Chapter 7

Scheduling

7.1 Introduction

This chapter presents basic (but important) results on task graph scheduling. We start with a motivating example before providing a quick overview of models and complexity results.

7.1.1 Where Do Task Graphs Come From?

Consider the following algorithm to solve the linear system $Ax = b$, where $A$ is an $n \times n$ nonsingular lower triangular matrix and $b$ is a vector with $n$ components:

\begin{verbatim}
for $i = 1$ to $n$ do
    Task $T_{i,i}$: $x_i = b_i/a_{i,i}$
for $j = i + 1$ to $n$ do
    Task $T_{j,i}$: $b_j = b_j - a_{j,i} \times x_i$
\end{verbatim}

For a given value of $i$, $1 \leq i \leq n$, each task $T_{i,i}$ represents some computations executed during the $i$-th iteration of the external loop. The computation of $x_i$ is performed first (task $T_{i,i}$). Then, each component $b_j$ of vector $b$ such that $j > i$ is updated (task $T_{j,i}$).

In the original program, there is a total precedence order between tasks. Let us write $T \prec_{seq} T'$ if task $T$ is executed before task $T'$ in the original sequential code. We have

$$T_{1,1} \prec_{seq} T_{1,2} \prec_{seq} T_{1,3} \prec_{seq} \cdots \prec_{seq} T_{1,n} \prec_{seq} T_{2,2} \prec_{seq} T_{2,3} \prec_{seq} \cdots \prec_{seq} T_{n,n}.$$  

However, there are independent tasks that can be executed in parallel. Intuitively, independent tasks are tasks whose execution orders can be interchanged without modifying the result of the program execution. A necessary condition for tasks to be independent is that they do not update the same variable. They can read the same value, but they cannot write into the same memory location (otherwise there would be a race condition and the result would be non-deterministic). For instance, tasks $T_{1,2}$ and $T_{1,3}$ both read $x_1$ but modify distinct components of $b$, hence they are independent.
7.1. Introduction

automatically satisfied. We say that \( T \) is a predecessor of \( T' \) if \( T < T' \) and if there is no task \( T'' \) in between, i.e., such that \( T < T'' \) and \( T'' < T' \). We will give formal definitions in the next section. In our example, the predecessor relationships are as follows:

- \( T_{i,j} \prec T_{i',j} \) for \( 1 \leq i < j \leq n \) (the computation of \( x_i \) must be done before updating \( b_j \) at step \( i \) of the outer loop).
- \( T_{i,j} \prec T_{i+1,j} \) for \( 1 \leq i < j \leq n \) (updating \( b_j \) at step \( i \) of the outer loop must be done before reading it at step \( i + 1 \)).

We end up with the graph shown in Figure 7.1. We will use this graph several times in this chapter for illustrative purposes.

7.1.2 Overview

This chapter presents classic theorems and algorithms from scheduling theory. The communication model used by this theory is rather unrealistic, but it makes it possible to obtain fundamental complexity results.

We start with the most simple (one might say crude) model where all communication delays between processors are neglected. We introduce basic definitions in Section 7.2. When there is no restriction on the number of available processors, optimal schedules can be found in polynomial time, as shown in Section 7.3. Section 7.4 deals with a limited number of processors; the scheduling problem becomes NP-complete, and so-called list scheduling heuristics are the typical approach. An elegant and powerful theorem shows that any list scheduling algorithm generates a schedule that is no longer than twice the optimal schedule (Section 7.4.2). We continue on the theoretical side: In Section 7.4.5, we discuss the scheduling of independent tasks, and we derive arbitrarily good approximation algorithms, i.e., polynomial algorithms whose performance can be guaranteed within a \((1 + \epsilon)\) of the optimal, for any arbitrary \( \epsilon > 0 \).

Next we move to the classical scheduling model in which communication costs are taken into account each time two dependent tasks are not assigned to the same processor. We detail this model in Section 7.5. In this case, even the problem with unlimited processors is NP-complete, as explained in Section 7.6. We present heuristics for \( p \) identical processors in Section 7.7 and briefly discuss how to extend these heuristics to handle heterogeneous processors in Section 7.8.

\[ \text{FIGURE 7.1: Task graph for the triangular system (} n = 6 \text{).} \]

7.2 Scheduling Task Graphs

**Definition 7.1.** A task system, or task graph, is a directed vertex-weighted graph \( G = (V, E, w) \), where:

- the set \( V \) of vertices represents the tasks (note that \( V \) is finite).
- the set \( E \) of edges represents precedence constraints between tasks: \( e = (u, v) \in E \) if and only if \( u \prec v \).
- the weight function \( w : V \rightarrow \mathbb{N}^* \) gives the weight (or duration) of each task. Task weights are assumed to be positive integers.\(^\text{1}\)

For the triangular system (Figure 7.1), we can assume that all tasks have equal weight; let \( w(T_{i,j}) = 1 \) for \( 1 \leq i \leq j \leq n \). We could also consider

\[ \text{This is not a restriction; task weights can be rational numbers. However, because there is a finite number of tasks, the weights can always be scaled up to integers.} \]

\(^\text{1}\)Well, “must” is slightly exaggerated, because the addition is an associative and commutative operation. This is true the way the program is written, reusing the same memory location \( b_j \) for all the updates. Technically, we have an output dependence that can be removed using standard techniques [92, 119].
that a division is more costly than a multiply-add and give extra weight to the diagonal tasks $T_{i,i}$.

A schedule $\sigma$ of a task system is a function that assigns a start time to each task:

**DEFINITION 7.2.** A schedule of a task system $G = (V, E, w)$ is a function $\sigma : V \rightarrow \mathbb{N}^+$ such that $\sigma(u) + w(u) \leq \sigma(v)$ whenever $e = (u, v) \in E$.

In other words, a schedule must preserve the dependence constraints induced by the precedence relation $\prec$ and embodied by the edges of the dependence graph; if $u \prec v$, then the execution of $u$ begins at time $\sigma(u)$ and requires $w(u)$ units of time, and the execution of $v$ at time $\sigma(v)$ must start after the end of the execution of $u$.

Often there are other constraints that must be met by schedules, namely, resource constraints. When there is an infinite number of processors, we say that we have a problem with unlimited processors, denoted $\text{Pb}(\infty)$. When there is only a fixed number $p$ of available processors, we speak of a problem with limited processors, $\text{Pb}(p)$. In this case we need an allocation function $\text{alloc} : V \rightarrow P$, where $P = \{1, \ldots, p\}$ denotes the set of available processors.

This function assigns a target processor to each task. The resource constraints simplify to specify that no processor can be allocated more than one task at the same time. This translates into the following conditions:

$$\text{alloc}(T) = \text{alloc}(T') \iff \sigma(T) + w(T) \leq \sigma(T') \quad \text{or} \quad \sigma(T') + w(T') \leq \sigma(T).$$

This condition expresses the fact that if two tasks $T$ and $T'$ are allocated to the same processor, then their executions cannot overlap in time.

Given a task system, there is a basic condition for a schedule to exist, regardless of resource constraints:

**THEOREM 7.1.** Let $G = (V, E, w)$ be a task system. There exists a schedule if and only if $G$ contains no cycle.

Proof. Clearly, if there is a cycle $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_1$ (where $v_1$ depends on itself), and a schedule $\sigma$ would satisfy $\sigma(v_1) + w(v_1) \leq \sigma(v_1)$, which is impossible because we assumed $w(v_1) > 0$.

Conversely, if $G$ has no cycle, then there exist vertices with no predecessors ($V$ is finite) and we can thus topologically sort the vertices and schedule them one after the other (i.e., assuming a single processor) according to the topological order. Formally, if $v_{\pi(1)}, \ldots, v_{\pi(n)}$ is the ordered list of vertices obtained by the topological sort, let $\sigma(v_{\pi(i)}) = 0$ and $\sigma(v_{\pi(i)}) = \sigma(v_{\pi(i-1)}) + w(v_{\pi(i-1)})$ for $2 \leq i \leq n$. Dependence constraints are respected, because if $v_{\pi(i)} \prec v_{\pi(j)}$, then the topological sort ensures that $\pi(i) < \pi(j)$.

In fact we need no more processors than the total number of tasks.

Theorem 7.1 states that scheduling deals with directed acyclic graphs (or DAGs).

**DEFINITION 7.3.** A DAG $G = (V, E, w)$ is a task system (as in Definition 7.2) where $G$ is a directed acyclic graph. We have the following:

1. Let $\sigma$ be a schedule for $G$. Assume $\sigma$ uses at most $p$ processors (let $p = \infty$ if the processors are unlimited). The makespan $MS(\sigma, p)$ of $\sigma$ is its total execution time:

$$MS(\sigma, p) = \max_{v \in V}(\sigma(v) + w(v)) - \min_{v \in V}(\sigma(v)).$$

2. $\text{Pb}(p)$ is the problem of determining a schedule $\sigma$ of minimal makespan $MS(\sigma, p)$ assuming $p$ processors (let $p = \infty$ if the processors are unlimited). Let $MS_{\text{opt}}(p)$ be the value of the makespan of an optimal schedule with $p$ processors:

$$MS_{\text{opt}}(p) = \min_p MS(\sigma, p).$$

If the first task is scheduled at time 0, which is a common assumption, the expression of the makespan can be reduced to $MS(\sigma, p) = \max_{e \in E}(\sigma(v) + w(e))$. We extend weights to paths in $G$ as usual; if $\Phi = (T_1, T_2, \ldots, T_n)$ denotes a path in $G$, then $w(\Phi) = \sum_{i=1}^{n} w(T_i)$. Because schedules respect dependencies, we have the following easy bound on the makespan:

**PROPOSITION 7.1.** Let $G = (V, E, w)$ be a DAG and $\sigma$ a schedule for $G$ with $p$ processors. Then, $MS(\sigma, p) \geq w(\Phi)$ for all paths $\Phi$ in $G$.

Proof. Consider any path $\Phi = (T_1, T_2, \ldots, T_n)$ in $G$: $e = (T_i, T_{i+1}) \in E$ for $1 \leq i < n$. Then, $\sigma(T_i) + w(T_i) \leq \sigma(T_{i+1})$ for $1 \leq i < n$, and thus $MS(\sigma, p) \geq w(T_1) + \sigma(T_n) - \sigma(T_1) \geq \sum_{i=1}^{n} w(T_i) = w(\Phi)$.

Our last definition introduces the notions of speedup and efficiency for schedules (see [70] for a detailed discussion of speedup and efficiency):

**DEFINITION 7.4.** Let $G = (V, E, w)$ be a DAG, and $\sigma$ a schedule for $G$ with $p$ processors:

1. The speedup is the ratio $s(\sigma, p) = \frac{\text{Seq}}{MS(\sigma, p)}$, where $\text{Seq} = \sum_{v \in V} w(v)$ is the sum of all task weights.

