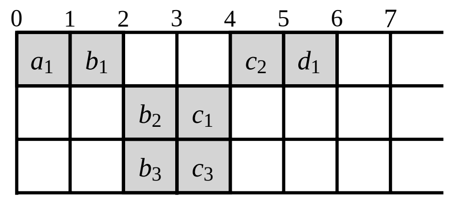

Figure 7.6. An in-time schedule for $(G, 3, D)$

Note that the schedules for (G, μ, ∞, D_0) constructed by Algorithm LIST SCHEDULING are greedy schedules.

Let S be a feasible schedule for (G, μ, ∞, D_0). Then S be transformed into a greedy schedule for (G, μ, ∞, D_0) as follows. Let u be a task of G. If u is available at time $t < S(u)$ and u can be scheduled at time t without violating the feasibility of S, then schedule u at time t. This is repeated until no task can be executed at an earlier time without violating the feasibility. The resulting schedule is a greedy schedule for (G, μ, ∞, D_0). Since no task is scheduled at a later time, the tardiness of this schedule is at most that of S. Hence there is a greedy minimum-tardiness schedule for (G, μ, ∞, D_0).

In a greedy schedule for (G, μ, ∞, D_0), the number of potential starting times of a task is bounded. Let $est(u)$ denote the earliest possible starting time of a task u in a communication-free schedule for (G,μ,∞,D_0).

$$est(u) = \begin{cases} 0 & \text{if } u \text{ is a source of } G \\ \max_{v\in Pred_{G,0}(u)}(est(v)+\mu(v)) & \text{otherwise} \end{cases}$$

In a greedy schedule for (G,μ,∞,D_0), every task u of G starts at most $n-1$ time units after $est(u)$.

Lemma 7.4.2. *Let S be a feasible greedy schedule for (G,μ,∞,D_0). Then for all tasks u of G, $est(u) \leq S(u) \leq est(u)+n-1$.*

Proof. Obviously, $S(u) \geq est(u)$ for all tasks u of G. For all tasks u of G, let $lpp(u)$ be the maximum number of tasks on a path from a source of G to a parent of u.

$$lpp(u) = \begin{cases} 0 & \text{if } u \text{ is a source of } G \\ \max_{v\in Pred_{G,0}(u)} lpp(v)+1 & \text{otherwise} \end{cases}$$

We will prove by induction that $S(u) \leq est(u)+lpp(u)$ for all tasks u of G. This is obvious for the sources of G. Let u be a task of G. Assume by induction that $S(v) \leq est(v)+lpp(v)$ for all predecessors v of u. Let w be a predecessor of u with a maximum completion time. Then u is available at time $S(w)+\mu(w)+1$. So u starts at time $S(w)+\mu(w)$ or at time $S(w)+\mu(w)+1$. Consequently,

$$\begin{aligned} S(u) &\leq \max_{v\in Pred_{G,0}(u)}(S(v)+\mu(v)+1) \\ &\leq \max_{v\in Pred_{G,0}(u)}(est(v)+lpp(v)+\mu(v)+1) \\ &\leq \max_{v\in Pred_{G,0}(u)}(est(v)+\mu(v))+\max_{v\in Pred_{G,0}(u)}(lpp(v)+1) \\ &= est(u)+lpp(u). \end{aligned}$$

Clearly, $lpp(u) \leq n-1$. So $est(u) \leq S(u) \leq est(u)+n-1$. By induction, $est(u) \leq S(u) \leq est(u)+n-1$ for all tasks u of G. □

The limited number of potential starting times will be used in the design of a dynamic-programming algorithm. Let U be a prefix of G. Then any feasible schedule for $(G[U],\mu,\infty,D_0)$ can be extended to a feasible schedule for (G,μ,∞,D_0) by assigning a starting time to the tasks of $G[V(G)\setminus U]$. This is the basis of the dynamic-programming algorithm.

Let S be a feasible schedule for $(G[U],\mu,\infty,D_0)$, such that $S(u) \leq t-1$ for all tasks u in U. Let V be a set of sources of $G[V(G)\setminus U]$. Then V is called *available* at time t with respect to (U,S) if

1. for all tasks u in V, all parents of u are completed at or before time t;
2. for all tasks u in V, at most one parent of u finishes at time t; and
3. for all tasks u in U, if u finishes at time t, then V contains at most one child of u.

Note that the availability of V only depends on the completion times of the sinks of $G[U]$. Moreover, if S is a feasible schedule for (G,μ,∞,D_0), then for all times $t \in \{0,\dots,\max_{u\in V(G)}(S(u)+\mu(u))\}$, the set $\{u \in V(G) \mid S(u) = t\}$ is available at time t with respect to (U,S_U), where $U = \{u \in V(G) \mid S(u) \leq t-1\}$ and S_U is the restriction of S to U.

Partial (greedy) schedules for (G,μ,∞,D_0) will be represented by tuples (U,S,t,ℓ): t is an integer, such that $est(u) \leq t \leq est(u)+n-1$ for some task u of G, U is a prefix of G and S is a schedule for $(G[U],\mu,\infty,D_0)$ with tardiness ℓ, such that $S(u) \leq t-1$ for all tasks in U. The time t denotes the next time at which a task of G can be scheduled. Such a tuple (U,S,t,ℓ) will be called a *feasible tuple* of (G,μ,∞,D_0).

Since partial (greedy) schedules for (G,μ,∞,D_0) can be extended by assigning a starting time to unscheduled tasks, we need a notion of extension of feasible tuples. Let (U,S,t,ℓ) and (U',S',t',ℓ') be two feasible tuples of (G,μ,∞,D_0). Then (U',S',t',ℓ') is called *available* with respect to (U,S,t,ℓ) if

1. $U' \setminus U$ is available at time t with respect to (U,S);
2. $t' \geq t+1$; and
3. $\ell' = \max\{\ell, \max_{u\in U'\setminus U}(t+\mu(u)-D_0(u))\}$.

Let $Av(U,S,t,\ell)$ denote the set of feasible tuples of (G,μ,∞,D_0) that are available with respect to (U,S,t,ℓ). Note that if $U \neq V(G)$, then $Av(U,S,t,\ell)$ cannot be empty, since the feasible tuple (U,S,t',ℓ), such that $t' = \min\{t'' \geq t+1 \mid t'' \in \bigcup_{u\in V(G)}\{est(u),\dots,est(u)+n-1\}\}$, is an element of $Av(U,S,t,\ell)$.

Let S be a greedy schedule for (G,μ,∞,D_0). Then for all times t, the tuple (U_t,S_{U_t},t,ℓ_t), such that $U_t = \{u \in V(G) \mid S(u) \leq t-1\}$, S_{U_t} is the restriction of S to U_t and ℓ_t is the tardiness of S_{U_t}, is a feasible tuple of (G,μ,∞,D_0). In addition, if $U_t \neq V(G)$, then the feasible tuple (U,S_U,t',ℓ_U), where $t' = \min\{t'' \geq t+1 \mid t'' \in \bigcup_{u\in V(G)}\{est(u),\dots,est(u)+n-1\}\}$, $U = \{u \in V(G) \mid S(u) \leq t'-1\}$, S_U is the restriction of S to U and ℓ_U is the tardiness of S_U, is available with respect to (U_t,S_{U_t},t,ℓ_t). So to construct a minimum-tardiness schedule for (G,μ,∞,D_0), we only need to consider feasible tuples of (G,μ,∞,D_0).

Let (U,S,t,ℓ) be a feasible tuple of (G,μ,∞,D_0). Define $T(U,S,t,\ell)$ as the tardiness of a minimum-tardiness schedule for (G,μ,∞,D_0) that extends S. Then for all feasible tuples (U,S,t,ℓ) of (G,μ,∞,D_0), if $U \neq V(G)$, then

$$T(U,S,t,\ell) \;=\; \min\{T(U',S',t',\ell') \mid (U',S',t',\ell') \in Av(U,S,t,\ell)\},$$

and if $U = V(G)$, then

$$T(U,S,t,\ell) \;=\; \ell.$$

Note that $T(\varnothing,\varnothing,0,0)$ equals the tardiness of a minimum-tardiness schedule for (G,μ,∞,D_0).

To implement the computation of $T(\varnothing,\varnothing,t,\ell)$, a table *Tab* is constructed. *Tab* contains an entry $Tab[U,S,t,\ell]$ for all feasible tuples (U,S,t,ℓ) of (G,μ,∞,D_0). We start by setting $Tab[U,S,t,\ell]=\infty$ for all feasible tuples (U,S,t,ℓ) of (G,μ,∞,D_0). Algorithm DYNAMIC PROGRAMMING presented in Figure 7.7 constructs a table *Tab*, such that $Tab[U,S,t,\ell]=T(U,S,t,\ell)$ for all feasible tuples (U,S,t,ℓ) of (G,μ,∞,D_0). This table is used to construct a minimum-tardiness schedule for (G,μ,∞,D_0).

```
Algorithm DYNAMIC PROGRAMMING
Input. An instance $(G,\mu,\infty,D_0)$.
Output. A minimum-tardiness schedule for $(G,\mu,\infty,D_0)$.
1.  for all feasible tuples $(U,S,t,\ell)$ of $(G,\mu,\infty,D_0)$
2.      do $Tab[U,S,t,\ell] := \infty$
3.  CONSTRUCT$(\varnothing,\varnothing,0,0)$
4.  $(U,S,t,\ell) := (\varnothing,\varnothing,0,0)$
5.  while $U \neq V(G)$
6.      do let $(U',S',t',\ell') = succ(U,S,t,\ell)$
7.          for $u \in U' \setminus U$
8.              do $S(u) := t$
9.          $(U,S,t,\ell) := (U',S',t',\ell')$
10.
11. Procedure CONSTRUCT$(U,S,t,\ell)$
12. if $Tab[U,S,t,\ell] = \infty$
13.   then if $U = V(G)$
14.            then $Tab[U,S,t,\ell] := \ell$
15.            else $T := \infty$
16.                 for $(U',S',t',\ell') \in Av(U,S,t,\ell)$
17.                     do CONSTRUCT$(U',S',t',\ell')$
18.                         if $Tab[U',S',t',\ell'] < T$
19.                           then $T := Tab[U',S',t',\ell']$
20.                                $succ(U,S,t,\ell) := (U',S',t',\ell')$
21.                 $Tab[U,S,t,\ell] := T$
```

Figure 7.7. Algorithm DYNAMIC PROGRAMMING

Now we will prove that the schedules constructed by Algorithm DYNAMIC PROGRAMMING are minimum-tardiness schedules.

Lemma 7.4.3. *Let S be the schedule for (G,μ,∞,D_0) constructed by Algorithm* DYNAMIC PROGRAMMING. *Then S is a minimum-tardiness schedule for (G,μ,∞,D_0).*

Proof. Let *Tab* be the table constructed by Algorithm DYNAMIC PROGRAMMING. We can prove by induction that $Tab[U,S,t,\ell]$ equals $T(U,S,t,\ell)$ for all feasible tuples (U,S,t,ℓ) of (G,μ,∞,D_0). Let (U,S,t,ℓ) be a feasible tuple of (G,μ,∞,D_0). Assume by induction that $Tab[U',S',t',\ell']$ equals $T(U',S',t',\ell')$ for all feasible tuples (U',S',t',ℓ') in $Av(U,S,t,\ell)$.

If $U = V(G)$, then $T(U,S,t,\ell) = \ell$. In that case, $Tab[U,S,t,\ell] = T(U,S,t,\ell)$. So we may assume that $U \neq V(G)$. Then $T(U,S,t,\ell)$ equals $\min\{T(U',S',t',\ell') \mid (U',S',t',\ell') \in Av(U,S,t,\ell)\}$. Algorithm DYNAMIC PROGRAMMING determines an element (U',S',t',ℓ') in $Av(U,S,t,\ell)$ with the smallest table entry. Hence $Tab[U,S,t,\ell] = T(U,S,t,\ell)$. By induction, $Tab[U,S,t,\ell] = T(U,S,t,\ell)$ for all feasible tuples (U,S,t,ℓ) of (G,μ,∞,D_0).

In addition, it is not difficult to see that for all feasible tuples (U,S,t,ℓ) of (G,μ,∞,D_0), if $U \neq V(G)$, then $succ(U,S,t,\ell) \in Av(U,S,t,\ell)$ and $Tab[succ(U,S,t,\ell)] = Tab[U,S,t,\ell]$. Since $Tab[U,S,t,\ell]$ equals $T(U,S,t,\ell)$ for all feasible tuples (U,S,t,ℓ) of (G,μ,∞,D_0), $Tab[\varnothing,\varnothing,0,0]$ equals the tardiness of a minimum-tardiness schedule for (G,μ,∞,D_0).

We inductively construct a sequence of feasible tuples (U_i,S_i,t_i,ℓ_i) of (G,μ,∞,D_0). Let $(U_0,S_0,t_0,\ell_0) = (\varnothing,\varnothing,0,0)$. If $U_i \neq V(G)$, then let $(U_{i+1},S_{i+1},t_{i+1},\ell_{i+1}) = succ(U_i,S_i,t_i,\ell_i)$. Assume (U_k,S_k,t_k,ℓ_k) is the last feasible tuple that can be constructed this way. Then $U_k = V(G)$. Then the schedule S_k is the schedule for (G,μ,∞,D_0) constructed by Algorithm DYNAMIC PROGRAMMING. S_k has tardiness ℓ_k. Because $T(U_k,S_k,t_k,\ell_k) = \ell_k = T(\varnothing,\varnothing,0,0)$ and $T(\varnothing,\varnothing,0,0)$ is the tardiness of a minimum-tardiness schedule for (G,μ,∞,D_0), Algorithm DYNAMIC PROGRAMMING constructs a minimum-tardiness schedule for (G,μ,∞,D_0). □

The time complexity of Algorithm DYNAMIC PROGRAMMING can be determined as follows. Consider an instance (G,μ,∞,D_0), such that G is a precedence graph of width w. Like in the analysis of the time complexity of Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING, we will assume that G is a transitive reduction.

Assume $C_1,\dots,C_w$ is a chain decomposition of G, such that $C_i = \{c_{i,1},\dots,c_{i,k_i}\}$ for all $i \in \{1,\dots,w\}$. From Lemma 7.1.4, $C_1,\dots,C_w$ can be constructed in $O(wn^2 + e^+\sqrt{n})$ time.

Algorithm DYNAMIC PROGRAMMING computes $T(U,S,t,\ell)$ for all feasible tuples (U,S,t,ℓ) of (G,μ,∞,D_0). There is a greedy minimum-tardiness schedule for (G,μ,∞,D_0). Hence we need to consider at most n^2 values of t and at most n^2 values of ℓ. A prefix U of G is a set $\bigcup_{i=1}^{w}\{c_{i,1},\dots,c_{i,b_i}\}$, such that $0 \leq b_i \leq k_i$ for all $i \in \{1,\dots,w\}$. Because the availability of a feasible tuple with respect to (U,S,t,ℓ) only depends on the starting times of the sinks of $G[U]$, S can be represented by a tuple $(t_1,\dots,t_w)$, such that $t_i \in \bigcup_{i=1}^{w}\{est(c_{i,b_i}),\dots,est(c_{i,b_i})+n-1\}$ for all $i \in \{1,\dots,w\}$. So a feasible tuple (U,S,t,ℓ) of (G,μ,∞,D_0) can be represented by a tuple $(b_1,\dots,b_w,t_1,\dots,t_w,t,\ell)$, such that $0 \leq b_i \leq k_i$ and $t_i \in \bigcup_{i=1}^{w}\{est(c_{i,b_i}),\dots,est(c_{i,b_i})+n-1\}$ for all $i \in \{1,\dots,w\}$, $t \in \bigcup_{u\in V(G)}\{est(u),\dots,est(u)+n-1\}$ and $\ell \in \bigcup_{u\in V(G)}\{est(u)+\mu(u)-D_0(u),\dots,est(u)+n-1+\mu(u)-D_0(u)\}$. So the number of feasible tuples of (G,μ,∞,D_0) is at most

$$n^4\prod_{i=1}^{w} n(k_i+1) \;\leq\; n^{w+4}\prod_{i=1}^{w} 2k_i \;\leq\; 2^w n^{w+4}\prod_{i=1}^{w}\frac{n}{w} \;\leq\; n^{2w+4}.$$

For each feasible tuple (U,S,t,ℓ) of (G,μ,∞,D_0), Algorithm DYNAMIC PROGRAMMING determines the set $Av(U,S,t,\ell)$. An element of $Av(U,S,t,\ell)$ corresponds to a subset of the sources of $G[V(G)\setminus U]$ and an integer t', such that $est(u) \leq t' \leq est(u)+n-1$ for some task u of G. Since G is a precedence graph of width w and the sources of a precedence graph are incomparable, $Av(U,S,t,\ell)$ contains at most $n^2 2^w$ elements. Since the availability of a feasible tuple only depends on the starting times of the sinks and every task of G has indegree and outdegree at most w, checking whether a feasible tuple (U', S', t, ℓ) of (G, μ, ∞, D_0) is available with respect to (U, S) takes $O(w^2)$ time. Consequently, Algorithm DYNAMIC PROGRAMMING uses $O(n^2 w^2 2^w)$ time for each feasible tuple (U, S, t, ℓ) of (G, μ, ∞, D_0). So the table *Tab* is constructed in $O(w^2 2^w n^{2w+6})$ time.

Using table *Tab*, Algorithm DYNAMIC PROGRAMMING constructs minimum-tardiness schedule for (G, μ, ∞, D_0). It is obvious that the construction of the schedule does not take as much time as the construction of the table. As a result, Algorithm DYNAMIC PROGRAMMING constructs a minimum-tardiness for (G, μ, ∞, D_0) in $O(w^2 2^w n^{2w+6})$ time. Since any feasible schedule for (G, μ, ∞, D_0) is a feasible schedule for (G, μ, ∞, D_0) for all $m \geq w$, we have proved the following result.

Theorem 7.4.4. *There is an algorithm with an $O(w^2 2^w n^{2w+6})$ time complexity that constructs minimum-tardiness schedules for instances (G, μ, m, D_0), such that G is a precedence graph of width w and $m \geq w$.*

For every fixed w, minimum-tardiness schedules can be constructed in polynomial time.

Theorem 7.4.5. *There is an algorithm with an $O(n^{2w+6})$ time complexity that constructs minimum-tardiness schedules for instances (G, μ, m, D_0), such that G is a precedence graph of constant width w and $m \geq w$.*

Proof. Obvious from Theorem 7.4.4. □

7.5 Concluding remarks

In this chapter, it is proved that minimum-tardiness schedules for precedence graphs of bounded width can be constructed in polynomial time. It is obvious that the dynamic-programming approaches presented in this chapter can be generalised in many ways. First of all, both algorithms can be generalised for scheduling with other objective functions [91]. The same is true for scheduling subject to $\{0,1\}$-communication delays and for scheduling with release dates and deadlines. Both generalisations do not increase the time complexity.

The dynamic-programming algorithm for scheduling precedence graphs with unit-length tasks can be generalised in other ways as well. For instance, if a task cannot be executed by every processor or the communication delays may have length at least two, then there is a minimum-tardiness schedule whose length is bounded by a polynomial in the number of tasks. Consequently, the dynamic-programming algorithm presented in Section 7.2 can be generalised to a polynomial-time algorithm for such problems. This is not true for the algorithm presented in Section 7.4. This algorithm does not construct minimum-tardiness schedules for precedence graphs of bounded width in polynomial time if the number of possible starting times in a minimum-tardiness schedule is not bounded by a polynomial in the number of tasks. So this algorithm cannot be used for scheduling preallocated tasks. In addition, Sotskov and Shakhlevich [83] proved that constructing a minimum-length schedule on three processors for a job shop with three jobs is an NP-hard optimisation problem. Hence it is unlikely that there is a polynomial-time algorithm that constructs minimum-tardiness schedules for precedence graphs of constant width w with preallocated tasks on $m \geq w$ processors.

II Scheduling in the LogP model

8 The LogP model

Part II is concerned with scheduling in the LogP model. In this chapter, the LogP model is presented as a scheduling model. In Section 8.1, the communication requirements of the LogP model are presented. The general problem instances for LogP scheduling are introduced in Section 8.2, feasible schedules for such instances are presented in Section 8.3. In Section 8.4, previous results concerning scheduling in the LogP model are presented. An outline of the second part of this thesis is presented in Section 8.5.

8.1 Communication requirements

The LogP model [21] is a model of a distributed memory computer. It consists of a number of identical processors connected by a communication network. Each processor has an unlimited amount of local memory. The processors execute a computer program in an asynchronous manner: one processor can execute a task while another is involved in a communication action. Communication is modelled by message-passing: data is transferred between the processors by sending messages through the communication network.

The LogP model captures the characteristics of a real parallel computer using four parameters.

1. The *latency* L is an upper bound on the time required to send a unit-length message from one processor to another via the communication network. The latency depends on the diameter of the network topology.
2. The *overhead* o is the amount of time during which a processor is involved in sending or receiving a message consisting of one word. During this time, a processor cannot perform other operations.
3. The *gap* g is the minimum length of the delay between the starting times of two consecutive message transmissions or two consecutive message receptions on the same processor. $\frac{1}{g}$ is the communication *bandwidth* available for each processor.
4. P is the *number of processors*.

We will assume that L, o and g are non-negative integers and that $P \in \{2,3,\ldots,\infty\}$.

In addition, Culler et al. [21] make the following assumptions. The communication network is assumed to be of finite capacity: at each time at most $\lceil \frac{L}{g} \rceil$ messages can be in transit from or to any processor. If a processor attempts to send a message that causes such a bound to be exceeded, then this processor stalls until the message can be sent without exceeding the bound of $\lceil \frac{L}{g} \rceil$ messages. Moreover, the time needed to transfer a message from one processor to another is assumed to be exactly L time units: any message arrives at its destination processor exactly L time units after it has been submitted to the communication network by its source processor.

We will consider a *common data semantics* [25]: the children of a task u all need the complete result of u. So the result of the execution of a task needs to be sent at most once to any other processor even if a processor executes more than one child of u.

The communication between processors in the LogP model works as follows. Consider two different processors p_1 and p_2. Assume processor p_1 executes a task u_1 and the processor p_2 a child u_2 of u_1. Then the result of the execution of u_1 must be transferred from processor p_1 to processor p_2 before u_2 can be executed. Assume the result of u_1 is contained in two messages. Then two messages must be sent from processor p_1 to processor p_2. Figure 8.1 shows the communication between processors p_1 and p_2. The *send operations* are represented by s_1 and s_2; r_1 and r_2 are the *receive operations* corresponding to s_1 and s_2, respectively.

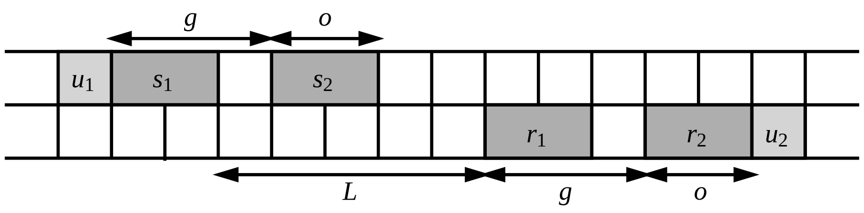

Figure 8.1. Communication between two processors in the LogP model

The first message can be sent by processor p_1 immediately after the completion of u_1. After this message has been submitted to the communication network, exactly L time units are used to send it to processor p_2 through the network. Then it can be received by processor p_2. The second message cannot be sent immediately after the first: there must be a delay of at least g time units between the starting times of two consecutive send operations on the same processor. The second message can be received L time units after it has been sent. Note that the starting times of the receive operations differ at least g time units. After the second message has been received by processor p_2, u_2 can be scheduled.

If another child of u_1 is scheduled after u_2 on processor p_2, then no additional communication is necessary: this child can be executed immediately after u_2. This is due to the fact that the result of u_1 has already been transferred from processor p_1 to processor p_2.

Under a *common data semantics* [25], the children of a task u all need the complete result of u and the result of a task has to be sent to any processor at most once. Under an *independent data semantics* [25], each child of a task u needs a separate part of the result of u. Using an independent data semantics, a separate set of messages has to be sent for every child of u that is not scheduled on the same processor as u. Note that if every task has at most one child, then there is no difference between a common data semantics and an independent data semantics: if a task u has exactly one child, then it requires the complete result of u. In addition, the problem of scheduling outforests under an independent data semantics is the same as scheduling inforests (under either an independent data semantics or a common data semantics).

8.2 Problem instances

The general scheduling instances introduced in Chapter 2 have to be extended to obtain LogP scheduling instances. These instances are extended with the parameters of the LogP model and the sizes of the results of the tasks. Hence we will consider instances (G,μ,c,L,o,g,P), such that tuple (G,μ,c) describes a computer program and (L,o,g,P) contains the parameters of the LogP model. In a tuple (G,μ,c,L,o,g,P), G is a precedence graph, $\mu : V(G) \to \mathbb{Z}^+$ is a function that assigns an execution length to every task of G and $c : V(G) \to \mathbb{N}$ is a function that specifies the number of messages needed to send the result of a task of G to another processor. Because the result of a sink of G is not sent to any processor, we will assume that $c(u)$ equals zero for all sinks u of G. In the remainder of Part II, we will only consider instances (G,μ,c,L,o,g,P), such that $c(u) \geq 1$ for every task u of G that is not a sink of G. All algorithms presented in the following chapters can be easily generalised to scheduling instances (G,μ,c,L,o,g,P) with arbitrary functions c.

Like for scheduling in the UCT model, some special instances will be considered. If all tasks have unit length, then μ will be omitted. In addition, if $c(u)$ equals one for all tasks u of G with outdegree at least one, then c will be left out. So the instance (G,L,o,g,P) corresponds to the instance (G,μ,c,L,o,g,P), such that $\mu(u) = 1$ for all tasks u of G and $c(u) = 1$ for all tasks u of G with outdegree at least one and $c(u) = 0$ for all sinks u of G.

8.3 Feasible schedules

In the LogP model, processors communicate by sending messages to each other. For each task u, messages have to be sent to all processors that execute a child of u except the processor that executes u. So the corresponding send and receive operations may be scheduled for all processors but one. Since we assume a common data semantics, no message needs to be sent to the same processor twice.

Consider a task u_1 and one of its children u_2 that are scheduled on different processors. Assume u_1 is executed on processor p_1 and u_2 on processor $p_2 \neq p_1$. Then $c(u_1)$ messages $m_{u,1},\dots,m_{u,c(u)}$ have to be sent from processor p_1 to processor p_2. Sending message $m_{u,i}$ to processor p_2 will be represented by the *send operation* $s_{u,p_2,i}$. This send operation must be executed on processor p_1. The reception of message $m_{u,i}$ is represented by a *receive operation* $r_{u,p_2,i}$ that must be executed by processor p_2.

We will define two sets $S(G,P,c)$ and $R(G,P,c)$ containing the send and the receive operations, respectively. $S(G,P,c)$ contains the send operations $s_{u,p,i}$, such that u is a task of G that is not a sink of G, $p \in \{1,\dots,P\}$ is a processor and $i \in \{1,\dots,c(u)\}$ is the index of a message of u. The set $R(G,P,c)$ contains the receive operations $r_{u,p,i}$, such that u is a task of G that is not a sink of G, $p \in \{1,\dots,P\}$ and $i \in \{1,\dots,c(u)\}$. Let $C(G,P,c)$ be the union of $S(G,P,c)$ and $R(G,P,c)$, the set of *communication operations*. Each communication operation u in $C(G,P,c)$ has *length* $\mu(u) = o$.

Note that the communication operations have length zero if o equals zero. Because there must be a delay of at least g time units between the starting times of two consecutive send operations or two consecutive receive operations on the same processor, the presence of zero-length communication operations is not the same as the absence of communication operations.

A *schedule* for an instance (G,μ,c,L,o,g,P) is a pair of functions (σ,π), such that $\sigma : V(G) \cup C(G,P,c) \to \mathbb{N} \cup \{\perp\}$ and $\pi : V(G) \cup C(G,P,c) \to \{1,\dots,P\} \cup \{\perp\}$. σ assigns a starting time to every element of $V(G) \cup C(G,P,c)$ and π assigns a processor to each operation in $V(G) \cup C(G,P,c)$. The value $\perp$ denotes the starting time and processor of communication operations that are not scheduled.

