CHAPTER 4

Task Scheduling

This chapter introduces the general problem of task scheduling for parallel systems. It formulates the static task scheduling problem and discusses its theoretic background. Hence, the chapter's aim is to introduce the reader to task scheduling and to lay the groundwork for the following chapters. This is done by providing a uniform and consistent conceptual framework.

The scheduling problem defined in this chapter is based on a strongly idealized model of the target parallel system. It has been shown that this model, referred to as the *classic model*, is often not sufficient to obtain schedules with high accuracy and short execution times. More sophisticated models are therefore discussed in Chapters 7 and 8. Yet, most algorithms found in the literature have been proposed for the classic model. In order to build a comprehensive understanding of task scheduling, it is thus indispensable to analyze and discuss this model and its algorithms. What is more, most of the techniques and algorithms can be employed for the more advanced models studied in later chapters.

The chapter starts with terminology and basic definitions for task scheduling and formulates the scheduling problem. Its associated decision problem is NP-complete and a corresponding proof is provided. Subsequently, task scheduling without communication costs is studied as a special case of the general problem. Again, the complexity is discussed, including the NP-completeness of this problem. The chapter then returns to the task graph model to analyze its properties in connection with task scheduling.

4.1 FUNDAMENTALS

In this chapter, and in the rest of this book, *static* task scheduling is addressed. Static scheduling usually refers to the scheduling at compile time, as opposed to *dynamic* scheduling, where tasks are scheduled during the execution of the program, hence at runtime. It follows that the computation and communication structure of the program and the target parallel system must be completely known at compile time and the implications of this condition were already discussed in connection with the task graph model in the previous chapter (Section 3.5.2).

The discussion in this chapter is oriented toward what can be considered the general scheduling problem: no restrictions are imposed on the input task graph—it may have arbitrary computation and communication costs as well as an arbitrary structure—and the number of processors is limited.

Another important, less general scheduling problem does not consider the costs of communication. Historically, the investigation of scheduling without communication delays preceded the consideration of communication costs in scheduling. Nevertheless, this chapter will start with the general problem that includes communication delays. This has the advantage that scheduling without communication delays can easily be treated afterwards as a special case of the general problem. This improves the understanding and interrelation of the task scheduling problems and also reduces the number of necessary definitions. Furthermore, the more advanced task scheduling discussed in later chapters is based on scheduling with communication delays.

Several other scheduling problems arise by restricting the task graph, for example, by having unit computation or communication costs, or by employing an unlimited number of processors. Again, these problems can be treated as special cases of the general problem and they are discussed later in Section 6.4.

Basics With the preparations of the previous chapter, the notion of a schedule and associated conditions can be defined right away. A system of tasks together with their precedence constraints is already defined by the task graph model.

In Section 2.3, when the parallelization process was outlined, the scheduling problem was introduced as the spatial and temporal assignment of tasks to processors. The spatial assignment, or mapping, is the allocation of tasks to the processors.

Definition 4.1 (Processor Allocation) *A processor allocation $\mathcal{A}$ of the task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ (Definition 3.13) on a finite set $\mathbf{P}$ of processors is the processor allocation function proc: $\mathbf{V} \rightarrow \mathbf{P}$ of the nodes of G to the processors of* $\mathbf{P}$.

The temporal assignment is the attribution of a start time to each task. Even though scheduling refers only to the attribution of start times, it presupposes the allocation of the tasks to processors and therefore commonly both are defined by a schedule.

Definition 4.2 (Schedule) *A schedule $\mathcal{S}$ of the task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ on a finite set $\mathbf{P}$ of processors is the function pair $(t_s, proc)$, where*

- *t_s: $\mathbf{V} \rightarrow \mathbb{Q}_0^+$ is the start time function of the nodes of G.*
- *proc: $\mathbf{V} \rightarrow \mathbf{P}$ is the processor allocation function of the nodes of G to the processors of* $\mathbf{P}$.

So the two functions t_s and *proc* describe the spatial and temporal assignment of tasks, represented by the nodes of the task graph, to the processors of a target parallel system, represented by the set $\mathbf{P}$. The node $n \in \mathbf{V}$ is scheduled to start execution at

$t_s(n)$ on processor $proc(n) = P$, $P \in \mathbf{P}$, which is denoted by $t_s(n, P)$; hence,

$$t_s(n, P) \Leftrightarrow t_s(n), proc(n) = P,\ P \in \mathbf{P}. \tag{4.1}$$

4.2 WITH COMMUNICATION COSTS

Definition 4.2 describes a schedule, but it does not ensure the compliance with the precedence constraints of the task graph. Conditions for this will be established shortly, but first the model of the target parallel system must be defined.

Definition 4.3 (Target Parallel System—Classic Model) *A target parallel system* **P** *consists of a set of identical processors connected by a communication network. This system has the following properties:*

1. Dedicated System. *The parallel system is dedicated to the execution of the scheduled task graph. No other program or task is executed on the system while the scheduled task graph is executed.*
2. Dedicated Processor. *A processor $P \in \mathbf{P}$ can execute only one task at a time and the execution is not preemptive.*
3. Cost-Free Local Communication. *The cost of communication between tasks executed on the same processor, local communication, is negligible and therefore considered zero. This assumption is based on the observation that for many parallel systems remote communication (i.e., interprocessor communication) is one or more orders of magnitude more expensive than local communication (i.e., intraprocessor communication).*
4. Communication Subsystem. *Interprocessor communication is performed by a dedicated communication subsystem. The processors are not involved in communication.*
5. Concurrent Communication. *Interprocessor communication in the system is performed concurrently; there is no contention for communication resources.*
6. Fully Connected. *The communication network is fully connected. Every processor can communicate directly with every other processor via a dedicated identical communication link.*

What kind of parallel systems does this model represent (Section 2.1)? Given the identical processors and the fully connected network of identical communication links, the system is completely homogeneous. The characteristic of expensive remote communication in comparison with local communication is typical for NUMA or message passing architectures, where the memory is distributed across the processors. Other parallel systems are also reflected by this model; essential is the characteristic of expensive remote communication. For example, on a UMA system remote communication can become expensive due to cache effects. Two tasks running on the same processor may communicate via the processor's cache, while they have to communicate through the main memory when they run on different processors. Another cause for expensive remote communication in a UMA system can be the employment of the message passing programming paradigm.

Based on this model, the meaning of the computation and communication costs of the nodes and edges in a task graph, respectively, can be defined. The weights of the task graph elements were introduced as abstract costs in Definition 3.13.

Definition 4.4 (Computation and Communication Costs) *Let* $\mathbf{P}$ *be a parallel system. The computation and communication costs of a task graph* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *expressed as weights of the nodes and edges, respectively, are defined as follows*:

- w: $\mathbf{V} \rightarrow \mathbb{Q}^+$ *is the computation cost function of the nodes* $n \in \mathbf{V}$. *The computation cost* $w(n)$ *of node n is the time the task represented by n occupies a processor of* $\mathbf{P}$ *for its execution.*
- c: $\mathbf{E} \rightarrow \mathbb{Q}_0^+$ *is the communication cost function of the edge* $e \in \mathbf{E}$. *The communication cost* $c(e)$ *of edge e is the time the communication represented by e takes from an origin processor in* $\mathbf{P}$ *until it completely arrives at a* different *destination processor in* $\mathbf{P}$. *In other words,* $c(e)$ *is the* communication delay *between sending the first data item until receiving the last.*

The finish time of a node is thus the node's start time plus its execution time (i.e., its cost).

Definition 4.5 (Node Finish Time) *Let* $\mathcal{S}$ *be a schedule for task graph* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *on system* $\mathbf{P}$. *The finish time of node n is*

$$t_f(n) = t_s(n) + w(n). \tag{4.2}$$

Again, $t_f(n, P)$ is written to denote $t_f(n)$ and $proc(n) = P$.

Because of Property 2 of the system model, it must be ensured that no two nodes occupy one processor at the same time.

Condition 4.1 (Exclusive Processor Allocation) *Let* $\mathcal{S}$ *be a schedule for task graph* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *on system* $\mathbf{P}$. *For any two nodes* $n_i, n_j \in \mathbf{V}$:

$$proc(n_i) = proc(n_j) \Rightarrow \left\{ \begin{array}{ll} & t_s(n_i) < t_f(n_i) \leq t_s(n_j) < t_f(n_j) \\ or & t_s(n_j) < t_f(n_j) \leq t_s(n_i) < t_f(n_i) \end{array} \right. . \tag{4.3}$$

In contrast to the execution time of a task in the target system, which is simply the cost of the respective node, the communication time of an edge e_{ij} depends on the processors of the origin and destination node. As Property 3 states, local communication, that is, communication between tasks on the same processor, is cost free; hence, it takes no time at all—there is no delay. The time of remote communication, that is, between tasks on different processors, is given by the weight of the representing edge. The time at which a communication arrives at the destination processor is defined as the edge finish time.

Definition 4.6 (Edge Finish Time) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $\mathbf{P}$ *a parallel system. The finish time of* $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, *communicated from processor* P_{src} *to* P_{dst}, $P_{\text{src}}, P_{\text{dst}} \in \mathbf{P}$, *is*

$$t_f(e_{ij}, P_{\text{src}}, P_{\text{dst}}) = t_f(n_i, P_{\text{src}}) + \begin{cases} 0 & \text{if } P_{\text{src}} = P_{\text{dst}} \\ c(e_{ij}) & \text{otherwise} \end{cases}. \tag{4.4}$$

So, the arrival time of e_{ij} at processor P_{dst} is given by the finish time of node n_i, that is, the time when the data for e_{ij} is made available by n_i (node strictness, Definition 3.8), plus the communication time of e_{ij}, either local (0) or remote $(c(e_{ij}))$. Following Properties 4–6 of the system model, a communication e_{ij} is always initiated as soon as its origin node n_i finishes: there is no delay due to any form of contention.

Finally, a condition can be formulated that warrants the compliance of the schedule $\mathcal{S}$ with the precedence constraints of the task graph G.

Condition 4.2 (Precedence Constraint) *Let* $\mathcal{S}$ *be a schedule for task graph* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *on system* $\mathbf{P}$. *For* $n_i, n_j \in \mathbf{V}$, $e_{ij} \in \mathbf{E}$, $P \in \mathbf{P}$,

$$t_s(n_j, P) \geq t_f(e_{ij}, proc(n_i), P). \tag{4.5}$$

Condition 4.2 formulates the consequences of a precedence constraint imposed by an edge $e_{ij} \in \mathbf{E}$ on the start time of the destination node $n_j \in \mathbf{V}$ in schedule $\mathcal{S}$. The node strictness obliges n_j to delay its start until the communication of e_{ij} has completely arrived at processor P.

Condition 4.2 guarantees the execution of the nodes in precedence order.

Lemma 4.1 (Feasible Start Order Is Precedence Order) *Let* $\mathcal{S}$ *be a schedule for task graph* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *on system* $\mathbf{P}$. *If Condition 4.2 is fulfilled for all edges* $e \in \mathbf{E}$ *then any ordering of the nodes* $n \in \mathbf{V}$ *in ascending start time complies with the precedence constraints of* G.