2. The efficiency is the ratio $e(\sigma, p) = \frac{s(\sigma, p)}{p} = \frac{\text{Seq}}{p \cdot MS(\sigma, p)}$.

Seq is the optimal execution time $MS_{\text{opt}}(1)$ of a schedule with a single processor. We have the following well-known result:

**THEOREM 7.2.** Let $G = (V, E, w)$ be a task system. For any schedule $\sigma$ with $p$ processors,

$$0 \leq e(\sigma, p) \leq 1.$$
7.2. Scheduling Task Graphs

Proof. Consider the execution of $\sigma$ as illustrated in Figure 7.2 (this is a fictitious example, not related to the triangular system example). At any time during execution, some processors are active, and some are idle. At the end, all tasks have been processed. Let $\text{Idle}$ denote the cumulated idle time of the $p$ processors during the whole execution. Because $\text{Seq}$ is the sum of all task weights, the quantity $\text{Seq} + \text{Idle}$ is equal to the area of the rectangle in Figure 7.2, i.e., the product of the number of processors by the makespan of the schedule: $\text{Seq} + \text{Idle} = p \times \text{MS}(\sigma, p)$. Hence, $\epsilon(\sigma, p) = \frac{\text{Seq} + \text{Idle}}{\text{MS}(\sigma, p)} \leq 1$. \hfill $\Box$

![Figure 7.2: Active and idle processors during execution.](image)

Another way to state Theorem 7.2 is to say that the speedup with $p$ processors is always bounded by $p$. No superlinear speedup with our model! Here is an easy result to conclude this section: The more processors, the smaller (or equal) the optimal makespan.

**THEOREM 7.3.** Let $G = (V, E, w)$ be a task system. We then have

$$\text{Seq} = \text{MS}_{\text{opt}}(1) \geq \ldots \geq \text{MS}_{\text{opt}}(p) \geq \text{MS}_{\text{opt}}(p + 1) \geq \ldots \geq \text{MS}_{\text{opt}}(\infty).$$

Proof. The proof is straightforward. Consider an optimal schedule $\sigma$ with $p$ processors, and view it as a schedule with $p + 1$ processors where the last processor is kept idle. Then, $\text{MS}(\sigma, p + 1) = \text{MS}(\sigma, p) = \text{MS}_{\text{opt}}(p)$; hence, $\text{MS}_{\text{opt}}(p + 1) \leq \text{MS}_{\text{opt}}(p)$.

Theorem 7.3 can be refined as follows. The number of processors actually used by a schedule $\sigma$ is $|\text{alloc}(V)|$, i.e., the number of processors that execute at least one task. If we define $\text{MS}_{\text{p}}(p)$ as the minimum makespan of all schedules that use exactly $p$ processors, we have $\text{MS}(\sigma, p) = \text{MS}_{\text{opt}}(p)$ for $1 \leq p \leq |V|$, so that Theorem 7.3 holds when replacing $\text{MS}_{\text{opt}}(p)$ by $\text{MS}_{\text{p}}(p)$. Intuitively, it cannot hurt to make use of more processors in a model where communication costs are not taken into account! In Section 7.5, we introduce communication costs, and we give an example where $\text{MS}_{\text{p}}(p) < \text{MS}(\sigma, p)$ with $p < p'$; the refined version of Theorem 7.3 is no longer true with this new model.

We are now ready to address the search for optimal schedules. Not surprisingly, it turns out that the problem $\text{Pb}(p)$ with limited processors is more difficult than $\text{Pb}(\infty)$, whose solution is explained in the next section.

7.3 Solving $\text{Pb}(\infty)$

Let $G = (V, E, w)$ be a given DAG and assume unlimited processors. Remember that a schedule $\sigma$ for $G$ is said to be optimal if its makespan $\text{MS}(\sigma, \infty)$ is minimal, i.e., if $\text{MS}(\sigma, \infty) = \text{MS}_{\text{opt}}(\infty)$.

**DEFINITION 7.5.** Let $G = (V, E, w)$ be a DAG.

1. For $v \in V$, $\text{PRED}(v)$ denotes the set of all immediate predecessors of $v$, and $\text{SUCC}(v)$ the set of all its immediate successors.
2. $v \in V$ is an entry (top) vertex if and only if $\text{PRED}(v) = \emptyset$.
3. $v \in V$ is an exit (bottom) vertex if and only if $\text{SUCC}(v) = \emptyset$.
4. For $v \in V$, the top level $t_l(v)$ is the largest weight of a path from an entry vertex to $v$, excluding the weight of $v$.
5. For $v \in V$, the bottom level $b_l(v)$ is the largest weight of a path from $v$ to an output vertex, including the weight of $v$.

In the example of the triangular system, there is a single entry vertex, $T_{1,1}$, and a single exit vertex, $T_{2,3}$. The top level of $T_{1,1}$ is 0, and $t_l(T_{2,3}) = t_l(T_{1,1}) + w(T_{1,1}) = 1$. We have

$$t_l(T_{2,3}) = \max\{w(T_{1,1}) + w(T_{2,2}), w(T_{1,1}) + w(T_{2,3})\} = 3$$

because there are two paths from the entry vertex to $T_{2,3}$.

The top level of a vertex can be computed by a traversal of the DAG; the top level of an entry vertex is 0, while the top level of a non-entry vertex $v$ is

$$t_l(v) = \max\{t_l(u) + w(u); u \in \text{PRED}(v)\}.$$

Similarly, $b_l(v) = \max\{b_l(u); u \in \text{SUCC}(v)\} + w(v)$ (and $b_l(v) = w(v)$ for an exit vertex $v$). The top level of a vertex is the earliest possible time at which it can be executed, while its bottom level represents a lower bound of the remaining execution time once starting its execution. This can be stated more formally as follows.

**THEOREM 7.4.** Let $G = (V, E, w)$ be a DAG and define $\sigma_{\text{free}}$, as follows:

$$\forall v \in V, \sigma_{\text{free}}(v) = t_l(v) .$$

Then, $\sigma_{\text{free}}$ is an optimal schedule for $G$. 

---

Chapter 7. Scheduling

Chapter 7. Scheduling
7.3. Solving P\textsubscript{b}(\infty)  

Proof. The proof has two parts. First we show that \( \sigma_{\text{free}} \) is indeed a schedule, then we derive its optimality. Both are easy:

1. \( \sigma_{\text{free}} \) respects all dependence constraints by construction; if \( (u,v) \in E \), then \( u \in \text{PRED}(v) \) and the constraint \( \sigma_{\text{free}}(v) \geq \sigma_{\text{free}}(u) + w(u) \) is taken into account in the computation of \( h(v) = \sigma_{\text{free}}(v) \).

2. To prove that \( \sigma_{\text{free}} \) is optimal, we use a topological sort order of the vertices of \( G \), and prove by induction that all vertices are scheduled as soon as possible, i.e., as soon as the execution of all their predecessors has been completed.

The free schedule \( \sigma_{\text{free}} \) is also known as the as soon as possible (ASAP) schedule.

From Theorem 7.4 we have

\[
MS_{\text{opt}}(\infty) = MS(\sigma_{\text{free}}, \infty) = \max_{v \in V} \{h(v) + w(v)\}.
\]

Hence, \( MS_{\text{opt}}(\infty) \) is simply the maximal weight of a path in the graph. Note that \( \sigma_{\text{free}} \) is not the only optimal schedule. For example, the \( \text{as late as possible} \) (ALAP) schedule \( \sigma_{\text{free}} \) is also optimal. We define \( \sigma_{\text{free}} \) as follows:

\[\forall v \in V, \sigma_{\text{free}}(v) = MS(\sigma_{\text{free}}, \infty) - b(v).\]

To understand the definition, note that \( b(v) \) is the maximal weight of a path from \( v \) to exit nodes, hence the need to start the execution of \( v \) no later than \( MS(\sigma_{\text{free}}, \infty) \) time units.

**Corollary 7.1.** Let \( G = (V,E,w) \) be a directed acyclic graph. \( P\textsubscript{b}(\infty) \) can be solved in time \( O(|V| + |E|) \).

Proof. From Theorem 7.4 we know that the optimal schedule \( \sigma_{\text{free}} \) can be computed using top levels and that \( MS_{\text{opt}}(\infty) \) is the maximal weight of a path in the graph. Because \( G \) is acyclic, these quantities can be computed by a traversal of the graph, hence the complexity \( O(|V| + |E|) \).

Going back to the triangular system (Figure 7.1), because all tasks have weight 1, the weight of a path is equal to its length plus 1. The longest path is

\[T_{n,1} \rightarrow T_{n,2} \rightarrow \cdots \rightarrow T_{n-1,n-1} \rightarrow T_{n-1,n} \rightarrow T_{n,n},\]

whose weight is \( 2n - 1 \). Note that we do not need as many processors as tasks to achieve execution within \( 2n - 1 \) time units. For example, we can use only \( n - 1 \) processors. Let \( 1 \leq i \leq n \), at time \( 2i - 2 \), processor \( P_i \) starts the execution of task \( T_{i,i} \), while at time \( 2i - 1 \), the first \( n - i \) processors \( P_1, P_2, \ldots, P_{n-i} \) execute tasks \( T_{n,j}, 1 \leq j \leq n \).

7.4 Solving P\textsubscript{b}(p)

Let \( G = (V,E,w) \) be a DAG. It turns out that the problem with limited processors \( P\textsubscript{b}(p) \) is NP-complete. Hence, a polynomial algorithm to determine an optimal schedule is unlikely to exist (unless \( \text{P} = \text{NP} \)). Therefore, in the next sections we will introduce heuristics to compute approximate solutions.

7.4.1 NP-completeness of P\textsubscript{b}(p)

In the following proofs, we use two well-know partition problems, although their names are somewhat misleading:

- **2-Partition** Given \( n \) positive integer numbers \( \{a_1, a_2, \ldots, a_n\} \), is there a subset \( I \) of indices such that \( \sum_{i \in I} a_i = \sum_{i \notin I} a_i \)?

- **3-Partition** Given \( 3n \) positive integer numbers \( \{a_1, a_2, \ldots, a_{3n}\} \) and a bound \( B \), assuming that \( \frac{a}{3} < a_i < \frac{2a}{3} \) for all \( i \) and that \( \sum_{i \in I} a_i = nB \), is there a partition of these numbers into \( n \) subsets \( I_1, I_2, \ldots, I_n \) of sum \( B \)? In other words, are there \( n \) subsets \( I_1, I_2, \ldots, I_n \) such that \( I_1 \cup I_2 \cup \ldots \cup I_n = \{1, 2, \ldots, 3n\}, I_i \cap I_j = \emptyset \) if \( i \neq j \) and \( \sum_{i \in I_i} a_i = B \) for all \( i \)?

We see that 3-Partition does not amount to partitioning the data into three sets of equal sums, which would have been the natural extension of 2-Partition. The technical condition \( \frac{a}{3} < a_i < \frac{2a}{3} \) ensures that all subsets will be of cardinal 3, hence the name 3-Partition. The last condition \( \sum_{i \in I} a_i = nB \) is obviously needed for a solution to exist.

Why deal with these two variants then? It turns out that 2-Partition, although NP-complete, is easier, if we may say so, than 3-Partition: There exists a pseudo-polynomial algorithm to solve 2-Partition, while 3-Partition is NP-complete in the strong sense [57]. Intuitively, problems involving numbers may turn out to be hard to solve only when these numbers are very large, or they may remain of combinatorial nature even for small numbers. The latter are said to be NP-hard in the strong sense. We refer the reader to [57] for a primer on NP-complete number problems.

Partitioning problems and scheduling independent tasks are closely related topics, and we come back to this important notion in Section 7.4.5. Beforehand, we turn to the complexity of P\textsubscript{b}(p) and to list scheduling heuristics:

**Definition 7.6.** The decision problem Dec\textsubscript{p}(p) associated with P\textsubscript{b}(p) is as follows. Given a DAG \( G = (V,E,w) \), a number of processors \( p \geq 1 \), and an execution bound \( K \in \mathbb{N} \), does there exist a schedule \( \sigma \) for \( G \) using at most \( p \) processors, such that MS(\sigma, p) \leq K? The restriction of Dec\textsubscript{p}(p) to independent tasks (no dependence, i.e., when \( E = \emptyset \)) is denoted Indep\textsubscript{tasks}(p). In both problems, \( p \) is arbitrary (it is part of the problem instance). When \( p \) is fixed a priori, say \( p = 2 \), we note Dec\textsubscript{2}(2) and Indep\textsubscript{tasks}(2).
THEOREM 7.5.