Definition 8.3.1. A schedule (σ,π) for (G,μ,c,L,o,g,P) is called *feasible* if

1. for all tasks u of G, $\sigma(u) \neq \perp$ and $\pi(u) \neq \perp$;
2. for all elements u_1 and u_2 of $V(G) \cup C(G,P,c)$, if $\pi(u_1) = \pi(u_2) \neq \perp$, then $\sigma(u_1) + \mu(u_1) \leq \sigma(u_2)$ or $\sigma(u_2) + \mu(u_2) \leq \sigma(u_1)$;
3. for all tasks u_1 and u_2 of G, if $u_1 \prec_G u_2$, then $\sigma(u_1) + \mu(u_1) \leq \sigma(u_2)$;
4. for all tasks u_1 and u_2 of G, if u_2 is a child of u_1 and $\pi(u_1) \neq \pi(u_2)$, then, for all $i \leq c(u_1)$, $\pi(s_{u_1,\pi(u_2),i}) = \pi(u_1)$, $\pi(r_{u_1,\pi(u_2),i}) = \pi(u_2)$, $\sigma(s_{u_1,\pi(u_2),i}) \geq \sigma(u_1) + \mu(u_1)$, $\sigma(r_{u_1,\pi(u_2),i}) = \sigma(s_{u_1,\pi(u_2),i}) + o + L$ and $\sigma(u_2) \geq \sigma(r_{u_1,\pi(u_2),i}) + o$;
5. for all send operations s_1 and s_2 in $S(G,P,c)$, if $\pi(s_1) = \pi(s_2) \neq \perp$, then $\sigma(s_1) + g \leq \sigma(s_2)$ or $\sigma(s_2) + g \leq \sigma(s_1)$;
6. for all receive operations r_1 and r_2 in $R(G,P,c)$, if $\pi(r_1) = \pi(r_2) \neq \perp$, then $\sigma(r_1) + g \leq \sigma(r_2)$ or $\sigma(r_2) + g \leq \sigma(r_1)$; and
7. for all tasks u of G and all processors p, if no children of u are scheduled on processor p or $p = \pi(u)$, then $\sigma(s_{u,p,i}) = \perp$ and $\pi(r_{u,p,i}) = \perp$.

The first constraint states that all tasks of G have to be executed. The second and third correspond to the constraints for feasible communication-free schedules: a processor cannot execute two tasks at the same time and a task must be scheduled after its predecessors. The fourth states that messages have to be sent if a task and one of its children are scheduled on different processors. Moreover, it states that a message must be received exactly L time units after it has been submitted to the communication network. The fifth and sixth constraint ensure that there is a delay of at least g time units between two consecutive send or receive operations on the same processor. Note that there need not be a delay between a send operation and a receive operation on the same processor. The last constraint states that some communication operations need not be executed.

In the definition of the LogP model [21], processors can send messages to other processors, unless the number of messages in transit from or to one processor exceeds $\lceil \frac{L}{g} \rceil$, in which case the sending processor stalls. The definition of feasible schedules in the LogP model states that a receive operation must be executed exactly L time units after the corresponding send operation has been completed. So each processor can send at most one message in g consecutive time units and at most one message can be sent to the same processor in g consecutive time units. Hence the number of messages in transit from or to any processor cannot be larger than $\lfloor \frac{L+\max\{o,g\}-1}{\max\{o,g\}} \rfloor \leq \lceil \frac{L-1}{g} \rceil + 1 \leq \lceil \frac{L}{g} \rceil$. So we do not need to consider stalling.

Constructing a schedule for an instance (G,μ,c,L,o,g,P) corresponds to assigning a starting time and a processor to every task of G and every communication operation in $C(G,P,c)$. Hence any algorithm that constructs feasible schedules for instances (G,μ,c,L,o,g,P) uses at least $\Theta(\sum_{u \in V(G)} c(u))$ time. If $c_{\max} = \max_{u \in V(G)} c(u)$ is not bounded by a polynomial in n and $\log \max_{u \in V(G)} \mu(u)$, then such an algorithm cannot have a polynomial time complexity.

In a well-structured computer program, the size of a result of a task is not very large. Hence we may assume that $c_{\max}$ is not exponentially large. In the rest of Part II, we do not want to focus on the time needed to schedule the communication operations. Hence we will assume that $c_{\max}$ is bounded by a constant. However, the time complexity of the algorithms presented in the remaining chapters of Part II remains polynomial if $c_{\max}$ is bounded by a polynomial in n and $\log \max_{u \in V(G)} \mu(u)$: the time complexity of the algorithms must be increased by $O(n c_{\max})$ to account for the assignment of a starting time and a processor to each communication operation.

This section will be concluded with two examples of feasible schedules. The first is a schedule for the same graph as the one in Sections 2.1 and 3.4.

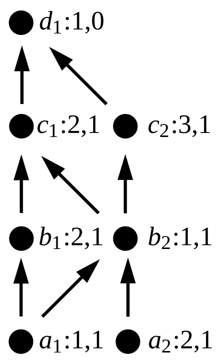

Figure 8.2. An instance $(G, \mu, 1, 1, 1, 2)$

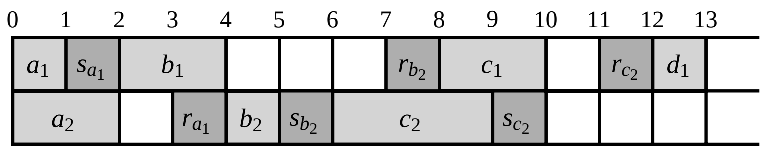

Figure 8.3. A feasible schedule for $(G, \mu, 1, 1, 1, 2)$

Example 8.3.2. Consider the instance $(G, \mu, 1, 1, 1, 2)$ shown in Figure 8.2. Each task of G is labelled with its name, its execution length and the number of messages required to send its result to another processor. The instance $(G, \mu, 1, 1, 1, 2)$ corresponds to the general scheduling instance $(G, \mu, 2)$ shown in Figure 2.1 and the UCT instance $(G, \mu, 2, D)$ shown in Figure 3.1. A feasible schedule for $(G, \mu, 1, 1, 1, 2)$ is shown in Figure 8.3. a_1 and a_2 are scheduled on different processors. b_2 is a common child of a_1 and a_2. So the result of a_1 is sent to the second processor. This is represented by tasks s_{a_1} and r_{a_1}. Note that there is a delay of one time unit between the completion time of s_{a_1} and the starting time of r_{a_1}. Since a_1 is the only parent of b_1 and b_2 is the only parent of c_2, these tasks can be scheduled without extra communication on the first and second processor, respectively. c_1 is a child of b_1 and b_2. Because its parents are scheduled on different processors, the result of b_2 is sent to the first processor before c_1 is executed. Similarly, the result of c_2 is sent to the first processor before d_1 starts.

The next example shows a schedule for an instance (G,μ,c,L,o,g,P) in which g exceeds o. It shows that the idle time between consecutive communication operations can be used to execute tasks.

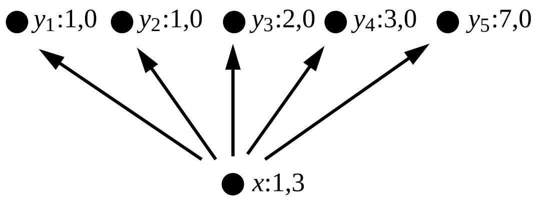

Figure 8.4. An instance $(G,\mu,c,2,1,2,2)$

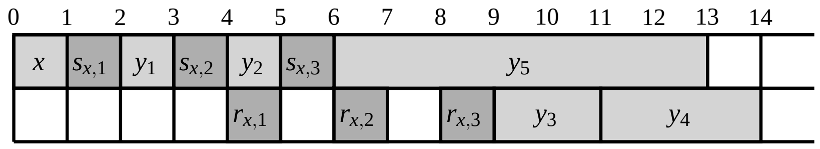

Figure 8.5. A feasible schedule for $(G,\mu,c,2,1,2,2)$

Example 8.3.3. Consider the instance $(G,\mu,c,2,1,2,2)$ shown in Figure 8.4. It is not difficult to see that the schedule shown in Figure 8.5 is a feasible schedule for $(G,\mu,c,2,1,2,2)$. Note that y_1 and y_2 are scheduled between the send operations on processor 1. No task can be executed between the receive operations on processor 2, since all three messages are needed to send the result of x to another processor. Although two children of x are executed on the second processor, only three send and receive operations are executed. This is due to the fact that we assume a common data semantics: the complete result of x is sent to the second processor and it has to be sent to this processor exactly once. Under an independent data semantics, two separate sets of messages must be sent to the second processor: a set of messages for y_3 and one for y_4.

Examples 8.3.2 and 8.3.3 show that schedules in the LogP model are very different from communication-free schedules and from schedules in the UCT model. However, communication-free scheduling and scheduling in the UCT model can be seen as special cases of scheduling in the LogP model: if all tasks have unit length or the number of processors is unrestricted, then any communication-free schedule can be viewed as a schedule in the LogP model with parameters $L = o = g = 0$ and any schedule in the UCT model as a schedule in the LogP model with parameters $L = 1$ and $o = g = 0$.

A feasible schedule (σ, π) for an instance (G, m, D) in the UCT model can be transformed into a feasible schedule for the instance $(G, c, 1, 0, 0, m)$ in the LogP model by scheduling the send and receive operations. For all tasks u of G, all processors $p \neq \pi(u)$ that execute a child of u and all $i \in \{1, \ldots, c(u)\}$, send operation $s_{u,p,i}$ must be executed at time $\sigma(u)+1$ on processor $\pi(u)$ and receive operation $r_{u,p,i}$ at time $\sigma(u)+2$ on processor p. Since $g = o = 0$, the resulting schedule is a feasible schedule for $(G, c, 1, 0, 0, m)$. A feasible communication-free schedule for an instance (G, μ, m), such that $\mu(u) = 1$ for all tasks u of G, can be transformed into a feasible schedule for the instance $(G, c, 0, 0, 0, m)$ in the LogP model in a similar way. Moreover, communication-free schedules for instances (G, μ, ∞) can be transformed into feasible schedules for instances $(G, \mu, c, 0, 0, 0, \infty)$ and schedules in the UCT model for instances (G, μ, ∞, D) into feasible schedules for instances $(G, \mu, c, 1, 0, 0, \infty)$. Both transformations do not change the starting time of any tasks, but they may schedule tasks on different processors.

8.4 Previous results

Like for many other models of parallel computation, little is known about scheduling in the LogP model. A few algorithms have been presented that construct schedules in the LogP model for common computer programs. These programs include sorting [1, 24], broadcast [54] and the Fast Fourier Transform [20].

In addition, Löwe and Zimmermann [63, 95] presented an algorithm that constructs schedules for communication structures of PRAMs on an unrestricted number of processors. The length of these schedules is at most $1 + \frac{1}{\gamma(G)}$ times the length of a minimum-length schedule, where $\gamma(G)$ is the grain size of G. Löwe et al. [64] proved the same result for a generalisation of the LogP model. Moreover, Löwe and Zimmermann [63] presented an algorithm that constructs schedules of length at most twice as long as a minimum-length schedule plus the duration of the sequential communication operations.

Simultaneously to my research on scheduling in the LogP model, Kort and Trystram [55] studied the problem of scheduling in the LogP model. They presented three algorithms for scheduling send graphs under an independent data semantics [25]. They proved that if g equals o and all sinks or all messages have the same length, then a minimum-length schedule for a send graph on an unrestricted number of processors can be constructed in polynomial time. Because scheduling send graphs under an independent data semantics corresponds to scheduling receive graphs (under a common data semantics), their result also shows that minimum-length schedules for receive graphs on an unrestricted number of processors can be constructed in polynomial time if g equals o and all sources have the same execution length or all message lengths are equal. In addition, Kort and Trystram [55] showed that if all sinks have the same length and this length is at least $\max\{g, 2o+L\}$, then a minimum-length schedule for a send graph on two processors can be constructed in linear time.

8.5 Outline of the second part

The remaining chapters of Part II are concerned with the problem of constructing minimum-length schedules in the LogP model. In the next chapter, we study the problem of scheduling send graphs. It is proved that constructing minimum-length schedules for a send graph on an unrestricted number of processors is a strongly NP-hard optimisation problem. A polynomial-time algorithm is presented that constructs schedules for send graphs on P processors that are at most twice as long as a minimum-length schedule on P processors. In addition, it is shown that if all task lengths are equal, then a minimum-length schedule for a send graph on P processors can be constructed in polynomial time.

In Chapter 10, two polynomial-time approximation algorithms for scheduling receive graphs are presented. The first is a 3-approximation algorithm for scheduling receive graphs on an unrestricted number of processors. For each constant $k \in \mathbb{Z}^+$, the second algorithm can construct schedules for receive graphs on P processors that are at most $3 + \frac{1}{k+1}$ times as long as minimum-length schedules on P processors. Moreover, it is proved that if all task lengths are equal, then a minimum-length schedule for a receive graph on an unrestricted number of processors can be constructed in polynomial time.

In Chapter 11, two algorithms are presented that decompose inforests into subforests whose sizes do not differ much. Using the decompositions constructed by the first algorithm, schedules for d-ary inforests on P processors are constructed that have a length that is at most the sum of $d + 1 - \frac{d^2+d}{d+P}$ times the length of a minimum-length schedule on P processors and the duration of $d(P-1)-1$ communication actions. The decompositions constructed by the other algorithm can be used to construct schedules on P processors with a length that is at most the sum of $3 - \frac{6}{P+2}$ times the length of a minimum-length schedule on P processors and the duration of $d(d-1)(P-1)-1$ communication actions.

9 Send graphs

In this chapter, the problem of scheduling send graphs in the LogP model is studied. In Section 9.1, it is proved that constructing minimum-length schedules for send graphs on an unrestricted number of processors is a strongly NP-hard optimisation problem. A polynomial-time 2-approximation algorithm for scheduling send graphs is presented in Section 9.2. In Section 9.3, it is shown that if all task lengths are equal, then a minimum-length schedule for a send graph can be constructed in polynomial time.

9.1 An NP-completeness result

In this section, we study the complexity of constructing minimum-length schedules for send graphs in the LogP model. If the number of processors is restricted, then it is not difficult to prove that this optimisation problem is NP-hard. Using a polynomial reduction from 3PARTITION, it will be shown that constructing minimum-length schedules for send graphs on an unrestricted number of processors is strongly NP-hard. 3PARTITION is defined as follows [33].

Problem. 3PARTITION
Instance. A set $A = \{a_1, \dots, a_{3m}\}$ of positive integers and an integer B, such that $\sum_{i=1}^{3m} a_i = mB$ and $\frac{1}{4}B < a_i < \frac{1}{2}B$ for all $i \in \{1, \dots, 3m\}$.
Question. Are there pairwise disjoint subsets $A_1, \dots, A_m$ of A, such that $\sum_{a \in A_j} a = B$ for all $j \in \{1, \dots, m\}$?

3PARTITION is a well-known strongly NP-complete decision problem [33]. SEND GRAPH SCHEDULING is the following decision problem.

Problem. SEND GRAPH SCHEDULING
Instance. An instance $(G, \mu, L, o, g, \infty)$, such that G is a send graph and an integer D.
Question. Is there a feasible schedule for $(G, \mu, L, o, g, \infty)$ of length at most D?

Lemma 9.1.1 shows the existence of a polynomial reduction from 3PARTITION to SEND GRAPH SCHEDULING. This reduction shows that SEND GRAPH SCHEDULING is a strongly NP-complete decision problem.

Lemma 9.1.1. *There is a polynomial reduction from* 3PARTITION *to* SEND GRAPH SCHEDULING.

Proof. Let $A = \{a_1, \dots, a_{3m}\}$ and B be an instance of 3PARTITION. Construct an instance $(G, \mu, L, o, g, \infty)$ of SEND GRAPH SCHEDULING as follows. G is a send graph with source x and sinks $y_1, \dots, y_{3m}$ and $z_1, \dots, z_{m+2}$. Let $\mu(x) = 1$, $\mu(y_i) = a_i$ for all $i \in \{1, \dots, 3m\}$, $\mu(z_1) = 3mB$ and $\mu(z_i) = 3mB + (m + 2 - i)B$ for all $i \in \{2, \dots, m + 2\}$. Let $c(x) = 1$, $c(y_i) = 0$ for all $i \in \{1, \dots, 3m\}$ and $c(z_i) = 0$ for all $i \in \{1, \dots, m + 2\}$. Let $L = 0$, $o = 0$ and $g = B$. In addition, let $D = 4mB + 1$. Now it is proved that there is a collection of pairwise disjoint subsets $A_1, \dots, A_m$ of A, such that $\sum_{a \in A_j} a = B$ for all $j \in \{1, \dots, m\}$ if and only if there is a feasible schedule for $(G, \mu, L, o, g, \infty)$ of length at most D.

($\Rightarrow$) Assume $A_1, \ldots, A_m$ is a collection of pairwise disjoint subsets of A, such that $\sum_{a \in A_j} a = B$ for all $j \in \{1, \ldots, m\}$. Then $A_1 \cup \cdots \cup A_m = A$. A schedule (σ, π) for $(G, \mu, L, o, g, \infty)$ can be constructed as follows. x starts at time 0 on processor 1. For all $i \in \{2, \ldots, m+2\}$, send operation $s_{x,i,1}$ is executed at time $(i-2)B+1$ on processor 1 and receive operation $r_{x,i,1}$ at time $(i-2)B+1$ on processor i. Sink z_1 is scheduled at time $mB+1$ on processor 1 and sink z_i at time $(i-2)B+1$ on processor i for all $i \in \{2, \ldots, m+2\}$. For all $j \in \{1, \ldots, m\}$, define $Y_j = \{y_i \mid a_i \in A_j\}$. Then $\sum_{y \in Y_j} \mu(y) = B$ for all $j \in \{1, \ldots, m\}$. The tasks of Y_j are scheduled without interruption from time $(j-1)B+1$ to time $jB+1$ on processor 1. Then the sinks $y_1, \ldots, y_{3m}$ are scheduled between the send operations on processor 1 and the sinks $z_1, \ldots, z_{m+2}$ after the communication operations. Hence (σ, π) is a feasible schedule for $(G, \mu, L, o, g, \infty)$. Its length equals $\max_{1 \leq i \leq m+2}(\sigma(z_i) + \mu(z_i))$. z_1 is completed at time $\sigma(z_1) + \mu(z_1) = mB + 1 + 3mB = 4mB + 1$. For all $i \in \{2, \ldots, m+2\}$, sink z_i finishes at time $\sigma(z_i) + \mu(z_i) = (i-2)B + 1 + 3mB + (m+2-i)B = 4mB + 1$. Hence (σ, π) is a feasible schedule for $(G, \mu, L, o, g, \infty)$ of length at most D.

($\Leftarrow$) Assume (σ, π) is a feasible schedule for $(G, \mu, L, o, g, \infty)$ of length at most D. Then $\pi(z_i) \neq \pi(z_j)$ for all $i \neq j$. So the tasks of G are scheduled on at least $m+2$ processors. Assume x is scheduled at time 0 on processor 1. There is a sink z_i that is scheduled after $m+1$ receive operations. This task cannot start until time $mg + 1 = mB + 1$. Since $\mu(z_i) \geq 3mB$ for all $i \in \{1, \ldots, m+2\}$, we may assume that z_{m+2} is scheduled at time $mB+1$. Since it starts at time $mB+1$, send operations must be executed at times $(i-2)B+1$ on processor 1 for all $i \in \{2, \ldots, m+2\}$. We may assume that send operation $s_{x,i,1}$ is scheduled at time $(i-2)B+1$ on processor 1 and receive operation $r_{x,i,1}$ at the same time on processor i. Hence we may assume that $\pi(z_{m+2}) = m+2$. The remaining sinks $z_1, \ldots, z_{m+1}$ must be scheduled on processors $1, \ldots, m+1$. Since the length of the sinks $z_2, \ldots, z_{m+1}$ is larger than $3mB$, z_1 must be scheduled on processor 1 at time $mB+1$. Similarly, sink z_i must be scheduled on processor i at time $(i-2)B+1$ for all $i \in \{2, \ldots, m+1\}$. Then all sinks $z_1, \ldots, z_{m+2}$ finish at time $4mB+1$. A sink y_i cannot be executed on processor $j \neq 1$ before sink z_j, because z_j is scheduled immediately after receive operation $r_{x,j,1}$. So sinks $y_1, \ldots, y_{3m}$ are scheduled between the send operations on processor 1. There is a delay of mB time units between the first and last send operation. Since the sum of the length of the sinks $y_1, \ldots, y_{3m}$ equals mB, processor 1 is not idle before time D. No sink y_i can start before a send operation and finish after it. For all $j \in \{2, \ldots, m+1\}$, define $Y_{j-1} = \{y_i \mid (j-2)B + 1 \leq \sigma(y_i) < (j-1)B + 1\}$ and $A_{j-1} = \{a_i \mid y_i \in Y_j\}$. Then the sets A_j are pairwise disjoint and $\sum_{a \in A_j} a = \sum_{y \in Y_j} \mu(y) = B$ for all $j \in \{1, \ldots, m\}$.

□

Lemma 9.1.1 shows that SEND GRAPH SCHEDULING is a strongly NP-complete decision and that constructing minimum-length schedules for send graphs on an unrestricted number of processors is strongly NP-hard.

Theorem 9.1.2. *Constructing minimum-length schedules for instances $(G, \mu, L, o, g, \infty)$, such that G is a send graph, is a strongly* NP*-hard optimisation problem.*

The reduction presented in the proof of Lemma 9.1.1 uses the fact that g may exceed o. Using a reduction from PARTITION [33], one can also prove that if $o \geq g$ and $o \geq 1$, then constructing a minimum-length schedule for a send graph on an unrestricted number of processors is an NP-hard optimisation problem. It is not clear whether constructing a minimum-length schedule for a send graph on an unrestricted number of processors remains NP-hard if o, g and $c(x)$ are bounded by a constant. If both o and g equal zero, then a minimum-length schedule for a send graph on an unrestricted number of processors can be constructed in polynomial time [13].

9.2 A 2-approximation algorithm

In this section, a simple 2-approximation algorithm for scheduling send graphs in the LogP model is presented. It is obvious that for a minimum-length schedule for an instance (G,μ,c,L,o,g,P), such that G is a send graph, the number of processors on which a task of G is scheduled need not exceed the number of sinks of G. For each possible number of processors m, the algorithm presented in this section constructs a schedule for (G,μ,c,L,o,g,P) that uses exactly m processors. It will be proved that the shortest of these schedules is at most twice as long as a minimum-length schedule for (G,μ,c,L,o,g,P).

Consider an instance (G,μ,c,L,o,g,P), such that G is a send graph with source x and sinks $y_1,\dots,y_n$. There is a minimum-length schedule for (G,μ,c,L,o,g,P) that uses at most $\min\{n,P\}$ processors. Let $m \leq \min\{n,P\}$ be a positive integer. A feasible schedule for (G,μ,c,L,o,g,P) will be called an *m-processor schedule* for (G,μ,c,L,o,g,P) if there are exactly m processors on which a task of G is executed. More precisely, a feasible schedule (σ,π) for (G,μ,c,L,o,g,P) is an m-processor schedule for (G,μ,c,L,o,g,P) if $|\{\pi(u) \mid u \in V(G)\}| = m$.

Consider an instance (G,μ,c,L,o,g,P), such that G is a send graph with source x and sinks $y_1,\dots,y_n$. Algorithm SEND GRAPH SCHEDULING shown in Figure 9.1 constructs an m-processor schedule for (G,μ,c,L,o,g,P) as follows. The source x of G is scheduled at time 0 on processor 1 and a set of $c(x)$ send and receive operations is scheduled for each of the processors $2,\dots,m$. To ensure that the constructed schedule is an m-processor schedule, a sink of G is scheduled after the last receive operation on each of these processors. The remaining sinks are scheduled by a straightforward modification of Graham's List scheduling algorithm [38, 39].

Example 9.2.1. Consider the instance $(G,\mu,c,2,1,2,\infty)$ shown in Figure 9.2. For this instance, Algorithm SEND GRAPH SCHEDULING constructs the 3-processor schedule shown in Figure 9.3. x is scheduled on processor 1 at time 0. The result of x is sent to processors 2 and 3. Sink y_1 is scheduled after the last receive operation on processor 2. Similarly, y_2 is scheduled after the last receive operation on processor 3. The other sinks are scheduled after the send operations on processor 1, after y_1 on processor 2, or after y_2 on processor 3.

Now we will prove that Algorithm SEND GRAPH SCHEDULING correctly constructs feasible m-processor schedules for send graphs.

Lemma 9.2.2. *Let G be a send graph with source x and sinks $y_1,\dots,y_n$. Let $m \leq \min\{n,P\}$ be a positive integer. Let (σ_m,π_m) be the schedule for (G,μ,c,L,o,g,P) constructed by*

Algorithm SEND GRAPH SCHEDULING
Input. An instance (G,μ,c,L,o,g,P), such that G is a send graph with source x and sinks $y_1,\dots,y_n$ and a positive integer $m \leq \min\{n,P\}$.
Output. A feasible m-processor schedule (σ_m,π_m) for (G,μ,c,L,o,g,P).

```
1.  $\sigma_m(x) := 0$
2.  $\pi_m(x) := 1$
3.  $idle(1) := \mu(x)$
4.  for $p := 2$ to $m$
5.      do $idle(p) := 0$
6.          for $j := 1$ to $c(x)$
7.              do $\sigma_m(s_{x,p,j}) := \mu(x) + ((p-2)c(x)+j-1)\max\{o,g\}$
8.                 $\pi_m(s_{x,p,j}) := 1$
9.                 $idle(1) := \sigma_m(s_{x,p,j}) + o$
10.                $\sigma_m(r_{x,p,j}) := \mu(x) + ((p-2)c(x)+j-1)\max\{o,g\} + L + o$
11.                $\pi_m(r_{x,p,j}) := p$
12.                $idle(p) := \sigma_m(r_{x,p,j}) + o$
13.         $\sigma_m(y_{p-1}) := idle(p)$
14.         $\pi_m(y_{p-1}) := p$
15.         $idle(p) := \sigma_m(y_{p-1}) + \mu(y_{p-1})$
16. for $i := m$ to $n$
17.     do assume $idle(p) = \min_{1\leq j\leq m} idle(j)$
18.         $\sigma_m(y_i) := idle(p)$
19.         $\pi_m(y_i) := p$
20.         $idle(p) := idle(p) + \mu(y_i)$
```

Figure 9.1. Algorithm SEND GRAPH SCHEDULING

Algorithm SEND GRAPH SCHEDULING. *Then (σ_m,π_m) is an m-processor schedule for (G,μ,c,L,o,g,P).*

Proof. x is executed at time 0 on processor 1. It is easy to see that all sinks of G are scheduled after x. For all processors $p \in \{2,\dots,m\}$ and all $j \in \{1,\dots,c(x)\}$, send operation $s_{x,p,j}$ is scheduled on processor 1 at time $\mu(x)+((p-2)c(x)+j-1)\max\{o,g\}$ and the corresponding receive operation $r_{x,p,j}$ on processor p at time $\mu(x)+((p-2)c(x)+j-1)\max\{o,g\}+o+L$. So the send operations are scheduled after x and there is a delay of $\max\{o,g\}$ time units between the starting times of two consecutive send operations or two consecutive receive operations on the same processor. Moreover, there is a delay of exactly L time units between the completion time of a send operation and the starting time of the corresponding receive operation. For all processors $p \in \{2,\dots,m\}$, a sink of G is scheduled on processor p at the completion time of the last receive operation on processor p. Clearly, the sinks of G are scheduled after all communication operations and no processor executes two tasks at the same time. So (σ_m,π_m) is a feasible schedule for (G,μ,c,L,o,g,P). Because every processor $p \in \{1,\dots,m\}$ executes at least one task of G, (σ_m,π_m) is an m-processor schedule for (G,μ,c,L,o,g,P). □

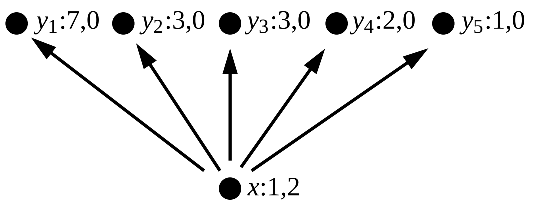

Figure 9.2. An instance $(G, \mu, c, 2, 1, 2, \infty)$

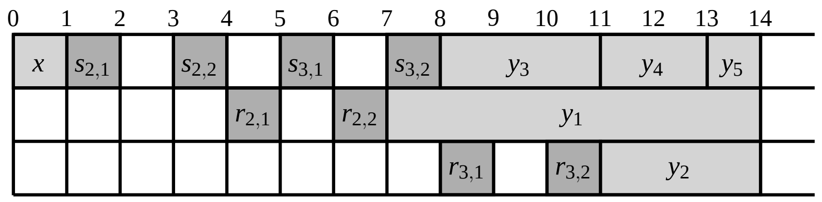

Figure 9.3. A 3-processor schedule constructed by Algorithm SEND GRAPH SCHEDULING

The time complexity of Algorithm SEND GRAPH SCHEDULING can be determined as follows. Consider an instance (G, μ, c, L, o, g, P), such that G is a send graph, and a positive integer $m \leq \min\{n, P\}$. Assigning a starting time and a processor to the source of G, $m-1$ sinks of G and the communication operations takes $O(n)$ time. If the processors are stored in a balanced search tree ordered by non-decreasing first idle time, then for each of the remaining $n-m+1$ sinks of G, $O(\log m)$ time is used to determine a starting time and a processor. Hence $O(n \log n)$ time is used to construct an m-processor schedule for (G, μ, c, L, o, g, P).