Proof. Condition 4.2 guarantees that for any edge $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, $t_s(n_j) \geq t_f(n_i)$, since $t_f(e_{ij}, proc(n_i), proc(n_j)) - t_f(n_i) \geq 0$ (Definitions 4.6 and 4.4). With Definition 4.5 of the node finish time, $t_s(n_j) \geq t_f(n_i) = t_s(n_i) + w(n_i) > t_s(n_i)$, since $w(n_i) > 0$ (Definition 4.4). Hence, in an ascending start time order of the nodes, the origin node n_i of e_{ij} appears before the destination node n_j. Per definition this is a topological order (Definition 3.6), thus a precedence order. □

The two Conditions 4.1 and 4.2 are the only constraints put on a schedule. A very noteworthy consequence is the possible overlap of computation and communication.

While the leaving communications of a node are sent over the network to their destinations, the node's processor can proceed with computation.

A schedule, whose nodes comply with these two conditions, respects the precedence constraints of the task graph and the properties of the system model and such a schedule is called *feasible*.

Definition 4.7 (Feasible Schedule) *Let $\mathcal{S}$ be a schedule for task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ on system $\mathbf{P}$. $\mathcal{S}$ is feasible if and only if all nodes $n \in \mathbf{V}$ and edges $e \in \mathbf{E}$ comply with Conditions 4.1 and 4.2.*

The feasibility of a schedule can easily be verified, as demonstrated by Algorithm 5. The complexity of verifying Condition 4.1 is $O(\mathbf{V} \log \mathbf{V})$ (e.g., with Mergesort, Cormen et al. [42]) and the complexity of verifying Condition 4.2 is $O(\mathbf{V} + \mathbf{E})$, as every edge is only considered once, resulting in the total complexity of $O(\mathbf{V} \log \mathbf{V} + \mathbf{E})$.

Algorithm 5 Verify Feasibility of Schedule $\mathcal{S}$ for $G = (\mathbf{V}, \mathbf{E}, w, c)$ on $\mathbf{P}$

```
▷ Verify Condition 4.1
for each $P \in \mathbf{P}$ do
    Sort $\{n \in \mathbf{V}: proc(n) = P\}$ according to $t_s(n)$. Let sorted nodes be
    $n_{P,0}, n_{P,1}, \ldots, n_{P,l}$.
    Verify that $t_s(n_{P,i}) < t_f(n_{P,i}) \leq t_s(n_{P,i+1}) < t_f(n_{P,i+1})$ for $i = 0, \ldots, l-1$.
end for
▷ Verify Condition 4.2
for each $n_j \in \mathbf{V}$ do
    Verify that $t_s(n_j) \geq t_f(e_{ij}, proc(n_i), proc(n_j))\ \forall e_{ij} \in \mathbf{E}, n_i \in \mathbf{pred}(n_j)$.
end for
```

From now on, all considered schedules are feasible, unless stated otherwise, and the attribute "feasible" is therefore omitted.

Condition 4.2 requires node n_j to wait until all entering communications $e_{ij} \in \mathbf{E}, n_i \in \mathbf{pred}(n_j)$ have arrived at its executing processor. The earliest time when node n_j can start execution is called the *data ready time*.

Definition 4.8 (Data Ready Time (DRT)) *Let $\mathcal{S}$ be a schedule for task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ on system $\mathbf{P}$. The data ready time of a node $n_j \in \mathbf{V}$ on processor $P \in \mathbf{P}$ is*

$$t_{\mathrm{dr}}(n_j, P) = \max_{n_i \in \mathbf{pred}(n_j)} \{t_f(e_{ij}, proc(n_i), P)\}. \tag{4.6}$$

If $\mathbf{pred}(n_j) = \emptyset$, that is, n_j is a source node, $t_{\mathrm{dr}}(n_j) = t_{\mathrm{dr}}(n_j, P) = 0$, for all $P \in \mathbf{P}$.

Note that the DRT can be determined for any processor P of $\mathbf{P}$, independently of the processor to which n_j is allocated. The constraints on the start time $t_s(n)$ of a node n can now be formulated using the DRT.

Condition 4.3 (DRT Constraint) *Let $\mathcal{S}$ be a schedule for task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ on system $\mathbf{P}$. For $n \in \mathbf{V}$, $P \in \mathbf{P}$,*

$$t_s(n, P) \geq t_{\mathrm{dr}}(n, P). \tag{4.7}$$

Thus, Condition 4.3 merges the constraints imposed on the start time of a node by all entering edges according to Eq. (4.5).

The finish time of a processor P is the time when the last node scheduled on P terminates.

Definition 4.9 (Processor Finish Time) *Let $\mathcal{S}$ be a schedule for task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ on system $\mathbf{P}$. The finish time of processor $P \in \mathbf{P}$ is*

$$t_f(P) = \max_{n \in \mathbf{V}: proc(n) = P} \{t_f(n)\}. \tag{4.8}$$

A schedule terminates when the last of the nodes of the task graph G finishes.

Definition 4.10 (Schedule Length) *Let $\mathcal{S}$ be a schedule for task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ on system $\mathbf{P}$. The schedule length of $\mathcal{S}$ is*

$$sl(\mathcal{S}) = \max_{n \in \mathbf{V}} \{t_f(n)\} - \min_{n \in \mathbf{V}} \{t_s(n)\}. \tag{4.9}$$

If $\min_{n \in \mathbf{V}} \{t_s(n)\} = 0$, *this expression reduces to*

$$sl(\mathcal{S}) = \max_{n \in \mathbf{V}} \{t_f(n)\}. \tag{4.10}$$

All schedules considered in this text start at time unit 0; thus, expression (4.10) suffices as the definition of the schedule length. An alternative designation for schedule length, which is quite common in the literature, is makespan.

The set of processors used in a given schedule $\mathcal{S}$ is defined as the set of all processors on which at least one node is executed.

Definition 4.11 (Used Processors) *Let $\mathcal{S}$ be a schedule for task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ on system $\mathbf{P}$. The set of used processors is*

$$\mathbf{Q} = \bigcup_{n \in \mathbf{V}} proc(n). \tag{4.11}$$

Obviously, for any schedule $\mathcal{S}$

$$|\mathbf{Q}| \leq |\mathbf{P}|. \tag{4.12}$$

To conclude the basic definitions, the sequential time of a task graph is defined.

Definition 4.12 (Sequential Time) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph. G's* sequential time *is*

$$seq(G) = \sum_{n \in \mathbf{V}} w(n), \tag{4.13}$$

which corresponds to G's execution time on one processor only (remember, local communication is cost free).

4.2.1 Schedule Example

The above definitions and conditions are illustrated by examining a sample schedule. Figure 4.1(a) depicts a Gantt chart of a schedule for the sample task graph of Figure 3.15 on three processors. A Gantt chart is an intuitive and common graphical representation of a schedule, in which each scheduled object (i.e., node) is drawn as a rectangle. The node's position (i.e., the position of its rectangle) in the coordinate system spanned by the time and space axis (i.e., processor axis) is determined by the node's allocated processor and its start time, while the size of the rectangle reflects the node's execution time.

Some examples from this figure illustrate the foregoing definitions and conditions. The start time of node a executed on processor P_1 is $t_s(a) = 0$ (i.e., $t_s(a, P_1) = 0$) and, with a computation cost of $w(a) = 2$, its finish time is $t_f(a) = t_f(a, P_1) = 2$.

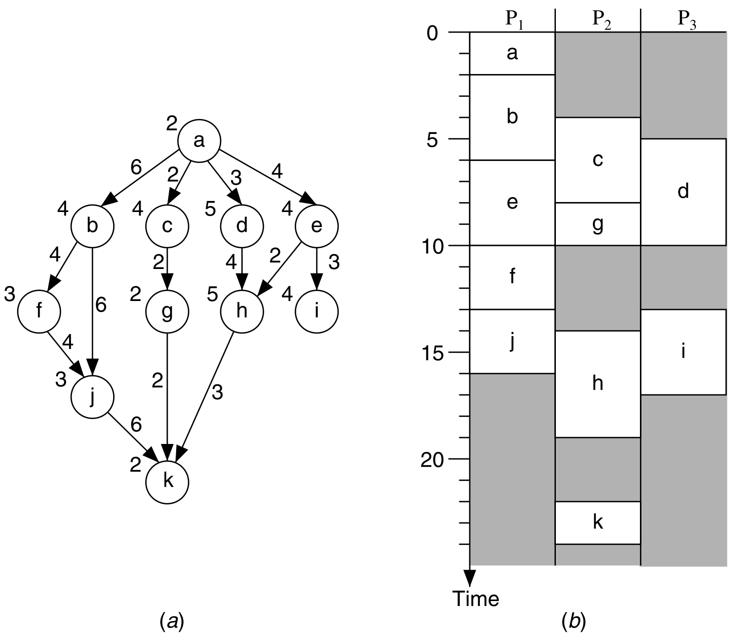


Figure 4.1. The Gantt chart (b) of a schedule for the sample task graph (a) of Figure 3.15 on three processors.

Node d begins at $t_s(d) = 5$ and finishes at $t_f(d) = t_s(d) + w(d) = 5 + 5 = 10$. Its start time $t_s(d) = 5$ is the earliest possible, because it has to wait for the data from node a, $t_f(e_{ad}, P_1, P_3) = t_f(a) + c(e_{ad}) = 2 + 3 = 5$. So in this case, the communication is remote and therefore causes a delay of 3 time units corresponding to the weight of edge e_{ad}. Node b, on the other hand, also receives data from node a, but as the communication is local—both a and b are on the same processor—it takes no time: $t_s(b) = t_f(e_{ab}, P_1, P_1) = t_f(a) = 2$. A good example for the influence of various precedence constraints on the start time of a node is node h. The two entering communications e_{dh} and e_{eh} result in a data ready time of $t_{dr}(h, P_2) = 14$. Responsible is the communication from node d, since d finishes at $t_f(d) = 10$ and the communication of e_{dh} lasts 4 time units, while e_{eh} arrives at P_2 at $t_f(e_{eh}, P_1, P_2) = 10 + 2 = 12$. So node h starts at its DRT. In contrast, node e starts at $t_s(e) = 6$, which is later than its DRT ($t_{\text{dr}}(e, P_1) = t_f(e_{ae}, P_1, P_1) = t_f(a) = 2$), because processor P_1 is busy with node b until that time. The length of the schedule is $sl = t_f(k) = 24$, that is, the finish time of the last node k, as the first node a starts at time unit 0.

4.2.2 Scheduling Complexity

The process of creating a schedule $\mathcal{S}$ for a task graph G on a set of processors $\mathbf{P}$ is called *scheduling*. It should be obvious from the example in Figure 4.1 that there is generally more than one possible schedule for a given graph and set of processors (e.g., a could be executed on processor P_2, which of course would have consequences for the scheduling of the other nodes). Since the usual purpose in employing a parallel system is the fast execution of a program, the aim of scheduling is to produce a schedule of minimal length.

Definition 4.13 (Scheduling Problem) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph and* $\mathbf{P}$ *a parallel system. The scheduling problem is to determine a feasible schedule $\mathcal{S}$ of minimal length sl for G on* $\mathbf{P}$.

Unfortunately, finding a schedule of minimal length (i.e., an optimal schedule) is in general a difficult problem. This becomes intuitively clear as one realizes that an optimal schedule is a trade-off between high parallelism and low interprocessor communication. On the one hand, nodes should be distributed among the processors in order to balance the workload. On the other hand, the more the nodes are distributed, the more interprocessor communications, which are expensive, are performed. In fact, the general decision problem (a decision problem is one whose answer is either “yes” or “no”) associated with the scheduling problem is NP-complete.