- Indep-tasks(2) is NP-complete but can be solved by a pseudo-polynomial algorithm.
- Indep-tasks(p) is NP-complete in the strong sense.
- Dec(2) (and hence Dec(p)) is NP-complete in the strong sense.

Proof. First, Dec(p) (and hence all the other problems, which are restrictions of it) belongs to NP. If we are given a schedule \( \sigma \) whose makespan is less than or equal to \( K \), we can check in polynomial time that both dependences and resource constraints are satisfied. Indeed, we have to ensure that each dependence constraint (each edge in \( E \)) is satisfied, which is straightforward. Also, we need to check that no more than \( p \) tasks ever execute simultaneously.

We can sort the tasks by their starting times, and check the latter condition by scanning the sorted array. This can easily be done in time polynomial in the size of the problem instance.

For proving the NP-completeness of Indep-tasks(2), we show that 2-Partition can be polynomially reduced to Indep-tasks(2). Consider an arbitrary instance \( I_{\text{inst}} \) of 2-Partition, with \( n \) integers \( \{a_1,a_2,\ldots,a_n\} \) and let \( S = \sum_{i=1}^{n} a_i \) be even (otherwise we know there is no solution). We build an instance \( I_{\text{inst}}(2) \) as follows. We let \( p = 2 \) (of course), \( G = (V,E,w) \) with \( V = \{v_1,v_2,\ldots,v_n\} \), \( E = \emptyset \), and \( w(v_i) = a_i \), \( 1 \leq i \leq n \). We also let \( K = \frac{S}{2} \).

The construction of \( I_{\text{inst}}(2) \) is polynomial (and even linear) in the size of \( I_{\text{inst}} \). Moreover, \( I_{\text{inst}} \) has a solution if and only if there exists a schedule that meets the bound \( K \), hence if and only if \( I_{\text{inst}}(2) \) has a solution.

The pseudo-polynomial algorithm to solve Indep-tasks(2) is a simple dynamic programming algorithm. For \( 1 \leq i \leq n \) and \( 0 \leq T \leq S \), let the boolean variable \( c(i,T) \) be true if there exists a subset of \( \{a_1,a_2,\ldots,a_n\} \) whose sum is \( T \). We need to determine the value of \( c(n,\frac{S}{2}) \). We use the induction

\[
c(i,T) = c(i-1,T) \vee (T \geq a_i) \wedge c(i-1,T-a_i),
\]

which basically states that either \( a_i \) is involved in the target subset, or not. The initialization is \( c(1,a_1) = 1 \), \( c(1,0) = 1 \) for all \( i \) and all other boundary values set to 0. The complexity of the algorithm is \( O(nS) \), which is not polynomial in the problem size, whose typical binary encoding would be \( O(n+\sum_{i=1}^{n} \log a_i) \). (However, if the \( a_i \)'s are encoded in unary, we have polynomial complexity, which is the definition of a pseudo-polynomial algorithm.)

The reduction for the strong NP-completeness of Indep-tasks(p) is straightforward. Consider an arbitrary instance \( I_{\text{inst}}(2) \) of 3-Partition, with \( n \) integers \( \{a_1,a_2,\ldots,a_n\} \) and bound \( K \) as stated above. The instance \( I_{\text{inst}}(2) \) is built with \( 3n \) independent tasks of weight \( a_i \), \( p = n \) processors and \( K = B \). Clearly, \( I_{\text{inst}}(2) \) has a solution if and only if there exists a schedule that meets the bound \( K \), hence if and only if \( I_{\text{inst}}(2) \) has a solution.

The reduction for the strong NP-completeness of Dec(2) is more interesting. Consider again an arbitrary instance \( I_{\text{inst}}(2) \) of 3-Partition, with \( 3n \) integers \( \{a_1,a_2,\ldots,a_{3n}\} \) and bound \( B \). The instance \( I_{\text{inst}}(2) \) is built with \( p = 2 \) processors; 3n independent tasks \( \{T_1,\ldots,T_{3n}\} \), where the weight of \( T_i \) is \( a_i \) and 3n other tasks, \( \{X_1,Y_1,Z_1,\ldots,X_n,Y_n,Z_n\} \), all of weight \( B \), linked by the following dependences:

![Diagram](image)

Finally, we let \( K = 2nB \).

Assume first that \( I_{\text{inst}} \) has a solution \( I_1 \cup I_2 \cup I_3 \), where each \( I_i \) is composed of three numbers whose sum is \( B \). The solution to \( I_{\text{inst}}(2) \) is the following schedule \( \sigma \):

- The first processor \( P_1 \) executes all 2n tasks \( X_i \) and \( Y_i \). These tasks are totally ordered along a dependence path of length \( K = 2nB \).
- The second processor \( P_2 \) executes \( Z_i \) while \( P_1 \) executes \( Y_i \). While \( P_1 \) executes a task \( X_i \), it has a slot of size \( B \) to execute the three tasks \( T_i \) that belong to \( I_i \).

Altogether, all dependence and resource constraints are satisfied, and \( \sigma \) is a valid schedule of makespan \( K \).

Now, assume that \( I_{\text{inst}} \) has a solution schedule \( \sigma \). We see that the schedule \( \sigma \) is quite constrained. Because \( X_1 \rightarrow Y_1 \rightarrow X_2 \rightarrow \ldots \rightarrow X_n \rightarrow Y_n \) is a dependence path of length \( K \), these tasks must be processed as soon as possible too. This enforces that \( \sigma(X_1) = (2i-2)B \) and that \( \sigma(Y_1) = \sigma(Z_1) = (2i-1)B \). Up to some exchanges, we can assume that \( P_1 \) executes all the \( X_i \) and \( Y_i \), and that \( P_2 \) executes all the \( Z_i \). We see that the 3n tasks \( T_i \) are executed by \( P_2 \) during \( n \) intervals of length \( B \), hence the solution to \( I_{\text{inst}} \).
 decide which tasks are given priority in the (frequent) case where there are more free tasks than available processors. But a key result due to Coffman [42] is that any list algorithm can be shown to achieve at most twice the optimal makespan. We express this more formally after giving some definitions.

**DEFINITION 7.7.** Let $G = (V, E, w, u, t)$ be a DAG and let $\sigma$ be a schedule for $G$. A task $v \in V$ is free at time $t$ (we note $v \in FREE(\sigma, t)$) if and only if its execution has not yet started ($\sigma(w(v)) \geq t$) but all its predecessors have been executed ($\forall u \in PRED(v), \sigma(u) + w(u) \leq t$).

A list schedule is a schedule such that no processor is deliberately left idle; at each time $t$, if $|FREE(\sigma, t)| = r \geq 1$, and if $q$ processors are available, then $\min(r, q)$ free tasks start executing.

**THEOREM 7.6.** Let $G = (V, E, w)$ be a DAG and assume there are $p$ available processors. Let $\sigma$ be an any list schedule of $G$. Let $MS_{opt}(p)$ be the makespan of an optimal schedule. Then,

$$MS(\sigma, p) \leq \left(2 - \frac{1}{p}\right) MS_{opt}(p).$$

It is important to point out that Theorem 7.6 holds for any list schedule, regardless of the strategy to choose among free tasks when there are more free tasks than available processors.

**Proof.** We first need a lemma:

**LEMMA 7.1.** There exists a dependence path $\Phi$ in $G$ whose weight $w(\Phi)$ satisfies

$$\text{Idle} \leq (p - 1) \times w(\Phi),$$

where Idle is the cumulated idle time of the $p$ processors during the whole execution of the list schedule.

**Proof.** Let $T_i$ be a task whose execution terminates at the end of the schedule:

$$\sigma(T_i) + w(T_i) = MS(\sigma, p).$$

Let $t_i$ be the largest time smaller than $\sigma(T_i)$ and such that there exists an idle processor during the time interval $[t_i, t_i + 1]$ (let $t_i = 0$ if such a time does not exist). Why is this processor idle? Because $\sigma$ is a list schedule, no task is free at $t_i$, otherwise the idle processor would start executing a free task. Therefore, there must be a task $T_j$ that is an ancestor of $T_i$ and that is being executed at time $t_i$; otherwise $T_i$ would have been started at time $t_i$ by the idle processor. Because of the definition of $t_i$ we know that all processors

\[\text{are active between the end of the execution of } T_j \text{ and the beginning of the execution of } T_i.\]

We start the construction again from $T_j$ so that we obtain a task $T_k$ such that all processors are active between the end of $T_k$ and the beginning of $T_i$. Iterating the process, we end up with $r$ tasks $T_1, T_2, \ldots, T_r$ that belong to a dependence path $\Phi$ of $G$ and such that all processors are active except perhaps during their execution. In other words, the idleness of some processors can only occur during the execution of these $r$ tasks, during which at least one processor is active (the one that executes the task). Hence, Idle $\leq (p - 1) \times \sum_{i=1}^{r} w(T_i) \leq (p - 1) \times w(\Phi)$.

Going back to the proof of Theorem 7.6, we know that $p \times MS(\sigma, p) = \text{Idle} + \text{Seq}$, where Seq $= \sum_{i=1}^{r} w(\Phi)$ is the sequential time, i.e., the sum of all task weights (see Figure 7.2). Now take the dependence path $\Phi$ constructed in Lemma 7.1. We have $w(\Phi) \leq MS_{opt}(p)$, because the makespan of any schedule is greater than the weight of all dependence paths in $G$ (simply because dependence constraints are met). Furthermore, Seq $\leq p \times MS_{opt}(p)$ (with equality only if all $p$ processors are active all the time). Putting this together, we get

$$p \times MS(\sigma, p) = \text{Idle} + \text{Seq} \leq (p - 1)w(\Phi) + \text{Seq} \leq (p - 1)MS_{opt}(p) + pMS_{opt}(p) = (2p - 1)MS_{opt}(p),$$

which proves the theorem. \qed

Fundamentally, Theorem 7.6 says that any list schedule is within 50% of the optimum. Therefore, list scheduling is guaranteed to achieve half the best possible performance, regardless of the strategy to choose among free tasks. Before presenting the most widely used strategy to perform this choice (in order to obtain a practical scheduling algorithm), we make a short digression to show that the bound $\frac{2p}{p}$ cannot be improved.

**PROPOSITION 7.2.** Let $MS_{opt}(p)$ be the shortest possible makespan produced by a list scheduling algorithm. The bound

$$MS_{opt}(p) \leq \frac{2p - 1}{p} MS_{opt}(p)$$

is tight.

**Proof.** Let $K$ be an arbitrarily large integer. We build a DAG $G = (V, E, w)$, for which any list schedule $\sigma$ has a makespan $MS(\sigma, p) = \frac{2p + 1}{p} MS_{opt}(p)$ (see Figure 7.3). There are $2p + 1$ vertices, whose weights are as follows: $w(T_i) = K(p - 1)$ for $1 \leq i \leq p - 1$; $w(T_p) = 1$; $w(T_i) = K$ for $p + 1 \leq i \leq 2p$; and $w(T_{2p + 1}) = K(p - 1)$. Precedence edges are indicated in the figure. There are exactly $p$ entry vertices, hence $\sigma(T_i) = 0, 1 \leq i \leq p$ for any list schedule
7.4. Solving $P_b(p)$

The overall scheme can be outlined as follows: We give an example instantiation of this generic algorithm in Section 7.4.4.