Lemma 9.2.3. *For all instances (G, μ, c, L, o, g, P), such that G is a send graph and all positive integers $m \leq \min\{n, P\}$, Algorithm* SEND GRAPH SCHEDULING *constructs a feasible m-processor schedule for (G, μ, c, L, o, g, P) in $O(n \log n)$ time.*

Now it will be proved that the m-processor schedules constructed by Algorithm SEND GRAPH SCHEDULING are at most twice as long as m-processor schedules of minimum length. Let G be a send graph with source x and sinks $y_1, \ldots, y_n$. Let $m \leq \min\{n, P\}$ be a positive integer. Let (σ_m, π_m) be the m-processor schedule for (G, μ, c, L, o, g, P) constructed by Algorithm SEND GRAPH SCHEDULING. Let ℓ_m be the length of (σ_m, π_m) and ℓ_m^* the length of a minimum-length m-processor schedule for (G, μ, c, L, o, g, P). In any m-processor schedule for (G, μ, c, L, o, g, P), $c(x)$ receive operations have to be executed on $m-1$ processors. Hence if $m \neq 1$, then every m-processor schedule for (G, μ, c, L, o, g, P) has length at least

$$\ell_m^* \quad \geq \quad \mu(x) + ((m-1)c(x) - 1)\max\{o, g\} + 2o + L.$$

Obviously, every 1-processor schedule for (G, μ, c, L, o, g, P) has length at least $\mu(x) + \sum_{i=1}^{n} \mu(y_i)$ and if $m = 1$, then Algorithm SEND GRAPH SCHEDULING constructs a schedule of this length. Hence we will assume that $m \geq 2$.

Assume y is a sink of G that finishes at time ℓ_m. Then y has been assigned a starting time and a processor in Lines 13 and 14 or in Lines 18 and 19 of Algorithm SEND GRAPH SCHEDULING.

Case 1. y has been assigned a starting time and a processor in Lines 13 and 14.
Assume $\pi(y) = p$. Then $p \neq 1$ and y is scheduled immediately after receive operation $r_{x,p,c(x)}$. This receive operation finishes at time $\mu(x) + ((p-1)c(x) - 1)\max\{o,g\} + 2o + L \leq \ell^*_m$. Obviously, $\mu(y) \leq \ell^*_m$. So

$$\begin{aligned}
\ell &= \sigma_m(y) + \mu(y) \\
&= (\mu(x) + ((p-1)c(x) - 1)\max\{o,g\} + 2o + L) + \mu(y) \\
&\leq 2\ell^*_m.
\end{aligned}$$

Case 2. y has been assigned a starting time and a processor in Lines 18 and 19.
Assume y is scheduled on processor p. If $p = 1$, then y is scheduled after x and the send operations. Otherwise, y is scheduled after sink y_{p-1}. If processor 1 is idle at a time t, such that $\mu(x) + ((m-1)c(x) - 1)\max\{o,g\} + o \leq t < \sigma_m(y)$, then y would have been scheduled at time t on processor 1. Similarly, if a processor $p' \in \{2,\ldots,m\}$ is idle at a time t, such that $\mu(x) + ((p'-1)c(x) - 1)\max\{o,g\} + 2o + L + \mu(y_{p'-1}) \leq t < \sigma_m(y)$, then y would have been scheduled at time t on processor p'. Hence processor 1 is busy from time $\mu(x) + ((m-1)c(x) - 1)\max\{o,g\} + o$ until time $\sigma_m(y)$ and each processor $p' \in \{2,\ldots,m\}$ from time $\mu(x) + ((p'-1)c(x) - 1)\max\{o,g\} + 2o + L + \mu(y_{p'-1})$ until time $\sigma_m(y)$.

No sink of G can be executed before a receive operation on a processor $p \in \{2,\ldots,m\}$. Because the communication operations are executed as early as possible, the idle periods in (σ_m, π_m) on processors $2,\ldots,m$ before the first sink cannot be avoided. Hence the only idle time in (σ_m, π_m) that can be avoided is the idle time between the send operations on processor 1. As a result,

$$\begin{aligned}
\ell^*_m &\geq \tfrac{1}{m}(m\sigma_m(y) + \mu(y) - ((m-1)c(x) - 1)(\max\{o,g\} - o)) \\
&= \sigma_m(y) + \tfrac{1}{m}\mu(y) - \tfrac{1}{m}((m-1)c(x) - 1)(\max\{o,g\} - o).
\end{aligned}$$

In addition, $\ell^*_m \geq \mu(y)$ and $\ell^*_m \geq \mu(x) + ((m-1)c(x) - 1)\max\{o,g\} + 2o + L$, since the last receive operation on the m^{th} processor cannot be completed before this time. Consequently,

$$\begin{aligned}
\ell_m &= \sigma_m(y) + \mu(y) \\
&\leq \ell^*_m + (1 - \tfrac{1}{m})\mu(y) + \tfrac{1}{m}(((m-1)c(x) - 1)(\max\{o,g\} - o)) \\
&\leq \ell^*_m + (1 - \tfrac{1}{m})\ell^*_m + \tfrac{1}{m}\ell^*_m \\
&= 2\ell^*_m.
\end{aligned}$$

Consequently, (σ_m, π_m) is at most twice as long as a minimum-length m-processor schedule for (G,μ,c,L,o,g,P).

For each positive integer $m \leq \min\{n,P\}$, Algorithm SEND GRAPH SCHEDULING is used to construct an m-processor schedule (σ_m, π_m) for (G,μ,c,L,o,g,P) of length ℓ_m. Assume (σ_k, π_k) is the shortest of these schedules. Let $\ell^* = \min_{1 \leq m \leq \min\{n,P\}} \ell^*_m$. Assume $\ell^* = \ell^*_p$. Then $\ell_k \leq \ell_p \leq 2\ell^*_p = 2\ell^*$. Hence we have proved the following result.

Theorem 9.2.4. *There is an algorithm with an $O(n^2 \log n)$ time complexity that constructs feasible schedules for instances (G,μ,c,L,o,g,P), such that G is a send graph, with length at most $2\ell^*$, where ℓ^* is the length of a minimum-length schedule for (G,μ,c,L,o,g,P).*

9.3 A polynomial special case

In Section 9.1, it was shown that constructing minimum-length schedules for send graphs is a strongly NP-hard optimisation problem. In Section 9.2, a 2-approximation algorithm was presented. In this section, it will be proved that if all task lengths are equal, then a minimum-length schedule can be constructed in polynomial time.

Let G be a send graph. Consider an instance (G,μ,c,L,o,g,P), such that $\mu(y) = \mu$ for all sources y of G. There is a minimum-length schedule for (G,μ,c,L,o,g,P) that uses at most $\min\{n,P\}$ processors. A minimum-length schedule for (G,μ,c,L,o,g,P) is constructed by computing the length of a minimum-length m-processor schedule for all positive integers $m \leq \min\{n,P\}$. These lengths are used to construct a minimum-length schedule for (G,μ,c,L,o,g,P).

Let G be a send graph. Consider an instance (G,μ,c,L,o,g,P), such that all sinks y of G have execution length $\mu(y) = \mu$. In an m-processor schedule for (G,μ,c,L,o,g,P), $c(x)$ receive operations have to be executed on $m-1$ processors and at least one sink is scheduled after the last receive operation on each of these processors. Hence $C_m = (m-1)c(x)$ send and receive operations have to be scheduled. Because the length of a minimum-length 1-processor schedule (G,μ,c,L,o,g,P) equals $\mu(x)+n\mu$, we will only consider the computation of the length of minimum-length m-processor schedules for (G,μ,c,L,o,g,P), where $m \geq 2$.

First we will consider an m-processor schedule $(\sigma_{m,0},\pi_{m,0})$ for (G,μ,c,L,o,g,P), in which the communication operations are executed as early as possible. We may assume that x is scheduled at time 0 on processor 1 and that send operations $s_{x,p,i}$ are executed before send operations $s_{x,p+1,j}$ for all processors $p \in \{2,\dots,m-1\}$ and all $i,j \in \{1,\dots,c(x)\}$. So we may assume that for all processors $p \in \{2,\dots,m\}$ and all $i \in \{1,\dots,c(x)\}$, send operation $s_{x,p,i}$ is scheduled at time $\mu(x)+((p-2)c(x)+i-1)\max\{o,g\}$ and receive operation $r_{x,p,i}$ at time $\mu(x)+((p-2)c(x)+i-1)\max\{o,g\}+L+o$. Hence the last send operation finishes at time

$$idle_{m,0}(1) \;=\; \mu(x)+((m-1)c(x)-1)\max\{o,g\}+o.$$

Since we may assume that the sinks of G are scheduled immediately after the last communication operation on processors $2,\dots,m$, the first sink on processor $p \in \{2,\dots,m\}$ finishes at time

$$idle_{m,0}(p) \;=\; \mu(x)+((p-1)c(x)-1)\max\{o,g\}+L+2o+\mu.$$

Now consider a minimum-length m-processor schedule (σ_m,π_m) for (G,μ,c,L,o,g,P). We may assume that the communication operations are scheduled in the same order as in $(\sigma_{m,0},\pi_{m,0})$. The sinks of G are scheduled after the communication operations or between the send operations. There is a delay of at least $\max\{o,g\}-o$ time units between the completion time of a send operation and the starting time of the next one. Let $\alpha(o,g) = \frac{\max\{o,g\}-o}{\mu}$. If there is a delay of $\max\{o,g\}$ time units between the starting times of two consecutive send operations, then at most

$\lfloor \alpha(o,g) \rfloor$ sinks can be scheduled between them. If at least $\lceil \alpha(o,g) \rceil$ sinks are scheduled between two consecutive send operations, then we may assume that processor 1 is not idle between these send operations. It is not difficult to see that if more than $\lceil \alpha(o,g) \rceil$ sinks are scheduled between two consecutive send operations, then one of them can be scheduled at a later time without increasing the schedule length. Hence we may assume that at most $\lceil \alpha(o,g) \rceil$ sinks are scheduled between two consecutive send operations. In addition, we may assume that no sink is scheduled before the first send operation on processor 1. So the total number of sinks scheduled between the send operations of processor 1 is at most $(C_m - 1)\lceil \alpha(o,g) \rceil$.

If $\lceil \alpha(o,g) \rceil$ sinks are scheduled between two consecutive send operations s_1 and s_2, then the starting times of these send operations differs exactly $o + \lceil \alpha(o,g) \rceil \mu$. So compared to the starting times of s_1 and s_2 in $(\sigma_{m,0}, \pi_{m,0})$, the starting time of s_2 is increased by

$$inc(o,g) \;=\; \lceil \alpha(o,g) \rceil \mu - (\max\{o,g\} - o).$$

Assume k sinks are scheduled between the send operations on processor 1. We may assume that $k \leq (C_m - 1)\lceil \alpha(o,g) \rceil$ and $k \leq n - m + 1$. In addition, because $\lfloor \alpha(o,g) \rfloor$ sinks can be scheduled between any pair of consecutive send operations without increasing the schedule length, we may assume that $k \geq \min\{n - m + 1, (C_m - 1)\lfloor \alpha(o,g) \rfloor\}$. If $k = k_0 + (C_m - 1)\lfloor \alpha(o,g) \rfloor$ for some non-negative integer k_0, then $\lceil \alpha(o,g) \rceil$ sinks have to be scheduled before the last k_0 send operations and $\lfloor \alpha(o,g) \rfloor$ before the other send operations except the first. If $k \leq (C_m - 1)\lfloor \alpha(o,g) \rfloor$, then at most $\lfloor \alpha(o,g) \rfloor$ sinks have to be scheduled between any pair of consecutive send operations on processor 1. Hence the last send operation on processor 1 finishes

$$inc_{m,k}(1) \;=\; \max\{0, k - (C_m - 1)\lfloor \alpha(o,g) \rfloor\}\, inc(o,g)$$

time units later than in $(\sigma_{m,0}, \pi_{m,0})$. Moreover, the completion times of the first sinks on processors $2, \ldots, m$ are increased compared to their completion times in $(\sigma_{m,0}, \pi_{m,0})$. The send operations $s_{x,p,i}$ are scheduled before send operations $s_{x,p+1,j}$ for all processors $p \in \{2, \ldots, m-1\}$ and all $i, j \in \{1, \ldots, c(x)\}$. Because $\lceil \alpha(o,g) \rceil$ sinks are scheduled between the last k_0 pairs of consecutive send operations on processor 1, the completion times of the first sink on the last $\lceil \frac{k_0}{c(x)} \rceil$ processors are increased. The completion time of the first sink on processor $p \in \{2, \ldots, m\}$ is increased by

$$inc_{m,k}(p) \;=\; \max\{0, k - (C_m - 1)\lfloor \alpha(o,g) \rfloor - (m-p)c(x)\}\, inc(o,g),$$

because $\lceil \alpha(o,g) \rceil$ sinks are scheduled before the last $k_0 = k - (C_m - 1)\lfloor \alpha(o,g) \rfloor$ send operations on processor 1 and the $(m-p)c(x)$ send operations scheduled on processor 1 after send operation $s_{x,p,c(x)}$ does not increase the starting time of the first sink on processor p.

Let $\ell_{m,k}$ be the minimum length of an m-processor schedule for (G, μ, c, L, o, g, P) in which k sinks are scheduled between the send operations on processor 1. Then $\ell_{m,k}$ is the length of (σ_m, π_m). So we may assume that the last send operation on processor 1 finishes at time

$$idle_{m,k}(1) \;=\; idle_{m,0}(1) + inc_{m,k}(1)$$

and that for all processors $p \in \{2,\ldots,m\}$, the completion time of the first sink on processor p equals

$$idle_{m,k}(p) \;=\; idle_{m,0}(p) + inc_{m,k}(p).$$

Note that $idle_{m,k}(m) \geq idle_{m,k}(p)$ for all processors $p \in \{1,\ldots,m\}$. Since the remaining $n-k$ sinks have to be scheduled after the send operations on processor 1 or after the first sink on a processor $p \in \{2,\ldots,m\}$, $\ell_{m,k}$ is the smallest integer ℓ, such that

$$\ell \;\geq\; idle_{m,k}(m) \qquad \text{and} \qquad \sum_{p=1}^{m} \left\lfloor \frac{\ell - idle_{m,k}(p)}{\mu} \right\rfloor \;\geq\; n-k.$$

Define

$$\ell_{m,k,0} \;=\; \min\{\ell \in \mathbb{Q} \mid \ell \geq idle_{m,k}(m) \wedge \sum_{p=1}^{m} \frac{\ell - idle_{m,k}(p)}{\mu} \geq n-k\}.$$

Then $\ell_{m,k,0} \leq \ell_{m,k} < \ell_{m,k,0} + \mu$. $\ell_{m,k,0}$ can be computed in $O(m)$ time:

$$\ell_{m,k,0} \;=\; \max\{idle_{m,k}(m), \frac{1}{m}((n-k)\mu + \sum_{p=1}^{m} idle_{m,k}(p))\}.$$

If $\ell_{m,k,0} = idle_{m,k}(m)$, then $\ell_{m,k,0} = \ell_{m,k} = idle_{m,k}(m)$. So we will assume that $\ell_{m,k,0} \neq idle_{m,k}(m)$. Then

$$\ell_{m,k} \;=\; \min\{\ell \in \mathbb{Z} \mid \sum_{p=1}^{m} \left\lfloor \frac{\ell - idle_{m,k}(p)}{\mu} \right\rfloor = \sum_{p=1}^{m} \frac{\ell_{m,k,0} - idle_{m,k}(p)}{\mu}\}.$$

Since $\ell_{m,k,0} \neq idle_{m,k}(m)$, $\sum_{p=1}^{m} \frac{\ell_{m,k,0} - idle_{m,k}(p)}{\mu} \in \mathbb{N}$. Define

$$D \;=\; \sum_{p=1}^{m} \frac{\ell_{m,k,0} - idle_{m,k}(p)}{\mu} - \sum_{p=1}^{m} \left\lfloor \frac{\ell_{m,k,0} - idle_{m,k}(p)}{\mu} \right\rfloor.$$

Note that $D \in \mathbb{N}$ and $D \leq m$. Assume that for all processors $p \in \{1,\ldots,m\}$,

$$\ell_{m,k,0} - idle_{m,k}(p) \;=\; q_p\mu + r_p,$$

such that $0 \leq r_p < \mu$. Then $\ell_{m,k} - \ell_{m,k,0}$ equals the smallest $d \in \mathbb{Q}$, such that $\ell_{m,k,0} + d \in \mathbb{Z}$ and for at least D processors p, $r_p + d \geq \mu$. Then $\ell_{m,k}$ can be computed as follows. Select the D^{th} element in the list of processors ordered by non-increasing r_p-values. Assume the D^{th} processor in this list is processor p_0. Then

$$\ell_{m,k} \;=\; \lceil \ell_{m,k,0} + \mu - r_{p_0} \rceil.$$

Selecting the D^{th} processor takes $O(m)$ time [18], so $\ell_{m,k}$ can be computed in $O(m)$ time.

Let $\ell^*_m = \min_k \ell_{m,k}$ and $\ell^* = \min_{1\leq m\leq \min\{n,P\}} \ell^*_m$. Then ℓ^*_m is the length of a minimum-length m-processor schedule for (G,μ,c,L,o,g,P) and ℓ^* the length of a minimum-length schedule for (G,μ,c,L,o,g,P). For each positive integer $m \leq \min\{n,P\}$, ℓ^*_m can be computed in $O(n^2)$ time, because $c(x)$ is bounded by a constant. So ℓ^* can computed in $O(n^3)$ time. If ℓ^* equals $\ell_{m,k}$, then m and k can be used to construct a minimum-length schedule in linear time. Hence we have proved the following result.

Theorem 9.3.1. *There is an algorithm with an $O(n^3)$ time complexity that constructs minimum-length schedules for instances (G,μ,c,L,o,g,P), such that G is a send graph and there is a positive integer μ, such that $\mu(y) = \mu$ for all sinks y of G.*

If $\max\{o,g\} - o$ is divisible by μ (for instance, if $g \leq o$ or if $\mu = 1$), then the length of a minimum-length schedule for (G,μ,c,L,o,g,P) can be computed more efficiently. Assume $\max\{o,g\} - o$ is divisible by μ. Then $\alpha(o,g) \in \mathbb{N}$. So we may assume that in a minimum-length m-processor schedule for (G,μ,c,L,o,g,P), exactly $k_m = \min\{n,(C_m - 1)\alpha(o,g)\}$ sinks of G are scheduled between the send operations on processor 1. Obviously, $inc_{m,k_m}(p) = 0$ for all processors $p \in \{1,\ldots,m\}$. So in a minimum-length m-processor schedule for (G,μ,c,L,o,g,P), the last send operation on processor 1 finishes at time

$$idle_{m,k_m}(1) \;=\; idle_{m,0}(1) \;=\; \mu(x) + ((m-1)c(x) - 1)\max\{o,g\} + o.$$

The completion time of the first sink on processor $p \in \{2,\ldots,m\}$ equals

$$idle_{m,k_m}(p) \;=\; idle_{m,0}(p) \;=\; \mu(x) + ((p-1)c(x) - 1)\max\{o,g\} + L + 2o + \mu.$$

Moreover, ℓ^*_m is the smallest integer ℓ, such that

$$\ell \;\geq\; idle_{m,k_m}(m) \qquad \text{and} \qquad \sum_{p=1}^{m} \left\lfloor \frac{\ell - idle_{m,k_m}(p)}{\mu} \right\rfloor \;\geq\; n - k_m.$$

ℓ^*_m can be computed in $O(n)$ time. Hence $\ell^* = \min_{1\leq m\leq \min\{n,P\}} \ell^*_m$ can be computed in $O(n^2)$ time. Given the number of processors m, such that $\ell^* = \ell^*_m$, a minimum-length schedule for (G,μ,c,L,o,g,P) can be constructed in linear time. So we have proved the following result.

Theorem 9.3.2. *There is an algorithm with an $O(n^2)$ time complexity that constructs minimum-length schedules for instances (G,μ,c,L,o,g,P), such that G is a send graph and there is a positive integer μ, such that $\mu(y) = \mu$ for all sinks y of G and $\max\{o,g\} - o$ is divisible by μ.*

9.4 Concluding remarks

In this chapter, two polynomial-time algorithms were presented that construct schedules for send graphs in the LogP model. Both algorithms use the knowledge of the order in which the send operations have to be scheduled in a minimum-length m-processor schedule. For more general classes of outforests, it is not obvious what the communication structure of minimum-length schedules looks like. Hence even for instances (G,L,o,g,P), such that G is an outtree of height three, it is not known whether a minimum-length schedule can be constructed in polynomial time.

Some results concerning scheduling in the UCT model can be generalised for scheduling in the LogP model. Because the UCT model can be viewed as the LogP model with parameters $L = 1$ and $o = g = 0$, the NP-completeness proof of Lenstra et al. [61] also shows that constructing minimum-length schedules for instances $(G, 1, 0, 0, P)$, such that G is an outtree, is an NP-hard optimisation problem.

Some algorithms for scheduling subject to communication delays can be generalised for scheduling in the LogP model. Chrétienne [12] presented an algorithm that constructs minimum-length schedules for outforests on an unrestricted number of processors subject to small communication delays. It is not difficult to transform the schedules constructed by this algorithm into feasible LogP schedules by introducing the communication operations. The resulting algorithm constructs minimum-length schedules for instances $(G, \mu, L, 0, g, \infty)$, such that G is a binary outforest and $L \leq \mu(u)$ for all tasks u of G, and for instances $(G, \mu, L, 0, 0, \infty)$, such that G is an outforest and $L \leq \mu(u)$ for all tasks u of G.

Munier [71] presented another algorithm that can be generalised for scheduling in the LogP model by introducing the communication operations. The generalised algorithm constructs schedules for instances $(G, \mu, c, L, 0, 0, \infty)$, such that G is an outforest, that are at most $2 - \frac{1}{L+1}$ times as long as a minimum-length schedule for $(G, \mu, c, L, 0, 0, \infty)$. Moreover, a more involved generalisation constructs schedules for instances $(G, \mu, L, o, g, \infty)$, such that G is a d-ary outforest, that are at most $2 + (d + 1)\max\{o, g\}$ times as long as a minimum-length schedule for $(G, \mu, L, o, g, \infty)$. Munier [71] also presented an algorithm that can be generalised to an algorithm that constructs schedules for instances $(G, c, L, 0, 0, P)$, such that G is an outforest. The length of the schedules constructed by this generalised algorithm are at most $1 + (1 + \frac{1}{P})(2 - \frac{1}{L+1})$ times as long as minimum-length schedules for $(G, c, L, 0, 0, P)$.

Another possible generalisation is scheduling with a different kind of communication. The communication in the schedules constructed by the algorithms presented in this chapter works as follows: if the result of a task u scheduled on processor p is needed by tasks scheduled on processors p_1 and p_2, then processor p must send the result of u to processors p_1 and p_2. However, the result of u could also be sent from processor p_1 to processor p_2. If such communication is allowed, then a schedule constructed by Algorithm SEND GRAPH SCHEDULING should start with a minimum-length schedule for a $c(x)$-item broadcast operation. If $c(x)$ equals one, then such a schedule can be constructed in polynomial time [20, 54]. So if broadcast communication is allowed and only one message is needed to send the result of the source to another processor, then schedules for send graphs that are at most twice as long as minimum-length schedules can be constructed in polynomial time. If $c(x)$ is at least two, then it is difficult to construct a minimum-length broadcast schedule. In that case, it is not easy to construct schedules that are at most twice as long as minimum-length schedules.

10 Receive graphs

In this chapter, we will consider the problem of scheduling receive graphs in the LogP model. Note that this problem is equivalent to the problem of scheduling send graphs under an independent data semantics. Like in Chapter 9, the structure of minimum-length schedules will be used to construct good schedules for receive graphs.

In Section 10.1, it is shown that constructing minimum-length schedules for receive graphs on an unrestricted number of processors is a strongly NP-hard optimisation problem. This is proved using a polynomial reduction similar to the one presented in the proof of Lemma 9.1.1.

In Section 10.2, two polynomial-time approximation algorithms are presented. Both algorithms assume that g does not exceed o. The first approximation algorithm constructs schedules for receive graphs on an unrestricted number of processors that are at most three times as long as a minimum-length schedule on an unrestricted number of processors. In Section 10.2.2, it is shown that a schedule on P processors that is at most $3+\frac{1}{k+1}$ times as long as a minimum-length schedule on P processors can be constructed in polynomial time for all constant $k \in \mathbb{Z}^+$.

In Section 10.3, it is shown that if all task lengths are equal, then a minimum-length schedule for a receive graph on an unrestricted number of processors can be constructed in polynomial time. This is an improvement over the result of Kort and Trystram [55] who proved that a minimum-length schedule for a receive graph on an unrestricted number of processors can be constructed in polynomial time if g does not exceed o and all sources have the same execution length.

10.1 An NP-completeness result

In Chapter 9, it was proved that constructing minimum-length schedules for send graphs on an unrestricted number of processors is a strongly NP-hard optimisation problem. This was proved using the polynomial reduction from 3PARTITION presented in the proof of Lemma 9.1.1. Let (G,μ,L,o,g,∞) be the instance constructed by this reduction for an instance of 3PARTITION. The send graph G contains $m+2$ large tasks that must be scheduled on different processors. These are the only tasks that are scheduled after the communication operations in a minimum-length schedule for (G,μ,L,o,g,∞).

By reversing all arcs in send graph G, we obtain a receive graph G'. In a minimum-length schedule for (G',μ,L,o,g,∞), the large tasks are the only ones that are scheduled before the communication operations. Hence the reversal of the minimum-length schedule for the send graph can be viewed as a minimum-length schedule for the receive graph. Thus a similar reduction as the one presented in the proof Lemma 9.1.1 can be used to prove that constructing minimum-length schedules for receive graphs on an unrestricted number of processors is a strongly NP-hard optimisation problem.

Theorem 10.1.1. *Constructing minimum length schedules for instances* (G,μ,L,o,g,∞)*, such that G is a receive graph, is a strongly* NP*-hard optimisation problem.*

Theorem 10.1.1 shows that it is unlikely that a minimum-length schedule for an instance

(G,μ,c,L,o,g,∞), such that G is a receive graph and $g > o$, can be constructed in polynomial time. It is unknown whether minimum-length schedules on an unrestricted number of processors can be constructed in polynomial time if g does not exceed o. Kort and Trystram [55] proved that if $g \leq o$ and all tasks have the same length, then a minimum-length schedule for a receive graph can be constructed in polynomial time.

10.2 Two approximation algorithms

In this section, two polynomial-time approximation algorithms for scheduling receive graphs in the LogP model are presented. The first is presented in Section 10.2.1. It constructs schedules for receive graphs on an unrestricted number of processors. The length of these schedules are at most three times as long as a minimum-length schedule on an unrestricted number of processors. The algorithm presented in Section 10.2.2 constructs schedules for receive graphs on a restricted number of processors. It is shown that for each constant $k \in \mathbb{Z}^+$, a schedule on P processors that is at most $3+\frac{1}{k+1}$ times as long as a minimum-length schedule on P processors can be constructed in polynomial time.

Both algorithms divide the set of sources of a receive graph into two sets. Let G be a receive graph. Consider an instance (G,μ,c,L,o,g,P). A source y of G is called *communication intensive* if $\mu(y) \leq c(y)o$. Otherwise, it is called *computation intensive*. Hence a source y of G is communication intensive if the total duration of the send operations needed to send the result of y to another processor exceeds the execution length of y. The sets of communication-intensive and computation-intensive sources will be used to compute lower bounds on the length of minimum-length schedules for receive graphs.

10.2.1 An unrestricted number of processors

In this section, an approximation algorithm for scheduling receive graphs on an unrestricted number of processors is presented. For this algorithm, we will assume that g does not exceed o. The algorithm constructs schedules for receive graphs on an unrestricted number of processors that are at most three times as long as a minimum-length schedule on an unrestricted number of processors. The algorithm is similar to the 3-approximation algorithm of Hollerman et al. [46] for scheduling send and receive graphs in a model of parallel computation that resembles the LogP model.

We start by proving some properties of minimum-length schedules for receive graphs on an unrestricted number of processors. The next lemma shows that if a source of a receive graph G is not scheduled on the same processor as the sink of G, then the receive operations corresponding to this source may be scheduled after the sources of G that are scheduled on the same processor as the sink of G. This result is not true if g exceeds o. If g exceeds o, then some sources of G may have to be scheduled between the receive operations in a minimum-length schedule for G on an unrestricted number of processors.