The complexity class of NP-complete problems (Cook [41], Karp [99]) has the unpleasant property that for none of its members has an algorithm been found that runs in polynomial time. It is unknown if such an algorithm exists, but it is improbable, given that if any NP-complete problem is polynomial-time solvable then all NP-complete problems are polynomial-time solvable (i.e., P = NP). Introductions to NP-completeness, its proof techniques, and the discussion of many

NP-complete problems can be found in Coffman [37], Cormen et al. [42], and Garey and Johnson [73].

Theorem 4.1 (NP-Completeness) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph and* $\mathbf{P}$ *a parallel system. The decision problem* SCHED(G, $\mathbf{P}$) *associated with the scheduling problem is as follows. Is there a schedule $\mathcal{S}$ for G on $\mathbf{P}$ with length $sl(\mathcal{S}) \leq T, T \in \mathbb{Q}^+$?* SCHED($G$, $\mathbf{P}$) *is NP-complete in the strong sense.*

Proof. First, it is argued that SCHED belongs to NP, then it is shown that SCHED is NP-hard by reducing the well-known NP-complete problem 3-PARTITION (Garey and Johnson [73]) in polynomial time to SCHED. 3-PARTITION is NP-complete in the strong sense. The reduction is inspired by a proof presented in Sarkar [167].

The 3-PARTITION problem is stated as follows. Given a set $\mathbf{A}$ of $3m$ positive integer numbers a_i and a positive integer bound B such that $\sum_{i=1}^{3m} a_i = mB$ with $B/4 < a_i < B/2$ for $i = 1, \ldots, 3m$, can $\mathbf{A}$ be partitioned into m disjoint sets $\mathbf{A}_1, \ldots, \mathbf{A}_m$ (triplets) such that each $\mathbf{A}_i$, $i = 1, \ldots, m$, contains exactly 3 elements of $\mathbf{A}$, whose sum is B?

Clearly, for any given solution $\mathcal{S}$ of SCHED(G, $\mathbf{P}$) it can be verified in polynomial time that $\mathcal{S}$ is feasible (Algorithm 5) and $sl(\mathcal{S}) \leq T$; hence, SCHED(G, $\mathbf{P}$) $\in$ NP.

From an arbitrary instance of 3-PARTITION $\mathbf{A} = \{a_1, a_2, \ldots, a_{3m}\}$, an instance of SCHED($G$, $\mathbf{P}$) is constructed in the following way.

A so-called fork, or send, graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ is constructed as shown in Figure 4.2. It consists of $|\mathbf{V}| = 3m + 1$ nodes $(n_0, n_1, \ldots, n_{3m})$, where node n_0 is the entry node with $3m$ successors. Hence, one edge goes from n_0 to each of the other nodes of G and there are no other edges apart from these. The edge weight of the $3m$ edges is $c(e) = \frac{1}{2} \forall e \in \mathbf{E}$, node n_0 has weight $w(n_0) = 1$, and the $3m$ nodes have weights that correspond to the integer numbers of $\mathbf{A}$, $w(n_i) = a_i, i = 1, \ldots, 3m$. The number of processors of the target system is $|\mathbf{P}| = m$ and the time bound is set to $T = B + 1.5$.

Clearly, the construction of the instance of SCHED is polynomial in the size of the instance of 3-PARTITION.

It is now shown how a schedule $\mathcal{S}$ is derived for SCHED(G, $\mathbf{P}$) from an arbitrary instance of 3-PARTITION $\mathbf{A} = \{a_1, a_2, \ldots, a_{3m}\}$, that admits a solution to 3-PARTITION: let $\mathbf{A}_1, \ldots, \mathbf{A}_m$ (triplets) be m disjoint sets such that each $\mathbf{A}_i$, $i = 1, \ldots, m$, contains exactly 3 elements of $\mathbf{A}$, whose sum is B.

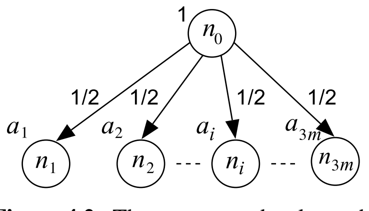


Figure 4.2. The constructed task graph.

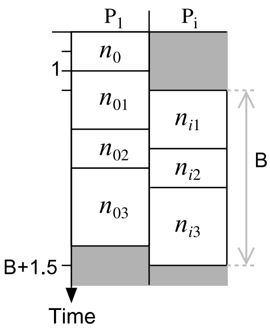


Figure 4.3. Extract of the constructed schedule for P_1 and P_i.

Node n_0 is allocated to a processor, which shall be called P_1. The remaining nodes $n_1, \ldots, n_{3m}$ are allocated in triplets to the processors $P_1, \ldots, P_m$. Let a_{i1}, a_{i2}, a_{i3} be the elements of $\mathbf{A}_i$. The nodes n_{i1}, n_{i2}, n_{i3}, corresponding to the elements of triplet $\mathbf{A}_i$, are allocated to processor P_i. The resulting schedule is illustrated for P_1 and P_i in Figure 4.3.

What is the length of this schedule? The time to execute n_{i1}, n_{i2}, n_{i3} on each processor is B, n_0 is executed in 1 time unit, and the communication from n_0 to its successor nodes takes $\frac{1}{2}$ time unit. Hence, the finish time of all processors P_i, $i = 2, \ldots, m$, is $t_f(P_i) = B + 1.5$. Since the communication on P_1 is local, the finish time on P_1 is $t_f(P_1) = B + 1$. The resulting length of the constructed schedule $\mathcal{S}$ is $sl(\mathcal{S}) = B + 1.5 \leq T$ and therefore the constructed schedule $\mathcal{S}$ is a solution to SCHED(G, $\mathbf{P}$).

Conversely, assume that an instance of SCHED admits a solution, given by the feasible schedule $\mathcal{S}$ with $sl(\mathcal{S}) \leq T$. It will now be shown that $\mathcal{S}$ is necessarily of the same kind as the schedule constructed above.

In this schedule $\mathcal{S}$, each processor can at most spend B time units in executing nodes from $\{n_1, \ldots, n_{3m}\}$. Otherwise the finish time of a processor, say, P_i, spending more that B time units is $t_f(P_i) > w(n_0) + B = 1 + B$. Due to the fact that all a_i's (i.e., node weights) are positive integers, this means that $t_f(P_i) \geq 2 + B$. However, this is larger than the bound T.

Furthermore, with $\sum_{i=1}^{3m} w(n_i) = mB$ and $|\mathbf{P}| = m$, it follows that each processor spends exactly B time units in executing nodes from $\{n_1, \ldots, n_{3m}\}$. Finally, due to $w(n_i) = a_i, B/4 < a_i < B/2, i = 1, \ldots, 3m$, only three nodes can have the exact execution time of B. Hence, the distribution of the nodes on the m processors corresponds to a solution of 3-PARTITION. □

This proof demonstrated the NP-completeness of the scheduling problem SCHED in the strong sense (Garey and Johnson [73]). Alternatively, the NP-completeness can also be shown with a proof based on a reduction from PARTITION (Garey and Johnson [73]). Such a proof (Chrétienne [32]) is slightly less involved, but only demonstrates NP-completeness in the weak sense, since PARTITION is NP-complete in the weak sense. In Exercise 4.3 you are asked to devise a proof based on PARTITION.

The NP-completeness of scheduling has been investigated extensively in the literature (e.g., Brucker [27], Chrétienne et al. [34], Coffman [37], Garey and Johnson [73], Ullman [192]). Several further scheduling problems arise by restricting the task graph, for example, by having unit computation or communication costs or by limiting the number of processors. These problems can be treated as special cases, but only a few of them are known to be solvable in polynomial time. A comprehensive overview of scheduling problems and their complexity is given in Section 6.4.

It will be demonstrated in Theorem 5.1 of Section 5.1 that the scheduling problem is equivalent to finding a node order and a processor allocation. Given an optimal order and allocation, the optimal schedule can be constructed in polynomial time. Of course, finding the order and the allocation is still NP-hard.

Task scheduling remains NP-complete even when an unlimited number of processors is available (Chrétienne [32]). A system with equal or more processors than nodes

$$|\mathbf{P}| \geq |\mathbf{V}| \tag{4.14}$$

can be considered as having an unlimited number of processors, because the task execution is nonpreemptive (Property 2 of Definition 4.3 of the target system model) and only entire tasks are scheduled. Hence, if there are more processors than nodes, the surplus processors will have no tasks to execute and can be removed from the system without influencing the schedule.

Theorem 4.2 (NP-Completeness—Unlimited Number of Processors) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $\mathbf{P}_\infty$ *a parallel system, with* $|\mathbf{P}_\infty| \geq |\mathbf{V}|$. *The decision problem* SCHED(G, $\mathbf{P}_\infty$) *associated with the scheduling problem is as follows. Is there a schedule* $\mathcal{S}$ *for* G *on* $\mathbf{P}_\infty$ *with length* $sl(\mathcal{S}) \leq T, T \in \mathbb{Q}^+$? SCHED($G$, $\mathbf{P}_\infty$) *is NP-complete.*

The proof for this theorem can be based on a reduction from PARTITION, as mentioned earlier, and is left as an exercise for the reader (Exercise 4.3).

As a logical consequence of the NP-completeness of scheduling, the scientific community has been eager to investigate efficient scheduling algorithms based on heuristics or approximation techniques that produce near optimal solutions. In practice, task scheduling must rely on these algorithms due to the intractability of the problem. In the following chapters, many task scheduling algorithms are analyzed, whereby the focus is on common techniques encountered in many algorithms.

This section finishes with a simple lemma regarding the optimal schedule length and the number of processors. In general, the more processors are *available* for the task graph to be scheduled on, the smaller (or equal) the optimal schedule length (Darte et al. [52]).

Lemma 4.2 (Relation Available Processors—Schedule Length) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $\mathbf{P}$, $\mathbf{P}_{+1}$ *two parallel systems with the only difference* *that* $|\mathbf{P_{+1}}| = |\mathbf{P}| + 1$. *For the optimal schedule of G on the two systems, it holds that*

$$sl(\mathcal{S}_{\mathrm{opt}}(\mathbf{P_{+1}})) \leq sl(\mathcal{S}_{\mathrm{opt}}(\mathbf{P})). \tag{4.15}$$

Proof. Assume the optimal schedule $\mathcal{S}_{\mathrm{opt}}(\mathbf{P})$ for the smaller system is given. By adding one processor to the schedule without using it (i.e., the processor is idle), we are not changing the original schedule. Yet, this schedule is a feasible schedule $\mathcal{S}(\mathbf{P_{+1}})$ for the larger system. Hence, $sl(\mathcal{S}(\mathbf{P_{+1}})) = sl(\mathcal{S}_{\mathrm{opt}}(\mathbf{P}))$ and for the optimal schedule on $\mathbf{P_{+1}}$ it must hold that $sl(\mathcal{S}_{\mathrm{opt}}(\mathbf{P_{+1}})) \leq sl(\mathcal{S}_{\mathrm{opt}}(\mathbf{P}))$. □

In Section 4.3.2 it will be seen that the situation is less trivial, when the number of *used* processors (Definition 4.11), that is, nonidle, is considered.

4.3 WITHOUT COMMUNICATION COSTS

As was said at the beginning of this chapter, scheduling without communication costs is a special case of the general scheduling problem that was discussed in the previous section. Nevertheless, it is worthwhile to have a closer look at this "special case," since it has some interesting properties. As already mentioned, historically the discussion of scheduling problems started without consideration of communication costs. As a consequence, the literature of the 1970s regarding task scheduling for parallel systems focused on this scheduling problem that will be defined in the following (Coffman [37], Ullman [192]). Only in the late 1980s and the 1990s were communication delays integrated into the scheduling problem (Chrétienne and Picovleav [35], El-Rewini and Ali [61], Rayward-Smith [158]).