Tasks are placed in a priority queue, meaning that we do not specify how to choose free tasks when there are more free tasks than available processors. We simply assume that free processors are assigned tasks in a first-come, first-served manner. Therefore, the remaining $p$-th task will be executed at time $K(p-1)$ by another processor. Only at time $K(p-1) + K = Kp$ will task $T_{2p+1}$ be free, which leads to $MS(\sigma, p) = Kp + K(p-1) = K(2p-1)$.

$$x_t^{(K(p-1)} \rightarrow T_t^{(K(p-1)} \rightarrow \ldots \rightarrow x_t^{(p)} \rightarrow T_t^{(p-1)}.$$  

**FIGURE 7.3:** The DAG used to bound list scheduling performance; task weights are indicated as exponents inside parentheses.

However, the DAG can be scheduled in only $Kp+1$ time units. The key is to deliberately keep $p-1$ processors idle while executing task $T_p$ at time 0 (which is forbidden in a list schedule). Then, at time 1, each processor executes one of the $p$ tasks $T_{p+1}, T_{p+2}, \ldots, T_{2p}$. At time 1, one processor starts executing $T_{2p+1}$ while the other $p-1$ processors execute tasks $T_1, T_2, \ldots, T_{p-1}$. This defines a schedule with a makespan equal to $1 + K + K(p-1) = Kp + 1$, which is optimal because it is equal to the weight of the path $T_p \rightarrow T_{p+1} \rightarrow T_{2p+1}$. Hence, we obtain the ratio

$$\frac{MS(\sigma, p)}{MS_{opt}(p)} \geq \frac{K(2p-1)}{Kp+1} = \frac{2p-1}{p} \frac{2p-1}{p} - \epsilon(K),$$

where $\lim_{K \to +\infty} \epsilon(K) = 0$.

7.4.3 Implementing a List Schedule

In this section, we show how to implement a “generic” list scheduling algorithm, meaning that we do not specify how to choose free tasks when there are more free tasks than available processors. We simply assume that free tasks are placed in a priority queue, according to some to-be-defined priority. We give an example instantiation of this generic algorithm in Section 7.4.4.

The implementation is not difficult but is somewhat lengthy to describe. The overall scheme can be outlined as follows:

1. **Initialization:**
   (a) Compute the priority of all tasks, for some definition of priority.
   (b) Place the free tasks in a priority queue, sorted by non-increasing priority.
   (c) Let $t$ be the current time: $t = 0$.

2. **While there remain tasks to schedule:**
   (a) Add new free tasks, if any, to the priority queue. If the execution of a task terminates at time $t$, remove this task from the predecessor lists of all its successors. Add those tasks whose predecessor lists have become empty to the priority queue.
   (b) If there are $q$ available processors and $r$ tasks in the priority queue, remove the first $\min(q, r)$ tasks from the priority queue and schedule them; for each such task $T$ set $\sigma(T) = t$.
   (c) Increment $t$ by one (recall that task weights are integers).

Let $G = (V, E, w)$ be a DAG and assume there are $p$ available processors. Let $\sigma$ be any list schedule of $G$. Our aim is to derive an implementation whose complexity is $O(|V|\log |V| + |E|)$ for computing the schedule. Clearly, the above scheme must be modified because time varies from $t = 0$ up to $t = MS(\sigma, p)$, implying that the complexity depends on task weights. Indeed, $MS(\sigma, p)$ may be of the order of $Seq$, the sum of all task weights, and we would have a pseudo-polynomial algorithm instead of a true polynomial algorithm; a binary encoding of the problem instance is of size $\log(Seq)$, not of size $Seq$. We outline a possible solution written in pseudo-code in Algorithm 7.1. Rather than using time $t$ we use events, which correspond to times when tasks become free or processors become idle.

A few words of explanation are in order for Algorithm 7.1. We use a heap $Q$ (see [44]) to store free tasks for two reasons. We can access the task with the highest priority in constant time, and we can insert a task in the heap, according to its priority level, in time proportional to the logarithm of the heap size, which is bounded by $|V|$. We use another heap $P$ to handle active processors; a processor executing a task $v \in V$ is valued by the time at which the execution of $v$ terminates. Thereby we can compute the next event in constant time, and we can insert a new active processor in the heap in $O(\log P) \leq O(\log |V|)$ time. When we extract a processor from the processor heap, meaning that a task $v$ has terminated, we need to update the in-degree of each successor of $v$ in array $A$. On the fly, if the in-degree of a given successor $v'$ becomes zero, we insert $v'$ in the priority heap $Q$. This way, we process each edge of $G$ only once, for a global cost $O(|E|)$. Overall, each task causes two insertions: the first is the insertion of the task itself in heap $Q$; the second is the insertion of the processor that executes it in heap $P$. Because each operation costs at most $O(\log |V|)$, we obtain the desired complexity $O(|V|\log |V| + |E|)$ for computing the schedule.
7.4. Solving $P_b(p)$

1. **List Schedule**($G = (V, E, w), p$)
   - $p$ processors, with $1 \leq p \leq |V|$
   - In addition to the data structure representing $G$, store in array $A$ the number of predecessors of each task (its in-degree)
   - Insert all the tasks without predecessors in priority heap $Q$
   - Initialize processor heap $P$ to Empty
   - $t \leftarrow 0$
   - While $Q \neq Empty$
     - $t' \leftarrow NEXT_EVENT($P$, t)$
     - Update($t', A, Q$)
     - Allocate_Tasks($t', P, Q$)
     - $t \leftarrow t'$

**Algorithm 7.1:** Outline of a list scheduling algorithm.

7.4.4 Critical Path Scheduling

In this section, we detail a widely used list scheduling technique, known as critical path scheduling. We have seen the basic principle of the list scheduling technique and assessed its performance. We now have to explain how to choose among free tasks to get a practical list scheduling algorithm. The most popular selection criterion is based on the value of the bottom level of the tasks. Intuitively, the larger the bottom level, the more “urgent” the task.

The critical path of a task is defined as its bottom level and is used to assign priority levels to tasks. Critical path scheduling is list scheduling where the priority level of a task is given by the value of its critical path. Ties are broken arbitrarily.

Let us work out a small example. Consider the DAG shown in Figure 7.4.

There are eight tasks, whose weights and critical paths are listed in Table 7.1. Assume there are $p = 3$ available processors and let $Q$ be the priority queue of free tasks. At $t = 0$, $Q$ is initialized as $Q = (T_3, T_2, T_1)$. Because $q = r = 3$,

**Table 7.1:** Weights and critical paths for DAG in Figure 7.4.

| Tasks | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$
|-------|-------|-------|-------|-------|-------|-------|-------|-------
| Weights | 3 | 2 | 1 | 3 | 4 | 4 | 3 | 0
| Critical Paths | 0 | 0 | 7 | 2 | 4 | 4 | 0 | 0

**Figure 7.4:** A small example DAG.

**Figure 7.5:** Critical path schedule for the example DAG in Figure 7.4.

**Figure 7.6:** Optimal schedule for the example DAG in Figure 7.4.

Note that it is possible to schedule the DAG in only 9 time units, as shown in Figure 7.6. The trick is to leave a processor idle at time $t = 1$ deliberately; although it has the highest critical path, $T_3$ can be delayed by two time units. $T_2$ and $T_4$ are given preference to achieve a better load balance between processors. How do we know that the schedule shown in Figure 7.6 is optimal? Because Seq = 26, so that three processors require at least $\lceil \frac{26}{3} \rceil = 9$ time units.

This small example illustrates the difficulty of scheduling with a limited number of processors.

7.4.5 Scheduling Independent Tasks

We come back to the problem Indep-tasks($p$) introduced in Section 7.4.1. Recall that the objective is to schedule a set of independent tasks (no dependence relation). In addition to being a scheduling problem, this problem can be viewed as a load balancing problem where we aim at partitioning the tasks so as to assign (almost) equal load to each processor.

For fixed $p$, we have shown that Indep-tasks($p$) is NP-complete but solvable
7.4. Solving $P_b(p)$

in pseudo-polynomial time. In this section, we aim at deriving $\lambda$-approximation algorithms, i.e., polynomial-time schedules whose makespans are at most $\lambda$ times the optimal makespan for any problem instance.

Because all tasks are independent, any greedy (or list scheduling) algorithm will keep some processors idle only at the end of the execution. Can we improve the performance bound $\lambda = 2 - \frac{1}{3}$ of Theorem 7.6 for this particular case? For the sake of simplicity we only consider the case $p = 2$ in this section. Although simple, this case enables us to introduce approximation schemes that play a major role in scheduling theory.

Assume that $P_b = \{P_1, P_2\}$, where the weight of $P_1$ is $a_1$, how do we schedule these tasks on two identical processors? Let $MS_{opt}$ be the optimal schedule and $P_{sum} = \sum_{i \in P} a_i$. Of course $MS_{opt} \geq \frac{P_{sum}}{2}$. There are two natural greedy algorithms:

- GREEDY: Consider the tasks in some arbitrary order; at each step, schedule the current task on the least-loaded processor.

- SORTED-GREEDY: Same as above, but considering the tasks sorted by non-increasing weight.

The rationale to sort the tasks is that the arrival of a big task in the end may unbalance the whole execution. However, we need to know all the tasks (for sorting) before starting the execution of the algorithm. For this reason SORTED-GREEDY is called an on-line algorithm. By contrast, GREEDY can be applied to an on-line problem in which new tasks dynamically arrive (e.g., to a dual-processor computer).

**Theorem 7.7.** GREEDY is a $\frac{1}{2}$-approximation and SORTED-GREEDY is a $\frac{1}{3}$-approximation for Indep-tasks(2), and these approximation factors cannot be improved.

**Proof.** We first show that the bounds are tight (note that the list scheduling bound is $2 - \frac{1}{2} = \frac{1}{2}$). For GREEDY, take an instance with three tasks of weights 1, 1, and 2. GREEDY has a makespan of 3 while the optimal is 2. For SORTED-GREEDY, take five tasks, three of weight 2 and two of weight 3. SORTED-GREEDY has a makespan of 7, while the optimal is 6.

For the approximations, recall that $MS_{opt} \geq \frac{P_{sum}}{2}$ and that $MS_{opt} \geq a_i$ for all $i$. Let us start with GREEDY. Let $P_1$ and $P_2$ be the two processors. Assume that $P_1$ finishes execution last. Let $M_1$ be the execution time on $P_1$ (the sum of all task weights assigned to it) and $M_2$ the execution time on $P_2$. Because $P_1$ terminates last, $M_1 \geq M_2$. Of course $M_1 + M_2 = P_{sum}$.

Let $T_1$ be the last task that executes on $P_1$. Let $M_0 = M_1 - a_1$ be the load of $P_1$ before $T_1$ (of weight $a_2$) is assigned to it. Why does the GREEDY algorithm choose to assign $T_j$ to $P_1$? It can only be because at that time $P_2$ has more load than (or the same load as) $P_1$. $M_0$ is not larger than the current load of $P_2$ at that time, which itself is not larger than its final load $M_2$ (note that $P_2$ may have been assigned more tasks after $T_1$ was scheduled on $P_1$). Therefore, $M_0 \leq M_2$. To summarize, the makespan of the schedule computed by GREEDY is $M_1$, and

$$M_1 = M_0 + a_j = \left(\left(\frac{M_0 + M_0 + a_j}{2}\right) + a_j\right) \leq \frac{P_{sum}}{2} + a_j$$

$$\leq MS_{opt} + \frac{a_j}{2} \leq MS_{opt} + \frac{MS_{opt}}{2}.$$ 

hence proving the $\frac{1}{2}$-approximation result for GREEDY.