Lemma 10.2.1. *Let G be a receive graph with sink x and sources $y_1,\dots,y_n$. If $g \leq o$, then there is a minimum-length schedule (σ,π) for (G,μ,c,L,o,g,∞), such that for all sources y_i and y_j of G, if $\pi(y_i)=\pi(x)$ and $\pi(y_j)\neq\pi(x)$, then $\sigma(y_i) < \sigma(r_{y_j,\pi(x),k})$ for all $k \leq c(y_j)$.*

Proof. Assume $g \leq o$. Let (σ,π) be a minimum-length schedule for (G,μ,c,L,o,g,∞). We may assume that x is scheduled on processor 1. Let y_i and y_j be two sources of G. Assume $\pi(y_i) = 1$ and $\pi(y_j) \neq 1$. Assume $\sigma(y_i) > \sigma(r_{y_j,1,k})$ for some $k \leq c(y_j)$. We may assume that $\sigma(y_i) = \sigma(r_{y_j,1,k}) + o$. Then y_i can be scheduled at time $\sigma(r_{y_j,1,k})$, $r_{y_j,1,k}$ at time $\sigma(r_{y_j,1,k}) + \mu(y_i)$ and $s_{y_j,1,k}$ at time $\sigma(r_{y_j,1,k}) + \mu(y_i) - o - L$ without violating the feasibility of (σ,π) or increasing its length. By repeating this step, a minimum-length schedule (σ,π) for (G,μ,c,L,o,g,∞) is constructed in which no source of G is scheduled after a receive operation on processor $\pi(x)$. $\square$

Lemma 10.2.2 proves that in a minimum-length schedule for a a receive graph G on an unrestricted number of processors, all processors that do not execute the sink of G need to execute at most one task. Unlike Lemma 10.2.1, this result is true for scheduling with arbitrary o and g.

Lemma 10.2.2. *Let G be a receive graph with sink x and sources $y_1,\ldots,y_n$. There is a minimum-length schedule (σ,π) for (G,μ,c,L,o,g,∞), such that for all processors $p \neq \pi(x)$, at most one source of G is executed on processor p.*

Proof. Let (σ,π) be a minimum-length schedule for (G,μ,c,L,o,g,∞). We may assume that x is scheduled on processor 1. Assume two sources y_i and y_j of G are scheduled on processor $p \neq 1$. Let processor p' be a processor on which no task of G is executed. Then y_j can be scheduled on processor p' at time $\sigma(y_j)$ and send operation $s_{y_j,1,k}$ on the same processor at time $\sigma(s_{y_j,1,k})$ for all $k \leq c(y_j)$. This does not violate the feasibility of (σ,π) nor does it increase its length. By repeating this step, we obtain a minimum-length schedule (σ,π) for (G,μ,c,L,o,g,∞), such that at most one source of G is executed on processor p for all processors $p \neq \pi(x)$. $\square$

The following lemma shows that there is a minimum-length schedule for a receive graph G on an unrestricted number of processors, in which the receive operations corresponding to the sources of G with a small execution length are scheduled before the receive operations corresponding to the sources of G with a large execution length.

Lemma 10.2.3. *Let G be a receive graph with sink x and sources $y_1,\ldots,y_n$. There is a minimum-length schedule (σ,π) for (G,μ,c,L,o,g,∞), such that for all sources y_i and y_j of G, if $\mu(y_i) < \mu(y_j)$ and $\pi(y_i),\pi(y_j) \neq \pi(x)$, then $\sigma(r_{y_i,\pi(x),k_i}) < \sigma(r_{y_j,\pi(x),k_j})$ for all $k_i \leq c(y_i)$ and $k_j \leq c(y_j)$.*

Proof. Let (σ,π) be a minimum-length schedule for (G,μ,c,L,o,g,∞). We may assume that x is scheduled on processor 1. From Lemma 10.2.2, we may assume that all processors $p \neq 1$ execute at most one task of G. Let y_i and y_j be two sources of G that are not scheduled on processor 1. Assume $\mu(y_i) < \mu(y_j)$ and $\sigma(y_i) = \sigma(y_j) = 0$. Receive operations $r_{y_i,1,k}$ can start at time $\mu(y_i) + L + o$ on processor 1, receive operations $r_{y_j,1,k}$ at time $\mu(y_j) + L + o$. Assume $\sigma(r_{y_j,1,k_j}) < \sigma(r_{y_i,1,k_i})$ for some $k_i \leq c(y_i)$ and $k_j \leq c(y_j)$. Then $r_{y_j,1,k_j}$ can be scheduled at time $\sigma(r_{y_i,1,k_i})$ and $r_{y_i,1,k_i}$ at time $\sigma(r_{y_j,1,k_j})$. In addition, send operations $s_{y_i,1,k_i}$ and $s_{y_j,1,k_j}$ can be scheduled $L+o$ time units before receive operations $r_{y_i,1,k_i}$ and $r_{y_j,1,k_j}$, respectively. This does not violate the feasibility of (σ,π) or increase its length, because all receive operations have length o. By repeating this step, we obtain a minimum-length schedule (σ,π) for (G,μ,c,L,o,g,P), such that for all sources y_i and y_j of G, if $\pi(y_i),\pi(y_j) \neq \pi(x)$ and $\mu(y_i) < \mu(y_j)$, then receive operation $r_{y_i,\pi(x),k_i}$ is scheduled before receive operation $r_{y_j,\pi(x),k_j}$ for all $k_i \leq c(y_i)$ and $k_j \leq c(y_j)$. $\square$

Lemma 10.2.4 shows that in a minimum-length schedule for a receive graph G on an unrestricted number of processors, all communication-intensive sources of G may be scheduled on the same processor as the sink of G.

Lemma 10.2.4. *Let G be a receive graph with sink x and sources $y_1,\dots,y_n$. If $g \leq o$, then there is a minimum-length schedule (σ,π) for (G,μ,c,L,o,g,∞), such that for all sources y_i of G, if $\mu(y_i) \leq c(y_i)o$, then $\pi(y_i) = \pi(x)$.*

Proof. Assume $g \leq o$. Let (σ,π) be a minimum-length schedule for (G,μ,c,L,o,g,∞). We may assume that x is executed on processor 1. From Lemmas 10.2.1 and 10.2.3, we may assume that the sources on processor 1 are scheduled before the receive operations of the sources scheduled on another processor and that for each source y_i of G, if y_i is not scheduled on processor 1, then the receive operations $r_{y_i,1,j}$ are scheduled on processor 1 without interruption. Assume y_i is a source of G, such that $\mu(y_i) \leq c(y_i)o$ and $\pi(y_i) \neq 1$. We may assume that $\sigma(r_{y_i,1,1}) < \cdots < \sigma(r_{y_i,1,c(y_i)})$. Then $r_{y_i,1,c(y_i)}$ finishes at time $\sigma(r_{y_i,1,1}) + c(y_i)o \geq \sigma(r_{y_i,1,1}) + \mu(y_i)$. Then y_i can be scheduled at time $\sigma(r_{y_i,1,1})$ on processor 1 without increasing the length of (σ,π) or violating its feasibility. By repeating this step, we obtain a minimum-length schedule (σ,π) for (G,μ,c,L,o,g,∞), such that for all sources y_i of G, if $\mu(y_i) \leq c(y_i)o$, then y_i is scheduled on processor $\pi(x)$. □

The next lemma proves that it can be determined in polynomial time whether the schedule for a receive graph G in which all tasks of G are scheduled on the same processor is a minimum-length schedule for G on an unrestricted number of processors.

Lemma 10.2.5. *Let G be a receive graph with sink x and sources $y_1,\dots,y_n$. If $g \leq o$, then a schedule for (G,μ,c,L,o,g,∞) of length $\mu(x) + \sum_{i=1}^{n} \mu(y_i)$ is a minimum-length schedule for (G,μ,c,L,o,g,∞) if and only if for all sources y_i of G, if $\mu(y_i) > c(y_i)o$, then $\sum_{j=1}^{n} \mu(y_j) \leq (c(y_i)+1)o + L + \mu(y_i)$.*

Proof. Assume $g \leq o$. We will prove that a minimum-length schedule for (G,μ,c,L,o,g,P) has length $\mu(x) + \sum_{i=1}^{n} \mu(y_i)$ if and only if for all computation-intensive sources y_i of G, $\sum_{j=1}^{n} \mu(y_j) \leq (c(y_i)+1)o + L + \mu(y_i)$.

($\Rightarrow$) Assume a minimum-length schedule for (G,μ,c,L,o,g,∞) has length $\mu(x) + \sum_{i=1}^{n} \mu(y_i)$. Let y_i be a source of G. Assume $\mu(y_i) > c(y_i)o$. It will be proved by contradiction that $\sum_{j=1}^{n} \mu(y_j) \leq (c(y_i)+1)o + L + \mu(y_i)$. Suppose $\sum_{j=1}^{n} \mu(y_j) > (c(y_i)+1)o + L + \mu(y_i)$. Then construct a schedule (σ,π) for (G,μ,c,L,o,g,∞) as follows. Tasks $y_1,\dots,y_{i-1},y_{i+1},\dots,y_n$ are scheduled without interruption on processor 1 from time 0 onward. y_i is scheduled on processor 2 at time 0. For all $k \leq c(y_i)$, receive operation $r_{y_i,1,k}$ is scheduled on processor 1 at time $\max\{\sum_{j\neq i} \mu(y_j), \mu(y_i) + o + L\} + (k-1)o$. For all $k \leq c(y_i)$, send operation $s_{y_i,1,k}$ is scheduled on processor 2 at time $\sigma(r_{y_i,1,k}) - L - o$. x is scheduled immediately after $r_{y_i,1,c(y_i)}$ on processor 1. Then (σ,π) is a feasible schedule for (G,μ,c,L,o,g,∞) of length

$$\mu(x) + \max\{\mu(y_i) + (c(y_i)+1)o + L, \sum_{j\neq i} \mu(y_j) + c(y_i)o\} \quad < \quad \mu(x) + \sum_{j=1}^{n} \mu(y_j).$$

Contradiction.

($\Leftarrow$) Assume for all sources y_i of G, if $\mu(y_i) > c(y_i)o$, then $\sum_{j=1}^{n} \mu(y_j) \leq (c(y_i)+1)o + L + \mu(y_i)$. Let (σ,π) be a minimum-length schedule for (G,μ,c,L,o,g,∞). Since there is a schedule for (G,μ,c,L,o,g,∞) of length $\mu(x) + \sum_{i=1}^{n} \mu(y_i)$, the length of (σ,π) is at most $\mu(x) + \sum_{i=1}^{n} \mu(y_i)$. It is proved by contradiction that (σ,π) has length $\mu(x) + \sum_{i=1}^{n} \mu(y_i)$. Suppose the length of (σ,π) is less than $\mu(x) + \sum_{i=1}^{n} \mu(y_i)$. Then at least one source y_i of G is not scheduled on the same processor as x. From Lemma 10.2.4, we may assume that all communication-intensive sources y_i of G are scheduled on processor $\pi(x)$. Hence we may assume that $\mu(y_i) > c(y_i)o$. So (σ,π) has length at least

$$\mu(y_i) + (c(y_i)+1)o + L + \mu(x) \quad \geq \quad \mu(x) + \sum_{i=1}^{n} \mu(y_i).$$

Contradiction.

□

The properties of minimum-length schedules proved in the preceding lemmas will be used to compute upper bounds on the length of the schedules constructed by Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING. Consider an instance (G,μ,c,L,o,g,∞), such that G is a receive graph and $g \leq o$. Assume G has sink x and sources $y_1,\ldots,y_n$. Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING constructs a schedule (σ,π) for (G,μ,c,L,o,g,∞) as follows. The communication-intensive sources of G and its sink x are scheduled on processor 1. All computation-intensive sources of G are scheduled on a separate processor. The receive operations are scheduled after the sources on processor 1, such that if $\mu(y_i) < \mu(y_j)$ and y_i and y_j are not scheduled on processor 1, then receive operations $r_{y_i,1,k_i}$ are executed before receive operations $r_{y_j,1,k_j}$. Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING is presented in Figure 10.1.

Example 10.2.6. Consider the instance $(G,\mu,c,1,2,2,\infty)$ shown in Figure 10.2. Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING constructs a schedule for $(G,\mu,c,1,2,2,\infty)$ as follows. The set $Y_1 = \{y_1,y_2,y_3\}$ contains the communication-intensive sources of G. These tasks are scheduled on processor 1 from time 0 onward. The other tasks are scheduled on a separate processor. Since the execution length of y_4 is smaller than that of y_5, the communication operations of y_4 are executed before those of y_5. Sink x is scheduled on processor 1 after the last receive operation. So Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING constructs the schedule for $(G,\mu,c,1,2,2,\infty)$ shown in Figure 10.3.

Now we will prove that Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING correctly constructs feasible schedules for receive graphs on an unrestricted number of processors.

Lemma 10.2.7. *Let G be a receive graph. Let (σ,π) be the schedule for (G,μ,c,L,o,g,∞) constructed by Algorithm* UNRESTRICTED RECEIVE GRAPH SCHEDULING. *If $g \leq o$, then (σ,π) is a feasible schedule for (G,μ,c,L,o,g,∞).*

```
Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING
Input. An instance $(G,\mu,c,L,o,g,\infty)$, such that $g \leq o$ and $G$ is a receive graph with sink $x$ and
       sources $y_1,\dots,y_n$, such that $\mu(y_1) \leq \cdots \leq \mu(y_n)$.
Output. A feasible schedule $(\sigma,\pi)$ for $(G,\mu,c,L,o,g,\infty)$.
1.  $idle(1) := 0$
2.  $p := 1$
3.  for $i := 1$ to $n$
4.      do if $\mu(y_i) \leq c(y_i)o$
5.          then $\sigma(y_i) := idle(1)$
6.               $\pi(y_i) := 1$
7.               $idle(1) := idle(1) + \mu(y_i)$
8.          else $p := p+1$
9.               $\sigma(y_i) := 0$
10.              $\pi(y_i) := p$
11. for $i := 2$ to $p$
12.     do let $y$ be the sink of $G$ executed on processor $i$
13.        for $j := 1$ to $c(y)$
14.            do $\sigma(r_{y,1,j}) := \max\{idle(1), \mu(y)+L+jo\}$
15.               $\pi(r_{y,1,j}) := 1$
16.               $\sigma(s_{y,1,j}) := \sigma(r_{y,1,j}) - L - o$
17.               $\pi(r_{y,1,j}) := i$
18.               $idle(1) := \sigma(r_{y,1,j}) + o$
19. $\sigma(x) := idle(1)$
20. $\pi(x) := 1$
```

Figure 10.1. Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING

Proof. Assume $g \leq o$. Let (σ,π) be the schedule for (G,μ,c,L,o,g,∞) constructed by Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING. Obviously, processor 1 does not execute two tasks or communication operations at the same time. For all sinks y of G, such that $\pi(y) \neq 1$, and all $j \in \{1,\dots,c(y)\}$, send operation $s_{y,i,j}$ starts after the completion time of y. Because all processors $p \neq 1$ execute at most one task, no processor executes two tasks or communication operations at the same time. Since $g \leq o$ and no two communication operations are executed on the same processor at the same time, there is a delay of at least g time units between two consecutive send or receive operations on the same processor. In addition, the receive operations are scheduled $L+o$ time units after the corresponding send operations. So (σ,π) is a feasible schedule for (G,μ,c,L,o,g,∞). □

The time complexity of Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING can be determined as follows. Let G be a receive graph. Sorting the sources of G by non-decreasing execution length takes $O(n\log n)$ time. Clearly, assigning a starting time and a processor to the tasks of G and the communication operations takes $O(n)$ time. It is easy to see that the remaining operations take $O(n)$ time.

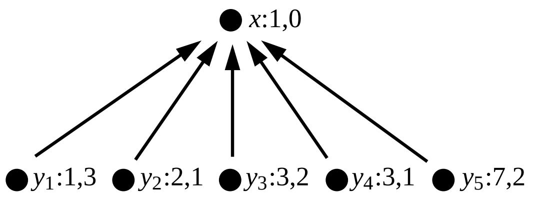

Figure 10.2. An instance $(G,\mu,c,1,2,2,\infty)$

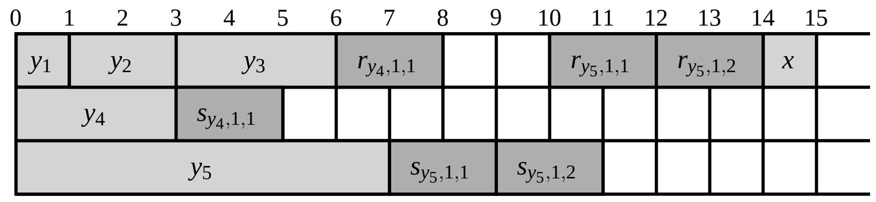

Figure 10.3. A feasible schedule for $(G,\mu,c,1,2,2,\infty)$

Lemma 10.2.8. *For all instances (G,μ,c,L,o,g,∞), such that G is a receive graph and $g \leq o$, Algorithm* UNRESTRICTED RECEIVE GRAPH SCHEDULING *constructs a feasible schedule for (G,μ,c,L,o,g,∞) in $O(n\log n)$ time.*

Now we will prove that Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING is a 3-approximation algorithm. Let G be a receive graph with sink x and sources $y_1,\dots,y_n$, such that $\mu(y_1) \leq \cdots \leq \mu(y_n)$. Assume $g \leq o$. Let (σ,π) be the schedule for (G,μ,c,L,o,g,∞) constructed by Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING. Let $y_{i_1},\dots,y_{i_k}$ be the sources of G that are not scheduled on processor 1. Then $\mu(y_{i_j}) > c(y_{i_j})o$ for all $j \leq k$. We will assume that $i_1 \leq \cdots \leq i_k$. Let $y_{i_{k+1}},\dots,y_{i_n}$ be the sources of G scheduled on processor 1, such that $i_{k+1} \leq \cdots \leq i_n$.

Then x is scheduled immediately after receive operation $r_{y_{i_k},1,c(y_{i_k})}$. If processor 1 is not idle before time $\sigma(x)$, then (σ,π) has length

$$\sum_{j=k+1}^{n} \mu(y_{i_j}) + \sum_{j=1}^{k} c(y_{i_j})o + \mu(x).$$

Otherwise, there is a $j \in \{1,\dots,k\}$, such that receive operation $r_{y_{i_j},1,1}$ starts at time $\mu(y_{i_j}) + L + o$ and processor 1 executes receive operations $r_{y_{i_l},1,i}$, such that $l \geq j$ and $i \leq c(y_{i_l})$, without interruption from time $\mu(y_{i_l}) + L + o$ until time $\sigma(x)$. In this case, (σ,π) has length

$$\mu(y_{i_j}) + \sum_{l=j}^{k} c(y_{i_l})o + L + o + \mu(x).$$

Let ℓ the length of (σ, π). Then

$$\ell \leq \mu(x) + \max\{\sum_{j=1}^{k} c(y_{i_j})o + \sum_{j=k+1}^{n} \mu(y_{i_j}), \max_{1 \leq j \leq k}(\mu(y_{i_j}) + \sum_{l=j}^{k} c(y_{i_l})o + L + o)\}.$$

Let ℓ^* be the length of a minimum-length schedule for $(G, \mu, c, L, o, g, \infty)$. Clearly, $\ell^* \geq \mu(x) + \mu(y)$ for all sources y of G. In addition, for each source y_i of G, either y_i itself or $c(y_i)$ receive operations are scheduled on the same processor as x in a feasible schedule for $(G, \mu, c, L, o, g, \infty)$. Hence

$$\ell^* \geq \mu(x) + \sum_{i=1}^{n} \min\{\mu(y_i), c(y_i)o\}.$$

Consequently,

$$\begin{aligned} \ell &\leq \mu(x) + \max\{\textstyle\sum_{j=1}^{k} c(y_{i_j})o + \sum_{j=k+1}^{n} \mu(y_{i_j}), \max_{1 \leq j \leq k}(\mu(y_{i_j}) + \sum_{l=j}^{k} c(y_{i_l})o + L + o)\} \\ &\leq \max\{\ell^*, \ell^* + \ell^* + L + o\} \\ &= 2\ell^* + L + o. \end{aligned}$$

If the length of a minimum-length schedule for $(G, \mu, c, L, o, g, \infty)$ equals $\mu(x) + \sum_{j=1}^{n} \mu(y_j)$, then this can be checked in linear time using Lemma 10.2.5. In that case, we can construct a minimum-length schedule for $(G, \mu, c, L, o, g, \infty)$ by scheduling all tasks on one processor. Otherwise, in a minimum-length schedule for $(G, \mu, c, L, o, g, \infty)$, there is a sink that is scheduled on a different processor than x. Hence $\ell^* \geq \mu(x) + 2o + L$ and $\ell \leq 2\ell^* + L + o \leq 3\ell^*$. Hence we have proved the following result.

Theorem 10.2.9. *There is an algorithm with an $O(n \log n)$ time complexity that constructs feasible schedules for instances $(G, \mu, c, L, o, g, \infty)$, such that G is a receive graph and $g \leq o$, with length at most $3\ell^*$, where ℓ^* is the length of a minimum-length schedule for $(G, \mu, c, L, o, g, \infty)$.*

Note that if L and o are bounded by a constant, then Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING is an approximation algorithm with asymptotic approximation ratio two.

10.2.2 A restricted number of processors

In this section, an approximation algorithm is presented that constructs schedules for receive graphs on a restricted number of processors. Consider an instance (G, μ, c, L, o, g, P), such that G is a receive graph, $g \leq o$ and $P \neq \infty$. Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING constructs a schedule for (G, μ, c, L, o, g, P). Like Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING, the communication-intensive sources of G will be scheduled on the same processor as its sink, the other sources of G can be scheduled on any processor. A schedule for (G, μ, c, L, o, g, P) is constructed by extending a feasible schedule for the subgraph of G induced by the set of computation-intensive sources of G. Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING is presented in Figure 10.4.

```
Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING
Input. An instance $(G,\mu,c,L,o,g,P)$, such that $g \leq o$, $P \neq \infty$ and $G$ is a receive graph with sink
       $x$ and sources $y_1,\dots,y_n$.
Output. A feasible schedule $(\sigma,\pi)$ for $(G,\mu,c,L,o,g,P)$.
 1.  $Y_1 := \{y_i \mid \mu(y_i) \leq c(y_i)o\}$
 2.  $Y_2 := \{y_i \mid \mu(y_i) > c(y_i)o\}$
 3.  let $(\sigma,\pi)$ be a feasible schedule for $(G[Y_2],\mu,c,L,o,g,P)$
 4.  for $p := 1$ to $P$
 5.      do $idle(p) := \max\{\sigma(y)+\mu(y) \mid y \in Y_2 \wedge \pi(y) = p\}$
 6.         $Y_{2,p} := \{y \in Y_2 \mid \pi(y) = p\}$
 7.  assume $idle(1) \leq \cdots \leq idle(P)$
 8.  for $y \in Y_1$
 9.      do $\sigma(y) := idle(1)$
10.         $idle(1) := idle(1) + \mu(y)$
11.  for $p := 2$ to $P$
12.      do for $y \in Y_{2,p}$
13.             do for $j := 1$ to $c(y)$
14.                    do $\sigma(r_{y,1,j}) := \max\{idle(1), idle(p) + L + jo\}$
15.                       $\pi(r_{y,1,j}) := 1$
16.                       $\sigma(s_{y,1,j}) := \sigma(r_{y,1,j}) - L - o$
17.                       $\pi(s_{y,1,j}) := p$
18.                       $idle(1) := \sigma(r_{y,1,j}) + o$
19.                       $idle(p) := \sigma(s_{y,1,j}) + o$
20.  $\sigma(x) := idle(1)$
21.  $\pi(x) := 1$
```

Figure 10.4. Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING

Example 10.2.10. Consider the instance $(G,\mu,c,1,2,2,2)$ shown in Figure 10.5. Apart from the number of processors, this instance equals the one shown in Figure 10.2. The set $Y_1 = \{y_1,y_2,y_3\}$ contains the communication-intensive sources of G. These tasks are scheduled on processor 1. Assume Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING starts with a schedule in which y_4 starts at time 0 on processor 1 and y_5 at time 0 on processor 2. Then y_1, y_2 and y_3 are scheduled on the same processor as y_4, because the execution length of y_4 is smaller than that of y_5. Receive operations $r_{y_5,1,i}$ are scheduled after y_3 on processor 2. x is executed after the last receive operation on processor 1. So Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING constructs the schedule for $(G,\mu,c,1,2,2,2)$ shown in Figure 10.6.

Now we will prove that Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING correctly constructs feasible schedules for receive graphs on a restricted number of processors.

Lemma 10.2.11. *Let G be a receive graph. Let (σ,π) be the schedule for (G,μ,c,L,o,g,P) constructed by Algorithm* RESTRICTED RECEIVE GRAPH SCHEDULING. *If $g \leq o$, then (σ,π) is a feasible schedule for (G,μ,c,L,o,g,P).*

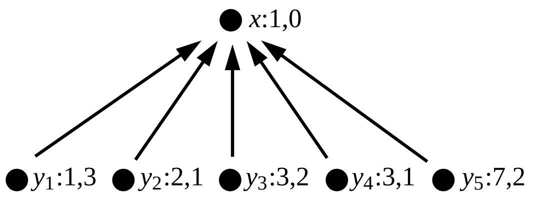

Figure 10.5. An instance $(G,\mu,c,1,2,2,2)$

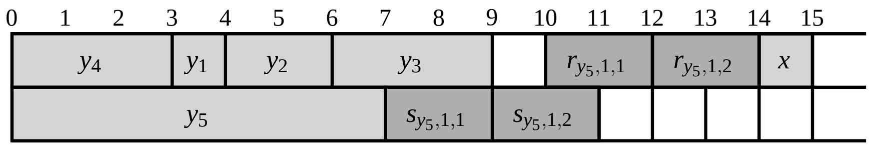

Figure 10.6. A feasible schedule for $(G,\mu,c,1,2,2,2)$

Proof. Assume $g \le o$ and G has sink x and sources $y_1,\ldots,y_n$. Define $Y_1 = \{y_i \mid \mu(y_i) \le c(y_i)o\}$ and $Y_2 = \{y_i \mid \mu(y_i) > c(y_i)o\}$. Let (σ_0,π_0) be a feasible schedule for $(G[Y_2],\mu,c,L,o,g,P)$. Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING extends (σ_0,π_0) to a schedule (σ,π) for (G,μ,c,L,o,g,P). It is obvious that no processor executes two tasks at the same time. It is easy to see that there is a delay of exactly L time units between the completion time of a send operation and the starting time of the corresponding receive operation. Because $g \le o$ and all receive operations are scheduled on processor 1, there is a delay of at least g time units between a pair of consecutive send and receive operations on the same processor. So (σ,π) is a feasible schedule for (G,μ,c,L,o,g,P). □

The time complexity of Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING can be determined as follows. Let G be a receive graph with sink x and sources $y_1,\ldots,y_n$. Let $Y_1 = \{y_i \mid \mu(y_i) \le c(y_i)o\}$ and $Y_2 = \{y_i \mid \mu(y_i) > c(y_i)o\}$. Y_1 and Y_2 can be computed in $O(n)$ time. Let (σ_0,π_0) be a feasible schedule for $(G[Y_2],\mu,c,L,o,g,P)$. Sorting the processors by non-decreasing maximum completion time takes $O(P\log P)$ time. Assigning a starting time and a processor to every task of Y_1 takes $O(n)$ time. It is easy to see that the starting times and processors for the communication operations can be assigned in linear time as well. So Algorithm UNRESTRICTED RECEIVE GRAPH SCHEDULING uses $O(n\log n)$ time apart from the time needed to construct (σ_0,π_0).

Lemma 10.2.12. *For all instances (G,μ,c,L,o,g,P), such that G is a receive graph and $g \le o$, if a feasible schedule for n incomparable tasks can be constructed in $O(T(n))$ time, then Algorithm* RESTRICTED RECEIVE GRAPH SCHEDULING *constructs a feasible schedule for (G,μ,c,L,o,g,P) in $O(T(n)+n\log n)$ time.*

Consider an instance (G,μ,c,L,o,g,P), such that $g \le o$ and G is a receive graph with sink x and sources $y_1,\ldots,y_n$. Define $Y_1 = \{y_i \mid \mu(y_i) \le c(y_i)o\}$ and $Y_2 = \{y_i \mid \mu(y_i) > c(y_i)o\}$. Let

(σ_0,π_0) be a feasible schedule for $(G[Y_2],\mu,c,L,o,g,P)$. Assume Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING extends (σ_0,π_0) to a feasible schedule (σ,π) for (G,μ,c,L,o,g,P). Let ℓ^* be the length of a minimum-length schedule for (G,μ,c,L,o,g,P) and ℓ the length of (σ,π). Because any schedule on a restricted number of processors can be viewed as a schedule on an unrestricted number of processors,

$$\ell^* \geq \mu(x)+\sum_{i=1}^{n}\min\{\mu(y_i),c(y_i)o\} = \mu(x)+\sum_{y\in Y_1}\mu(y)+\sum_{y\in Y_2}c(y)o.$$

In addition, $\ell^* \geq \mu(x)+\frac{1}{P}\sum_{i=1}^{n}\mu(y_i)$. If the schedule in which all tasks are scheduled on one processor is not of minimum length, then $\ell^* \geq \mu(x)+L+2o$.