Not considering communication costs corresponds to $c(e) = 0\,\forall e \in \mathbf{E}$ in the task graph model. Thus, a task graph without communication costs can be defined as $G = (\mathbf{V}, \mathbf{E}, w)$ and its edges represent the dependence relations of the nodes, but do not reflect the cost of communication. As a consequence, it can be meaningful to reflect all types of data dependence (flow, output, and antidependence (Section 2.5.1)), not only flow dependence as was done by the general task graph. This was discussed in Section 3.5, when the task graph model was introduced.

Only two small modifications have to be made to the definitions of the foregoing section to completely define the scheduling problem without communication costs. First, the target system model can be significantly simplified and second, the definition of the edge finish time (Definition 4.6) must be modified.

Definition 4.14 (Target Parallel System—Cost-Free Communication) *A target parallel system* $\mathbf{P}_{c0}$ *consists of a set of identical processors connected by a cost-free communication network. This system has the following properties:*

1. Dedicated System. *The parallel system is dedicated to the execution of the scheduled task graph. No other program or task is executed on the system while the scheduled task graph is executed.*

2. Dedicated Processor. *A processor $P \in \mathbf{P}_{c0}$ can execute only one task at a time and the execution is not preemptive.*
3. Cost-Free Communication. *The cost of communication between tasks is negligible and therefore considered zero. Hence, no further considerations regarding communication subsystem, concurrency of communication, and the network structure are necessary for this model.*

The first two properties of this model are identical to Properties 1 and 2 of the target system of the classic model that considers communication costs (Definition 4.3). The third property is new, which supersedes Properties 3–6 of Definition 4.3. There are no communication costs, thus nothing has to be specified or defined regarding the communication subsystem.

Based on this model, it is obvious how to redefine the edge finish time (Definition 4.6), because the communication time is now always zero.

Definition 4.15 (Edge Finish Time—Cost-Free Communication) *Let $G = (\mathbf{V}, \mathbf{E}, w)$ be a task graph and $\mathbf{P}_{c0}$ a parallel system. The finish time of $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, communicated from processor P_{src} to P_{dst}, $P_{\text{src}}, P_{\text{dst}} \in \mathbf{P}$, is*

$$t_f(e_{ij}, P_{\text{src}}, P_{\text{dst}}) = t_f(n_i). \tag{4.16}$$

All other definitions, conditions, and so on of the previous section were carefully made in such a way that no further modifications are necessary to treat the scheduling problem without communication costs.

Setting $t_f(e_{ij}, P_{\text{src}}, P_{\text{dst}})$ to $t_f(n_i)$ has the interesting consequence that the DRT (Definition 4.8) is no longer a function of the processors to which the involved nodes are scheduled. Since this has important consequences for scheduling under this model, as will be seen later, the DRT for scheduling without communication costs is here explicitly redefined.

Definition 4.16 (Data Ready Time (DRT)—Cost-Free Communication) *Let $\mathcal{S}$ be a schedule for task graph $G = (\mathbf{V}, \mathbf{E}, w)$ on system $\mathbf{P}_{c0}$. The data ready time of a node $n_j \in \mathbf{V}$ is*

$$t_{\text{dr}}(n_j) = \max_{n_i \in \mathbf{pred}(n_j)} \{t_f(n_i)\}. \tag{4.17}$$

If $\mathbf{pred}(n_j) = \emptyset$, that is, n_j is a source node, $t_{\text{dr}}(n_j) = 0$.

4.3.1 Schedule Example

To illustrate the difference between scheduling with and without communication costs, a schedule is examined where communication costs are zero. Essentially, the same example as in Figure 4.1 is used. Again a schedule for the sample task graph of Figure 3.15 on three processors is depicted in Figure 4.4. The obvious difference is that

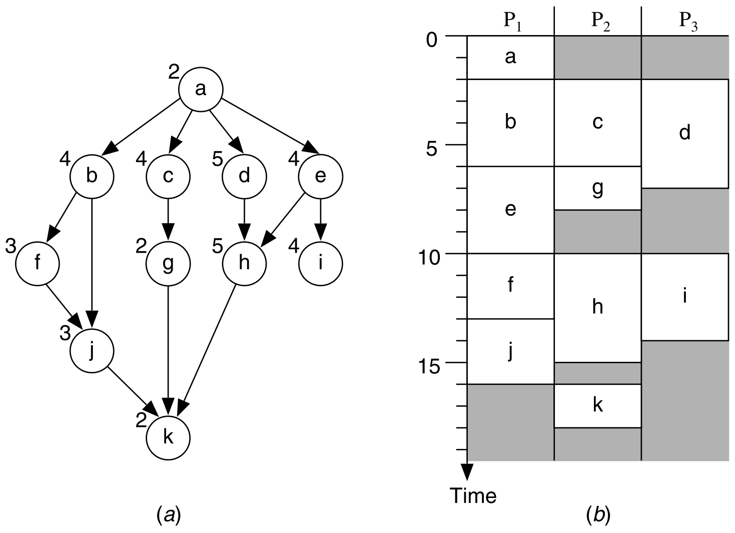


Figure 4.4. Scheduling without communication costs: schedule(*b*) for the sample task graph (*a*) (without edge weights) of Figure 3.15 on three processors. Compare to Figure 4.1.

the edges of the task graph have no weights (which is equivalent to $c(e) = 0\,\forall e \in \mathbf{E}$) as shown in Figure 4.4(*a*). The nodes are scheduled on the same processors and in the same order as in the schedule of Figure 4.1, however with some significant differences in their start times, due to the lack of communication time consideration.

Some examples from Figure 4.4 will illustrate that. The start time of node a on processor P_1 is $t_s(a) = 0$; there is no change in comparison to Figure 4.1, since a is an entry node. In contrast, nodes c and d can now start earlier than in Figure 4.1 and they do so at $t_s(c) = t_s(d) = 2$, immediately after a finishes, $t_f(a) = 2$. This is in compliance with the precedence constraints, $t_s(c) \geq t_f(a)$ and $t_s(d) \geq t_f(a)$, since both node c and node d depend on a through the edges e_{ac} and e_{ad} and the communication time is zero. There is no difference for node b in the schedules of Figure 4.1 and Figure 4.4, since b only depends on a. Hence, the communication is local and thereby no communication costs are inflicted in either of the models; it starts immediately after node a finishes, $t_s(b) = t_f(a) = 2$. Nodes h and i can only start at $t_s(h) = t_s(i) = 10$ (which is, however, much earlier than in Figure 4.1), because both depend on node e, which finishes at $f(e) = 10$. In other words, their DRT is $t_{\mathrm{dr}}(h) = t_{\mathrm{dr}}(i) = 10$. Note that their DRT does not depend on the processor on which they are scheduled. The same is true for node k, which starts at $t_s(k) = 16$. It depends on the three nodes g, h, j, and node j is the last one to finish; hence, k's DRT is $t_{\mathrm{dr}}(k) = \max\{t_f(g), t_f(h), t_f(j)\} = t_j(j) = 16$. The length of the schedule is $sl = t_f(k) = 18$.

4.3.2 Scheduling Complexity

Without communication costs, scheduling remains an NP-complete problem. This is a little bit surprising, given that there is no conflict between high parallelism and low communication costs, as there is in the general problem. On the other hand, problems that are similar to the scheduling problem and seem to be easy at first sight, such as the KNAPSACK or the BINPACKING problems (Cormen et al. [42], Garey and Johnson [73]), are NP-complete, too.

Theorem 4.3 (NP-Completeness—Cost-Free Communication) *Let $G = (\mathbf{V}, \mathbf{E}, w)$ be a task graph and $\mathbf{P}_{c0}$ a parallel system. The decision problem* SCHED-C0(G, $\mathbf{P}_{c0}$) *associated with the scheduling problem is as follows. Is there a schedule $\mathcal{S}$ for G on $\mathbf{P}_{c0}$ with length $sl(\mathcal{S}) \leq T, T \in \mathbb{Q}^+$? SCHED-C0(G, $\mathbf{P}_{c0}$) is NP-complete.*

Proof. The same proof as for Theorem 4.1 is used with one small modification for the construction of an instance of SCHED-C0(G, $\mathbf{P}_{c0}$):

- There are no edge weights, that is, $c(e) = 0 \forall e \in \mathbf{E}$ (instead of $c(e) = \frac{1}{2} \forall e \in \mathbf{E}$ in SCHED(G, $\mathbf{P}$)).

The argumentation is then identical to that in the proof of Theorem 4.1; even the time bound T can remain unchanged. □

Polynomial for Unlimited Processors While in general the scheduling problem without communication costs is NP-complete, it is solvable in polynomial time for an unlimited number of processors. Note that this is in contrast to the general scheduling problem; see Theorem 4.2. A simple algorithm to find an optimal schedule is based on two ideas:

1. Each node is assigned to a distinct processor.
2. Each node starts execution as soon as possible.

Since there is only one task on each processor, the earliest start time is only determined by the precedence constraints of the task graph; processor constraints cannot occur (Condition 4.1). A node can start its execution as soon as its data is ready. From Eq. (4.17) of Definition 4.16, it is clear that the DRT is not a function of the processors on which the involved nodes are executed. This is the essential difference from the general case, where communication costs must be considered. Hence, the earliest possible start time of a node is its DRT, and each node should start at that time:

$$t_s(n) = t_{\text{dr}}(n) \ \forall n \in \mathbf{V}. \tag{4.18}$$

The job of the algorithm is to calculate the DRTs and to assign the start times in the correct order. For the calculation of a node's DRT, the finish time of all its predecessors must be known. This is easily accomplished by handling the nodes in a topological order (Definition 3.6), which guarantees that the finish times of all predecessors $\mathbf{pred}(n_i)$ of a node n_i have already been defined when n_i's DRT is calculated. Algorithm 6 shows this procedure in algorithmic form. It is assumed that the processors and nodes are consecutively indexed, that is, $\mathbf{V} = \{n_1, n_2, \ldots, n_{|\mathbf{V}|}\}$ and $\mathbf{P}_{c0,\infty} = \{P_1, P_2, \ldots, P_{|\mathbf{P}_{c0,\infty}|}\}$.

Algorithm 6 Optimal Scheduling $G = (\mathbf{V}, \mathbf{E}, w)$ on $\mathbf{P}_{c0,\infty}$, with $|\mathbf{P}_{c0,\infty}| \geq |\mathbf{V}|$

```
Insert $\{n \in \mathbf{V}\}$ in topological order into sequential list $L$.
for each $n_i$ in $L$ do
    $DRT \leftarrow 0$
    for each $n_j \in \mathbf{pred}(n_i)$ do
        $DRT \leftarrow \max\{DRT, t_f(n_j)\}$
    end for
    $t_s(n_i) \leftarrow DRT$; $t_f(n_i) \leftarrow t_s(n_i) + w(n_i)$
    $proc(n_i) \leftarrow P_i, P_i \in \mathbf{P}_{c0,\infty}$
end for
```

Theorem 4.4 (Optimal Scheduling on Unlimited Processors) *Let $G = (\mathbf{V}, \mathbf{E}, w)$ be a task graph and $\mathbf{P}_{c0,\infty}$ a parallel system, with $|\mathbf{P}_{c0,\infty}| \geq |\mathbf{V}|$. Algorithm 6 produces an optimal schedule $\mathcal{S}_{\text{opt}}$ of G on $\mathbf{P}_{c0,\infty}$.*

Proof. The produced schedule $\mathcal{S}_{\text{opt}}$ is feasible: (1) each node is scheduled on a distinct processor, hence the processor constraint (Condition 4.1) is always adhered to; and (2) the nodes comply with the precedence constraints (Condition 4.2), since the start time of each node is not earlier than its DRT. The correct calculation of the DRTs is guaranteed by the topological order in which the nodes are processed.