For SORTED-GREEDY the same line of reasoning is used, but with a tighter bounding of $a_1$ than by $MS_{opt}$. First, if $a_1 \leq \frac{1}{2}MS_{opt}$, we obtain what we need, i.e., $M_1 \leq \frac{2}{3}MS_{opt}$. But if $a_1 > \frac{1}{2}MS_{opt}$, then necessarily $j \leq 4$. Indeed, if $T_j$ was the fifth task or higher, because task weights are sorted, there would be at least five tasks of weight greater than $\frac{1}{2}MS_{opt}$; in any schedule, including the optimal schedule, one processor would receive three of these tasks, a contradiction. Next we observe that the makespan achieved by SORTED-GREEDY is the same when scheduling all tasks as when scheduling only the first four tasks. But for any problem instance with $n \leq 4$, SORTED-GREEDY is optimal, and $M_1 = MS_{opt}$.

We now show that Indep-tasks(2) has a $(1 + \varepsilon)$-approximation for any value of $\varepsilon > 0$. We say that Indep-tasks(2) has a polynomial time approximation scheme, or PTAS. For any value of $\varepsilon > 0$, we build an algorithm whose makespan is guaranteed up to a factor $1 + \varepsilon$ from the optimal, and whose complexity is polynomial. A word of caution here: “polynomial” means polynomial in the problem size only. Even if $\varepsilon$ can tend to infinity, $\varepsilon$ is a constant for the algorithm. Later we present a fully polynomial time approximation scheme, or FPTAS, where the complexity of the $1 + \varepsilon$ approximation is polynomial both in the problem size and in $\frac{1}{\varepsilon}$.

**Theorem 7.8.** $\forall \varepsilon > 0$, Indep-tasks(2) has a $(1 + \varepsilon)$-approximation.

**Proof.** Let $P_{max} = \max_i a_i$ and $L = \max(P_{max}, P_{max})$. We already know that $L \leq MS_{opt}$. We call big jobs those tasks $T_j$ whose weights are such that $a_j > \varepsilon L$ and small jobs those such that $a_i \leq \varepsilon L$. The number of big jobs is at most

$$\frac{P_{max}}{\varepsilon L} \leq \frac{2 L}{\varepsilon L} = \frac{2}{\varepsilon} = B.$$ 

Because $\varepsilon$ is fixed, $B$ is a constant, so there is a (possibly large but) constant number of big jobs. We temporarily forget about small jobs, consider only big jobs and search for the best schedule. There are $2^B$ possible schedules.
(each big job can be assigned to either processor), which is a constant number again, so we try them all and keep the best one, say \( \sigma^* \). The resulting makespan \( MS_{\text{opt}}^* \) satisfies \( MS_{\text{opt}}^* \leq MS_{\text{opt}} \) because there are fewer jobs than in the original problem.

Now we extend \( \sigma^* \) and schedule the small jobs after the big jobs, using GREEDY, and we obtain a schedule \( \sigma \) for the original problem. We claim that \( \sigma \) solves the problem, i.e., that \( MS(\sigma) \leq (1 + \epsilon)MS_{\text{opt}} \). If the makespans of \( \sigma^* \) and \( \sigma \) are equal, then \( \sigma \) is optimal. Otherwise, \( \sigma \) terminates with a small job \( T_j \), say on the first processor \( P_1 \). But the load of \( P_1 \) before this last assignment could not exceed \( MS_{\text{opt}} \), otherwise GREEDY would have assigned \( T_j \) on \( P_2 \) (same proof as in Theorem 7.7). Hence,

\[
MS(\sigma) \leq \frac{P_{\text{sum}}}{2} + a_j \leq L + \epsilon L \leq (1 + \epsilon)MS_{\text{opt}},
\]

which proves the theorem.

Theorem 7.8 is interesting, but the combinatorial search for the best assignment of big jobs can be prohibitively expensive when \( \epsilon \) tends to 0. This motivates our last result, which states that Indep-tasks(2) has an FPTAS. (This will conclude our iteration to the fascinating world of approximation schemes.)

**THEOREM 7.9.** \( \forall \epsilon > 0 \), Indep-tasks(2) has a \((1 + \epsilon)\)-approximation whose complexity is polynomial in \( \frac{1}{\epsilon} \).

Proof. We encode schedules as vector sets (VS). The first component of each vector represents the load of the first processor \( P_1 \), and the second component is the load of \( P_2 \). Here is the construction:

**Initialization** Set \( VS_1 = \{[a_1, 0], [0, a_1]\} \).

**Phase \( k \) for \( 2 \leq k \leq n \)** For every vector \([x, y] \in VS_{k-1}\), put \([x + a_k, y]\) and \([x, y + a_k]\) in \( VS_k \).

**Output** Vector \([x, y] \in VS_n\) that minimizes max\((x, y)\).

Instead of building all possible vectors (whose number is exponential), we will prune some of them during the construction. The difficulty is to retain a vector close to the optimal when pruning. Let \( \Delta = 1 + \frac{\epsilon}{n} \). All considered vectors lie in the rectangle \([0, P_{\text{sum}}] \times [0, P_{\text{sum}}] \). We subdivide this rectangle into many boxes. Horizontal and vertical cuts are made at the coordinates \( \Delta^i \) for \( 1 \leq i \leq M \), where

\[
M = \left\lceil \log_\Delta P_{\text{sum}} \right\rceil = \left\lceil \frac{\ln P_{\text{sum}}}{\ln \Delta} \right\rceil \leq \left(1 + \frac{2n}{\epsilon} \right) \ln(P_{\text{sum}}).
\]

The last inequality comes from the fact that \( \ln(z) \geq 1 - \frac{z}{4} \) for all \( z \geq 1 \).
7.5 Taking Communication Costs into Account

Distributed-memory parallel computing platforms pose many challenges to the algorithm designer and the programmer. An obvious factor contributing to this complexity is the need for network communication, whose performance is difficult to model in a way that is both precise and conducive to understanding the performance of algorithms (see Chapter 3). Older parallel computers used a store-and-forward approach to communicate messages, which was not efficient but simple to understand and to model. Essentially, the time for sending a message from a processor to a processor \( p' \) is \( c(p, p') = \text{dist}(p, p') \times (L + \beta b) \), where \( b \) is the length of the message, \( \text{dist}(p, p') \) is the distance between \( p \) and \( p' \) in number of hops, \( L \) is the communication start-up cost, and \( \beta \) is the inverse of the steady-state bandwidth. In modern computers, messages are split into packets that are dynamically routed between processors, possibly using different paths. Messages can be routed efficiently if there are no congestions on the communication links (or “hot spots”). The distance between communicating processors is no longer the single most important factor for communication performance. In fact, if several processors are to exchange data simultaneously, then the more structured the communication patterns, the more efficient they are, making the role of locality on performance at best indirect.

In light of the complexity of performance modeling for network communications, the vast majority of scheduling works and results are for a very simple model, which is as follows. If a task \( T \) communicates data to a successor task \( T' \), the cost is modeled as

\[
\text{cost}(T, T') = \begin{cases} 
0 & \text{if } \text{alloc}(T) = \text{alloc}(T') \\
\epsilon (T, T') & \text{otherwise},
\end{cases}
\]

where \( \text{alloc}(T) \) denotes the processor that executes task \( T \) (see Section 7.2), and \( \epsilon (T, T') \) is defined by the application specification. The above model states that the time for communication between two tasks running on the same processor is negligible. The model also assumes that the processors are part of a fully connected clique. This so-called macro-dataflow model makes two main assumptions: (i) communication can occur as soon as data are available; and (ii) there is no contention for network links. Assumption (i) is reasonable as communication can overlap with (independent) computations in most modern computers. Assumption (ii) is much more questionable. Indeed, there is no physical device capable of sending, say, 1,000 messages to 1,000 distinct processors, at the same speed as if there were a single message. In the worst case, it would take 1,000 times longer (serializing all messages). In the best case, the output bandwidth of the network card of the sender would be a limiting factor. In other words, assumption (ii) amounts to assuming infinite network resources! Nevertheless, this assumption is omnipresent in the traditional scheduling literature. Perhaps it is the price to pay to derive tractable mathematical results? We use this mode for now and turn to more realistic communication models in Chapter 8.

We conclude this discussion by stating the model more formally.

**DEFINITION 7.8.** A communication DAG (or cDAG) is a direct acyclic graph \( G = (V, E, w, c) \), where vertices represent tasks and edges represent precedence constraints. The computation weight function is \( w : V \rightarrow N^* \) and the communication cost function is \( c : E \rightarrow N^* \). A schedule \( \sigma \) must preserve dependences, which is written as

\[
\forall e = (T, T') \in E : \begin{cases} 
\sigma(T) + w(T) \leq \sigma(T') & \text{if } \text{alloc}(T) = \text{alloc}(T') \\
\sigma(T) + w(T) + c(T, T') \leq \sigma(T') & \text{otherwise}.
\end{cases}
\]

The expression of resource constraints is the same as in the no-communication case.

7.6 \( Pb(\infty) \) with communications

Including communication costs in the model makes everything difficult, including solving \( Pb(\infty) \). The intuitive reason is that we hesitate between allocating tasks to either many processors (hence balancing the load but communicating intensively) or few processors (leading to less communication but less parallelism as well). We illustrate this with a small example, borrowed from [60].

Consider the cDAG in Figure 7.7. Task weights are indicated close to the tasks within parentheses, and communication costs are shown along the edges, underlined. For the sake of this example, we use two non-integer communication costs: \( c(T_1, T_2) = c(T_1, T_3) = 1.5 \). Of course, we could scale every weight \( w \) and cost \( c \) to have only integer values. We can check the following:

- On the one hand, if we allocate all tasks to the same processor, the makespan will be equal to the sum of all task weights, i.e., 13.
6. Pb(∞) with communications

- On the other hand, if we have as many processors as we want (we need no more than seven processors because there are seven tasks), we can allocate one task per processor. Then, we can check that the makespan of the ASAP schedule is equal to 14. To see this, it is important to point out that once the allocation of tasks to processors is given, we can compute the makespan easily: For each edge \((T_i, T_j)\) of weight \(w(T_i, T_j)\) if the edge links two different processors (\(\text{alloc}(T) \neq \text{alloc}(T')\)), and do nothing otherwise. Then, consider the new graph as a DAG (without communications) and traverse it to compute the length of the longest path, as explained in Section 7.3. In our case, because all tasks are allocated to different processors, we add a virtual node on each edge. The longest path is \(T_1 \rightarrow T_2 \rightarrow T_5\), whose length is \(w(T_1) + w(T_2) + c(T_2, T_3) + w(T_3) = 14\).