Let y^* be a source of Y_2 with a maximum completion time. Then its completion time equals the length of (σ_0,π_0). It is possible that every task in Y_1 is scheduled after y^*. Hence

$$\ell \leq \sigma(y^*)+\mu(y^*)+\sum_{y\in Y_1}\mu(y)+\sum_{y\in Y_2:\pi(y)\neq 1}c(y)o+L+o+\mu(x).$$

Assume ℓ_0 is the length of (σ_0,π_0) and ℓ_0^* is the length of a minimum-length schedule for $(G[Y_2],\mu,c,L,o,g,P)$. Clearly, $\ell_0^* < \ell^*$. Assume $\ell_0 \leq \rho\ell_0^*$. Then

$$\begin{aligned}\ell &\leq \sigma(y^*)+\mu(y^*)+\textstyle\sum_{y\in Y_1}\mu(y)+\sum_{y\in Y_2:\pi(y)\neq 1}c(y)o+L+o+\mu(x)\\ &\leq \rho\ell_0^*+\ell^*+L+o\\ &\leq (\rho+1)\ell^*+L+o.\end{aligned}$$

So if $\ell^* > \mu(x)+\sum_{i=1}^{n}\mu(y_i)$, then $\ell \leq (\rho+2)\ell^*$. If the schedule in which all tasks are executed on one processor is of minimum length, then its length is at most ℓ. If (σ,π) is longer than $\mu(x)+\sum_{i=1}^{n}\mu(y_i)$, then replace (σ,π) by the schedule in which all tasks are executed by the same processor. Then this schedule is at most $\rho+2$ times as long as a minimum-length schedule for (G,μ,c,L,o,g,P).

Note that if L and o are bounded by a constant, then Algorithm RESTRICTED RECEIVE GRAPH SCHEDULING is an approximation algorithm with asymptotic approximation ratio $\rho+1$.

There are many algorithms for scheduling incomparable tasks on P identical processors. Using Graham's List scheduling algorithm [38, 39], we obtain an algorithm that constructs schedules on P processors that are at most $4-\frac{2}{P}$ times as long as a minimum-length schedule on P processors [92].

By using different algorithms, we obtain better approximation bounds. Coffman et al. [14] presented Algorithm MULTIFIT. k iterations of this algorithm construct schedules on P processors that are at most $\frac{13}{11}+2^{-k}$ time as long as a minimum-length schedule on P processors [94]. k iterations of Algorithm MULTIFIT take $O(n\log n+kn\log P)$ time. Hence we have proved the following result.

Theorem 10.2.13. *For all constant $k\in\mathbb{Z}^+$, there is an algorithm with an $O(n\log n)$ time complexity that constructs feasible schedules for instances (G,μ,c,L,o,g,P), such that G is a receive graph and $g\leq o$, with length at most $(\frac{35}{11}+2^{-k})\ell^*$, where ℓ^* is the length of a minimum-length schedule for (G,μ,c,L,o,g,P).*

Hochbaum and Shmoys [45] presented a polynomial approximation scheme for scheduling incomparable tasks on identical processors. For each $k \in \mathbb{Z}^+$, a schedule on P processors that is at most $1+\frac{1}{k+1}$ times as long as the length of a minimum-length schedule on P processors can be constructed in $O(((k+1)n)^{(k+1)\log(k+1)})$ time using this approximation scheme [62]. Hence we have proved the following result.

Theorem 10.2.14. *For all constant $k \in \mathbb{Z}^+$, there is an algorithm with an $O(n^{(k+1)\log(k+1)})$ time complexity that constructs feasible schedules for instances (G,μ,c,L,o,g,P), such that G is a receive graph and $g \leq o$, with length at most $(3+\frac{1}{k+1})\ell^*$, where ℓ^* is the length of a minimum-length schedule for (G,μ,c,L,o,g,P).*

10.3 A polynomial special case

In Section 10.2, two approximation algorithms for scheduling receive graphs were presented. Constructing minimum-length schedules for receive graphs on an unrestricted number of processors is a strongly NP-hard optimisation problem. Kort and Trystram showed that if g does not exceed o and all sources of a receive graph have the same execution length, then a minimum-length schedule for this receive graph on an unrestricted number of processors can be constructed in polynomial time. In this section, this result is improved: it is proved that if all sources have the same execution length, then a minimum-length schedule on an unrestricted number of processors can be constructed in polynomial time even if g exceeds o.

Consider an instance (G,μ,c,L,o,g,∞), such that G is a receive graph with sink x and sources $y_1,\dots,y_n$. Assume $\mu(y_1) = \cdots = \mu(y_n) = \mu$. There is a minimum-length schedule for (G,μ,c,L,o,g,∞) in which the tasks and the communication operations are scheduled on at most n processors. From Lemma 10.2.2, we may assume that all processors, expect that one that executes x, execute at most one source of G. To obtain a minimum-length schedule for (G,μ,c,L,o,g,∞), the sources y with minimum $c(y)$ should be scheduled on another processor than x. Assume $c(y_1) \leq \cdots \leq c(y_n)$. In a minimum-length m-processor schedule for (G,μ,c,L,o,g,∞), x is scheduled on processor 1, y_i on processor $i+1$ for all $i \leq m-1$ and the remaining sources of G on processor 1. Sources $y_1,\dots,y_{m-1}$ are completed at time μ. Then $C_m = \sum_{i=1}^{m-1} c(y_i)$ receive operations have to be scheduled on processor 1.

The sinks $y_1,\dots,y_n$ have to be scheduled before the first receive operation or between the receive operations on processor 1. There is a delay of least $\max\{o,g\}-o$ time units between two consecutive receive operations on processor 1. Let $\alpha(o,g) = \frac{\max\{o,g\}-o}{\mu}$. Because there is a delay of at least $\max\{o,g\}-o$ time units between a pair of consecutive receive operations, at least $\lfloor \alpha(o,g) \rfloor$ sources can be scheduled between a pair of consecutive receive operations. If at least $\lceil \alpha(o,g) \rceil$ sources are scheduled between two consecutive receive operations, then we may assume that processor 1 is not idle between these receive operations. We may assume that at most $\lceil \alpha(o,g) \rceil$ sources are scheduled between two consecutive receive operations: if more than $\lceil \alpha(o,g) \rceil$ sources are scheduled between two consecutive receive operations, then the first of these receive operations can be scheduled at a later time without increasing the schedule length.

The length of an m-processor schedule depends on the number of sources executed between the receive operations. Let k be this number. We may assume that $k \leq (C_m-1)\lceil \alpha(o,g) \rceil$ and $k \leq$

$n-m+1$. Let $\ell_{m,k}$ be the minimum length of an m-processor schedule for (G,μ,c,L,o,g,P) in which k sources are scheduled between the receive operations. In such an m-processor schedule, the first receive operation can start at time

$$\max\{(n-k-(m-1))\mu, \mu+L+o\}.$$

If $\lceil\alpha(o,g)\rceil$ sources are scheduled between two consecutive receive operations, then the starting times of these receive operations differ $\lceil\alpha(o,g)\rceil\mu+o$. This is

$$inc(o,g) \quad = \quad \lceil\alpha(o,g)\rceil\mu-(\max\{o,g\}-o)$$

more than when the receive operations are scheduled with as little delay as possible. So each time $\lceil\alpha(o,g)\rceil$ sources are scheduled between two consecutive receive operations, the starting time of x increases by $inc(o,g)$.

Hence $\ell_{m,k}$ equals

$$\max\{(n-k-(m-1))\mu, \mu+L+o\}+(C_m-1)\max\{o,g\}+o+inc_{m,k}(o,g)+\mu(x),$$

where $inc_{m,k}(o,g)=\max\{0,k-(C_m-1)\lfloor\alpha(o,g)\rfloor\}inc(o,g)$.

Let $\ell^*_m=\min_k \ell_{m,k}$. Then ℓ^*_m is the length of a minimum-length m-processor schedule for (G,μ,c,L,o,g,P). Since $c(y_i)$ is bounded by a constant for all sources y_i of G, ℓ^*_m can be computed in $O(n)$ time. The length ℓ^* of a minimum-length schedule for (G,μ,c,L,o,g,P) equals $\min_{1\leq m\leq n}\ell^*_m$. This can be computed in $O(n^2)$ time. If $\ell^*=\ell_{m,k}$, then m and k can be used to construct a schedule of length ℓ^* in linear time. Hence we have proved the following result.

Theorem 10.3.1. *There is an algorithm with an $O(n^2)$ time complexity that constructs minimum-length schedules for instances (G,μ,c,L,o,g,∞), such that G is a receive graph and there is a positive integer μ, such that $\mu(y)=\mu$ for all sources y of G.*

If $\max\{o,g\}-o$ is divisible by μ, then a minimum-length schedule for (G,μ,c,L,o,g,∞) can be constructed more efficiently. Let G be a receive graph with sink x and sources $y_1,\dots,y_n$, such that $c(y_1)\leq\cdots\leq c(y_n)$. Assume $\max\{o,g\}-o$ is divisible by μ. Then we may assume that in a minimum-length m-processor schedule for (G,μ,c,L,o,g,∞), exactly $k_m=\min\{n-m+1,(C_m-1)\alpha(o,g)\}$ sources of G are scheduled between the receive operations on processor 1 and that the remaining sources are scheduled before the first receive operation. Because $inc_{m,k_m}(o,g)$ equals zero, the length of such a schedule equals

$$\max\{(n-k_m-(m-1))\mu, \mu+L+o\}+(C_m-1)\max\{o,g\}+o+\mu(x).$$

The values ℓ_{m,k_m} can be computed in linear time, because we assumed that $c(y_i)$ is bounded by a constant for all sources y_i of G. Let $\ell^*=\min_{1\leq m\leq n}\ell_{m,k_m}$. Assume $\ell^*=\ell_{m,k_m}$. Using m, a schedule for (G,μ,c,L,o,g,∞) of length ℓ^* can be constructed in $O(n)$ time. Because $c(y_i)$ is bounded by a constant for all sources y_i of G, sorting the sources of G by non-decreasing message lengths tasks $O(n)$ time. Hence we have proved the following result.

Theorem 10.3.2. *There is an algorithm with an $O(n)$ time complexity that constructs minimum-length schedules for instances (G,μ,c,L,o,g,∞), such that G is a receive graph and there is a positive integer μ, such that $\mu(y) = \mu$ for all sources y of G and* $\max\{o,g\} - o$ *is divisible by μ.*

Both Theorem 10.3.1 and 10.3.2 improve a result of Kort and Trystram [55], who presented an algorithm that constructs minimum-length schedules for receive graphs with sources of equal length in $O(n^2)$ time if g does not exceed o.

10.4 Concluding remarks

The problem of scheduling send and receive graphs in the LogP model was studied in Chapters 9 and 10, respectively. Although send and receive graphs can be transformed into each other by reversing the arcs, scheduling send graphs is less complicated than scheduling receive graphs. This is due to the fact that we consider a common data semantics. For receive graphs, there is no difference between a common data semantics and an independent data semantics. For send graphs, there is a difference. Scheduling send graphs under an independent semantics is the same as scheduling receive graphs: messages have to be sent for all sinks that are not scheduled on the same processor as the source. Scheduling send graphs under a common data semantics is less complicated, because at most one set of messages has to be sent to any processor.

Like for scheduling send graphs, there are a lot of possible generalisations. If $g \geq o$, then we can prove properties of minimum-length schedules similar to those proved in Section 10.2.1. However, these results do not allow us to prove that Algorithms UNRESTRICTED RECEIVE GRAPH SCHEDULING and RESTRICTED RECEIVE GRAPH SCHEDULING are approximation algorithms with a constant approximation ratio for scheduling with arbitrary o and g. This is due to the fact that the number of communication operations that must be scheduled in an m-processor schedule for a receive graph depends on the processor assignment. Because the number of communication operations in an m-processor schedule for a send graph is independent of the processor assignment, we were able to present a 2-approximation algorithm for scheduling send graphs with arbitrary o and g.

It is unknown whether minimum-length schedules on a restricted number of processors can be constructed in polynomial time if all sources have the same execution length. Kort and Trystram proved that if all sources have the same execution length and this length exceeds $\max\{g, 2o+L\}$, then a minimum-length schedule on two processors can be constructed in polynomial time. They also proved that if $c(y)$ is the same for all sources y of a receive graph, then a minimum-length schedule for this receive graph on an unrestricted number of processors can be constructed in polynomial time.

Like for send graphs, the structure of minimum-length schedules for more general inforests is far more complicated than that of minimum-length schedules for receive graphs. Hence it is difficult to construct approximation algorithms with a constant approximation ratio for more general inforests. In Chapter 11, two algorithms are presented for scheduling general inforests in the LogP model.

11 Decomposition algorithms

In this chapter, two approximation algorithms are presented for scheduling intrees in the LogP model. The basis of these algorithms are two algorithms that decompose intrees into a number of subforests whose sizes do not differ much. Using such decompositions, communication-free schedules are constructed. These are transformed into feasible schedules by introducing the communication operations.

The decompositions of an intree are defined in Section 11.1. The algorithm presented in Section 11.2 uses these decompositions to construct communication-free schedules. In Section 11.3, two algorithms are presented that construct decompositions of d-ary intrees and of arbitrary intrees, respectively. Using these decompositions, the algorithm presented in Section 11.2 constructs communication-free schedules on P processors for d-ary intrees that are at most $d+1-\frac{d^2+d}{d+P}$ times as long as a minimum-length communication-free schedule on P processors. For arbitrary intrees, the communication-free schedules on P processors constructed using the decompositions of the second algorithm are at most $3-\frac{6}{P+2}$ times as long as a minimum-length communication-free schedule on P processors.

The constructed communication-free schedules are transformed into feasible schedules by introducing the communication operations. For both types of decompositions, the number of communication operations that must be introduced is independent of the number of tasks. The length of the schedules for a d-ary intree constructed using the first decomposition algorithm are increased by the total duration of at most $d(P-1)$ communication actions. The length of the schedules constructed using the second decomposition algorithm increases by the total duration of at most $d(d-1)(P-1)-1$ communication actions.

Hence the schedules constructed using the decompositions constructed by the first decomposition algorithm have a large computation part and a small communication part and the schedules constructed using the decompositions constructed by the second decomposition algorithm have a small computation part and a large communication part.

11.1 Decompositions of intrees

In this section, the decompositions of an intree will be defined. A decomposition of an intree is a collection of disjoint subforests whose roots have the same child.

Definition 11.1.1. Let G be an intree. A *decomposition* of G is a non-empty sequence of subforests $(G_1,\dots,G_k)$ of G, such that

1. $V(G_1)\cup\cdots\cup V(G_k)=V(G)$;
2. for all $i\neq j$, $V(G_i)\cap V(G_j)=\varnothing$;
3. for all $i\in\{1,\dots,k\}$, the roots of G_i all have the same child in G; and
4. for all $i\in\{1,\dots,k\}$, no task of G_i has a predecessor in $G_{i+1},\dots,G_k$.

A sequence of instances $((G_1,\mu,c,L,o,g,P),\dots,(G_k,\mu,c,L,o,g,P))$ will be called a *decomposition* of the instance (G,μ,c,L,o,g,P) if $(G_1,\dots,G_k)$ is a decomposition of G.

The fact that all roots of a subforest in a decomposition of an intree have the same parent will play an important role in the analysis of the algorithms presented in this chapter.

Let G be an intree. Let $((G_1,\mu,c,L,o,g,P),\dots,(G_k,\mu,c,L,o,g,P))$ be a decomposition of (G,μ,c,L,o,g,P). We will use a shorthand notation: $(G_1,\dots,G_k)$ is said to be a decomposition of (G,μ,c,L,o,g,P). Each forest G_i will be called *decomposition forest*. If a forest G_i has only one root, it will also be called a *decomposition tree*.

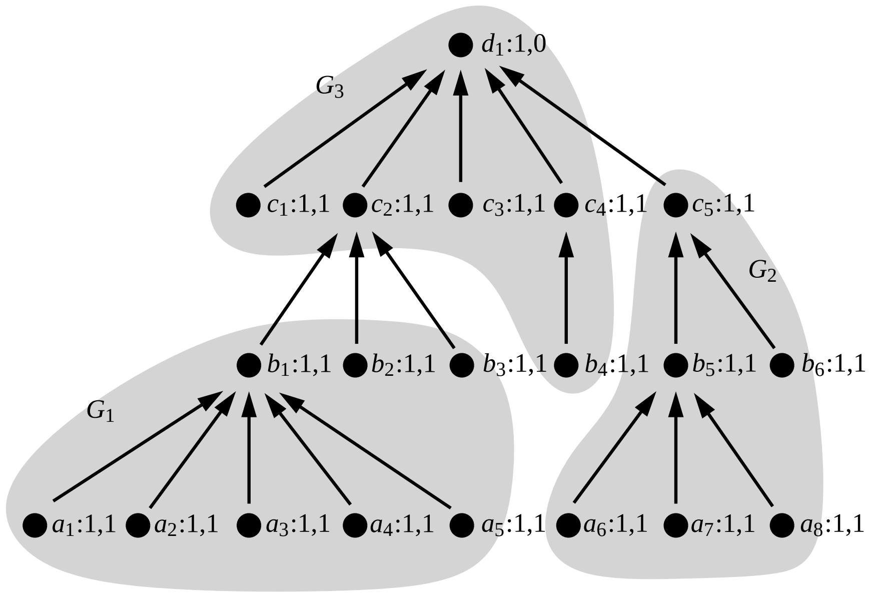

Figure 11.1. A decomposition (G_1,G_2,G_3) of an instance (G,L,o,g,P)

Example 11.1.2. Let G be the intree shown in Figure 11.1. A decomposition (G_1,G_2,G_3) of G is shown as well. The roots of G_1 are the tasks b_1, b_2 and b_3. These are all parents of c_2. G_2 and G_3 have only one root. It is obvious that no successor of a task of G_1 is a task of G_2 or G_3 and that a task of G_2 has no predecessor in G_3.

Let G be an intree and let $(G_1,\dots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P). Since a task of G_i has no predecessors in $G_{i+1},\dots,G_k$ and the root of G is a successor of all other tasks of G, G_k must be an intree whose root is the root of G.

Observation 11.1.3. *Let G be an intree. Let $(G_1,\dots,G_k)$ be a decomposition of G. Then G_k is an intree and its root is the root of G.*

Let G be an intree. Let $(G_1,\dots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P). We will divide each decomposition forest G_i into two parts. For each $i\in\{1,\dots,k\}$, the set $A(G_i)$ contains all tasks of G_i that have a predecessor outside G_i and $B(G_i)$ is the set of tasks of G_i do not have a predecessor outside G_i. More precisely,

$$A(G_i) \quad = \quad \{u \in V(G_i) \mid Pred_G(u) \setminus V(G_i) \neq \varnothing\}$$

and

$$B(G_i) \quad = \quad \{u \in V(G_i) \mid Pred_G(u) \subseteq V(G_i)\}.$$

Note that $A(G_i)$ does not contain any sources of G and that every task in $A(G_i)$ has a predecessor outside $B(G_i)$. Let $A(G_1,\dots,G_k)$ be the subforest of G induced by $A(G_1) \cup \dots \cup A(G_k)$. It is not difficult to see that if $A(G_1,\dots,G_k)$ is not the empty precedence graph, then $A(G_1,\dots,G_k)$ is a subtree of G with the same root as G. Moreover, if $k \geq 2$, then $A(G_1,\dots,G_k)$ cannot be the empty precedence graph. In addition, it is easy to see that the tasks in a set $B(G_i)$ are incomparable with tasks in a set $B(G_j)$ for all $j \neq i$.

Example 11.1.4. Let G be the intree shown in Figure 11.1. Let (G_1,G_2,G_3) be the decomposition of (G,L,o,g,P) shown in Figure 11.1. Since no task of G_1 has a predecessor outside G_1, $A(G_1) = \varnothing$ and $B(G_1) = \{a_1,a_2,a_3,a_4,a_5,b_1,b_2,b_3\}$. Similarly, $A(G_2) = \varnothing$ and $B(G_2) = \{a_6,a_7,a_8,b_5,b_6,c_5\}$. Tasks c_2 and d_1 of G_3 have a predecessor outside G_3: c_2 is a successor of all tasks of G_1 and d_2 of all tasks of G_1 and G_2. Hence $A(G_3) = \{c_2,d_1\}$ and $B(G_3) = \{b_4,c_1,c_3,c_4\}$. So $A(G_1,\dots,G_k)$ is the intree with tasks c_2 and d_1 and an arc from c_2 to d_1.

Let G be an intree. Let $(G_1,\dots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P). The number of roots of G_i is denoted by $\#G_i$. The following lemma will be used to bound the number of communication operations that must be introduced in a communication-free schedule for (G,μ,c,L,o,g,P).

Lemma 11.1.5. *Let G be a d-ary intree. If $(G_1,\dots,G_k)$ is a decomposition of (G,μ,c,L,o,g,P) into $k \geq 2$ subforests, then*

$$\sum_{u \in V(A(G_1,\dots,G_k))} (|Pred_{G,0}(u)|-1) \quad \leq \quad d(\#G_1 + \dots + \#G_k - 1) - 1.$$

Proof. Assume $(G_1,\dots,G_k)$ is a decomposition of (G,μ,c,L,o,g,P) into $k \geq 2$ subforests. Let U be the union of $V(A(G_1,\dots,G_k))$ and the set of parents of the tasks of $A(G_1,\dots,G_k)$. Let u be a task in U. If $|Pred_{G[U],0}(u)| \geq 1$, then u is a task of $A(G_1,\dots,G_k)$. Since $G[U]$ is an intree, the number of arcs of $G[U]$ equals $|U|-1$. Hence

$$\begin{aligned}
\textstyle\sum_{u \in V(A(G_1,\dots,G_k))} (|Pred_{G,0}(u)|-1) &= \textstyle\sum_{u \in V(A(G_1,\dots,G_k))} (|Pred_{G[U],0}(u)|-1) \\
&= \textstyle\sum_{u \in U} |Pred_{G[U],0}(u)| - |V(A(G_1,\dots,G_k))| \\
&= |U| - 1 - |V(A(G_1,\dots,G_k))| \\
&= |U \setminus V(A(G_1,\dots,G_k))| - 1.
\end{aligned}$$

The tasks in $U \setminus V(A(G_1,\dots,G_k))$ do not have a predecessor outside their subforests, but their children in $A(G_1,\dots,G_k)$ do. These children have a parent that is a root of a decomposition forest. The root of G is also the root of G_k and cannot be an element of $U \setminus V(A(G_1,\dots,G_k))$. So the number of tasks of $A(G_1,\dots,G_k)$ with a parent outside $A(G_1,\dots,G_k)$ is at most $\#G_1 + \dots + \#G_k - 1$. Every task of G has indegree at most d. So $U \setminus V(A(G_1,\dots,G_k))$ contains at most $d(\#G_1 + \dots + \#G_k - 1)$ tasks. Hence $\sum_{u \in V(A(G_1,\dots,G_k))}(|Pred_{G,0}(u)| - 1) \leq d(\#G_1 + \dots + \#G_k - 1) - 1$. □

Let G be an intree. Let $(G_1,\dots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P). For all $i \in \{1,\dots,k\}$, let $r_{i,1},\dots,r_{i,\#G_i}$ be the roots of G_i. Define an intree $D(G_1,\dots,G_k)$ as follows. $V(D(G_1,\dots,G_k)) = \bigcup_{i=1}^{k}\{r_{i,1},\dots,r_{i,\#G_i}\}$ and $D(G_1,\dots,G_k)$ contains an arc form r_{i_1,j_1} to r_{i_2,j_2} if there is a path in G from r_{i_1,j_1} to r_{i_2,j_2} that does not contain another task in $V(D(G_1,\dots,G_k))$. If $D(G_1,\dots,G_k)$ contains an arc from r_{i_1,j_1} to r_{i_2,j_2}, then r_{i_2,j_2} is called a *decomposition child* of r_{i_1,j_1} and r_{i_1,j_1} a *decomposition parent* of r_{i_2,j_2}.

Example 11.1.6. Let G be the intree shown in Figure 11.1. Let (G_1,G_2,G_3) be the decomposition of (G,L,o,g,P) shown in Figure 11.1. G_1 has roots b_1, b_2 and b_3; c_5 is the only root of G_2 and G_3 has root d_1. Hence $D(G_1,\dots,G_k)$ contains tasks b_1, b_2, b_3, c_5 and d_1. Moreover, it contains arcs (b_1,d_1), (b_2,d_1), (b_3,d_1) and (c_5,d_1).

11.2 Scheduling decomposition forests

The decompositions defined in Section 11.1 will be used to construct communication-free schedules for instances (G,μ,c,L,o,g,P), such that G is an intree and $P \neq \infty$. The communication operations are introduced in these communication-free schedules for every pair of tasks u_1 and u_2, such that u_1 is a parent of u_2 and u_1 and u_2 are scheduled on different processors. Such a pair of tasks will be called a *communicating pair* and the number of communicating pairs will be called the *communication requirement* of the communication-free schedule.

Hu [49] proved that a minimum-length communication-free schedule for an inforest with unit-length tasks on P processors can be constructed in polynomial time. Kunde [57] showed that critical path scheduling constructs communication-free schedules for inforests with arbitrary task lengths on P processors that are at most $2 - \frac{2}{P+1}$ times as long as a minimum-length schedule. Unfortunately, the communication requirements of the schedules constructed by the algorithms of Kunde and Hu may be as high as $(1-\frac{1}{d})n + \frac{1}{d}$ for d-ary intrees. As a result, introducing communication operations in such schedules will greatly increase the length of the schedule.

Using a decomposition of an intree, we will construct communication-free schedules that are longer than those constructed by critical path scheduling, but have only a small communication requirement. Algorithm DECOMPOSITION FOREST SCHEDULING presented in Figure 11.2 uses a decomposition of an intree to construct a communication-free schedule. Let G be an intree and let $(G_1,\dots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P) into $k \leq P$ subforests. Algorithm DECOMPOSITION FOREST SCHEDULING works as follows. For each $i \in \{1,\dots,k\}$, the tasks in $B(G_i)$ are scheduled without interruption from time 0 onward on processor i. The tasks in $A(G_i)$ are scheduled on one of the processors $1,\dots,i-1$ not before the maximum completion time of a task in $B(G_i)$.

```
Algorithm DECOMPOSITION FOREST SCHEDULING
Input. An instance $(G,\mu,c,L,o,g,P)$, such that $G$ is an intree and a decomposition $(G_1,\ldots,G_k)$
       of $(G,\mu,c,L,o,g,P)$ consisting of $k \le P$ decomposition forests.
Output. A feasible communication-free schedule $(\sigma,\pi)$ for $(G,\mu,c,L,o,g,P)$.
1.  for $i := 1$ to $k$
2.      do $idle(i) := 0$
3.         $U := B(G_i)$
4.         while $U \neq \varnothing$
5.             do let $u$ be a source of $G[U]$
6.                $\sigma(u) := idle(i)$
7.                $\pi(u) := i$
8.                $idle(i) := idle(i) + \mu(u)$
9.                $U := U \setminus \{u\}$
10.        $last(i) := idle(i)$
11.        $U := A(G_i)$
12.        while $U \neq \varnothing$
13.            do let $u$ be a source of $G[U]$
14.               let $v \notin B(G_i)$ be a parent of $u$ with maximum completion time
15.               $\sigma(u) := \max\{idle(\pi(v)), last(i)\}$
16.               $\pi(u) := \pi(v)$
17.               $idle(\pi(v)) := \sigma(u) + \mu(u)$
18.               $U := U \setminus \{u\}$
```

Figure 11.2. Algorithm DECOMPOSITION FOREST SCHEDULING

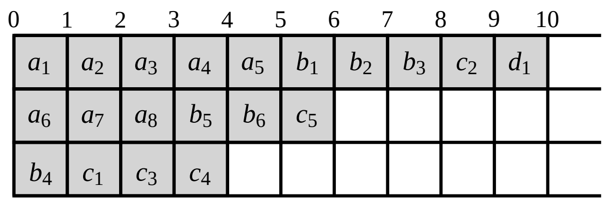

Figure 11.3. A schedule built by Algorithm DECOMPOSITION FOREST SCHEDULING

Example 11.2.1. Let $(G,L,o,g,3)$ be the instance shown in Figure 11.1. Consider its decomposition (G_1,G_2,G_3) that is also shown in Figure 11.1. Algorithm DECOMPOSITION FOREST SCHEDULING constructs a communication-free schedule for $(G,L,o,g,3)$ as follows. The tasks in $B(G_1) = \{a_1,a_2,a_3,a_4,a_5,b_1,b_2,b_3\}$ are scheduled on processor 1 from time 0 onward. Similarly, the tasks in $B(G_2) = \{a_6,a_7,a_8,b_5,b_6,c_5\}$ are scheduled on processor 2 from time 0 onward. $B(G_3)$ contains tasks b_4, c_1, c_3 and c_4; these are scheduled on processor 3 from time 0 onward. $A(G_3)$ contains tasks c_2 and d_1. b_3 is the parent of c_2 outside $B(G_3)$ with the largest completion time. So c_2 is scheduled on processor 1 after b_3. Because c_2 is the parent of d_1 with the largest completion time and d_1 is not an element of $B(G_3)$, d_1 is scheduled on processor 1 after c_2. The resulting schedule is shown in Figure 11.3. It has communication requirement 4, because (c_1,d_1), (c_3,d_1), (c_4,d_1) and (c_5,d_1) are communication pairs.