The produced schedule $\mathcal{S}_{\text{opt}}$ is optimal, because each node starts execution at the earliest possible time, the time when its data is ready. The DRT cannot be reduced, since it is not a function of the processors (Definition 4.16); that is, scheduling the involved nodes on other processors does not change the DRT. □

Algorithm 6 has a time complexity of $O(\mathbf{V} + \mathbf{E})$: calculating a topological order is $O(\mathbf{V} + \mathbf{E})$ (Algorithm 4). To calculate the DRT of each node ($O(\mathbf{V})$), every predecessor is considered, which amortizes to $O(\mathbf{E})$ predecessors for the entire graph. Hence, this part is also $O(\mathbf{V} + \mathbf{E})$.

While Algorithm 6 is simple, it does not utilize the processors in an efficient way. Usually, there will be a schedule $\mathcal{S}$ that uses less than $|\mathbf{V}|$ processors, but whose schedule length is also optimal, that is, $sl(\mathcal{S}) = sl(\mathcal{S}_{\text{opt}})$. Yet again, finding the minimum number of processors for achieving the optimal schedule length is an NP-hard problem (Darte et al. [52]). A simple heuristic to reduce the number of processors, while maintaining the optimal schedule length, is linear clustering, which is presented in Section 5.3.2.

Used Processors In Lemma 4.2 of Section 4.2.2 it was stated that the more processors are *available* for the task graph to be scheduled on, the smaller (or equal) the optimal schedule length. The situation becomes less trivial and more interesting when one considers the *used* processors (Definition 4.11) in a schedule.

For scheduling without communication costs, it still holds that the more *used* processors, the smaller (or equal) the optimal schedule length.

Theorem 4.5 (Relation Used Processors—Schedule Length) *Let $G = (\mathbf{V}, \mathbf{E}, w)$ be a task graph and $\mathbf{P}_{c0}$ a parallel system. Let $\mathcal{S}^q$ and $\mathcal{S}^{q+1}$ be the classes of all schedules of G on $\mathbf{P}_{c0}$ that use q, $q < |\mathbf{P}_{c0}|$, and $q+1$, $q+1 \leq |\mathbf{P}_{c0}|$, processors, respectively. For an optimal schedule of each of the two classes it holds that*

$$sl(\mathcal{S}^{q+1}_{\text{opt}}) \leq sl(\mathcal{S}^{q}_{\text{opt}}). \tag{4.19}$$

Proof. Assume an optimal schedule $\mathcal{S}^q_{\text{opt}}$ is given and $\mathbf{Q}$ is the set of processors used by this schedule (Definition 4.11). Let $n \in \mathbf{V}$ be a node scheduled on processor $Q = proc(n), Q \in \mathbf{Q}$ with start time $t_s(n, Q)$. When n is rescheduled from Q onto an idle processor $P \in \mathbf{P}_{c0}, \notin \mathbf{Q}$ then it can start there at its DRT, $t_s(n, P) = t_{\text{dr}}(n)$, since it is the only node on P. This start time cannot be later than its start time on Q, since for the start time $t_s(n)$ it must hold on any processor that $t_s(n) \geq t_{\text{dr}}(n)$ (Condition 4.2); hence, $t_s(n, P) \leq t_s(n, Q)$. Remember, the DRT remains unaltered, since it is not a function of the processors (Definition 4.16). As n does not start later on P than on Q, it also does not finish later on P than on Q. In turn, the DRTs of all successor nodes of n do not increase, which means the entire schedule remains feasible. This new schedule $\mathcal{S}_{\text{new}}$ has $q+1$ used processors and thus belongs to the class $\mathcal{S}^{q+1}_{\text{opt}}$. Its schedule length is the same as that of $\mathcal{S}^q_{\text{opt}}$, $sl(\mathcal{S}_{\text{new}}) = sl(\mathcal{S}^q_{\text{opt}})$. Thus, for the optimal schedule in $\mathcal{S}^{q+1}_{\text{opt}}$ it must hold that $sl(\mathcal{S}^{q+1}_{\text{opt}}) \leq sl(\mathcal{S}^q_{\text{opt}})$. □

It should be stressed that this theorem is only valid for scheduling without communication costs. As can be seen, the proof is based on the fact that the DRT is not a function of the involved processors. This does not hold for the general scheduling problem with communication costs (Definition 4.8). The following example illustrates this. Consider a task graph with a chain structure, as illustrated in Figure 4.5(*a*).

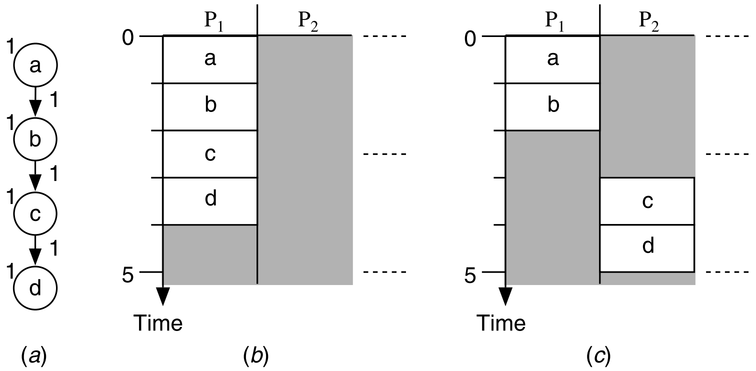


Figure 4.5. Counterexample to Theorem 4.5 when communication costs are considered: chain task graph (*a*) scheduled on one (*b*) and two processors (*c*).

Due to the precedence constraints, the graph must be executed in sequential order, namely, a, b, c, d. Hence, the optimal schedule of this task graph uses only one processor (Figure 4.5(b)). If more than one processor is used (e.g., two as illustrated in Figure 4.5(c)), at least one communication will be remote, which inevitably increases the schedule length by at least one time unit.

4.4 TASK GRAPH PROPERTIES

This section returns to the task graph model as discussed in Section 3.5 of the previous chapter. The concepts presented in the following are employed in the scheduling techniques of the remaining text, in particular, for the assignment of priorities to nodes. In many scheduling algorithms the order in which nodes are considered has a significant influence on the resulting schedule and its length. Gauging the importance of the nodes with a priority scheme is therefore a fundamental part of scheduling.

The discussion begins with the definition of the length of a path in the task graph, based on the computation and communication costs defined in Section 4.1.

Definition 4.17 (Path Length) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph. The length of a path p in G is the sum of the weights of its nodes and edges:*

$$len(p) = \sum_{n \in p, \mathbf{V}} w(n) + \sum_{e \in p, \mathbf{E}} c(e). \tag{4.20}$$

The computation length of a path p in G is the sum of the weights of its nodes:

$$len_w(p) = \sum_{n \in p, \mathbf{V}} w(n). \tag{4.21}$$

Note that the definition of the path length in a task graph differs from the general Definition 3.2 in Section 3.1, where the length of a path equals the number of edges. The path length $len(p)$ can be interpreted as the time the path p takes for its execution if all communications between its nodes are interprocessor communications, which happens, for instance, when each node of p is allocated to a different processor. Due to the sequential order inherent in the path, none of the nodes is executed concurrently with any other node. The computation path length $len_w(p)$ can be interpreted as the execution time of the path p when all communications between its nodes are local, that is, they have zero costs. Consequently, all nodes of p are executed on the same processor. In a task graph $G = (\mathbf{V}, \mathbf{E}, w)$ without communication costs, the path length is necessarily defined as it is in Eq. (4.21).

When the processor allocations of the nodes of p are known, the path length can be determined taking into account that local communications are cost free. In a scheduling algorithm it might be desirable to calculate a path length based on a partial schedule: that is, some processor allocations are given and others are not. The path length determined for a given (partial) processor allocation $\mathcal{A}$ (Definition 4.1) is denoted by $len(p, \mathcal{A})$, the *allocated path length*. Communications between nodes whose processor allocations are unknown are assumed to be remote. The allocated path length $len(p, \mathcal{A})$ is thus in between the path length $len(p)$ and the computation path length $len_w(p)$:

$$len(p) \geq len(p, \mathcal{A}) \geq len_w(p). \tag{4.22}$$

The above scheme for the distinction between the different path lengths is used throughout this text. All definitions based on path lengths will only be formulated for $len(p)$ but are implicitly valid for $len(p, \mathcal{A})$ and $len_w(p)$, too. The corresponding definitions use the same scheme for distinction: that is, the allocated path length is parameterized with $\mathcal{A}$ and the subscript w is used for the computation length.

4.4.1 Critical Path

An important concept for scheduling is the critical path—the longest path in the task graph.

Definition 4.18 (Critical Path (CP)) *A critical path cp of a task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ is a longest path in G*

$$len(cp) = \max_{p \in G}\{len(p)\}. \tag{4.23}$$

The computation critical path cp_w and the allocated critical path $cp(\mathcal{A})$ for a processor allocation $\mathcal{A}$ are defined correspondingly.

The nodes of a critical path cp, consisting of l nodes, are denoted by $n_{cp,1}, n_{cp,2}, \ldots, n_{cp,l}$. Clearly, there might be more than one critical path as several paths can have the same maximum length.

Note that in general

$$cp \neq cp_w \neq cp(\mathcal{A}), \tag{4.24}$$

from which follows for their path lengths

$$len(cp_w) \leq len(cp) \quad \text{and} \quad len(cp(\mathcal{A})) \leq len(cp). \tag{4.25}$$

Equivalent inequalities hold for the computation path length and the allocated path length and the corresponding critical paths.

Lemma 4.3 (Critical Path: From Source to Sink) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph. A critical path cp of G always starts in a source node and finishes in a sink node of G.*

Proof. By contradiction: Suppose cp does not start in a source node. Then, per definition, the first node of cp, here denoted by n_1, has at least one predecessor $n_0 \in \mathbf{pred}(n_1)$; hence, there is the edge $e_{01} \in \mathbf{E}$. A new path q can be concatenated from n_0, e_{01}, and cp, whose length is $len(q) = w(n_0) + c(e_{01}) + len(cp)$. As $w(n_0) > 0$, it follows that $len(q) > len(cp)$—a contradiction. Likewise for the sink node. □

The critical path gains its importance for scheduling from the fact that its length is a lower bound for the schedule length.

Lemma 4.4 (Critical Path Bound on Schedule Length) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph and cp_w a computation critical path of G. For any schedule $\mathcal{S}$ of G on any system* **P**,

$$sl \geq len_w(cp_w). \tag{4.26}$$

Proof. Due to their precedence constraints (Condition 4.2), the nodes of cp_w can only be executed in sequential order, which takes $len_w(cp_w)$ time, independently of the schedule or the number of processors. Thus, the duration of G's execution is at least $len_w(cp_w)$. □

In the worst case that all communications among nodes are remote,

$$sl \geq len(cp), \tag{4.27}$$

yet Eq. (4.26) is also fulfilled.

For the special case of scheduling without communication costs on an unlimited number of processors (Section 4.3.2), the lower bound of the schedule length established by Eq. (4.26) is tight. In other words, the length of the optimal schedule is the length of the critical path.