There is a difficult trade-off between executing tasks in parallel (hence with no more than seven processors because there are seven tasks), we can allocate one task per processor. Then, we can check that the makespan of the ASAP schedule is equal to 14. To see this, it is important to point out that once the allocation of tasks to processors is given, we can compute the makespan easily: For each edge \((T_i, T_j)\) of weight \(w(T_i, T_j)\) if the edge links two different processors (\(\text{alloc}(T) \neq \text{alloc}(T')\)), and do nothing otherwise. Then, consider the new graph as a DAG (without communications) and traverse it to compute the length of the longest path, as explained in Section 7.3. In our case, because all tasks are allocated to different processors, we add a virtual node on each edge. The longest path is \(T_1 \rightarrow T_2 \rightarrow T_5\), whose length is \(w(T_1) + w(T_2) + c(T_2, T_3) + w(T_3) = 14\).

There is a difficult trade-off between executing tasks in parallel (hence with several distinct processors) and minimizing communication costs. In our example, it turns out that the best solution is to use two processors, according to the schedule in Figure 7.8, whose makespan is equal to 9.

```
<table>
<thead>
<tr>
<th>P2</th>
<th>T5</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
<th>T6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 7.8: An optimal schedule for the example.
```

Note that dependence constraints are satisfied in Figure 7.8. For example, \(T_2\) can start at time 1 on processor \(P_1\) because this processor executes \(T_1\), hence there is no need to pay the communication cost \(c(T_1, T_2)\). By contrast, \(T_3\) is executed on processor \(P_2\), hence we need to wait until time 2 to start it even though \(P_2\) is idle: \(\sigma(T_1) + w(T_1) + c(T_1, T_3) = 0 + 1 + 1 = 2\).

How did we find the schedule shown in Figure 7.8? And how do we know it is optimal? By a tedious case-by-case analysis! This example shows that using more processors does not always lead to a shorter execution time. Using the notations of Section 7.2, the minimum makespan of a schedule making actual use of seven processors is \(\text{MS}(7) = 14\) while \(\text{MS}(1) = 13\) (or \(\text{MS}(2) = 9\)). In other words, Theorem 7.3 is no longer true when communication costs are taken into account.

### 7.6.1 NP-Completeness of Pb(∞)

Communication costs make solving Pb(∞), the scheduling problem with unlimited processors, very difficult.

**THEOREM 7.10.** Pb(∞) is NP-complete.

**Proof.** The decision problem \(\text{Comm}(\infty)\) associated with Pb(∞) is the following. Given a cDAG \(G = (V, E, w, c)\) and an execution bound \(K \in \mathbb{N}\), does there exist a schedule \(\sigma\) for \(G\) such that \(\text{MS}(\sigma, \infty) \leq K\)? We want to show that \(\text{Comm}(\infty)\) is NP-complete. First, \(\text{Comm}(\infty)\) belongs to NP. If we are given a schedule \(\sigma\) whose makespan is less than or equal to \(K\), we can check in polynomial time that dependence constraints are satisfied. For each task we know the beginning \(\sigma(T)\) of its execution and the processor \(\text{alloc}(T)\) that executes it, hence we just have to check for constraints.

```

FIGURE 7.9: Reduction for the NP-completeness proof.
```

To prove NP-completeness, we use 2-Partition as in the proof of Theorem 7.5, but the reduction is more involved. Consider any instance \(\text{Inst}_1\) of 2-Partition. Given \(n\) positive integer numbers \(\{a_1, \ldots, a_n\}\), is there a subset \(I\) of indices such that \(\sum_{i \in I} a_i = \sum_{i \notin I} a_i?\) We build an instance \(\text{Inst}_2\) of \(\text{Comm}(\infty)\) as follows. We let \(G = (V, E, w, c)\) be a fork-join graph (see Figure 7.9). There are \(n + 2\) tasks: \(V = \{T_0, T_1, T_2, \ldots, T_n, T_{n+1}\} \). The task weights are defined as follows: \(w(T_i) = 2a_i\) for \(1 \leq i \leq n\), and \(w(T_n) = w(T_{n+1}) = A\), where \(A\)
is a positive integer. There are 2n edges, and the communication costs are all equal: \(c(T_i, T_j) = c(T_i, T_{i+1}) = C\) for \(1 \leq i \leq n\). Here, \(C\) is any integer in the interval \([a - \min_{i \in [n]} 2a_i, a]\), where \(a = \sum_{i=1}^{n} a_i\). Note that this interval does contain an integer, as \(\min_{i \in [n]} 2a_i \geq 2\). Finally, let \(K = 2A + C + 1\). The size of Inst1 is clearly polynomial in the size of Inst1. The difficult part is to show that Inst1 has a solution if and only if there exists a schedule that meets the bound \(K\), i.e., if and only if Inst2 has a solution.

First, assume that Inst1 has a solution. Let \(T\) be a subset of indices such that \(\sum_{i \in T} \delta_i = \sum_{i \in \bar{T}} \delta_i = \frac{\delta}{2}\). Let \(T_1 = \{T, i \in I\}\) and \(T_2 = \{T, i \notin I\}\). By hypothesis, \(w(T_1) = w(T_2) = \alpha\). Consider the schedule in Figure 7.10.

![Figure 7.10: A schedule with makespan \(K = 2A + C + 1\).](image)

The makespan of this schedule is equal to \(K = 2A + C + 1\). All dependence constraints are satisfied. Indeed:

- Processor \(P_2\) starts executing the tasks in \(T_2\) at time \(A + C\) or \(w(T_2) + C\).
- Processor \(P_1\) finishes executing the tasks in \(T_1\) at time \(A + \alpha\). Hence, at time \(A + \alpha + C\), task \(T_{n+1}\) is ready to be executed by processor \(P_2\). Its execution can start right then, as \(P_2\) finishes executing the tasks in \(T_2\) at time \(A + C + 1\).

Conversely, assume that Inst2 has a solution. Let \(\sigma\) be a schedule whose makespan \(MS(\sigma)\) is less than or equal to \(K\). We need the two following lemmas.

**Lemma 7.4.** Tasks \(T_0\) and \(T_{n+1}\) are not executed by the same processor in schedule \(\sigma\).

**Proof.** Assume that the same processor \(P\) executes both \(T_0\) and \(T_{n+1}\). Then, \(P\) executes all \(n\) other tasks \(T_i\), \(1 \leq i \leq n\). Otherwise, let \(T_{n+1}\) be a task executed by another processor. The makespan of \(\sigma\) is greater than or equal to the length of the path \(T_0 \rightarrow T_{n} \rightarrow T_{n+1}\):

\[
MS(\sigma) \geq A + C + 2a_n + C + A = (2A + C) + (2a_n + C) > (2A + C) + \alpha = K,
\]

hence a contradiction. Therefore, \(P\) does not execute the \(n + 2\) tasks, which is a contradiction, as the sum of all task weights is \(2A + 2a_n > 2A + \alpha + C = K\). \(\square\)

Let \(P_1\) be the processor that executes \(T_0\) and \(P_2\) be the processor that executes \(T_{n+1}\).

**Lemma 7.5.** Each task \(T_i\), \(1 \leq i \leq n\), is executed by either \(P_1\) or \(P_2\).

**Proof.** Assume that there exists a task \(T_{6i}\), \(1 \leq i \leq n\), executed by a processor other than \(P_1\) and \(P_2\). Then, the makespan of \(\sigma\) is greater than or equal to the length of the path \(T_0 \rightarrow T_{6i} \rightarrow T_{n+1}\): \(MS(\sigma) \geq A + C + 2a_n + C + A > K\)

(as in Lemma 7.4), hence a contradiction. \(\square\)

Let \(T_1\) be the set of tasks \(T_i\), \(1 \leq i \leq n\), executed by \(P_1\). Define similarly \(T_2\) for \(P_2\). The makespan of \(\sigma\) satisfies:

\[
MS(\sigma) \geq w(T_0) + w(T_1) + C + w(T_{n+1}) = 2A + C + w(T_1).
\]

To understand this note that \(P_1\) takes at least \(w(T_0) + w(T_1)\) time units to execute its tasks, and that a communication must occur before \(P_2\) can start \(T_{n+1}\).

Similarly, \(MS(\sigma) \geq 2A + C + w(T_2)\), because \(P_2\) must wait at least \(A + C\) time units before starting execution. Because \(MS(\sigma) \leq K = 2A + C + 1\), we have \(w(T_1) \leq \alpha\) and \(w(T_2) \leq \alpha\). But \(w(T_1) + w(T_2) = 2\alpha\). Therefore, \(w(T_1) = w(T_2) = \alpha\). Let \(I\) denote the set of indices of the tasks in \(T_1\); \(I\) is a solution to Inst1, our instance of 2-Partition. \(\square\)

Theorem 7.10 only shows that \(PB(\infty)\) is NP-complete in the weak sense. In fact, \(PB(\infty)\) is NP-complete in the strong sense. Even the problem in which all task weights and communication costs have the same (unit) value, the so-called UET-UET problem (unit execution time-unit communication time), is NP-hard [93, 94].

### 7.6.2 A Guaranteed Heuristic for \(PB(\infty)\)

In this section, we present a guaranteed heuristic, due to Hanen and Munier [66], to solve \(PB(\infty)\). The heuristic is guaranteed within a factor at most \(\frac{4}{3}\) of the optimal under the assumption that all communication costs are smaller than all computation costs. Such a task graph is said to be coarse-grain, as stated in the following definition:

**Definition 7.9.** Let \(G = (V, E, w, c)\) be a cDAG. The granularity of \(G\) is the computation to communication ratio

\[
g(G) = \frac{\min_{T \in V} w(T)}{\max_{T \in V} c(T, T')}.
\]

\(G\) is coarse-grain if \(g(G) \geq 1\).

Before stating Hanen and Munier’s heuristic formally, we explain the main idea, that of “favorite successors.”
TABLE 7.2: Favorite successors for the schedule in Figure 7.8.

<table>
<thead>
<tr>
<th>Task</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Favorite Successor</td>
<td>T2</td>
<td>T4</td>
<td>T1</td>
<td>T6</td>
<td>T5</td>
<td>T4</td>
</tr>
</tbody>
</table>

Favorite successors – Let \( G = (V, E, w, c) \) be a coarse-grain cDAG and \( \sigma \) be any schedule for \( G \). Let \( T \in V \) be any task. The favorite successor of \( T \), if it exists, is the unique immediate successor \( T' \) of \( T \) such that
\[
\sigma(T') < \sigma(T) + w(T) + c(T, T').
\]
(FS)

If it exists, the favorite successor of \( T \) is executed by the same processor as \( T \), otherwise a communication cost would be paid and condition (FS) would not hold. Now, to see why the favorite successor is unique (if it exists), assume that two successors \( T'' \) and \( T''' \) of \( T \) satisfy condition (FS). \( T'' \) and \( T''' \) are executed by the same processor. Without loss of generality, assume that \( T'' \) is executed before \( T''' \). This implies that \( \sigma(T) + w(T') < \sigma(T'') \) and that \( \sigma(T') + w(T'') < \sigma(T'''') \). But, by hypothesis, \( \sigma(T') + w(T) + c(T, T') < \sigma(T) + w(T) + c(T, T'') \); hence \( w(T') < c(T, T''') \), a result that contradicts the fact that \( G \) is coarse-grain. Table 7.2 gives all the favorite successors for the optimal schedule shown in Figure 7.8.

For each edge \( e = (T, T') \in E \), we introduce a boolean variable \( x_{T,T'} \): \( x_{T,T'} = 0 \) if \( T' \) is the favorite successor of \( T \), and \( x_{T,T'} = 1 \) otherwise. The inequality
\[
\sigma(T) + w(T) + x_{T,T'} c(T, T') \leq \sigma(T')
\]
holds for any edge \((T, T') \in E\), whether \( T' \) is the favorite successor of \( T \) or not. Casting all such inequalities into a linear program is the main idea underlying the heuristic.