Now we will prove that Algorithm DECOMPOSITION FOREST SCHEDULING correctly constructs feasible communication-free schedules.

Lemma 11.2.2. *Let G be an intree. Let $(G_1,\ldots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P) into $k \leq P$ subforests. Let (σ,π) be the schedule for $(G_1,\ldots,G_k)$ constructed by Algorithm* DECOMPOSITION FOREST SCHEDULING. *Then (σ,π) is a feasible communication-free schedule for (G,μ,c,L,o,g,P).*

Proof. Let u be a task of G. Assume u is a task of G_i. First we will assume that u is an element of $B(G_i)$. Then u is scheduled on processor i and obviously, no other task is scheduled at the same time on this processor. Moreover, because the order in which the tasks of $B(G_i)$ are executed is a topological order of $G[B(G_i)]$, u is scheduled after its predecessors. Second we will assume that u is an element of $A(G_i)$. Then u has a parent outside $B(G_i)$. So u is scheduled after one of its parents v outside $B(G_i)$ on processor $\pi(v)$. Clearly, processor $\pi(v)$ does not execute another task at the same time. Since u does not start before the completion time of the last task in $B(G_i)$, u is scheduled after its predecessors. Hence (σ,π) is a feasible communication-free schedule for (G,μ,c,L,o,g,P). □

The time complexity of Algorithm DECOMPOSITION FOREST SCHEDULING can be determined as follows. Let G be an intree and let $(G_1,\ldots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P) into $k \leq P$ subforests. Let $i \in \{1,\ldots,k\}$. The tasks in $B(G_i)$ can be scheduled using a topological order of $G[B(G_i)]$. Such an order can be constructed in $O(|B(G_i)|)$ time [18]. Using a topological order of $G[B(G_i)]$, the tasks in $B(G_i)$ can be scheduled in $O(|B(G_i)|)$ time. The tasks in $A(G_i)$ can be scheduled using a topological order of $G[A(G_i)]$. Let u be a task in $A(G_i)$. The parents of u outside $B(G_i)$ can be found in $O(|Pred_{G,0}(u)|+|B(G_i)|)$ time. Then determining a parent of u outside $B(G_i)$ with the largest completion time requires $O(|Pred_{G,0}(u)|)$ time. So assigning a starting time and a processor to every task in $A(G_i)$ takes $O(\sum_{u\in A(G_i)}|Pred_{G,0}(u)|+|A(G_i)||B(G_i)|)$ time. Since the sets $A(G_i)$ and $B(G_i)$ are all disjoint, Algorithm DECOMPOSITION FOREST SCHEDULING constructs a feasible communication-free schedule in $O(n^2)$ time.

Lemma 11.2.3. *For all instances (G,μ,c,L,o,g,P), such that G is an intree, and all decompositions $(G_1,\ldots,G_k)$ of (G,μ,c,L,o,g,P) into at most P decomposition forests, Algorithm* DECOMPOSITION FOREST SCHEDULING *constructs a feasible communication-free schedule for (G,μ,c,L,o,g,P) in $O(n^2)$ time.*

The following lemma gives an important property of the communication-free schedules constructed by Algorithm DECOMPOSITION FOREST SCHEDULING. This result will be used to construct upper bounds on the length of a communication-free schedule constructed by Algorithm DECOMPOSITION FOREST SCHEDULING.

Lemma 11.2.4. *Let G be an intree. Let $(G_1,\ldots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P), such that $k \leq P$. Let (σ,π) be the communication-free schedule for (G,μ,c,L,o,g,P) constructed* *by Algorithm* DECOMPOSITION FOREST SCHEDULING*. Then for all $i \in \{1,\dots,k\}$, all roots r of G_i and all tasks u of G, if $u \notin V(G_i)$, $\pi(u) = \pi(r)$ and $\sigma(u) > \sigma(r)$, then $r \prec_G u$.*

Proof. We will prove by induction that for all $i \in \{1,\dots,k\}$, for all roots r_i of G_i and all tasks u of G, if u is not a task of G_i, $\pi(u) = \pi(r_i)$ and $\sigma(u) > \sigma(r_i)$, then $r_i \prec_G u$. Let $i \in \{1,\dots,k\}$. Assume by induction that for all $j \leq i-1$, for all roots r_j of G_j and all tasks u of G, if $u \notin V(G_j)$, $\pi(u) = \pi(r_j)$ and $\sigma(u) > \sigma(r_j)$, then $r_j \prec_G u$. Let r_i be a root of G_i. We will prove by induction that for all tasks u of G, if $u \notin V(G_i)$, $\pi(u) = \pi(r_i)$ and $\sigma(u) > \sigma(r_i)$, then $r_i \prec_G u$. Let u be a task of G. Assume by induction that for all predecessors v of u, if $v \notin V(G_i)$, $\pi(v) = \pi(r_i)$ and $\sigma(v) > \sigma(r_i)$, then $r_i \prec_G v$. Assume u is not a task of G_i, $\pi(u) = \pi(r_i)$ and $\sigma(u) > \sigma(r_i)$. Then u must be a task in a set $A(G_{i'})$ for some $i' \geq i+1$. Hence a parent v of u is scheduled on processor $\pi(r)$.

Case 1. v is a task of G_i.
Because u is not a task of G_i and v is a parent of u, v must be a root of G_i. Because all roots of G_i have the same child, r_i is a predecessor of u.

Case 2. v is not a task of G_i.

Case 2.1. $\sigma(v) > \sigma(r_i)$.
By induction, v is a successor of r_i. Hence u is a successor of r_i.

Case 2.2. $\sigma(v) \leq \sigma(r_i)$.
Since (σ,π) is a feasible communication-free schedule for (G,μ,c,L,o,g,P), $\sigma(v) < \sigma(r_i)$. Hence v must be a task of a decomposition forest $G_{j'}$, such that $j' < i$. Because u is not a task of $G_{j'}$, v must be a root of $G_{j'}$. By induction, r_i is a successor of v. Because G is an inforest, all successors of v are comparable. Because u is scheduled after r_i, u is a successor of r_i.

□

Next we will compute an upper bound on the length of the communication-free schedules constructed by Algorithm DECOMPOSITION FOREST SCHEDULING. Let G be an intree and let $(G_1,\dots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P) into at most P decomposition forests. Let (σ,π) be the communication-free schedule for (G,μ,c,L,o,g,P) constructed by Algorithm DECOMPOSITION FOREST SCHEDULING using $(G_1,\dots,G_k)$. Assume decomposition forest G_i has roots $r_{i,1},\dots,r_{i,\#G_i}$. Let $C(r_{i,j})$ be the completion time of $r_{i,j}$.

Consider a root $r_{i,j}$ of G_i. From Lemma 11.2.4, all tasks scheduled after $r_{i,j}$ on processor $\pi(r_{i,j})$ are either tasks of G_i or successors of $r_{i,j}$. Let r_{i_1,j_1} and r_{i_2,j_2} be roots of decompositions forests G_{i_1} and G_{i_2}. If r_{i_1,j_1} and r_{i_2,j_2} are both decomposition parents of $r_{i,j}$ and $i_1 \neq i_2$, then r_{i_1,j_1} and r_{i_2,j_2} are incomparable and must be scheduled on different processors.

Consider a root $r_{i,j}$ of decomposition forest G_i. Since (σ,π) is a communication-free schedule and all decomposition parents of $r_{i,j}$ are scheduled on different processors, there is a decomposition parent $r_{i',j'}$ of $r_{i,j}$, such that the path from the child of $r_{i',j'}$ to $r_{i,j}$ is scheduled without interruption. The first task of such a path starts either at the completion time of $r_{i',j'}$ or at the maximum completion time of a task in $B(G_i)$. Let $p(u,v)$ denote the unique path from the child of u to v if it exists. Then for all $i \leq k$ and $j \leq \#G_i$,

$$\begin{aligned} C(r_{i,j}) &\leq \max_{r_{i',j'} \in Pred_{D(G_1,\dots,G_k),0}(r_{i,j})} (\max\{\mu(B(G_i)), C(G_{i'})\} + \mu(p(r_{i',j'}, r_{i,j}))) \\ &\leq \max\{\mu(G_i), \max_{r_{i',j'} \in Pred_{D(G_1,\dots,G_k),0}(r_{i,j})} (C(G_{i'}) + \mu(p(r_{i',j'}, r_{i,j})))\}. \end{aligned}$$

We can prove by induction that for all $i \leq k$ and all $j \in \{1,\dots,\#G_i\}$,

$$C(r_{i,j}) \leq \max\{\mu(G_i), \max_{r_{i',j'} \in Pred_{D(G_1,\dots,G_k)}(r_{i,j})} (\mu(G_{i'}) + \mu(p(r_{i',j'}, r_{i,j})))\}.$$

Since $r_{k,1}$ is the root of G, the length of (σ,π) is at most

$$\max\{\mu(G_k), \max_{1 \leq i < k} (\mu(G_i) + \mu(p(r_{i,1}, r_{k,1})))\}.$$

Finally, we will compute an upper bound on the communication requirement of the schedules constructed by Algorithm DECOMPOSITION FOREST SCHEDULING. Let G be an intree and $(G_1,\dots,G_k)$ a decomposition of (G,μ,c,L,o,g,P) into $k \leq P$ decomposition forests. Let (σ,π) be the communication-free schedule for (G,μ,c,L,o,g,P) constructed by Algorithm DECOMPOSITION FOREST SCHEDULING using $(G_1,\dots,G_k)$. Let v be a task of G. If a parent of v is not scheduled on the same processor as v, then v must be a task of $A(G_1,\dots,G_k)$. Any task of $A(G_1,\dots,G_k)$ is scheduled on the same processor as one of its parents. So at most $|Pred_{G,0}(v)| - 1$ parents of v are executed on a different processor. From Lemma 11.1.5, the communication requirement of (σ,π) is at most

$$\sum_{u \in V(A(G_1,\dots,G_k))} (|Pred_{G,0}(u)| - 1) \leq d(\#G_1 + \cdots + \#G_k - 1) - 1.$$

Hence we have proved the following result.

Lemma 11.2.5. *For all instances (G,μ,c,L,o,g,P), such that G is an intree, and all decompositions $(G_1,\dots,G_k)$ of (G,μ,c,L,o,g,P) consisting of at most P decomposition forests, Algorithm* DECOMPOSITION FOREST SCHEDULING *constructs a communication-free schedule for (G,μ,c,L,o,g,P) with length at most* $\max\{\mu(G_k), \max_{1\leq i<k}(\mu(G_i) + \mu(p(r_{i,1}, r_{k,1})))\}$ *and communication requirement at most $d(\#G_1 + \cdots + \#G_k - 1) - 1$ in $O(n^2)$ time.*

Now we will shown how to introduce the communication operations in the communication-free schedules constructed by Algorithm DECOMPOSITION FOREST SCHEDULING. Let G be an intree. Let $(G_1,\dots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P) into at most P decomposition forests. Consider the communication-free schedule (σ,π) for (G,μ,c,L,o,g,P) constructed by Algorithm DECOMPOSITION FOREST SCHEDULING. A feasible schedule (σ_c,π_c) for (G,μ,c,L,o,g,P) can be constructed by introducing communication operations between all communicating pairs. This is done as follows. Assume (u_1,u_2) is a communicating pair. Let $U = \{u \in V(G) \mid \sigma(u) \geq \sigma(u_2)\}$. Increase the starting time of all tasks in U by $(c(u_1)-1)\max\{o,g\} + L + 2o$. For all $i \leq c(u_1)$, schedule send operation $s_{u_1,\pi(u_2),i}$ at time $\sigma(u_2) + (i-1)\max\{o,g\}$ on processor $\pi(u_1)$ and receive operation $r_{u_1,\pi(u_2),i}$ at time $\sigma(u_2)+(i-1)\max\{o,g\}+L+o$ on processor $\pi(u_2)$. If these communication operations are introduced for all communicating pairs, then the length of (σ_c,π_c) is at most the sum of the length of (σ,π) and $(d(\#G_1+\cdots+\#G_k-1)-1)(L+o+c_{\max}\max\{o,g\})$. It is easy to see that the introduction of these communication operations takes $O(n^2)$ time. Hence we have proved the following result.

Theorem 11.2.6. *For all instances (G,μ,c,L,o,g,P), such that G is an intree, and all decompositions $(G_1,\dots,G_k)$ of (G,μ,c,L,o,g,P) into at most P decomposition forests, a feasible schedule for (G,μ,c,L,o,g,P) of length at most* $\max\{\mu(G_k),\max_{1\le i<k}(\mu(G_i)+\mu(p(r_{i,1},r_{k,1})))\}+(d(\#G_1+\cdots+\#G_k-1)-1)(L+o+c_{\max}\max\{o,g\})$ *can be constructed in $O(n^2)$ time.*

11.3 Constructing decompositions of intrees

In this section, two algorithms are presented for constructing decompositions of intrees that are to be used by Algorithm DECOMPOSITION FOREST SCHEDULING for the construction of communication-free schedules. Both algorithms construct decompositions for a special class of instances, called β-restricted instances. Such instances will be defined in Section 11.3.1. In addition, it is shown how decompositions of β-restricted instances can be used to construct schedules for arbitrary instances.

The first decomposition algorithm is presented in Section 11.3.2. This algorithm constructs decompositions of d-ary intrees. The second decomposition algorithm, that is presented in Section 11.3.3, constructs decompositions of arbitrary intrees. Both algorithms decompose an intree into a sequence of subforests whose sizes do not differ much.

11.3.1 β-restricted instances

Let G be an intree. Consider an instance (G,μ,c,L,o,g,P). If the lengths of the tasks of G can be arbitrarily large, then it is impossible to construct decompositions of (G,μ,c,L,o,g,P) into a small number of decomposition forests whose total execution lengths do not differ much. Hence we will construct instances in which the maximum task length is bounded. Such instances will be called β-restricted and are defined as follows.

Definition 11.3.1. Let β be a positive integer. An instance (G,μ,c,L,o,g,P) is called β-*restricted* if for all tasks u of G,

1. $\mu(u)\le\beta$; and
2. if $|Pred_{G,0}(u)|\ge 2$, then $\mu(u)=1$.

We will show that any instance (G,μ,c,L,o,g,P), such that G is an intree, can be transformed into a β-restricted instance $(G_\beta,\mu_\beta,c_\beta,L,o,g,P)$ and that the schedules for $(G_\beta,\mu_\beta,c_\beta,L,o,g,P)$ constructed by Algorithm DECOMPOSITION FOREST SCHEDULING using a decomposition of $(G_\beta,\mu_\beta,c_\beta,L,o,g,P)$ can be transformed into feasible schedules for (G,μ,c,L,o,g,P) without increasing the schedule length. The choice of β will be delayed until the analysis of the schedules constructed by Algorithm DECOMPOSITION FOREST SCHEDULING using the decompositions of

β-restricted instances.

The following observation is used for the construction of decompositions of β-restricted instances (G,μ,c,L,o,g,P).

Observation 11.3.2. *Let β be a positive integer. Let (G,μ,c,L,o,g,P) be a β-restricted instance. Let U be a set of tasks of G. Then $(G[U],\mu,c,L,o,g,P)$ is a β-restricted instance.*

β-restricted instances can be constructed as follows. Let β be a positive integer. Consider an instance (G,μ,c,L,o,g,P). If (G,μ,c,L,o,g,P) is not β-restricted, then a β-restricted instance $(G_\beta,\mu_\beta,c_\beta,L,o,g,P)$ can be constructed as follows. Let u be a task of G. Assume $\mu(u)=k_1\beta+k_2+1$, such that $0\leq k_2\leq\beta-1$. If $k_2=0$, then let $k_u=k_1$. Otherwise, let $k_u=k_1+1$. Then u is replaced by a chain of k_u+1 tasks $z_{u,0},z_{u,1},\ldots,z_{u,k_u}$, such that $z_{u,0}\prec_{G,0} z_{u,1}\prec_{G,0}\cdots\prec_{G,0} z_{u,k_u}$, $\mu_\beta(z_{u,0})=1$, $\mu_\beta(z_{u,1})=\cdots=\mu_\beta(z_{u,k_u-1})=\beta$ and $\mu_\beta(z_{u,k_u})=\mu(u)-1-(k_u-1)\beta$. In addition, let $c_\beta(z_{u,0})=\cdots=c_\beta(z_{u,k_u-1})=1$ and $c_\beta(z_{u,k_u})=c(u)$. Then $(G_\beta,\mu_\beta,c_\beta,L,o,g,P)$ is a β-restricted instance. It is not difficult to see that G_β contains at most $(\frac{\mu(G)}{\beta}+2)n$ tasks and at most $(\frac{\mu(G)}{\beta}+1)n+e$ arcs.

The following lemma is used to transform a schedule for a β-restricted instance into a schedule for the corresponding original instance.

Lemma 11.3.3. *Let G be an intree. Let $(G_1,\ldots,G_k)$ be a decomposition of (G,μ,c,L,o,g,P) consisting of at most P decomposition forests. Let (σ,π) be the communication-free schedule for (G,μ,c,L,o,g,P) constructed by Algorithm* DECOMPOSITION FOREST SCHEDULING *using $(G_1,\ldots,G_k)$. Let u_1 and u_2 be two tasks of G. If u_1 is the only parent of u_2 and u_2 is the only child of u_1, then $\pi(u_1)=\pi(u_2)$.*

Proof. Assume u_1 is the only parent of u_2 and u_2 is the only child of u_1. Assume u_2 is a task of G_i.

Case 1. u_2 is an element of $A(G_i)$.
Then u_2 is scheduled on the same processor as one of its parents. Because u_1 is the only parent of u_2, $\pi(u_1)=\pi(u_2)$.

Case 2. u_2 is an element of $B(G_i)$.
Then u_2 has no predecessors outside G_i. Hence u_1 is a task of G_i. Because every predecessor of u_1 is a predecessor of u_2, u_1 is an element of $B(G_i)$. So $\pi(u_1)=\pi(u_2)$.

□

Let G be an intree. Consider an instance (G,μ,c,L,o,g,P). Let $(G_1,\ldots,G_k)$ be a decomposition of the β-restricted instance $(G_\beta,\mu_\beta,c_\beta,L,o,g,P)$ consisting of $k\leq P$ decomposition forests. Let (σ_β,π_β) be the communication-free schedule for $(G_\beta,\mu_\beta,c_\beta,L,o,g,P)$ constructed by Algorithm DECOMPOSITION FOREST SCHEDULING using $(G_1,\ldots,G_k)$. Let u be a task of G. Lemma 11.3.3 shows all subtasks $z_{u,i}$ of u are scheduled on the same processor.

It is not difficult to reschedule the tasks on each processor, such that the subtask $z_{u,i+1}$ is scheduled immediately after $z_{u,i}$ for all tasks u of G and all $i \in \{0,\ldots,k_u-1\}$. Let (σ,π) be the schedule for $(G_\beta,\mu_\beta,c_\beta,L,o,g,P)$ in which all subtasks of the same task of G are scheduled without interruption on one processor. It is not difficult to see that the length and communication requirement of (σ,π) do not differ from those of (σ_β,π_β). The schedule (σ,π) can be transformed into a feasible communication-free schedule for (G,μ,c,L,o,g,P): u can be scheduled at time $\sigma(z_{u,0})$ on processor $\pi(z_{u,0})$. Let (σ',π') be the resulting schedule for (G,μ,c,L,o,g,P). The length of (σ',π') equals that of (σ_β,π_β). Moreover, because no task is scheduled on a different processor, the communication requirement of (σ',π') equals that of (σ_β,π_β).

It is not difficult to see that (σ',π') can be constructed from (σ_β,π_β) in $O(|V(G_\beta)|)$ time. Hence we have proved the following result.

Lemma 11.3.4. *Let G be an intree. Let (σ_β,π_β) be the communication-free schedule for the β-restricted instance $(G_\beta,\mu_\beta,c_\beta,L,o,g,P)$ constructed by Algorithm* DECOMPOSITION FOREST SCHEDULING *using a decomposition $(G_1,\ldots,G_k)$ of $(G_\beta,\mu_\beta,c_\beta,L,o,g,P)$ into $k \leq P$ subforests. Then a feasible communication-free schedule for (G,μ,c,L,o,g,P) with the same length and communication requirement as (σ_β,π_β) can be constructed in $O(\frac{\mu(G)}{\beta}n)$ time.*

Lemma 11.3.4 shows that we only need to construct decompositions of β-restricted instances.

11.3.2 Constructing decompositions of d-ary intrees

In this section, an algorithm is presented that constructs decompositions of d-ary intrees. Let G be a d-ary intree. Let (G,μ,c,L,o,g,P) be a β-restricted instance. The next lemma allows the decomposition of (G,μ,c,L,o,g,P) into intrees whose sizes do not differ much. Let u be a task of G. The subgraph of G induced by a task u and its predecessors is an intree with root u. This intree is denoted by $T_G(u)$. So $T_G(u) = G[Pred_G(u) \cup \{u\}]$. The following lemma is similar to a lemma of Kosaraju [56] that considers the number of leafs of binary trees.

Lemma 11.3.5. *Let G be a d-ary intree. Let (G,μ,c,L,o,g,P) be a β-restricted instance. If $\mu(G) \geq \beta$, then G contains a task u, such that $\beta \leq \mu(T_G(u)) \leq d(\beta-1)+1$.*

Proof. It will be proved by induction that for all d-ary intrees G, if (G,μ,c,L,o,g,P) is β-restricted and $\mu(G) \geq \beta$, then G contains a task u, such that $\beta \leq \mu(T_G(u)) \leq d(\beta-1)+1$. Let G be a d-ary intree. Let r be the root of G. If G contains exactly one task, then $\mu(T_G(r)) = \beta$. So we may assume that G contains $n \geq 2$ tasks. Assume by induction that for all d-ary intrees G' with at most $n-1$ tasks, if (G',μ',c',L',o',g',P') is β-restricted and $\mu(G') \geq \beta$, then G' contains a task u, such that $\beta \leq \mu'(T_{G'}(u)) \leq d(\beta-1)+1$. Assume (G,μ,c,L,o,g,P) is β-restricted and $\beta \leq \mu(G)$.

Case 1. r has indegree one.
Let u be the parent of r. If $\mu(T_G(u)) \leq \beta-1$, then $\beta \leq \mu(T_G(r)) \leq \mu(r)+\mu(T_G(u)) \leq 2\beta-1 \leq d(\beta-1)+1$. Otherwise, by induction, $T_G(u)$ contains a task v, such that $\beta \leq \mu(T_G(v)) \leq d(\beta-1)+1$.

Case 2. r has indegree at least two.

Then r has length one. If $\mu(T_G(u)) \leq \beta - 1$ for all parents u of r, then $\beta \leq \mu(T_G(u)) \leq d(\beta-1)+1$. Otherwise, r has a parent u, such that $\mu(T_G(u)) \geq \beta$. By induction, $T_G(u)$ contains a task v, such that $\beta \leq \mu(T_G(v)) \leq d(\beta-1)+1$.

□

By repeatedly applying this result, one can construct a decomposition of a β-restricted instance (G,μ,c,L,o,g,P), such that G is a d-ary intree. This is done by Algorithm d-ARY INTREE DECOMPOSITION shown in Figure 11.4. Algorithm d-ARY INTREE DECOMPOSITION executes at most $P-1$ steps. In each step, it determines a subtree of G and removes the tasks of this subforest from G.

Algorithm d-ARY INTREE DECOMPOSITION
Input. A β-restricted instance (G,μ,c,L,o,g,P), such that G is a d-ary intree and $P \neq \infty$.
Output. A decomposition $(G_1,\dots,G_k)$ of (G,μ,c,L,o,g,P), such that $k \leq P$, for all $i \leq k-1$, $\beta \leq \mu(G_i) \leq d(\beta-1)+1$ and if $k<P$, then $\mu(G_k) \leq d(\beta-1)+1$.

```
1.  $i := 1$
2.  while $\mu(G) > d(\beta-1)+1$ and $i < P$
3.     do let $u_i$ be a task of $G$, such that $\beta \leq \mu(T_G(u_i)) \leq d(\beta-1)+1$
4.        $G_i := T_G(u_i)$
5.        $G := G[V(G) \setminus V(G_i)]$
6.        $i := i+1$
7.  $G_i := G$
```

Figure 11.4. Algorithm d-ARY INTREE DECOMPOSITION

Example 11.3.6. Let $(G,L,o,g,3)$ be the instance shown in Figure 11.5 and its decomposition (G_1,G_2,G_3) that is also shown in Figure 11.5. (G_1,G_2,G_3) is constructed by Algorithm d-ARY INTREE DECOMPOSITION using $\beta = 3$. G_1 contains seven tasks, G_2 contains six tasks and G_3 contains the remaining three tasks.

Now we will prove that Algorithm d-ARY INTREE DECOMPOSITION correctly constructs decompositions of intrees.

Lemma 11.3.7. *Let (G,μ,c,L,o,g,P) be a β-restricted instance, such that G is a d-ary intree and $P \neq \infty$. Let $(G_1,\dots,G_k)$ be the sequence of subforests of G constructed by Algorithm d-*ARY INTREE DECOMPOSITION. *Then $(G_1,\dots,G_k)$ is a decomposition of (G,μ,c,L,o,g,P), $k \leq P$, for all $i \leq k-1$, $\beta \leq \mu(G_i) \leq d(\beta-1)+1$ and if $k<P$, then $\mu(G_k) \leq d(\beta-1)+1$.*

Proof. Algorithm d-ARY INTREE DECOMPOSITION executes $k-1 \leq P-1$ steps. Before each step i, G contains at least $d(\beta-1)+2$ task. Then Algorithm d-ARY INTREE DECOMPOSITION chooses a task u_i, such that $\beta \leq \mu(T_G(u_i)) \leq d(\beta-1)+1$. From Lemma 11.3.5, there is such a task. Then G_i equals $T_G(u_i)$ and the tasks of G_i are removed from G. So for all $i \leq k-1$,

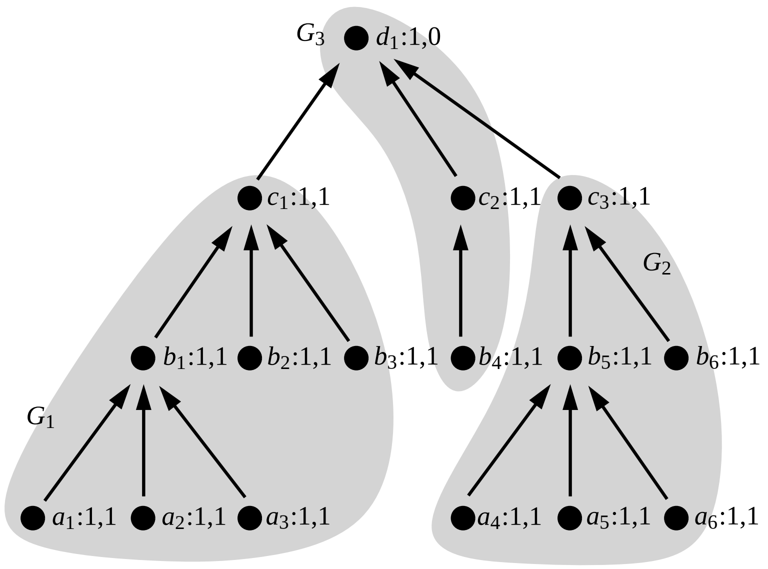

Figure 11.5. A decomposition constructed by Algorithm d-ARY INTREE DECOMPOSITION

$\beta \leq \mu(G_i) \leq d(\beta-1)+1$. After $k-1$ steps, the remaining tasks of G form decomposition tree G_k. Obviously, if $k \leq P-1$, then $\mu(G_k) \leq d(\beta-1)+1$. Otherwise, Algorithm d-ARY INTREE DECOMPOSITION would have executed another step. Because the tasks of decomposition forest G_i are removed after step i, the subforests G_i are pairwise disjoint and a task in G_i cannot have a predecessor in $G_{i+1},\ldots,G_k$. So $(G_1,\ldots,G_k)$ is a decomposition of (G,μ,c,L,o,g,P). $\square$

The time complexity of Algorithm d-ARY INTREE DECOMPOSITION can be determined as follows. Let (G,μ,c,L,o,g,P) be a β-restricted instance, such that G is a d-ary intree and $P \neq \infty$. For each task u of G, compute $\mu(T_G(u))$. These values can be computed in $O(n)$ time for all tasks. By traversing the tasks of G as described in the proof of Lemma 11.3.5, a task u of G, such that $\beta \leq \mu(T_G(u)) \leq d(\beta-1)+1$ can be determined in $O(n)$ time. Then the subtree $T_G(u)$ can be removed by subtracting $\mu(T_G(u))$ from $\mu(T_G(v))$ for all successors v of u. This takes linear time for each subforest, so $O(nP)$ time in total. Using the roots of the decomposition forests, the decomposition forests itself can be constructed in $O(n)$ time by traversing the tasks of G from its root to the sources.