Lemma 4.5 (Optimal *sl* on Unlimited Processors—Cost-Free Communication) *Let $G = (\mathbf{V}, \mathbf{E}, w)$ be a task graph, cp_w a computation critical path of G, and $\mathbf{P}_{c0,\infty}$ a parallel system, with $|\mathbf{P}_{c0,\infty}| \geq |\mathbf{V}|$. For an optimal length schedule $\mathcal{S}_{\text{opt}}$ of G on system $\mathbf{P}_{c0,\infty}$,*

$$sl(\mathcal{S}_{\text{opt}}) = len_w(cp_w). \tag{4.28}$$

Proof. Theorem 4.4 establishes that Algorithm 6 produces an optimal schedule $\mathcal{S}_{\text{opt}}$ of G on $\mathbf{P}_{c0,\infty}$. In this algorithm, the start time $t_s(n)$ of each node n is set to its DRT

$$t_s(n) = t_{\text{dr}}(n)\ \forall n \in \mathbf{V}. \tag{4.29}$$

By Definition 4.16, node n's DRT is the maximum of the finish times of its predecessor nodes (Eq. (4.17)). Together with $t_f(n) = t_s(n) + w(n)$ (Eq. (4.2) of Definition 4.5), it holds that

$$t_{\text{dr}}(n) = \max_{n_i \in \mathbf{pred}(n)} \{t_{\text{dr}}(n_i\} + w(n_i)\}. \tag{4.30}$$

By induction it is clear that node n's DRT is a sum of ancestor node weights. As per Definition 4.17, this is the computation length len_w of a path composed of ancestor nodes of n. In particular, for the node with the maximum finish time n_{last}, that is, $sl(\mathcal{S}_{\text{opt}}) = t_f(n_{\text{last}})$ (Definition 4.10), it holds that

$$t_f(n_{\text{last}}) = t_{\text{dr}}(n_{\text{last}}) + w(n_{\text{last}}).$$

Thus, $t_f(n_{\text{last}})$ is the length of a path ending with node n_{last}. Per definition of the computation critical path, this path cannot be longer than the critical path, $t_f(n_{\text{last}}) \leq len(cp_w)$. With Eq. (4.26), it follows that $sl(\mathcal{S}_{\text{opt}}) = len(cp_w)$. □

The critical path of the sample task graph (e.g., in Figure 4.1) is $cp = \langle a, b, f, j, k \rangle$ with length $len(cp) = 34$, while there are two computation critical paths cp_w, $\langle a, b, f, j, k \rangle$ and $\langle a, d, h, k \rangle$, with computation length $len_w(cp_w) = 14$.

4.4.2 Node Levels

The critical path identifies the nodes of a task graph, whose constrained sequential execution takes at least the execution time of any other path in the task graph. This makes the nodes of the critical path important, as a late execution start of any of these nodes directly results in an extended schedule length. The scheduling of a less important node, that is, a node not belonging to the critical path, is not that important, as long as it does not delay the execution of any critical path node. But how can one differentiate between nodes not belonging to the critical path? In general, some nodes are more important than others.

This problem is tackled by the notion of node levels, which is a natural extention of the critical path concept on a node-specific basis. Given a node $n \in \mathbf{V}$, in general, paths exist in G that end in n, that is, whose last node is n, and that start with n, that is, whose first node is n (see also the notion of ancestors and descendants in Definition 3.4, Section 3.1). Analogous to the critical path, the longest path in each of the two sets of paths can be distinguished. A node level is then the length of the longest path.

Definition 4.19 (Bottom and Top Levels) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph and $n \in \mathbf{V}$.*

- Bottom Level. *The bottom level $bl(n)$ of n is the length of the longest path starting with n:*

$$bl(n) = \max_{n_i \in \mathbf{desc}(n) \cap \mathbf{sink}(G)} \{len(p(n \to n_i))\}. \tag{4.31}$$

If $\mathbf{desc}(n) = \emptyset$, then $bl(n) = w(n)$. A path starting with n of length $bl(n)$ is called a bottom path of n and denoted by $p_{bl(n)}$.

- Top Level. *The top level $tl(n)$ of n is the length of the longest path ending in n, excluding $w(n)$*:

$$tl(n) = \max_{n_i \in \mathbf{ance}(n) \cap \mathbf{source}(G)} \{len(p(n_i \to n))\} - w(n). \tag{4.32}$$

If $\mathbf{ance}(n) = \emptyset$, *then* $tl(n) = 0$. *A path ending in n of length $tl(n) + w(n)$ is called a top path of n and denoted by $p_{tl(n)}$.*

Considering the proof of Lemma 4.3, it is evident that the longest path starting with a node n must always terminate in a sink node (hence, $n_i \in \mathbf{desc}(n) \cap \mathbf{sink}(G)$ in Eq. (4.31)) and the longest path terminating in n must always start in a source node (hence, $n_i \in \mathbf{ance}(n) \cap \mathbf{source}(G)$ in Eq. (4.32)). This fact gave name to the top level and the bottom level, as the longest paths start at the "top" of the task graph and end at its "bottom." Figure 4.6 illustrates the top and bottom levels of a node n with the corresponding paths in a task graph with several source and sink nodes.

It should be noted that the paths corresponding to the computation node levels—computation bottom level $bl_w(n)$ and computation top level $tl_w(n)$—are not identical to the paths of the levels defined earlier. In general,

$$p_{bl(n)} \neq p_{bl_w(n)} \quad \text{and} \quad p_{tl(n)} \neq p_{tl_w(n)}. \tag{4.33}$$

The significance of the node levels becomes apparent by the following considerations. At the time the execution of a node $n \in G$ is initiated, the minimum remaining time until the execution of G is completed is given by n's bottom level $bl(n)$ (assuming the worst case where all communications are interprocessor communications). The

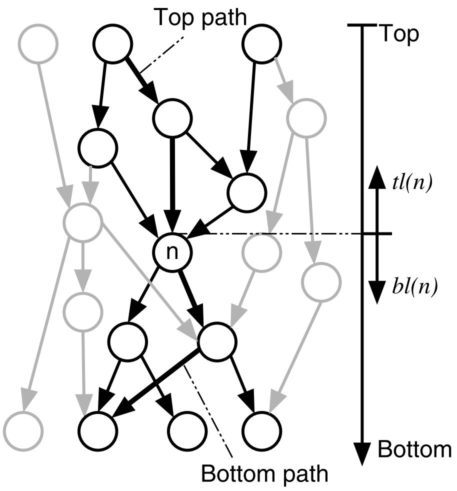


Figure 4.6. Top and bottom levels and the corresponding paths.

precedence constraints of G do not permit any earlier termination, independent of the number of involved processors. The computation bottom level $bl_w(n)$ reflects the best-case minimum time until the termination of G's execution, since the costs of communication are neglected, so they must all be local.

Likewise, a node cannot start earlier than at the time given by its top level $tl(n)$ (once more, assuming that all communications are remote; $tl_w(n)$ corresponds to the case that all communications are local). Recall that the top level does not include the cost $w(n)$ of n, while the bottom level does.

The above observations are formalized in the following lemma.

Lemma 4.6 (Level Bounds on Start Time) *Let $\mathcal{S}$ be a schedule for task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ on system* $\mathbf{P}$. *For $n \in \mathbf{V}$,*

$$sl \geq t_s(n) + bl_w(n) \tag{4.34}$$

$$t_s(n) \geq tl_w(n) \tag{4.35}$$

Proof. Both Eqs. (4.34) and (4.35) are obvious from the definition of $bl_w(n)$ and $tl_w(n)$. For Eq. (4.34), due to precedence constraints, all descendant nodes of n must be executed after n (Condition 4.2). Their execution in precedence order takes at least $bl_w(n)$ time, including the execution time of n. For Eq. (4.35), due to precedence constraints, all ancestors of n must be executed before n (Condition 4.2). Their execution in precedence order takes at least $tl_w(n)$ time. □

Note that Lemma 4.6 relies on the computation levels, to reflect the best possible case. A lemma based on the top and bottom levels needs the additional restriction that all communications among the ancestor and descendent nodes are remote. Lemma 4.6, however, is always true.

The top path $p_{tl(n)}$ and the bottom path $p_{bl(n)}$ of a node $n \in G$ together form a path

$$p_{tb(n)} = p_{tl(n)} \cup p_{bl(n)}, \tag{4.36}$$

which is a longest path of G through n, starting in a source node and ending in a sink node. This path is called a *level path* of n and its length is

$$len(p_{tb(n)}) = tl(n) + bl(n). \tag{4.37}$$

Equation (4.37) holds for every node $n_i \in p_{tb(n)}$, $n_i \in \mathbf{V}$. For the nodes of a critical path, this lemma follows.

Lemma 4.7 (Critical Path Length and Node Levels) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph. For any node $n_{cp,i}$ of a critical path cp*

$$len(cp) = tl(n_{cp,i}) + bl(n_{cp,i}). \tag{4.38}$$

Proof. By contradiction: Divide cp into two subpaths: cp_t ending in $n_{cp,i}$ and cp_b starting with $n_{cp,i}$. The length of cp_t must be $len(cp_t) = tl(n_{cp,i}) + w(n_{cp,i})$: if it was shorter, cp_t could be substituted with the top path $p_{tl(n_{cp,i})}$ of n and the path concatenated from $p_{tl(n_{cp,i})}$ and cp_b would be longer than cp; if it was longer, $tl(n_{cp,i})$ would not be the top level of $n_{cp,i}$, but instead $len(cp_t) - w(n_{cp,i})$. The length of cp_b must be $len(cp_b) = bl(n_{cp,i})$ with the analogous argumentation. Then, $len(cp) = len(cp_t) - w(n_{cp,i}) + len(cp_b) = tl(n_{cp,i}) + bl(n_{cp,i})$. □

With the argument of the above proof, it can also be seen that cp is a level path of each $n_{cp,i}$.

For any source node $n_{\text{src}} \in G$,

$$bl(n_{\text{src}}) = len(p_{tb(n_{\text{src}})}), \tag{4.39}$$

since by definition $tl(n_{\text{src}}) = 0$ if $\mathbf{pred}(n_{\text{src}}) = \emptyset$. Consequently, a source node with the highest bottom level of all nodes is the first node $n_{cp,1}$ of a critical path cp of G:

$$bl(n_{cp,1}) = len(cp) \geq bl(n_i)\,\forall\, n_i \in \mathbf{V}. \tag{4.40}$$

As-Soon/Late-as-Possible Start Times As a result of Eq. (4.35), the top level $tl(n)$ of a node n is sometimes called the as-soon-as-possible (ASAP) start time of n,

$$\text{ASAP}(n) = tl(n). \tag{4.41}$$

This property of the (computation) top level was already used in Theorem 4.4. It establishes that an optimal schedule on an unlimited number of processors for a task graph without communication costs can be constructed by starting each node as-soon-as-possible, Eq. (4.18); hence at its computation top level.

The counterpart, the as-late-as-possible (ALAP) start time of a node n, directly correlates to the bottom level of n. As stated by the inequality Eq. (4.27), the critical path of a task graph is a lower bound for the schedule length. A node n ought to start early enough so that it does not increase the schedule length beyond that bound. Together with Eq. (4.34) the ALAP start time of n is thus

$$\text{ALAP}(n) = len(cp) - bl(n). \tag{4.42}$$

Consequently, arranging nodes in decreasing bottom level order is equivalent to arranging them in increasing ALAP order.