Hanen and Munier’s heuristic – Given a coarse-grain cDAG, we define the following integer linear program.

**DEFINITION 7.10.** Let \( G = (V, E, w, c) \) be a cDAG. We define the integer linear program ILP\((G)\) as follows:

**Minimize** \( M_{\text{opt}} \) subject to
\[
\begin{align*}
\forall(T, T') \in E \quad x_{T,T'} & \in \{0, 1\} & \quad \text{(A)} \\
\forall(T, T') \in E \quad x_{T,T'} & \in \{0, 1\} & \quad \text{(B)} \\
\forall(T, T') \in E \quad s(T) + w(T) + x_{T,T'} c(T, T') & \leq s(T') & \quad \text{(1)} \\
\forall(T, T') \in E \quad s(T) + w(T) + x_{T,T'} c(T, T') & \leq s(T') & \quad \text{(2)} \\
\forall(T, T') \in E \quad s(T) + w(T) & \leq M_{\text{opt}} & \quad \text{(3)} \\
\forall(T, T') \in E \quad s(T) + w(T) & \leq s(T') & \quad \text{(4)}
\end{align*}
\]

We refer to [106] for a primer on (integer) linear programming. Intuitively, the makespan \( M_{\text{opt}} \) is the maximum of the completion times of all tasks, which is expressed by constraint (4).

**LEMMA 7.6.** Let \( G = (V, E, w, c) \) be a cDAG. The solution \( M_{\text{opt}} \) of the integer linear program ILP\((G)\) is equal to the optimal makespan with unlimited processors \( M_{\text{opt}}(G) \).

**Proof.** We show that there is a one-to-one correspondence between valid schedules for \( G \) (with unlimited processors) and solutions to the integer linear program ILP\((G)\).

Let \( \sigma \) be a valid schedule for \( G \). Let \( s(T) = \sigma(T) \) for each task \( T \), and let \( x_{T,T'} = 0 \) if \( T' \) is the favorite successor of \( T \), and \( x_{T,T'} = 1 \) otherwise. Then, all constraints of ILP\((G)\) are met:

- Constraints (A) and (B) are met by construction.
- Constraint (1) was derived above.
- Constraint (2) expresses the fact that each task has at most one favorite successor.
- Constraint (3) expresses the fact that each task is the favorite successor of at most one task, which can be proved quite similarly to constraint (2).
- The definition of the makespan is \( M_{\text{opt}}(\sigma) = \max_{T \in V} (\sigma(T) + w(T)) \), hence constraint (4) is met.

Let \( M_{\text{opt}}(\sigma) \) be the value returned by ILP\((G)\) when all variables \( s(T) \) and \( x_{T,T'} \) are defined as above. Because constraint (4) is the only constraint on the objective function, we have \( M_{\text{opt}}(\sigma) = M_{\text{opt}}(G) \).

Reciprocally, consider a solution of the optimization problem ILP\((G)\). To define the induced schedule \( \sigma \), we need to determine for each task both a starting time and the processor that executes it. For starting times, we simply let \( s(T) = s(T) \) for any task \( T \in V \). We define the allocation function as follows:
\[
\forall e = (T, T') \in E, x_{T,T'} = 0 \Rightarrow \text{alloc}(T) = \text{alloc}(T').
\]

To be more precise, we allocate entry tasks to different processors, and we traverse the graph to compute the allocation function as follows: if \( T' \) is an immediate successor of \( T \) and \( x_{T,T'} = 1 \), we allocate \( T' \) to a new processor, otherwise we allocate \( T' \) to the same processor as \( T \). We have no conflict during this traversal. Indeed, due to condition (2), for each task \( T' \in V \), there is at most one immediate successor of \( T \) allocated to the same processor as \( T \). This successor, if it exists, is the unique task \( T'' \) such that \( x_{T,T''} = 0 \) (\( T'' \) is then the favorite successor of \( T \)). Similarly, due to condition (3), for each task \( T' \in V \), there is at most one predecessor of \( T' \) allocated to the same processor as \( T' \). Furthermore, constraint (1) together with the choice of the allocation function ensure that all dependence constraints are met: \( \sigma \) is a valid schedule for \( G \). Finally, condition (4) shows that \( M_{\text{opt}} \) is equal to the makespan of \( \sigma \). \( \square \)
Given a solution to ILP(G), we can interpret $s(T)$ as the top level of $T$, where the bottom and top levels are computed according to the allocation function induced by the variables $x_{T,T'}$. We add the communication cost $c(T,T')$ into the weight of a path going from $T$ to $T'$ if and only if $\text{alloc}(T) \neq \text{alloc}(T')$, i.e., if and only if $x_{T,T'} = 1$. Condition (4) shows that the solution to ILP(G) is indeed equal to the value of the maximal weight of a path in the dependence graph computed using the previous rules. The difficulty lies in determining the allocation function, i.e., in determining the $x_{T,T'}$ values. Because these values are integers (and even restricted to 0 or 1), the ILP problem is an integer linear program, which is NP-hard [106]. However, if we relax the condition that the $x_{T,T'}$'s are integers, we obtain a linear program with rational numbers, whose complexity is known to be polynomial [106].

DEFINITION 7.11. Let $G=(V,E,w,c)$ be a DAG:

- We define the relaxed linear program $\text{RLP}(G)$ as the program obtained by replacing equation (A) in the definition of ILP(G) by the equation

$$\forall (T,T') \in E, \ 0 \leq x_{T,T'} \leq 1 .$$

Now the variables $x_{T,T'}$ are rational numbers instead of integers.

- We let $(x_{T,T'}^{\text{rel}}, s^{\text{rel}}(T), \text{MS}^{\text{rel}})$ denote the solution of the relaxed problem $\text{RLP}(G)$ over the rational numbers.

Hanen and Munier define their schedule $\sigma^{\text{hm}}$ directly from the solution of the relaxed linear program $\text{RLP}(G)$. Let $T \in V$ be any task. Constraint (2) ensures that there is at most one successor $T'$ of $T$ such that $x_{T,T'}^{\text{rel}} < \frac{1}{2}$, and constraint (4) ensures that there is at most one predecessor $T''$ of $T$ such that $x_{T'',T}^{\text{rel}} < \frac{1}{2}$; Therefore, let $x_{T,T'}^{\text{rel}} = 0$ for any edge $e=(T,T') \in E$ such that $x_{T,T'}^{\text{rel}} < \frac{1}{2}$, and $x_{T,T'}^{\text{rel}} = 1$ otherwise. For any task $T \in V$, define $s_{T}^{\text{hm}}$ to be the top level of $T$, where the bottom and top levels are computed according to the allocation function induced by the $x_{T,T'}^{\text{rel}}$. We add the communication cost $c(T,T')$ to the weight of a path going from $T$ to $T'$ if and only if $\text{alloc}(T) \neq \text{alloc}(T')$, i.e., if and only if $x_{T,T'}^{\text{rel}} = 1$. As explained earlier, this defines a valid schedule for $G$.

THEOREM 7.11. Let $G=(V,E,w,c)$ be a coarse-grain DAG, with granularity $g(G) \geq 1$. Let $\sigma^{\text{hm}}$ be the schedule defined by Hanen and Munier. Then,

$$\text{MS}(\sigma^{\text{hm}}) \leq \text{MS}_{\text{opt}}(G) \leq 2g(G) + 2 2g(G) \leq 2g(G) + 1 .$$

Proof. For any path in the graph going from a task $T$ to one of its successors $T'$, we have the communication cost $x_{T,T'}^{\text{rel}}c(T,T')$ for Hanen and Munier’s schedule, instead of $x_{T,T'}^{\text{rel}}c(T,T')$ for the solution of $\text{RLP}(G)$. Two cases occur:

- $x_{T,T'}^{\text{rel}} = 0$: then $w(T) + x_{T,T'}^{\text{rel}}c(T,T') \leq w(T) + x_{T,T'}^{\text{rel}}c(T,T')$.

- $x_{T,T'}^{\text{rel}} = 1$: then $x_{T,T'}^{\text{rel}} \geq \frac{1}{2}$, we have

$$\frac{w(T) + x_{T,T'}^{\text{rel}}c(T,T')}{w(T) + x_{T,T'}^{\text{rel}}c(T,T')} \leq \frac{1 + \frac{c(T,T')}{w(T)}}{1 + \frac{c(T,T')}{w(T)}} \leq 1 + \frac{c(T,T')}{w(T)} \leq 2g(G) + 2 2g(G) + 1 .$$

In all cases, $w(T) + x_{T,T'}^{\text{rel}}c(T,T') \leq 2g(G) + 2 2g(G) + 1$. This inequality extends to all paths in the graph, which proves the theorem.

An immediate consequence of Theorem 7.11 is that Hanen and Munier’s heuristic is guaranteed with a factor at most $\frac{2}{3}$ for coarse-grain graphs.

7.7 List Heuristics for Pb(p) with Communications

As expected, the limited processors scheduling problem Pb(p) remains NP-complete when introducing communication costs. Pb(p) does remain in the NP class; once the allocation is known, dependence and resource constraints can be checked by traversing the graph. The true problem is to determine a good allocation. The most natural idea is to extend the critical path list algorithm introduced in Section 7.4.4. We explain how to modify it in a straightforward fashion and then how to design a much improved version.

7.7.1 Naive Critical Path

The idea of critical path scheduling remains the same: list scheduling with task priorities equal to the task bottom levels. The problem is that we do not know how to compute bottom levels. Without knowing the allocation, it is not possible to decide which communication costs should be taken into account. A conservative approach is to include all communication costs when computing bottom levels (which amounts to assuming one distinct processor per task).

Consider again the example DAG in Figure 7.7, with corresponding task bottom levels shown in Table 7.3. We check that the bottom level of task $T_1$ is 14, the length of the ASAP schedule with one processor per task. Let us build a list schedule based on the values of these bottom levels. The algorithm proceeds as explained in Section 7.4.4, but with an important
we see that we made a wrong decision when assigning worse than the execution on a single processor! There must be room for this is the only di

would have been executed by path (na

FIGURE 7.11: Naïve critical path scheduling for the example DAG in Figure 7.7.

TABLE 7.3: Task bottom levels for the DAG in Figure 7.7.

<table>
<thead>
<tr>
<th>Task</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical Path</td>
<td>14</td>
<td>8</td>
<td>11.5</td>
<td>6.5</td>
<td>6.5</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

difference. A task is free when all its predecessors have been executed. But a free task cannot start execution as soon as it is free, even if there are available processors; depending on the allocation decision, we may or may not need to wait for some communication delay.

Assume p = 3 available processors P1, P2, and P3. When there are more available processors than free tasks, assign the tasks to the processors with (say) the lowest indices. Let Q be the priority queue of free tasks. In our example, there is r = 1 free task at time t = 0: Q = \{T1\}. Processor P1 executes T1 at t = 0. At t = 1, we update Q as Q = \{(T2, T3)\} (T1 is given priority over T2 because its bottom level is larger). All processors are available, so according to our rule we allocate T2 to P1 and T3 to P2. Note that we are lucky to assign T2 to P1: Because P1 has executed T1, there is no communication delay to pay, hence it can start T3 at time t = 1. On the other hand, P2 must wait until time t = 6 to start T2. At time t = 2, we have Q = \{(T4, T5)\} (breaking ties arbitrarily, using task numbers), and two available processors P1 and P3. Indeed, although still idle, P3 has been marked busy until it returns from the execution of T1. Hence, we allocate T4 to P1 (execution can start immediately) and T5 to P3 (execution cannot start before t = 5). In the end, we obtain the schedule shown in Figure 7.11.