Lemma 11.3.8. *For all β-restricted instances (G,μ,c,L,o,g,P), such that G is a d-ary intree and $P \neq \infty$, Algorithm d-ARY INTREE DECOMPOSITION constructs a decomposition $(G_1,\ldots,G_k)$ of (G,μ,c,L,o,g,P), such that $k \leq P$, for all $i \leq k-1$, $\beta \leq \mu(G_i) \leq d(\beta-1)+1$ and if $k < P$, then $\mu(G_k) \leq d(\beta-1)+1$, in $O(nP)$ time.*

Now we will compute upper bounds on the lengths of the schedules constructed by Algorithm DECOMPOSITION FOREST SCHEDULING using the decompositions constructed by Algorithm d-ARY INTREE DECOMPOSITION. Consider a β-restricted instance (G,μ,c,L,o,g,P), such that G is a d-ary intree and $P \neq \infty$. Assume $(G_1, \dots, G_k)$ is the decomposition of (G, μ, c, L, o, g, P) constructed by Algorithm d-ARY INTREE DECOMPOSITION. Let (σ, π) be the communication-free schedule for (G, μ, c, L, o, g, P) constructed by Algorithm DECOMPOSITION FOREST SCHEDULING using decomposition $(G_1, \dots, G_k)$. Let ℓ be the length of (σ, π) and ℓ^* the length of a minimum-length communication-free schedule for (G, μ, c, L, o, g, P). Assume r_i is the root of G_i. Then for all $i \leq k$,

$$\ell^* \geq \frac{1}{P}\mu(G_i) + \mu(p(r_i, r_k)) \qquad \text{and} \qquad \ell^* \geq \frac{\mu(G)}{P}.$$

From Lemma 11.2.5, (σ, π) has length at most $\max\{\mu(G_k), \max_{1 \leq j < k}(\mu(G_j) + \mu(p(r_j, r_k)))\}$ and communication requirement at most $d(\#G_i + \cdots + \#G_k - 1) - 1 = d(k-1) - 1$. We will consider two cases.

Case 1. $k < P$, or $k = P$ and $\mu(G_k) \leq \max_{1 \leq j < k}(\mu(G_j) + \mu(p(r_j, r_k)))$.
In that case,

$$\begin{aligned} \ell &\leq \max\nolimits_{1 \leq j < k}(\mu(G_j) + \mu(p(r_j, r_k))) \\ &\leq \max\nolimits_{1 \leq j < k}(\ell^* + (1 - \tfrac{1}{P})\mu(G_j)) \\ &\leq \ell^* + (1 - \tfrac{1}{P})d\beta \\ &\leq \ell^* + (1 - \tfrac{1}{P})d\beta\tfrac{P}{\mu(G)}\ell^* \\ &= (1 + d(P-1)\tfrac{\beta}{\mu(G)})\ell^*. \end{aligned}$$

Case 2. $k = P$ and $\mu(G_k) > \max_{1 \leq j < k}(\mu(G_j) + \mu(p(r_j, r_k)))$.
Then

$$\begin{aligned} \ell &\leq \mu(G_k) \\ &\leq \mu(G) - (P-1)\beta \\ &\leq \tfrac{P}{\mu(G)}(\mu(G) - (P-1)\beta)\ell^* \\ &= (P - P(P-1)\tfrac{\beta}{\mu(G)})\ell^*. \end{aligned}$$

Hence the length of (σ, π) is at most

$$\max\{1 + d(P-1)\frac{\beta}{\mu(G)}, P - P(P-1)\frac{\beta}{\mu(G)}\}\ell^*.$$

This bound is as small as possible if $1 + (dP-1)\frac{\beta}{\mu(G)}$ equals $P - P(P-1)\frac{\beta}{\mu(G)}$. In that case, $\beta = \frac{\mu(G)}{d+P}$. Then

$$\ell \leq (P - \frac{P^2 - P}{d+P})\ell^* = (1 + \frac{d(P-1)}{d+P})\ell^* = (d + 1 - \frac{d^2+d}{d+P})\ell^*.$$

From Lemma 11.2.5, the communication requirement of (σ, π) is at most $d(\#G_1 + \cdots + \#G_k - 1) - 1$. Since all decompositions forests G_i are intrees and $k \leq P$, the communication requirement is at most $d(P-1) - 1$. Hence we have proved the following lemma.

Lemma 11.3.9. *There is an algorithm with an $O(nP)$ time complexity that constructs feasible schedules for $\frac{\mu(G)}{d+P}$-restricted instances (G,μ,c,L,o,g,P), such that G is a d-ary intree and $P \neq \infty$, with length at most $(d+1-\frac{d^2+d}{d+P})\ell^* + (d(P-1)-1)(L+o+c_{\max}\max\{o,g\})$, where ℓ^* is the length of a minimum-length schedule for (G,μ,c,L,o,g,P).*

In Section 11.3.1, it was shown how a schedule for a β-restricted instance can be transformed into a schedule for an arbitrary instance. If we choose $\beta = \frac{\mu(G)}{d+P}$, then the number of tasks in G_β is at most $(d+P+2)n$. Using Lemma 11.3.9, we obtain the following result.

Theorem 11.3.10. *There is an algorithm with an $O((d+P)n^2)$ time complexity that constructs feasible schedules for instances (G,μ,c,L,o,g,P), such that G is a d-ary intree and $P \neq \infty$, with length at most $(d+1-\frac{d^2+d}{d+P})\ell^* + (d(P-1)-1)(L+o+c_{\max}\max\{o,g\})$, where ℓ^* is the length of a minimum-length schedule for (G,μ,c,L,o,g,P).*

Proof. Obvious from Lemmas 11.3.9 and 11.3.4. □

11.3.3 Constructing decompositions of arbitrary intrees

In this section, we will construct different decompositions of intrees. These decompositions consist of inforests that are smaller than those constructed by Algorithm d-ARY INTREE DECOMPOSITION and consist of more than one tree. The decomposition algorithm can also be used for inforests by assuming that all roots have the same (dummy) parent. The basis of the decomposition algorithm is the following lemma.

Lemma 11.3.11. *Let G be an intree. Let (G,μ,c,L,o,g,P) be a β-restricted instance. If $\mu(G) \geq \beta$, then G contains a collection of tasks $u_1,\dots,u_k$, such that $\beta \leq \mu(T_G(u_1)) + \cdots + \mu(T_G(u_k)) \leq 2\beta$ and if $k \geq 2$, then $u_1,\dots,u_k$ have the same child v and v has at least $k+1$ parents.*

Proof. It will be proved by induction that for all intrees G, if (G,μ,c,L,o,g,P) is β-restricted and $\mu(G) \geq \beta$, then G contains a collection of tasks $u_1,\dots,u_k$, such that $\beta \leq \mu(T_G(u_1)) + \cdots + \mu(T_G(u_k)) \leq 2\beta$ and if $k \geq 2$, then $u_1,\dots,u_k$ have the same child v and v has at least $k+1$ parents. Let G be an intree. Let r be the root of G. If G contains exactly one task, then $\mu(T_G(r)) = \beta$. So we may assume that G contains $n \geq 2$ tasks. Assume by induction that for all intrees G' with at most $n-1$ tasks, if (G',μ',c',L',o',g',P') is β-restricted and $\mu(G') \geq \beta$, then G' contains a collection of tasks $u_1,\dots,u_k$, such that $\beta \leq \mu'(T_{G'}(u_1)) + \cdots + \mu'(T_{G'}(u_k)) \leq 2\beta$ and if $k \geq 2$, then $u_1,\dots,u_k$ have the same child v and v has at least $k+1$ parents. Assume (G,μ,c,L,o,g,P) is β-restricted and $\mu(G) \geq \beta$.

Case 1. r has indegree one.

Let u be the parent of r. If $\mu(T_G(u)) \leq \beta - 1$, then $\beta \leq \mu(T_G(r)) \leq \mu(r) + \mu(T_G(u)) \leq 2\beta - 1$. Otherwise, $\mu(T_G(u)) \geq \beta$ and, by induction, $T_G(u)$ contains a collection of tasks $v_1,\dots,v_k$ with the same child w, such that $\beta \leq \mu(T_G(v_1)) + \cdots + \mu(T_G(v_k)) \leq 2\beta$ and if $k \geq 2$, then w has at least $k+1$ parents.

Case 2. r has indegree at least two.

Then r has length one. If $\mu(T_G(r)) \leq 2\beta$, then $\beta \leq \mu(T_G(r)) \leq 2\beta$. So we may assume that $\mu(T_G(r)) \geq 2\beta + 1$. Let $u_1, \ldots, u_m$ be the parents of r. Assume $\mu(T_G(u_1)) \geq \cdots \geq \mu(T_G(u_m))$. If $\mu(T_G(u_1)) \geq \beta$, then, by induction, $T_G(u_1)$ contains a collection of tasks $v_1, \ldots, v_k$, such that $\beta \leq \mu(T_G(v_1)) + \cdots + \mu(T_G(v_k)) \leq 2\beta$ and if $k \geq 2$, then $v_1, \ldots, v_k$ have the same child w and w has at least $k+1$ parents. So we will assume that $\mu(T_G(u_1)) \leq \beta - 1$. We know that $\mu(T_G(u_1)) + \cdots + \mu(T_G(u_m)) = \mu(T_G(r)) - 1 \geq 2\beta$. Let k be the smallest integer, such that $\mu(T_G(u_1)) + \cdots + \mu(T_G(u_k)) \geq \beta$. Then $k \leq m-1$ and $\beta \leq \mu(T_G(u_1)) + \cdots + \mu(T_G(u_k)) \leq \beta - 1 + \mu(T_G(u_k)) \leq 2\beta - 2$.

□

Like Algorithm d-ARY INTREE DECOMPOSITION, Algorithm INTREE DECOMPOSITION shown in Figure 11.6 constructs decompositions of arbitrary intrees by repeatedly removing a subforest.

```
Algorithm INTREE DECOMPOSITION
Input. A $\beta$-restricted instance $(G,\mu,c,L,o,g,P)$, such that $G$ is an intree and $P \neq \infty$.
Output. A decomposition $(G_1,\ldots,G_k)$ of $(G,\mu,c,L,o,g,P)$, such that $k \leq P$, for all $i \leq k-1$,
        $\beta \leq \mu(G_i) \leq 2\beta$ and if $k < P$, then $\mu(G_k) \leq 2\beta$.
1.  $i := 1$
2.  while $\mu(G) > 2\beta$ and $i < P$
3.      do let $u_{i,1},\ldots,u_{i,n_i}$ be tasks of $G$ with the same child and $\beta \leq \sum_{j=1}^{n_i} \mu(T_G(u_{i,j})) \leq 2\beta$
4.          $G_i := G[V(T_G(u_{i,1})) \cup \cdots \cup V(T_G(u_{i,n_i}))]$
5.          $G := G[V(G) \setminus V(G_i)]$
6.          $i := i+1$
7.  $G_i := G$
```

Figure 11.6. Algorithm INTREE DECOMPOSITION

Example 11.3.12. Consider the instance $(G,L,o,g,3)$ shown in Figure 11.7 and its decomposition (G_1,G_2,G_3) that is also shown in Figure 11.7. This is the same instance as the one shown in Figure 11.5. (G_1,G_2,G_3) is constructed by Algorithm INTREE DECOMPOSITION using $\beta = 3$. Decomposition trees G_1 and G_3 contain five tasks, G_2 contains the other six tasks. The sizes of these decomposition forests differ less than those of the decomposition forests of the decomposition constructed by Algorithm d-ARY INTREE DECOMPOSITION shown in Figure 11.5.

Now we will prove that Algorithm INTREE DECOMPOSITION correctly constructs decompositions of intrees.

Lemma 11.3.13. *Let (G,μ,c,L,o,g,P) be a β-restricted instance, such that G is an intree and $P \neq \infty$. Let $(G_1,\ldots,G_k)$ be the sequence of subforests of G constructed by Algorithm* INTREE DECOMPOSITION. *Then $(G_1,\ldots,G_k)$ is a decomposition of (G,μ,c,L,o,g,P), $k \leq P$, for all $i \leq k-1$, $\beta \leq \mu(G_i) \leq 2\beta$ and if $k < P$, then $\mu(G_k) \leq 2\beta$.*

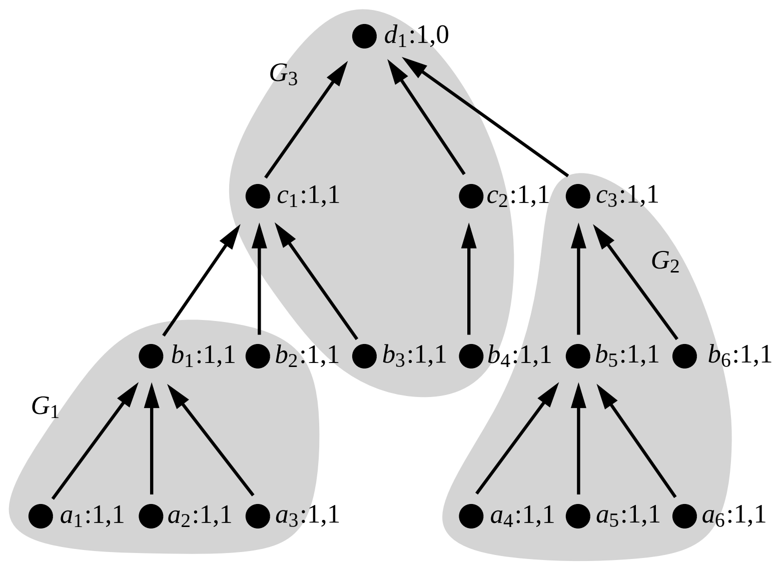

Figure 11.7. A decomposition constructed by Algorithm INTREE DECOMPOSITION

Proof. Algorithm INTREE DECOMPOSITION executes $k-1 \leq P-1$ steps. Before each step i, G contains at least $2\beta+1$ tasks. Then Algorithm INTREE DECOMPOSITION chooses a number of tasks $u_{i,1},\ldots,u_{i,n_i}$ with the same child, such that $\beta \leq \mu(T_G(u_{i,1}))+\cdots+\mu(T_G(u_{i,n_i})) \leq 2\beta$. From Lemma 11.3.11, there is such a collection of tasks. Then G_i equals the subgraph of G induced by the tasks $u_{i,1},\ldots,u_{i,n_i}$ and their predecessors. Hence $\beta \leq \mu(G_i) \leq 2\beta$ for all $i \leq k$. The tasks of G_i are removed from G. After $k-1$ steps, the remaining tasks form decomposition tree G_k. If $k \leq P-1$, then $\mu(G_k) \leq 2\beta$. Otherwise, Algorithm INTREE DECOMPOSITION would have executed another step. Because the tasks of decomposition forest G_i are removed after step i, the subforests G_i are pairwise disjoint and a task in G_i cannot have a predecessor in $G_{i+1},\ldots,G_k$. So $(G_1,\ldots,G_k)$ is a decomposition of (G,μ,c,L,o,g,P). □

The time complexity of Algorithm INTREE DECOMPOSITION can be determined as follows. Let G be an intree. Consider a β-restricted instance (G,μ,c,L,o,g,P), such that $P \neq \infty$. For each task u of G, compute $\mu(T_G(u))$. These values can be computed in $O(n)$ time for all tasks of G. By traversing the tasks of G as described in the proof of Lemma 11.3.11, a number of tasks $u_1,\ldots,u_m$ with the same child, such that $\beta \leq \mu(T_G(u_1))+\cdots+\mu(T_G(u_m)) \leq 2\beta$ can be chosen in $O(n)$ time. Then the subtrees $T_G(u_1),\ldots,T_G(u_n)$ can be removed by subtracting $\mu(T_G(u_1))+\cdots+\mu(T_G(u_m))$ from $\mu(T_G(v))$ for all successors v of u. Since the tasks $u_1,\ldots,u_m$ have the same successors, this takes linear time for each subforest, so $O(nP)$ time in total. Using the roots of the decomposition forests, the decomposition forests itself can be constructed in $O(n)$ time by traversing the tasks of G from its root to the sources.

Lemma 11.3.14. *Let (G,μ,c,L,o,g,P) be a β-restricted instance, such that G is an intree and*

$P \neq \infty$. Then Algorithm INTREE DECOMPOSITION *constructs a decomposition $(G_1,\dots,G_k)$ of (G,μ,c,L,o,g,P), such that $k \leq P$, for all $i \leq k-1$, $\beta \leq \mu(G_i) \leq 2\beta$ and if $k < P$, then $\mu(G_k) \leq 2\beta$, in $O(nP)$ time.*

Now we will prove an upper bound on the length of the schedules constructed by Algorithm DECOMPOSITION FOREST SCHEDULING using the decompositions constructed by Algorithm INTREE DECOMPOSITION. Let (G,μ,c,L,o,g,P) be a β-restricted instance, such that G is an intree and $P \neq \infty$. Let $(G_1,\dots,G_k)$ be the decomposition of (G,μ,c,L,o,g,P) constructed by Algorithm INTREE DECOMPOSITION. Using this decomposition, Algorithm DECOMPOSITION FOREST SCHEDULING constructs a communication-free schedule (σ,π) for (G,μ,c,L,o,g,P). From Lemma 11.2.5, its length ℓ is at most $\max\{\mu(G_k), \max_{1\leq i<k}(\mu(G_i)+\mu(p(r_{i,1},r_{k,1})))\}$, where $r_{i,j}$ is the the root of the jth subtree of G_i.

Let ℓ^* be the length of a minimum-length schedule for (G,μ,c,L,o,g,P). Obviously, for all $i \leq k$,

$$\ell^* \geq \frac{1}{P}\mu(G_i)+\mu(p(r_{i,1},r_{k,1})) \qquad \text{and} \qquad \ell^* \geq \frac{\mu(G)}{P}.$$

The length of (σ,π) equals the completion time of $r_{k,1}$, since $r_{k,1}$ is the root of G. Two cases need to be taken into account.

Case 1. $k < P$, or $k = P$ and $\mu(G_k) \leq \max_{1\leq j<k}(\mu(G_j)+\mu(p(r_{j,1},r_{k,1})))$.

Then

$$\begin{aligned}
\ell &\leq \max_{1\leq j<k}(\mu(G_j)+\mu(p(r_{j,1},r_{k,1})))\\
&\leq \max_{1\leq j<k}(\ell^*+(1-\tfrac{1}{P})\mu(G_j))\\
&\leq \ell^*+2(1-\tfrac{1}{P})\beta\\
&\leq \ell^*+2(1-\tfrac{1}{P})\beta\tfrac{P}{\mu(G)}\ell^*\\
&= (1+2(P-1)\tfrac{\beta}{\mu(G)})\ell^*.
\end{aligned}$$

Case 2. $k = P$ and $\mu(G_k) > \max_{1\leq j<k}(\mu(G_j)+\mu(p(r_{j,1},r_{k,1})))$.

In that case,

$$\begin{aligned}
\ell &\leq \mu(G_k)\\
&\leq \mu(G)-(P-1)\beta\\
&\leq \tfrac{P}{\mu(G)}(\mu(G)-(P-1)\beta)\ell^*\\
&= (P-P(P-1)\tfrac{\beta}{\mu(G)})\ell^*.
\end{aligned}$$

Hence the length of (σ,π) is at most

$$\max\{1+2(P-1)\frac{\beta}{\mu(G)}, P-P(P-1)\frac{\beta}{\mu(G)}\}\ell^*.$$

This bound is as small as possible if $1+(2P-1)\frac{\beta}{\mu(G)}$ equals $P-P(P-1)\frac{\beta}{\mu(G)}$. In that case, $\beta = \frac{\mu(G)}{P+2}$ and

$$\ell \;\leq\; (1+2\frac{P-1}{P+2})\ell^* \;=\; (3-\frac{6}{P+2})\ell^*.$$

From Lemma 11.2.5, the communication requirement of (σ,π) is at most $d(\#G_1+\cdots+\#G_k-1)-1$. Each decomposition forest consists of a collection of trees whose roots have the same parent. From Lemma 11.3.11, if a decomposition forest consists of more than one tree, then we may assume that the child of the roots of these trees has another parent. In addition, decomposition forest G_k consists of one tree. Hence the number of roots of the decomposition forests is at most $\max\{d-1,1\}(P-1)+1=(d-1)(P-1)+1$, where d is the maximum indegree in G.

Moreover, if G is an inforest instead of an intree, then a dummy root can be added. This dummy root is the child of the roots of G. For the constructed intree, a schedule can be constructed. By removing the dummy root, we obtain a feasible schedule for (G,μ,c,L,o,g,P). The indegree of the dummy root need not be taken into account. So we have proved the following lemma.

Lemma 11.3.15. *There is an algorithm with an $O(nP)$ time complexity that constructs feasible schedules for $\frac{\mu(G)}{P+2}$-restricted instances (G,μ,c,L,o,g,P), such that G is a d-ary intree and $P\neq\infty$, with length at most $(3-\frac{6}{P+2})\ell^*+(d(d-1)(P-1)-1)(L+o+c_{\max}\max\{o,g\})$, where ℓ^* is the length of a minimum-length schedule for (G,μ,c,L,o,g,P).*

Using the transformation of schedules for β-restricted instances into schedules for arbitrary instances, we can prove the following result.

Theorem 11.3.16. *There is an algorithm with an $O((d+P)n^2)$ time complexity that constructs feasible schedules for instances (G,μ,c,L,o,g,P), such that G is a d-ary intree and $P\neq\infty$, with length at most $(3-\frac{6}{P+2})\ell^*+(d(d-1)(P-1)-1)(L+o+c_{\max}\max\{o,g\})$, where ℓ^* is the length of a minimum-length schedule for (G,μ,c,L,o,g,P).*

Proof. Obvious from Lemmas 11.3.15 and 11.3.4. □

11.4 Concluding remarks

In Sections 11.3.2 and 11.3.3, two algorithms were presented that construct decompositions of d-ary intrees with arbitrary task lengths. The schedules constructed by Algorithm DECOMPOSITION FOREST SCHEDULING using the decompositions constructed by Algorithms d-ARY INTREE DECOMPOSITION and INTREE DECOMPOSITION consist of two parts: a computation part that depends on the execution lengths of the tasks and the precedence constraints and that is independent of the communication requirements, and a communication part that depends on the communication requirements and that is independent of the execution lengths of the tasks and the precedence constraints.

A decomposition of a d-ary intree constructed by Algorithm d-ARY INTREE DECOMPOSITION is a sequence of intrees. The size of the largest decomposition tree of such a decomposition can be d times as large as the size of the smallest one. Hence the schedules constructed using the decompositions of Algorithm d-ARY INTREE DECOMPOSITION have a large computation part. Moreover, because the total number of roots of the decomposition forests of a decomposition constructed by Algorithm d-ARY INTREE DECOMPOSITION is small, the communication part of these schedules is small.

A decomposition of a d-ary intree constructed by Algorithm INTREE DECOMPOSITION consists of inforests with at most $d-1$ roots. The size of the largest decomposition forest of such a decomposition can be at most twice as large as the size of the smallest one. As a result, the computation part of the schedules constructed using the decompositions of Algorithm INTREE DECOMPOSITION is small. However, because the number of roots of these decomposition forests of a decomposition constructed by Algorithm INTREE DECOMPOSITION can be large, the communication part of these schedules may be large.

Hence the schedules constructed using the decompositions of Algorithms d-ARY INTREE DECOMPOSITION and INTREE DECOMPOSITION give a trade-off between computation and communication.

The decompositions constructed by Algorithms d-ARY INTREE DECOMPOSITION and INTREE DECOMPOSITION are used to construct communication-free schedules in which subsequently the communication operations are introduced. By using different kinds of communication, these decompositions can be used to construct schedules in any model of parallel computation. Using the decompositions constructed by Algorithm d-ARY INTREE DECOMPOSITION, schedules on P processors for d-ary intrees can be constructed whose length is at most the sum of $d+1-\frac{d^2+d}{d+P}$ times the length of a minimum-length schedule on P processors and the duration of $d(P-1)-1$ communication actions. Similarly, the decompositions constructed by Algorithm INTREE DECOMPOSITION can be used to construct schedules on P processors of length at most the sum of $3-\frac{6}{P+2}$ times the length of a minimum-length schedule on P processors and the duration of $d(d-1)(P-1)-1$ communication actions.

Conclusion

12 Conclusion

In this thesis, we studied the complexity of scheduling in two models of parallel computation: the UCT model and the LogP model. In this chapter, we give an overview of the results presented in this thesis and some related problems that remain open. Section 12.1 is concerned with the results presented in Part I, Section 12.2 with those presented in Part II. In Section 12.3, we compare the complexity of scheduling in the UCT model and the LogP model.

12.1 Scheduling in the UCT model

In Part I, we studied the complexity of constructing minimum-tardiness schedules in the UCT model. In Chapters 4, 5 and 6, we presented several polynomial-time algorithms with the same structure: first these algorithms modify the deadlines and second they apply a list scheduling algorithm that uses the modified deadlines. In Chapter 4, consistent deadlines were computed by considering the set of successors of each task. These consistent deadlines are used by a list scheduling algorithm to construct a schedule. The resulting algorithm is proved to be an approximation algorithm with asymptotic approximation ratio $\max\{2, 3-\frac{3}{m}\}$ for scheduling precedence graphs with unit-length tasks and non-positive deadlines on m processors and a 2-approximation algorithm for scheduling precedence graphs with arbitrary task lengths and non-positive deadlines on an unrestricted number of processors. Moreover, the algorithm was shown to be an approximation algorithm with asymptotic approximation ratio $2-\frac{2}{m}$ for scheduling outforests with unit-length tasks and non-positive deadlines on m processors. The algorithm constructs minimum-tardiness schedules for outforests with arbitrary task lengths on an unrestricted number of processors and for outforests with unit-length tasks on two processors.

The least urgent parent property was introduced in Chapter 5. The least urgent parent property was used to construct an approximation algorithm for scheduling inforests. Using a transformation of inforests with consistent deadlines into inforests with the least urgent parent property, a polynomial-time algorithm for scheduling inforests was presented. This algorithm was shown to be a 2-approximation algorithm for scheduling inforests with unit-length tasks and non-positive deadlines on m processors. Moreover, it was proved that minimum-tardiness schedules can be constructed in polynomial time for chain-like task systems with unit-length tasks on m processors and for precedence graphs with the least urgent parent property and arbitrary task lengths on an unrestricted number of processors.

The deadline modification part of the algorithms presented in Chapter 6 considers pairs of tasks instead of individual tasks. It computes pairwise consistent deadlines that may be smaller than the consistent deadlines computed in Chapter 4. The pairwise consistent deadlines are used by a list scheduling algorithm. This approach is used by both algorithms that were presented in Chapter 6. The first algorithm constructs minimum-tardiness schedules for precedence graphs of width two with unit-length tasks on two processors in polynomial time; the second is a polynomial-time algorithm that constructs minimum-tardiness schedules for interval orders with unit-length tasks on m processors.

In Chapter 7, two dynamic-programming algorithms were presented. Both algorithms construct minimum-tardiness schedules for precedence graphs of bounded width in polynomial time. The first constructs minimum-tardiness schedules for precedence graphs of constant width w with unit-length tasks on m processors. For precedence graphs of constant width w with arbitrary task lengths, the second algorithm constructs minimum-tardiness schedules on $m \geq w$ processors. In addition, we proved that for precedence graphs of width w with arbitrary task lengths, constructing minimum-tardiness schedules on $m \leq w-1$ processors is an NP-hard optimisation problem.

Many generalisations of the problems studied in Part I remain open. For example, most algorithms presented in Chapters 4, 5 and 6 are approximation algorithms with a constant approximation ratio for scheduling precedence graphs with unit-length tasks on a restricted number of processors. It would be interesting to determine approximation ratios of similar algorithms for scheduling with arbitrary task lengths or with tasks with execution lengths taken from a restricted set of execution lengths.

The algorithm presented in Chapter 4 for scheduling precedence graphs with unit-length tasks and non-positive deadlines on m processors is an approximation algorithm with asymptotic approximation ratio $\max\{2, 3-\frac{3}{m}\}$. It would be interesting to know whether this algorithm has an asymptotic approximation ratio that is smaller than 2 for scheduling on two processors and whether there are polynomial-time approximation algorithms with better approximation ratios.