For consistency, the notation of top and bottom levels will be used as a substitute for ASAP and ALAP, respectively, even when algorithms are described that were proposed with the latter notations.

Table 4.1. Node Levels for All Nodes of the Sample Task Graph[a]

Nodes	tl/ASAP	bl	$tl + bl$	ALAP	tl_w/ASAP$_w$	bl_w	$tl_w + bl_w$	ALAP$_w$
a	0	34	34	0	0	14	14	0
b	8	26	34	8	2	12	14	2
c	4	12	16	22	2	8	10	6
d	5	19	24	15	2	12	14	2
e	6	16	22	18	2	11	13	3
f	16	18	34	16	6	8	14	6
g	10	6	16	28	6	4	10	10
h	14	10	24	24	7	7	14	7
i	13	4	17	30	6	4	10	10
j	23	11	34	23	9	5	14	9
k	32	2	34	32	12	2	14	12

[a]For example, as in Figure 4.1.

Examples Table 4.1 displays the various node levels for all nodes of the sample task graph (e.g., in Figure 4.1). The value of node a's bottom level $bl(a)$ is the length of the cp, $len(cp) = 34$ (see Eq. (4.40)). All nodes n_i, whose sum of $tl(n_i)$ and $bl(n_i)$ is 34, are critical path nodes (Lemma 4.7). In comparison, there are more computation critical path nodes: that is, nodes whose sum of $tl_w(n_i) + bl_w(n_i)$ is the length of the computation critical path $len_w(cp_w) = 14$ (see also Section 4.4.1).

Computing Levels and Critical Path To compute node levels, the following recursive definition of the levels is convenient. For a task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ and $n_i \in \mathbf{V}$,

$$bl(n_i) = w(n_i) + \max_{n_j \in \mathbf{succ}(n_i)} \{c(e_{ij}) + bl(n_j)\}, \tag{4.43}$$

$$tl(n_i) = \max_{n_j \in \mathbf{pred}(n_i)} \{tl(n_j) + w(n_j) + c(e_{ji})\}. \tag{4.44}$$

It can easily be shown by contradiction that these definitions are equivalent to Definition 4.19. For the computation levels (bl_w and tl_w), $c(e_{ij}) = 0$ in the above equations, and for the allocated levels ($bl(n_i, \mathcal{A})$ and $tl(n_i, \mathcal{A})$), $c(e_{ij}) = 0$ if the communication e_{ij} is local; that is, $proc(n_i) = proc(n_j)$.

To determine the levels with the above equations, a correct order of the nodes must be warranted, so that all levels on the right side of the expressions are already defined when calculating a level. For the bottom level this order is inverse topological and for the top level it is topological. Algorithms for the calculation of the levels can then be formulated as in Algorithms 7 and 8.

The time complexity of both algorithms is $O(\mathbf{V} + \mathbf{E})$: calculating a topological order is $O(\mathbf{V} + \mathbf{E})$ (Algorithm 4). Furthermore, the level of each node ($O(\mathbf{V})$) is calculated by considering all of its predecessors or successors, which amortizes to $O(\mathbf{E})$ predecessors or successors for the entire graph. Hence, this part is also $O(\mathbf{V} + \mathbf{E})$.

Algorithm 7 Compute Bottom Levels of $G = (\mathbf{V}, \mathbf{E}, w, c)$

```
Insert $\{n \in \mathbf{V}\}$ in inverse topological order into sequential list $L$.
for each $n_i$ in $L$ do
    $max \leftarrow 0$; $n_{bl_{\text{succ}}}(n_i) \leftarrow NULL$
    for each $n_j \in \mathbf{succ}(n_i)$ do
        if $c(e_{ij}) + bl(n_j) > max$ then
            $max \leftarrow c(e_{ij}) + bl(n_j)$; $n_{bl_{\text{succ}}}(n_i) \leftarrow n_j$
        end if
        $bl(n_i) \leftarrow w(n_i) + max$
    end for
end for
```

Algorithm 8 Compute Top Levels of $G = (\mathbf{V}, \mathbf{E}, w, c)$

```
Insert $\{n \in \mathbf{V}\}$ in topological order into sequential list $L$.
for each $n_i$ in $L$ do
    $max \leftarrow 0$; $n_{tl_{\text{pred}}}(n_i) \leftarrow NULL$
    for each $n_j \in \mathbf{pred}(n_i)$ do
        if $tl(n_j) + w(n_j) + c(e_{ji}) > max$ then
            $max \leftarrow tl(n_j) + w(n_j) + c(e_{ji})$; $n_{tl_{\text{pred}}}(n_i) \leftarrow n_j$
        end if
        $tl(n_i) \leftarrow max$
    end for
end for
```

Observe that the top and bottom paths are also computed with the presented algorithms. For each node n, the algorithm stores the first successor (predecessor) as $n_{bl_{\text{succ}}}(n)$ ($n_{tl_{\text{pred}}}(n)$) that led to the maximum value. Upon termination of the algorithm, a bottom path of a node n, for instance, is given by the recursive list of successors: $p_{bl(n)} = \langle n, n_{bl_{\text{succ}}}(n), n_{bl_{\text{succ}}}(n_{bl_{\text{succ}}}(n)), \ldots \rangle$.

Moreover, a critical path and its length are also computed by Algorithm 7. It suffices to store a node with the highest bottom level during the run of Algorithm 7. From Eq. (4.40), it is known that this node $n_{cp,1}$ is the first node of a critical path and the path cp is given by the recursive list of $n_{cp,1}$'s successors, $cp = \langle n_{cp,1}, n_{bl_{\text{succ}}}(n_{cp,1}), n_{bl_{\text{succ}}}(n_{bl_{\text{succ}}}(n_{cp,1})), \ldots \rangle$.

As stated before, the paths of node levels and the critical path are in general not unique. For instance, the sample task graph (e.g., in Figure 4.1) possesses two different computation critical paths. To avoid ambiguity, additional criteria must be used to distinguish between the various paths. The two algorithms presented return only one bottom path and one top path for each node, since they store only the first successor (predecessor) with the maximum value they encounter. If the nodes are considered in the same order in each run, the algorithms always compute the same level paths for a given task graph.

4.4.3 Granularity

For a task graph, the notion of granularity (Section 2.4.1) usually describes the relation between the computation and the communication costs. Often, the granularity of a task graph is defined as the relation between the minimal node weight and the maximal edge weight.

Definition 4.20 (Task Graph Granularity) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph. G's granularity is*

$$g(G) = \frac{\min_{n \in \mathbf{V}} w(n)}{\max_{e \in \mathbf{E}} c(e)}. \tag{4.45}$$

With this definition, a task graph is said to be coarse grained if $g(G) \geq 1$. Coarse granularity is a desirable property of a task graph. One objective of task scheduling is always to minimize the cost of communication. This is achieved by having as much local communication (i.e., communication between tasks on the same processor) as possible. Unfortunately, this objective conflicts with the other objective of scheduling, namely, the distribution of the tasks among the processors. Graphs with coarse granularity can be parallelized (i.e., scheduled) more efficiently, since the inflicted cost of communication through parallelization is reasonable to small relative to the cost of computation.

It is therefore not surprising that certain theoretical results of task scheduling can be established for coarse grained task graphs. For example, bounds on the schedule length in relation to the optimal schedule length can be established for coarse grained graphs (Darte et al. [52], Gerasoulis and Yang [77], Hanen and Munier [86]). See also Theorem 4.6 and Theorem 5.3 of Section 5.3.2.

Some of the algorithms that will be presented later in this text essentially try to improve (i.e., increase) the granularity of a task graph, for example, clustering (Section 5.3) and grain packing (Section 5.3.5). Communication costs are eliminated by grouping nodes together and thereby making communication local, thus cost free.

Weak Granularity A local ratio between the computation and communication costs is the task or node grain (Gerasoulis and Yang [77]). This ratio measures the communication received and sent by a node in relation to the computation of its predecessors and successors, respectively.

Definition 4.21 (Grain) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph. The grain of node $n_i \in \mathbf{V}$ is*

$$grain(n_i) = \min \left\{ \frac{\min_{n_j \in \mathbf{pred}(n_i)} w(n_j)}{\max_{e_{ji} \in \mathbf{E}, n_j \in \mathbf{pred}(n_i)} c(e_{ji})}, \frac{\min_{n_j \in \mathbf{succ}(n_i)} w(n_j)}{\max_{e_{ij} \in \mathbf{E}, n_j \in \mathbf{succ}(n_i)} c(e_{ij})} \right\}. \tag{4.46}$$

If $\mathbf{pred}(n_i) \cup \mathbf{succ}(n_i) = \emptyset$*, that is, the node is independent, the grain is not defined.*

As an example for this definition consider Figure 4.7, where node n has

$$grain(n) = \min \left\{ \frac{\min\{2,3,4\}}{\max\{1,1,1\}}, \frac{\min\{5,6\}}{\max\{2,2\}} \right\} = \min \left\{ \frac{2}{1}, \frac{5}{2} \right\} = 2.$$

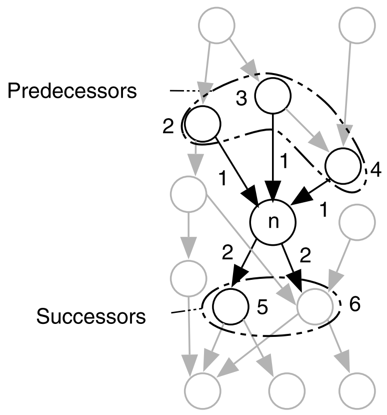


Figure 4.7. A task graph, where node n is featured as an example for Definition 4.21 of the grain; node n has $grain(n) = \min\{\min\{2,3,4\}/\max\{1,1,1\}, \min\{5,6\}/\max\{2,2\}\} = 2$.

So the grain is a local definition of granularity in the same way the node levels are local definitions of the critical path concept (Section 4.4.2). Of course, other definitions of the node grain are possible; for example, the incoming and outgoing communications could be set into relation with the weight $w(n)$ of the node n itself. However, the above definition of the grain leads to the following, weaker definition of global granularity, which is defined as the minimum of all node grains (Gerasoulis and Yang [77]).

Definition 4.22 (Task Graph Weak Granularity) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph. G's weak granularity is*

$$g_{\text{weak}}(G) = \min_{n \in \mathbf{V}, \mathbf{pred}(n) \cup \mathbf{succ}(n) \neq \emptyset} grain(n). \tag{4.47}$$

This definition of granularity is called weak granularity because

$$g(G) \leq g_{\text{weak}}(G), \tag{4.48}$$

as can be seen directly from the definitions. Hence, the desirable property of coarse granularity is easier to achieve with this latter definition. For some theoretical results of task scheduling, coarse granularity in this weak sense is enough (Gerasoulis and Yang [77]). This is the main motivation for defining the node grain as done in Definition 4.21.

While it might seem that the two definitions of granularity do not differ much, their values might be hugely different in practice. To comprehend this, consider the task graph G given in Figure 4.8. Its granularity is

$$g(G) = \frac{\min\{1, 100\}}{\max\{1, 100\}} = \frac{1}{100},$$

whereas its weak granularity is $g_{\text{weak}}(G) = \min\{grain(a), grain(b), grain(c)\} = \min\{1, 1, 1\} = 1$. Hence, the granularities differ by a factor of 100 in this example!

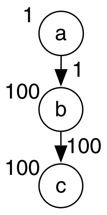


Figure 4.8. Example task graph G to illustrate difference between $g(G)$ and $g_{\text{weak}}(G)$.