FIGURE 7.11: Naive critical path scheduling for the example DAG in Figure 7.7.

We obtain a makespan equal to 14 time units. Note that the naïve critical path (naïve CP) scheduling with two processors leads to the same result: T2 would have been executed by P1 at time t = 4 rather than by P2 at time t = 5: this is the only difference. In both cases, we obtain the same makespan, even worse than the execution on a single processor! There must be room for improvement.

7.7.2 Modified Critical Path

If we analyze the execution of naïve CP scheduling on our small example, we see that we made a wrong decision when assigning T1 to P2. Indeed, at time t = 1 we had Q = \{(T3, T2)\}. The first allocation, that of T3 to P2, is fine. But the second, that of T2 to P3, is not. We should allocate it to P1 again, even though it is not available. The reason is that P1 can start T2 earlier than P2. The rule of modified critical path (MCP) is: Allocate a free task to the processor that allows its earliest execution, given previous task allocation decisions. It is important to explain further what "previous task allocation decisions" means. Free tasks from the queue are processed one after the other. At any moment, we know which processors are available and which ones are busy. Moreover, for the busy processors, we know when they will finish computing their currently allocated tasks. Hence, we can always select the processor that can start the execution of the task under consideration the earliest. It may well be the case that we select a processor that is currently busy, as discussed earlier for allocating task T2.

Rather than writing the details of MCP in algorithmic form, we apply it in our example with three processors. We obtain the schedule shown in Figure 7.12. As already discussed, T2 is allocated to P1. Similarly, T4 is allocated to P1, because it allows execution at time t = 8.5, against 9.5 on P1 or P3. The new makespan is 10.5, to be compared with the makespan of 14 of naïve CP.  

7.7.3 Hints for Comparison

The comparison of naïve CP and MCP for our little example should not lead to drastic conclusions. We are comparing two heuristics; neither of them is always superior to the other. There are examples where naïve CP is better than MCP. For the example shown in Figure 7.13 with two processors, one can check that the makespan of MCP is 15, while that of naïve CP is 14.

Our intuition, however, shows that MCP is likely to outperform naïve CP in most cases. How do we quantify this assertion? To be less specific than with particular examples, let us compare CP and MCP on simple graph types. We analyze two very simple cases, a fork with two nodes and a join with two nodes. In Figures 7.14(a) and 7.14(b), we have three tasks of the same weight w. The communication costs are all equal to c. Assume two processors are
ally speaking, there are three possible approaches: scheduling problem is NP-complete, and we only compare heuristics. Gener-

This short discussion shows how di-

FIGURE 7.14: An example where naïve CP is better than MCP.

FIGURE 7.13: Elementary graphs for comparing naïve CP and MCP.

available, $P_1$ and $P_2$:

1. **Fork Graph** (Figure 7.14(a)). Naïve CP schedules $T_1$ on $P_1$ at time $t = 0$. Then, it schedules $T_2$ on $P_1$ at time $t = w$, and $T_3$ on $P_2$ at time $w + c$, hence a makespan equal to $2w + c$. MCP does the same as naïve CP if $w > c$. But if $w < c$, the earliest execution time for $T_3$ is $t = 2w$ on $P_1$, hence MCP schedules all tasks on $P_1$, and its makespan is equal to $3w < 2w + c$. Conclusion: MCP outperforms naïve CP if $w < c$ and ties it otherwise.

2. **Join Graph** (Figure 7.14(b)). Naïve CP and MCP perform identically. At time $t = 0$, they schedule $T_2$ on $P_1$ and $T_3$ on $P_2$. At time $t = w + c$, $T_1$ is scheduled on either $P_1$ or $P_2$, and the makespan is $2w + c$. Note that is not optimal if $w < c$; it is better to schedule the three tasks on the same processor! Of course this is forbidden in list scheduling: No processor can be deliberately kept idle.

This short discussion shows how difficult it is to draw conclusions. The scheduling problem is NP-complete, and we only compare heuristics. Gener-

ally speaking, there are three possible approaches:

1. **Theoretical**: Prove that the heuristic is guaranteed, i.e., that it always leads to a makespan within some fraction of the optimal (in other words, make it an approximation algorithm).

2. **Experimental**: Use random graphs and “standard benchmark” graphs to compare heuristics. The difficulty is that there is little consensus on which benchmarks are representative of large and relevant application classes.

3. **Tricky**: Prove that the heuristic is optimal for certain classes of graphs: forks, joins, fork-joins, trees, etc.

The first approach is the strongest: One is ensured that the heuristic will perform within a certain factor of the optimal in the worst case. The second approach is quite useful (and used) in practice. And the third approach helps to tune the heuristics so as to be optimal for certain graph classes (and maybe to publish nice research papers!).

A small step in the first direction is the following (rather weak) counterpart of Theorem 7.6:

**THEOREM 7.12.** Let $G = (V, E, w, c)$ be a cDAG of granularity $g(G)$ (see Section 7.6.2), and let $MS_{opt}(p)$ be the makespan of an optimal sched-

ule. Then, we can derive a schedule $\sigma$ with $p$ processors whose makespan verifies

$$MS(\sigma, p) \leq \left(1 + \frac{1}{p}\right) (1 + g(G)) MS_{opt}(p).$$

**Proof.** The proof is straightforward. Neglect all communication costs and construct a list schedule $\sigma$: its makespan is such that

$$MS(\sigma, p) \leq \left(1 + \frac{1}{p}\right) MS_{opt}(p) \leq \left(2 - \frac{1}{p}\right) MS_{opt}(p),$$

where $MS_{opt}(p)$ is the optimal makespan without communication costs. Then, we stretch the schedule by a factor $1 + g(G)$, which allows us to pay for the communication cost incurred from the predecessors of each task $T_i$. We have an interval of length $(1 + g(G)) w_i$ to communicate data from the predecessors of $T_i$ and execute it. Therefore, we have derived a valid schedule whose makespan meets the desired bound. Note that this schedule is not necessarily a list schedule because we may have waited longer than needed to execute some tasks.

### 7.8 Extension to Heterogeneous Platforms

This section explains how to extend list scheduling techniques to hetero-

genous platforms, i.e., to platforms that consist of processors with different speeds and interconnection links with different bandwidths. We have discussed the issue of static load balancing on platforms with homogeneous processors (but homogeneous network links) in Chapter 6. Rather than defining all nota-
tions precisely, we proceed rather informally and focus on the key differences with the homogeneous case.

We start with a cDAG with $n$ tasks $T_1, \ldots, T_n$. The goal is to schedule this cDAG on a platform with $p$ heterogeneous processors $P_1, \ldots, P_p$. There are many parameters to instantiate:
7.8. Extension to Heterogeneous Platforms

Computation costs – The execution cost of $T_i$ on $P_j$ is modeled as $w_{ij}$. Therefore, an $n \times p$ matrix of values is needed to specify all computation costs. This matrix comes directly for the specific scheduling problem at hand. However, when attempting to evaluate competing scheduling heuristics over a large number of synthetic scenarios, one must generate this matrix. One can distinguish two approaches. In the first one, the execution costs are represented as a matrix with $w_{ij} = w_{ij} \times \gamma_i$, where $w_{ij}$ represents the number of operations required by $T_i$, and $\gamma_i$ is the inverse of the speed of $P_i$ (in operations per second). With this definition the relative speed of the processors does not depend on the particular task they execute. If instead some processors are faster for some tasks than some other processors, and slower for other tasks, one speaks of an inconsistent (or non-uniform) matrix. This corresponds to the case in which some processors are specialized for some tasks (e.g., specialized hardware or software).

Communication costs – Just as processors have different speeds, communication links may have different bandwidths. However, while the speed of a processor may depend upon the nature of the computation it performs, the bandwidth of a link does not depend on the nature of the bytes it transmits. It is therefore natural to assume consistent (or uniform) links. If there is a dependence $e_{ij} : T_i \rightarrow T_j$, if $T_i$ is executed on $P_q$ and $T_j$ executed on $P_r$, then the communication time is modeled as

$$\text{comm}((i, j, q, r) = \text{data}(i, j) \times v_{qr}.$$

where $\text{data}(i, j)$ is the data volume associated to $e_{ij}$ and $v_{qr}$ is the communication time for a unit-size message from $P_q$ to $P_r$ (i.e., the inverse of the bandwidth). Like in the homogeneous case, we let $v_{qr} = 0$ if $q = r$, i.e., if both tasks are assigned the same processor. If one wishes to generate synthetic scenarios to evaluate competing scheduling heuristics, one then must generate two matrices: one of size $n \times n$ for data and one of size $p \times p$ for $v_{qr}$.

The main list scheduling principle is unchanged. As before, we need to compute the priority of each task so as to decide which one to execute first when there are more free tasks than available processors. In other words, we need to find the equivalent of bottom levels for MCP. The most natural idea is to compute averages of computation and communication times, and use these to compute priority levels exactly as in the homogeneous case. We define:

- $w_{\text{avg}} = \frac{\sum_{i=1}^{n} w_{ii}}{n}$, the average execution time of $T_i$;
- $\text{comm}_{\text{avg}} = \text{data}(i, j) \times \frac{\sum_{e_{ij} \in E} \text{data}(i, j) \times v_{qr}}{p \times p}$, the average communication cost for edge $e_{ij} : T_i \rightarrow T_j$.

The last (but important) modification concerns the way in which tasks are assigned to processors: Instead of assigning the current task to the processor that will start its execution first (given all already taken decisions), we should assign it to the processor that will complete its execution first (given all already taken decisions). Both choices are equivalent with homogeneous processors, but intuitively the latter is likely to be more efficient in the heterogeneous case.

Altogether, we have re-discovered the list heuristic called HEFT, for heterogeneous earliest finish time [115]. The complexity of the algorithm as we have outlined it here is the same as that of MCP. More sophisticated versions attempt to insert tasks in intervals of time during which processors are idle. This technique is called insertion scheduling. Instead of scheduling a new task after those already assigned to a given processor, a good idea may be to try and schedule it at some earlier time, provided that there exists an interval long enough to accommodate the task and during which the processor was idle (most likely waiting for some communication to complete).

Bibliographical Notes

All the material covered in this chapter is rather basic. Without communication costs, we mention that pioneering work includes the book by Coffman [42], Chapter 9 of [82]; the book by El-Rewini, Lewis, and Ali [53]; and the IEEE compilation of papers [107] provide additional material. On the theoretical side, Appendix A5 of Garcia and Johnson [57] provides a list of NP-complete scheduling problems. Also, the book by Brucker [37] offers a comprehensive overview of many complexity results.

The literature with communication costs is more recent. Theorem 7.10 is due to Chrétienne [40]. Picouleau [93, 94] proves that PB(sc) remains NP-complete even when we assume all task weights and communication costs to have the same (unit) value — this is the so-called UET-UCT problem (unit execution time-unit communication time) — or even if communication costs are arbitrarily small (but non-zero). Several extensions to Theorem 7.10 are discussed in the survey paper by Chrétienne and Picouleau [41]. Hansen and Munier’s heuristic can be extended to cope with limited processors; see [66]. See also the book by Darte, Robert and Vivien [49], where many clustering heuristics are surveyed. Finally, a recent book by Sinnem [108] provides a thorough discussion on communication models. In particular, it describes several extensions for modeling and accounting for communication contention.