In Chapter 5, a 2-approximation algorithm for scheduling inforests was presented. This algorithm uses a transformation of inforests with consistent deadlines to inforests with the least urgent parent property to construct schedules for arbitrary inforests. The algorithm has a constant approximation ratio, because good schedules can be constructed for inforests with the least urgent parent property and because inforests with consistent deadlines can be transformed into inforests with the least urgent parent property without greatly increasing the deadlines. A generalisation could be extending this approach to a larger class of precedence graphs.

In Chapter 6, we considered pairs of tasks to compute smaller deadlines that are met in all in-time schedules. These pairwise consistent deadlines were used to construct minimum-tardiness schedules for precedence graphs of width two on two processors and for interval orders on m processors. If larger sets of tasks are taken into account, then we might be able to compute even smaller deadlines. It would be interesting to determine whether there are classes of precedence graphs for which the consistent deadlines computed by considering larger sets of tasks can be used to construct minimum-tardiness schedules.

12.2 Scheduling in the LogP model

In Part II, the problem of constructing minimum-length schedules in the LogP model was studied. In Chapter 9, we studied the problem of scheduling send graphs in the LogP model. Constructing minimum-length schedules for a send graph on an unrestricted number of processors was shown to be a strongly NP-hard optimisation problem. We presented a polynomial-time 2-approximation algorithm for scheduling send graphs on P processors. Moreover, we showed that if all sinks of a send graph have the same execution length, then a minimum-length schedule for this send graph on P processors can be constructed in polynomial time.

In Chapter 10, two polynomial-time approximation algorithms for scheduling receive graphs were presented. The first is a 3-approximation algorithm that constructs schedules for receive graphs on an unrestricted number of processors. For each constant $k \in \mathbb{Z}^+$, the second algorithm constructs schedules for receive graphs on P processors that are at most $3 + \frac{1}{k+1}$ times as long as minimum-length schedules on P processors. Moreover, we proved that if the execution length of the sources of a receive graph are all equal, then a minimum-length schedule for this receive graph on an unrestricted number of processors can be constructed in polynomial time.

In Chapter 11, two polynomial-time algorithms were presented that use decompositions to construct schedules for inforests. The first constructs schedules for d-ary intrees on P processors that have a length that is at most the sum of $d + 1 - \frac{d^2+d}{d+P}$ times the length of a minimum-length schedule on P processors and the duration of $d(P-1) - 1$ communication actions. The second algorithm constructs schedules for d-ary inforests on P processors with a length that is at most the sum of $3 - \frac{6}{P+2}$ times the length of a minimum-length schedule on P processors and the duration of $d(d-1)(P-1) - 1$ communication actions.

Because scheduling in the LogP model is a new field of research, many open problems remain. In Chapters 9 and 10, we considered very simple precedence graphs (send and receive graphs). Even for these precedence graphs, constructing minimum-length schedules was proved to be strongly NP-hard. It would be interesting to determine special cases for which these problems become solvable in polynomial time. For instance, it is unknown whether minimum-length schedules for send or receive graphs with a constant number of different execution lengths can be constructed in polynomial time. Another generalisation is focusing on a special choice of the LogP parameters (for instance, scheduling with gap zero).

Another interesting open problem is finding polynomial-time approximation algorithms with better approximation ratios than those of the algorithms presented in Chapters 9 and 10. In particular, there should be algorithms with better approximation ratios than those of the algorithms for scheduling receive graphs presented in Chapter 10.

In Chapters 9 and 10, it was shown that if the tasks of a send graph or a receive graph have the same execution length, then a minimum-length schedule can be constructed in polynomial time. An interesting generalisation of these results would be considering the problem of scheduling more general precedence graphs with tasks of equal length. Classes of precedence graphs that resemble send or receive graphs are inforests (outforests) of height three in which the root is the only task with indegree (outdegree) greater than one, and precedence graphs of height two with a constant number of sources (sinks) and an arbitrary number of sinks (sources). For such classes of precedence graphs, it would be interesting to construct approximation algorithms with a constant approximation ratio.

12.3 A comparison of the UCT model and the LogP model

As shown in Chapters 3 and 8, there is a great difference between the UCT model and the LogP model. The UCT model is a model of parallel computation in which communication is represented by delays with a small fixed duration. The LogP model characterises the communication in a parallel computer by latencies, overheads and gaps. In this section, we consider the effects of these types of communication on the complexity of multiprocessor scheduling.

The UCT model captures one aspect of communication in a parallel computer: a communication latency that models the time needed to send a message through the communication network. In a schedule in the UCT model the result of a task is available on all processors one time unit after its completion time. So the result of a task becomes available at the same time on all processors (except the sending processor). No processor is involved in the transfer of data. This makes it easy to construct good schedules in the UCT model: for small precedence graphs, near-optimal schedules can be constructed by hand. In addition, the simplicity of the UCT model allows the computation of good lower bounds on the length (or tardiness) of minimum-length (or minimum-tardiness) schedules. The lower bounds can be used to prove strong approximation ratios for algorithms for scheduling in the UCT model. As a result, there are many approximation algorithms for scheduling in the UCT model with a constant approximation ratio.

The LogP model is a more complicated model of parallel computation that captures several aspects of communication in a parallel computer by four parameters: latency L, overhead o, gap g and number of processors P. The existence of communication operations makes scheduling in the LogP model a very complicated problem. In a schedule in the LogP model the result of a task does not become available on all processors automatically: the processors have to execute communication operations to send and receive data. The data does not become available on all processors at the same time, because a result has to be sent to each processor separately and there is a minimum delay between consecutive communication operations on the same processor. Deciding to which processors a result must be sent is one of the difficulties in scheduling in the LogP model. A second difficulty is due to the gaps between consecutive communication operations on the same processor. If the length of the gaps exceeds that of the overheads (in other words, if g exceeds o), then a processor is available for the execution of tasks between two consecutive send or receive operations. Executing tasks between two consecutive send or receive operations may increase or decrease the schedule length. Hence choosing tasks to be scheduled between a pair of consecutive communication operations is another difficulty in scheduling in the LogP model. These communication-related difficulties make scheduling in the LogP model very complicated: even for small precedence graphs, it is difficult to construct near-optimal schedules by hand. Moreover, since it is not clear which communication operations must be executed in a minimum-length schedule and whether tasks should be scheduled between communication operations, most lower bounds of the length of schedules in LogP model are far below the actual length of minimum-length schedules. As a result, all known approximation algorithms for scheduling in the LogP model either have a parameter-dependent approximation ratio, or a constant approximation ratio for a very restricted class of precedence graphs.

The results show that the effect of the communication requirements is very different for the complexity of scheduling in the UCT model and scheduling in the LogP model. For simple precedence graphs, one can easily construct near-optimal schedules in the UCT model, but it is difficult to construct such schedules in the LogP model. Hence the complexity of scheduling in the UCT model mainly depends on the precedence constraints, whereas the complexity of scheduling in the LogP model is mainly determined by the existence of communication operations and the length of the overheads and the gaps (parameters o and g).

Bibliography

[1] M. Adler, J.W. Byers and R.M. Karp. Parallel sorting with limited bandwidth. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 129–136, 1995.

[2] A. Aggarwal, A.K. Chandra and M. Snir. On communication latency in PRAM computations. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[3] A. Aggarwal, A.K. Chandra and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, March 1990.

[4] H.H. Ali and H. El-Rewini. An optimal algorithm for scheduling interval ordered tasks with communication on n processors. *Journal of Computer and System Sciences*, 51(2):301–306, October 1995.

[5] H. Alt, N. Blum, K. Mehlhorn and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$. *Information Processing Letters*, 37(4):237–240, February 1991.

[6] E. Bampis, J.-C. König and D. Trystram. Optimal parallel execution of complete binary trees and grids into most popular interconnection networks. *Theoretical Computer Science*, 147(1–2):1–18, August 1995.

[7] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the Postal model for message-passing systems. *Mathematical Systems Theory*, 27(5):431–452, September/October 1994.

[8] M. Bartusch, R.H. Möhring and F.J. Radermacher. *m*-machine unit time scheduling: a report on ongoing research. Preprint 192/1988, Fachbereich Mathematik, Technische Universität Berlin, Berlin, Germany, 1988.

[9] H.L. Bodlaender and M.R. Fellows. W[2]-hardness of precedence constrained *k*-processor scheduling. *Operations Research Letters*, 18(2):93–97, September 1995.

[10] R.P. Brent. The parallel evaluation of arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[11] P. Brucker, M.R. Garey and D.S. Johnson. Scheduling equal-length tasks under treelike precedence constraints to minimize maximum lateness. *Mathematics of Operations Research*, 2(3):275–284, August 1977.

[12] P. Chrétienne. A polynomial algorithm to optimally schedule tasks on a virtual distributed system under tree-like precedence constraints. *European Journal of Operational Research*, 43:225–230, 1989.

[13] P. Chrétienne and C. Picouleau. Scheduling with communication delays: a survey. In P. Chrétienne, E.G. Coffman, Jr., J.K. Lenstra and Z. Liu, editors, *Scheduling Theory and its Applications*, Chapter 4, pages 65–90. John Wiley & Sons, Chichester, United Kingdom, 1995.

[14] E.G. Coffman, Jr., M.R. Garey and D.S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, February 1978.

[15] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.

[16] R. Cole and O. Zajicek. The expected advantage of asynchrony. *Journal of Computer and System Sciences*, 51(2):286–300, October 1995.

[17] D. Coppersmith and S. Winograd. Matrix multiplication via algorithmic progressions. *Journal of Symbolic Computation*, 9(3):251–280, March 1990.

[18] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge MA, United States, 1990.

[19] M. Cosnard and A. Ferreira. On the real power of loosely coupled parallel architectures. *Parallel Processing Letters*, 1(2):103–111, 1991.

[20] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM-SIGPLAN Symposium on Principles and Practice of Parallel Processing*, pages 1–12, 1993.

[21] D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramonian and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[22] R.P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, January 1950.

[23] R.G. Downey and M.R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24(4):873–921, August 1995.

[24] A.C. Dusseau, D.E. Culler, K.E. Schauser and R.P. Martin. Fast parallel sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, August 1996.

[25] L. Finta and Z. Liu. Scheduling of parallel programs in single-bus multiprocessor systems. Rapport de recherche 2302, Institut National de Recherche en Informatique et en Automatique, Sophia-Antipolis, France, May 1994.

[26] L. Finta and Z. Liu. Complexity of task graph scheduling with fixed communication capacity. Rapport de recherche 2959, Institut National de Recherche en Informatique et en Automatique, Sophia-Antipolis, France, August 1996.

[27] L. Finta, Z. Liu, I. Milis and E. Bampis. Scheduling UET-UCT series-parallel graphs on two processors. *Theoretical Computer Science*, 162(2):323–340, August 1996.

[28] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[29] D.R. Fulkerson. Note on Dilworth's decomposition theorem for partially ordered sets. *Proceedings of the AMS*, 7:701–702, 1956.

[30] H.N. Gabow and R.E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, April 1985.

[31] M.R. Garey and D.S. Johnson. Scheduling tasks with nonuniform deadlines on two processors. *Journal of the ACM*, 23(6):461–467, July 1976.

[32] M.R. Garey and D.S. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM Journal on Computing*, 6(3):416–426, September 1977.

[33] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York NY, United States, 1979.

[34] P.B. Gibbons. A more practical PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

[35] P.B. Gibbons, Y. Matias and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 638–648, 1994.

[36] P.B. Gibbons, Y. Matias and V. Ramachandran. Efficient low-contention parallel algorithms. *Journal of Computer and System Sciences*, 53(3):417–442, December 1996.

[37] A. Goralčíkova and V. Koubek. A reduct-and-closure algorithm for graphs. In J. Bečvář, editor, *Proceedings of the 8th Conference on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, volume 74, pages 301–307. Springer-Verlag, Berlin, Germany, 1979.

[38] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[39] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[40] F. Guinand. *Ordonnancement avec communications pour architectures multiprocesseurs dans divers modèles d'exécution*. Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, June 1995.

[41] F. Guinand, C. Rapine and D. Trystram. Worst case analysis of Lawler's algorithm for scheduling trees with communication delays. *IEEE Transactions on Parallel and Distributed Systems*, 8(10):1085–1086, October 1997.

[42] F. Guinand and D. Trystram. Optimal scheduling of UECT trees on two processors. Rapport APACHE 3, Laboratoire de Modélisation et Calcul, Institute d'Informatique et de Mathématiques Appliquées de Grenoble, Grenoble, France, November 1993.

[43] L.A. Hall and D.B. Shmoys. Approximation schemes for constrained scheduling problems. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 134–139, 1989.

[44] C. Hanen and A. Munier. Performances d'algorithmes de liste en presence de delais de communication unitaires. Unpublished manuscript, 1995.

[45] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM*, 34(1):144–162, January 1987.

[46] L. Hollerman, T.-S. Hsu, D.R. Lopez and K. Vertanen. Scheduling problems in a practical allocation model. *Journal of Combinatorial Optimization*, 1(2):129–149, 1997.

[47] J.A. Hoogeveen, J.K. Lenstra and B. Veltman. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters*, 16:129–137, 1994.

[48] J.E. Hopcroft and R.M. Karp. A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, December 1973.

[49] T.C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[50] G. Isaak. Scheduling rooted forests with communication delays. *Order*, 11:309–316, 1994.

[51] B.H.H. Juurlink. *Computational Models for Parallel Computers*. PhD thesis, Leiden University, Leiden, the Netherlands, 1997.

[52] K. Kalpakis and Y. Yesha. On the power of the linear array architecture for performing tree-structured computations. *Journal of Computer and System Sciences*, 50(1):1–10, February 1995.

[53] K. Kalpakis and Y. Yesha. Scheduling tree dags on parallel architectures. *Algorithmica*, 15:373–396, 1996.

[54] R.M. Karp, A. Sahay, E.E. Santos and K.E. Schauser. Optimal broadcast and summation in the LogP model. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 142–153, 1993.

[55] I. Kort and D. Trystram. Some results on scheduling trees of height one under logp. Unpublished manuscript, 1997.

[56] S.R. Kosaraju. Parallel evaluation of division-free arithmetic expressions. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 231–239, 1986.

[57] M. Kunde. Nonpreemptive LP-scheduling on homogeneous multiprocessor systems. *SIAM Journal on Computing*, 10(1):151–173, February 1981.

[58] B.J. Lageweg, J.K. Lenstra and A.H.G. Rinnooy Kan. Minimizing maximum lateness on one machine: computational experience and some applications. *Statistica Neerlandica*, 30:25–41, 1976.

[59] E.L. Lawler. Scheduling trees on multiprocessors with unit communication delays. Unpublished manuscript, 1993.

[60] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S.C. Graves, A.H.G. Rinnooy Kan and P.H. Zipkin, editors, *Handbooks in Operations Research and Management Science*, volume 4, Chapter 9, pages 445–522. Elsevier Science Publishers, Amsterdam, the Netherlands, 1993.

[61] J.K. Lenstra, M. Veldhorst and B. Veltman. The complexity of scheduling trees with communication delays. *Journal of Algorithms*, 20(1):157–173, January 1996.

[62] J.Y-T. Leung. Bin packing with restricted piece sizes. *Information Processing Letters*, 31:145–149, 1989.

[63] W. Löwe and W. Zimmermann. Upper time bounds for executing PRAM-programs on the LogP-machine. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 41–50, 1995.

[64] W. Löwe, W. Zimmermann and J. Eisenbiegler. On linear schedules for task graphs for generalized LogP-machines. In C. Lengauer, M. Griebl and S. Gorlatch, editors, *Proceedings of the Third Euro-Par Conference*, Lecture Notes in Computer Science, volume 1300, pages 895–904. Springer-Verlag, Berlin, Germany, 1997.

[65] C. Martel and A. Raghunathan. Asynchronous PRAMs with memory latency. *Journal of Parallel and Distributed Computing*, 23(1):10–26, October 1994.

[66] G. McMahon and M. Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23(3):475–482, May–June 1975.

[67] R.H. Möhring. Computationally tractable classes of ordered sets. In I. Rival, editor, *Algorithms and Order*, pages 105–194. Kluwer Academic Publishers, Dordrecht, the Netherlands, 1989.

[68] R.H. Möhring and M.W. Schäffter. Approximation algorithms for scheduling series-parallel orders subject to unit time communication delays. Preprint 483/1995, Fachbereich Mathematik, Technische Universität Berlin, Berlin, Germany, December 1995.

[69] R.H. Möhring and M.W. Schäffter. Approximation algorithms for scheduling series-parallel orders subject to unit time communication delays. Unpublished manuscript, December 1995.

[70] R.H. Möhring, M.W. Schäffter and A.S. Schulz. Scheduling jobs with communication delays: Using infeasible solutions for approximation. In J. Diaz and M. Serra, editors, *Proceedings of the Fourth Annual European Symposium on Algorithms*, Lecture Notes in Computer Science, volume 1136, pages 76–90. Springer-Verlag, Berlin, Germany, 1996.

[71] A. Munier. Approximation algorithms for scheduling trees with general communication delays. To appear in *Parallel Computing*, 1996.

[72] A. Munier and C. Hanen. An approximation algorithm for scheduling unitary tasks on m processors with communication delays. Rapport LITP 95/12, Institut Blaise Pascal, Université Pierre et Marie Curie, Paris, France, February 1995.

[73] A. Munier and J.-C. König. A heuristic for a scheduling problem with communication delays. *Operations Research*, 45(1):145–147, January–February 1997.

[74] I. Munro and M. Paterson. Optimal algorithms for parallel polynomial evaluation. *Journal of Computer and System Sciences*, 7:189–198, 1973.

[75] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, April 1990.

[76] C.H. Papadimitriou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM Journal on Computing*, 8(3):405–409, August 1979.

[77] C. Picouleau. *Etude de Problèmes d'Optimisation dans les Systèmes Distribués*. Thèse de doctorat, Université Pierre et Marie Curie, Paris, France, 1992.

[78] C. Picouleau. UET-UCT scheduling on 2 processors with constrained communications. Rapport LITP 96/13, Institut Blaise Pascal, Université Pierre et Marie Curie, Paris, France, March 1996.

[79] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18(1):55–71, January 1987.

[80] R. Saad. Scheduling with communication delays. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 18:214–224, 1995.

[81] M.W. Schäffter. *Scheduling Jobs with Communication Delays: Complexity Results and Approximation Algorithms*. Dissertation, Technische Universität Berlin, Berlin, Germany, November 1996.

[82] M. Snir. On parallel searching. *SIAM Journal on Computing*, 14(3):688–708, August 1985.

[83] Yu.N. Sotskov and N.V. Shakhlevich. NP-hardness of shop-scheduling problems with three jobs. *Discrete Applied Mathematics*, 59(3):237–266, May 1995.

[84] H. Stadtherr. Scheduling interval orders with communication delays in parallel. Unpublished manuscript, 1997.

[85] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[86] T.A. Varvarigou, V.P. Roychowdhury, T. Kailath and E. Lawler. Scheduling in and out forests in the presence of communication delays. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1065–1074, October 1996.

[87] B. Veltman. *Multiprocessor scheduling with communication delays*. PhD thesis, Eindhoven University of Technology, Eindhoven, the Netherlands, 1993.

[88] J. Verriet. Scheduling UET, UCT dags with release dates and deadlines. Technical report UU-CS-1995-31, Department of Computer Science, Utrecht University, Utrecht, the Netherlands, September 1995.

[89] J. Verriet. Scheduling interval ordered tasks with non-uniform deadlines. In T. Asano, Y. Igarashi, H. Nagamochi, S. Miyano and S. Suri, editors, *Proceedings of the 7th International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science, volume 1178, pages 366–375. Springer-Verlag, Berlin, Germany, 1996.

[90] J. Verriet. Scheduling interval orders with release dates and deadlines. Technical report UU-CS-1996-12, Department of Computer Science, Utrecht University, Utrecht, the Netherlands, March 1996.

[91] J. Verriet. The complexity of scheduling graphs of bounded width subject to non-zero communication delays. Technical report UU-CS-1997-01, Department of Computer Science, Utrecht University, Utrecht, the Netherlands, January 1997.

[92] J. Verriet. Scheduling tree-structured programs in the LogP model. Technical report UU-CS-1997-18, Department of Computer Science, Utrecht University, Utrecht, the Netherlands, June 1997.

[93] J. Verriet. Scheduling interval-ordered tasks with non-uniform deadlines subject to non-zero communication delays. To appear in *Parallel Computing*, 1998.

[94] M. Yue. On the exact upper bound for the multifit processor scheduling algorithm. *Annals of Operations Research*, 24(1–4):233–259, October 1990.

[95] W. Zimmermann and W. Löwe. An approach to machine-independent parallel programming. In B. Buchberger and J. Volkert, editors, *Proceedings of the Third Joint Conference on Vector and Parallel Processing*, Lecture Notes in Computer Science, volume 854, pages 277–288. Springer-Verlag, Berlin, Germany, 1994.

Acknowledgements

Throughout the nearly four years during which I was working as a research assistant, Marinus Veldhorst was my supervisor. I want to thank him for introducing me to the field of multiprocessor scheduling. Moreover, I thank him for many valuable discussions and for carefully reading all papers I have written including this thesis.

I thank Jan van Leeuwen for reading an earlier version of this thesis and suggesting many improvements. I also want to thank Iskander Kort, Peter van Rossum and Markus Schäffter for proofreading earlier versions of parts of this thesis. The discussions with Markus has been especially helpful to improve the presentation of the results.

I wish to thank the members of the review committee, Hans Bodlaender, Peter Hilbers, Jan Karel Lenstra, Denis Trystram and Harry Wijshoff, for reviewing this thesis and their useful comments.

Samenvatting

Multiprocessor scheduling houdt zich bezig met de planning van de uitvoering van computerprogramma's op een parallelle computer. Een *computerprogramma* kan worden gezien als een collectie instructies die gegroepeerd zijn in *taken*. Een parallelle computer is een computer met meerdere *processoren* die verbonden zijn door een *communicatie-netwerk*. Elke processor kan taken van een computerprogramma uitvoeren.

Tijdens de uitvoering van een computerprogramma op een parallelle computer wordt elke taak één maal uitgevoerd. In het algemeen kunnen de taken van een computerprogramma niet in een willekeurige volgorde worden uitgevoerd: het resultaat van een taak kan nodig zijn om een andere taak uit te voeren. Zulke taken worden *data-afhankelijk* genoemd. De data-afhankelijkheden definiëren de structuur van het computerprogramma: als taak u_2 het resultaat van taak u_1 nodig heeft, dan kan u_2 pas worden uitgevoerd nadat u_1 is voltooid. Als er geen data-afhankelijkheid bestaat tussen twee taken, dan kunnen ze in willekeurige volgorde of tegelijkertijd worden uitgevoerd.

Als twee data-afhankelijke taken u_1 en u_2 op verschillende processoren worden uitgevoerd, dan moet het resultaat van u_1 naar de processor die u_2 uitvoert worden overgebracht. Dit transport van informatie wordt *communicatie* genoemd. Het resultaat van u_1 kan naar een andere processor worden overgebracht door het sturen van berichten door het communicatie-netwerk.

Een *schedule* geeft voor elke taak aan welke processor hem uitvoert en op welk tijdstip. Het doel van multiprocessor scheduling is het construeren van een *schedule* van zo kort mogelijke duur, rekening houdend met de communicatie veroorzaakt door de data-afhankelijkheden tussen de taken. De duur van een schedule wordt in grote mate bepaald door de hoeveelheid communicatie in het schedule: de duur van een schedule kan toenemen doordat een processor lange tijd geen taken kan uitvoeren, omdat hij staat te wachten op het resultaat van een taak die op een andere processor wordt uitgevoerd.

Omdat de wijze waarop processoren van parallelle computers communiceren verschilt per computer, is het uiterst moeilijk om op efficiënte wijze goede schedules te construeren voor een computerprogramma op een parallelle computer. Daarom wordt in het algemeen een model van een parallelle computer gebruikt in plaats een echte parallelle computer. Zo'n model wordt een *parallel berekeningsmodel* genoemd. In een parallel berekeningsmodel kan men zich concentreren op die aspecten van communicatie die een grote invloed hebben op de kwaliteit van een schedule. Dit geeft de mogelijkheid deze aspecten beter te begrijpen.

In dit proefschrift worden twee parallelle berekeningsmodellen beschouwd: het UCT model en het LogP model. Het UCT model richt zich op het bestuderen van één aspect van communicatie: een tijdvertraging die nodig is om resultaten tussen processoren te transporteren. Het LogP model is een model dat meerdere aspecten van communicatie in acht neemt: door middel van een geschikt gekozen invulling van zijn parameters L, o, g en P kan het LogP model de communicatie in vele parallelle computers modelleren.

Communicatie in het UCT model werkt als volgt. Als taak u_2 het resultaat van taak u_1 nodig heeft en deze taken zijn op verschillende processoren uitgevoerd, dan moet er een vertraging van tenminste één tijdstap zijn tussen de tijd waarop u_1 wordt voltooid en de tijd waarop u_2 start.

Deze vertraging is nodig om het resultaat van u_1 naar de processor die u_2 uitvoert te sturen. Als u_1 en u_2 op dezelfde processor worden uitgevoerd, dan is het resultaat van u_1 al op de juiste processor beschikbaar en is er geen vertraging nodig. In dat geval kan u_2 direct na u_1 worden uitgevoerd.

Communicatie in het LogP model is veel ingewikkelder. Beschouw wederom twee data-afhankelijke taken u_1 en u_2 die op verschillende processoren worden uitgevoerd. Neem aan dat het resultaat van u_1 moet worden getransporteerd naar de processor die u_2 uitvoert. In vele gevallen kan het transporteren van het resultaat van een taak niet met één bericht, maar zijn meerdere berichten nodig. Deze moeten naar de processor die u_2 uitvoert worden gestuurd. Het versturen van één bericht kost o tijdstappen op de processor die u_1 uitvoert; het ontvangen ervan kost o tijdstappen op de processor die u_2 uitvoert. Daarnaast kan elke processor ten hoogste één bericht versturen of ontvangen in elke g opeenvolgende tijdstappen en is er een vertraging van precies L tijdstappen tussen het versturen en het ontvangen van een bericht.

In het eerste deel van dit proefschrift (hoofdstukken 3, 4, 5, 6 en 7) worden algoritmen beschreven die op efficiënte wijze schedules in het UCT model construeren. In hoofdstuk 4 wordt een algoritme beschreven dat goede schedules construeert voor willekeurige computerprogramma's. Voor computerprogramma's met een *outforest-structuur* construeert dit algoritme optimale schedules. In hoofdstuk 5 beschrijven we algoritmen die goede schedules construeren voor computerprogramma's met een *inforest-structuur*. De algoritmen die worden beschreven in hoofdstukken 6 en 7 construeren optimale schedules voor computerprogramma's waarin het maximum aantal paarsgewijs data-onafhankelijke taken klein is en voor computerprogramma's met een *interval order-structuur*.

Het tweede deel van dit proefschrift (hoofdstukken 8, 9, 10 en 11) houdt zich bezig met scheduling in het LogP model. In hoofdstukken 9 en 10 bewijzen we dat het construeren van optimale schedules voor computerprogramma's met een zeer eenvoudige boomstructuur (*send graph-structuur* of *receive graph-structuur*) waarschijnlijk niet op efficiënte wijze mogelijk is. In deze hoofdstukken worden efficiënte algoritmen beschreven die goede (maar niet noodzakelijk optimale) schedules construeren voor computerprogramma's met een dergelijke structuur. In hoofdstuk 11 worden decompositie-algoritmen gebruikt om op efficiënte wijze schedules te construeren voor computerprogramma's met een algemene *boomstructuur*.

Het blijkt dat optimale schedules in het UCT model op efficiënte wijze kunnen worden geconstrueerd als de structuur van de computerprogramma's eenvoudig is (bijvoorbeeld computerprogramma's met een boomstructuur). De eenvoudige aard van de communicatie in het UCT model maakt dit mogelijk. Vandaar dat de complexiteit van scheduling in het UCT model met name bepaald wordt door de structuur van de computerprogramma's. Daarentegen maakt de communicatie het moeilijk om goede schedules in het LogP model te construeren, zelfs als de structuur van de computerprogramma's zeer eenvoudig is (bijvoorbeeld computerprogramma's met een send graph-structuur). Hieruit blijkt dat de complexiteit van scheduling in het LogP model in grote mate wordt bepaald door de ingewikkelde vorm van communicatie in dit model.

Curriculum vitae

Jacobus Hendrikus Verriet

3 oktober 1970
Geboren te Ubbergen

augustus 1983 - juni 1989
Voorbereidend Wetenschappelijk Onderwijs aan het Canisius College-Mater Dei te Nijmegen
Diploma behaald op 8 juni 1989

september 1989 - augustus 1994
Studie Informatica aan de Katholieke Universiteit Nijmegen
Propedeutisch diploma (cum laude) behaald op 31 augustus 1990
Doctoraal diploma (cum laude) behaald op 26 augustus 1994

september 1994 - augustus 1998
Assistent in Opleiding bij de Vakgroep Informatica van Universiteit Utrecht

Index