Note that the task graph of Figure 4.8 is not unrealistic: in real programs there is often (but not always) some form of relation between the computation time and the communicated data. This is the case in the example graph: node a has a short computation time $w(a) = 1$ and consequently a short outgoing communication time $c(e_{ab}) = 1$ and the same principle holds for nodes b and c. In conclusion, the concept of weak granularity seems to reflect this quite well.

Granularity and Critical Paths Using the granularity, a relation between the critical path cp and the computation critical path cp_w of a task graph G can be established.

First, the ratio between the weight of an edge and weight of its origin node is studied. Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph. For any edge $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, it follows directly from Definitions 4.21 and 4.22 that

$$g_{\text{weak}}(G) \leq grain(n_j) \leq \frac{w(n_i)}{c(e_{ij})}. \tag{4.49}$$

With this consideration, the following theorem can be established.

Theorem 4.6 (Relation Between Critical Path and Computation Critical Path) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph, cp its critical path, and cp_w its computation critical path. The nodes of cp are denoted by $\mathbf{V}_{cp} \subseteq \mathbf{V}$, where $n_{\text{last}} \in \mathbf{V}_{cp}$ is the last node of cp, and its edges by $\mathbf{E}_c \subseteq \mathbf{E}$. It holds that*

$$len(cp) \leq \left(1 + \frac{1}{g_{\text{weak}}(G)}\right) len_w(cp_w). \tag{4.50}$$

Proof. Inequality (4.50) follows from Definition 4.17 of the path length, by substituting the edge weights using Eq. (4.49).

$$\begin{aligned} len(cp) &= \sum_{n \in \mathbf{V}_{cp}} w(n) + \sum_{e_{ij} \in \mathbf{E}_{cp}} c(e_{ij}) \\ &\leq \sum_{n \in \mathbf{V}_{cp}} w(n) + \sum_{e_{ij} \in \mathbf{E}_{cp}} \frac{w(n_i)}{g_{\text{weak}}(G)} \quad \text{with Eq. (4.49)} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{n\in\mathbf{V}_{cp}} w(n) + \sum_{n\in\mathbf{V}_{cp}-n_{\text{last}}} \frac{w(n)}{g_{\text{weak}}(G)} \\
&\leq \sum_{n\in\mathbf{V}_{cp}} w(n) + \frac{1}{g_{\text{weak}}(G)} \sum_{n\in\mathbf{V}_{cp}} w(n) \quad \text{adding } \frac{w(n_{\text{last}})}{g_{\text{weak}}(G)} \qquad (4.51) \\
&= \left(1 + \frac{1}{g_{\text{weak}}(G)}\right) \sum_{n\in\mathbf{V}_{cp}} w(n) \\
&= \left(1 + \frac{1}{g_{\text{weak}}(G)}\right) len_w(cp) \\
&\leq \left(1 + \frac{1}{g_{\text{weak}}(G)}\right) len_w(cp_w) \qquad \text{with Eq. (4.25).} \qquad \square
\end{aligned}
$$

A corresponding inequality is valid for any path p in task graph G:

$$len(p) \leq \left(1 + \frac{1}{g_{\text{weak}}(G)}\right) len_w(p), \qquad (4.52)$$

which can be shown with the above proof, leaving out the very last step of Eq. (4.51). Obviously, Theorem 4.6 is also valid when the weak granularity is substituted by the granularity $g(G)$, but using weak granularity results in a tighter inequality, as should be clear from the example of Figure 4.8. Actually, an even weaker granularity could be defined, where the node grain reflects only the relation between the node weight and its *outgoing* edge weights. The foregoing proof would also hold for such a definition of granularity. For the sake of comparability with other results, this is not done here.

Theorem 4.6 states that the larger the weak granularity, the smaller is the possible difference between the critical path and the computation critical path:

$$g_{\text{weak}}(G) \to \infty \Rightarrow len(cp) \to len_w(cp_w). \qquad (4.53)$$

The theorem is valuable for establishing bounds on the length of certain schedules (see Theorem 5.3 of Section 5.3.2).

Communication to Computation Ratio The measure of granularity considers extreme values (minimums and maximums) and consequently guarantees certain properties of a task graph. For example, selecting arbitrarily one node $n \in \mathbf{V}$ and one edge $e \in \mathbf{E}$ of a task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ with granularity $g(G)$, it always holds that

$$\frac{w(n)}{c(e)} \geq g(G). \qquad (4.54)$$

However, the general scheduling behavior of a task graph is not necessarily related to the granularity of the graph. After all, two almost identical graphs can have very different granularities due to the fact that one node or edge weight differs significantly between them. Average or total measures are usually better suited to reflect the scheduling behavior. An often employed measure, especially in experimental comparisons, is the ratio of the total costs.

Definition 4.23 (Communication to Computation Ratio (CCR)) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph. G's communication to computation ratio is*

$$CCR(G) = \frac{\sum_{e \in \mathbf{E}} c(e)}{\sum_{n \in \mathbf{V}} w(n)}. \tag{4.55}$$

Usually, a task graph is said to have high, medium, and low communication for CCRs of about 10, 1, and 0.1, respectively. With the help of the CCR, one can judge the importance of communication in a task graph, which strongly determines the scheduling behavior. In many experiments, the CCR is used to characterize the task graphs of the workload (Khan et al. [103], Macey and Zomaya [132], McCreary et al. [136], Sinnen and Sousa [179]). In the literature the CCR is sometimes the ratio of the average edge to the average node weight (Kwok and Ahmad [111]). This definition is insensible to the number of edges and thus does not reflect the total communication volume.

4.5 CONCLUDING REMARKS

This chapter was devoted to the formal definition of the task scheduling problem and its theoretical analysis. The chapter's aim is to provide the reader with a consistent framework of task scheduling. Using this framework, the following chapters will study heuristics and techniques of task scheduling.

Focus has been placed on the general scheduling problem, namely, the one including communication costs, without any restriction on the task graph and for a target parallel system with a finite number of processors. Later in Section 6.4 it will be seen that there are many variations of this problem, which often result from some form of restriction. Apart from these "special cases," there are many other scheduling problems, for instance, scheduling with deadlines or with preemption. They sometimes differ quite strongly from the ones discussed in this text and are often not related to parallel systems. The reader interested in such scheduling problems should refer to Brucker [27], Chrétienne et al. [34], Leung [121], Pinedo [151], and Veltman et al. [196].

4.6 EXERCISES

4.1 Figure 4.9(*a*) depicts a small task graph *G*, according to Definition 3.13. It is scheduled on a target system as defined in Definition 4.3, consisting of three processors. Three different schedules are shown in Figures 4.9(*b*) to 4.9(*d*).

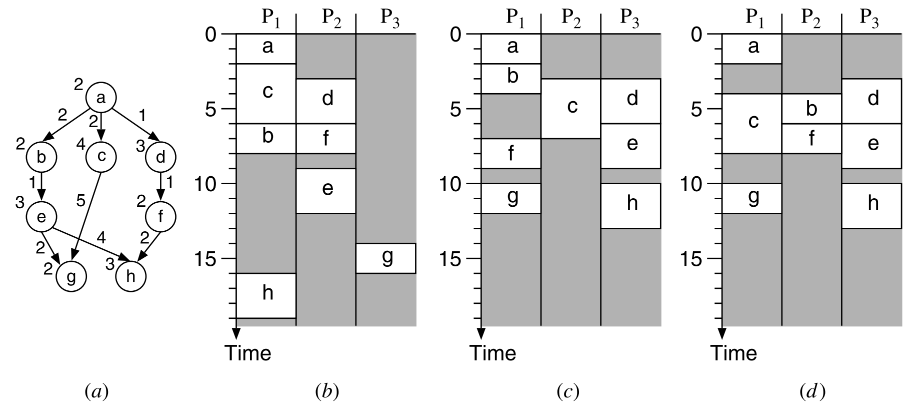


Figure 4.9. Task graph (*a*) and three different, possibly infeasible, schedules on three processors (*b*)–(*d*).

(a) For each schedule check if it is feasible.

(b) In case a schedule is infeasible, make it feasible with as little modifications as possible. What is the resulting schedule length?

(c) Create a feasible schedule for task graph G that is of equal length or shorter than the ones of Figure 4.9.

4.2 In Exercise 4.1 you are asked to evaluate the feasibility of the schedules depicted in Figure 4.9. The feasibility is to be checked for the general scheduling model that considers communication costs. Consider now scheduling without communication costs (Section 4.3), which is based on the target system model formulated in Definition 4.14.

(a) For each schedule of Figure 4.9 check if it is feasible when communication costs are ignored.

(b) In case a schedule is infeasible, make it feasible. Furthermore, try to reduce the schedule length for each schedule without changing either processor allocation or the node order on each processor. What is the resulting schedule length?

4.3 The proof of Theorem 4.1 demonstrated NP-completeness of the scheduling problem SCHED with a reduction from 3-PARTITION. A less involved proof, which shows NP-completeness in the weak sense, can be based on a reduction from the well known NP-complete problem PARTITION (Garey and Johnson [73]):

Given n positive integer numbers $\{a_1, a_2, \ldots, a_n\}$, is there a subset $\mathbf{I}$ of indices such that $\sum_{i\in\mathbf{I}} a_i = \sum_{i\notin\mathbf{I}} a_i$?

Devise such a proof. (*Hint*: Use a fork-join graph.) Why does this proof also show NP-completeness for an unlimited number of processors?

4.4 Consider again the task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ of Figure 4.9(a). Determine for each node $n \in \mathbf{V}$:

(a) Top level $tl(n)$ and computation top level $tl_w(n)$.

(b) Bottom level $bl(n)$ and computation bottom level $bl_w(n)$.

What is the graph's critical path and what is its computation critical path?

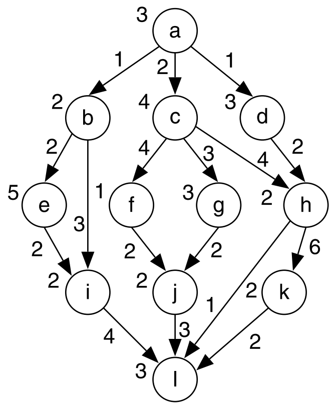
4.5 In Exercise 4.4, you are asked to calculate the node levels for the task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ of Figure 4.9(a). When the nodes are allocated to processors, these levels can change because local communication has zero costs. Figures 4.9(b) to 4.9(d) show three schedules for the task graph of Figure 4.9(a). Using the processor allocation given by these three schedules, calculate the following for each schedule $\mathcal{S}$:

(a) The allocated top level $tl(n, \mathcal{S})$ for each $n \in \mathbf{V}$.

(b) The allocated bottom level $bl(n, \mathcal{S})$ for each $n \in \mathbf{V}$.

What is the graph's allocated critical path for each schedule?

4.6 Given is this task graph:



Order its nodes $n \in \mathbf{V}$ in the following orders, while adhering to the precedence constraints:

(a) In descending order of the node weights $w(n)$.

(b) In descending bottom level $bl(n)$ order.

(c) In ascending top level $tl(n)$ order.

(d) In descending top level $tl(n)$ order.

(e) In descending order of $tl(n) + bl(n)$.

4.7 For the task graphs of Exercises 4.1 and 4.6, determine the following:

(a) Granularity $g(G)$.

(b) Weak granularity $g_{\text{weak}}(G)$.

(c) Communication to computation ratio $CCR(G)$.