CHAPTER 6

Advanced Task Scheduling

The previous chapter introduced the two fundamental scheduling algorithms: list scheduling and clustering. In this chapter, the focus is on more advanced aspects of task scheduling.

The first sections are dedicated to general techniques, which are not scheduling algorithms as such but are techniques that are used in combination with the fundamental algorithms. Following the framework approach utilized so far, they are studied detached from concrete algorithms in a general manner. The insertion technique (Section 6.1) is a generalization of the end technique (Definition 5.2), where nodes no longer need to be scheduled after already scheduled nodes. With node duplication (Section 6.2), one node might be executed on more than one processor. The goal is to save communication costs, as node duplication can make more communications local.

The classic target system model (Definition 4.3) used so far assumes that all processors are identical regarding their processing speed. In real parallel systems, this is not always given. Section 6.3 extends the scheduling model toward heterogeneous processors. This is done in a simple but powerful way, which leaves almost all scheduling concepts studied so far unaffected.

Section 6.4 returns to the more theoretical side of task scheduling. It studies the NP-hardness of many variations of the scheduling problem introduced in Chapter 4.

A completely different approach to scheduling, based on genetic algorithms, is analyzed in Section 6.5. A genetic algorithm is a stochastic search based on the principles of evolution.

The concluding remarks of this chapter will provide some references to further task scheduling publications.

6.1 INSERTION TECHNIQUE

In Section 5.1, the most common technique for the node's start time determination—the end technique (Definition 5.2)—was presented. On a given processor P, a node is scheduled at the end of all nodes already scheduled on this processor.

Task Scheduling for Parallel Systems, by Oliver Sinnen

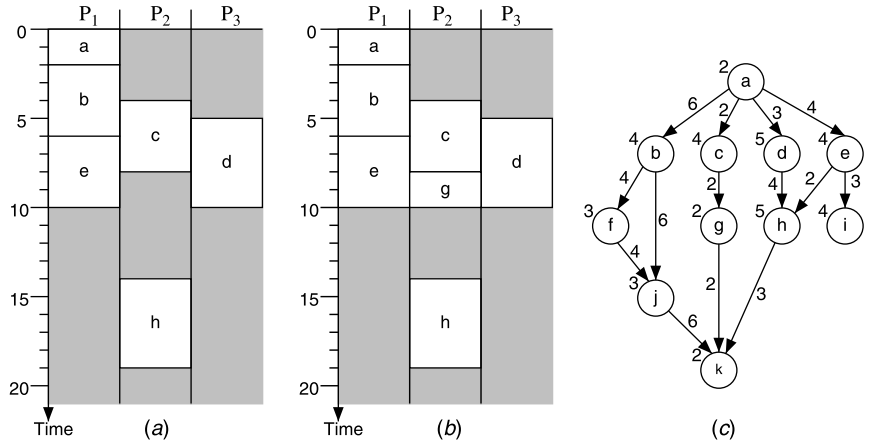


Figure 6.1. Partial schedule of sample task graph before the scheduling of g (a); partial schedule after node g is inserted in the idle period between c and h on P_2 (b); the sample task graph (c).

Under certain circumstances, a node can be scheduled earlier, if there is a time period between two nodes already scheduled on P, where P runs idle. This becomes clear with the following example. Consider the partial schedule of the sample task graph displayed in the Gantt chart of Figure 6.1(a): nodes a, b, c, d, e, h have already been scheduled and the next node is g. With the end technique g's start time on each processor is (see Eq. (5.1))

$$t_s(g, P_1) = \max\{t_{\text{dr}}(g, P_1), t_f(P_1)\} = \max\{10, 10\} = 10,$$
$$t_s(g, P_2) = \max\{8, 19\} = 19 \text{ or } t_s(g, P_3) = \max\{10, 10\} = 10.$$

Unfortunately, g's earliest DRT is on P_2, $t_{\text{dr}}(g, P_2) = 8$, where the processor only finishes at time unit 19. However, there is a large period between c and h during which P_2 runs idle, since h has to wait for its communications from nodes e and d. Node g fits easily into this period and can start immediately after c, because the communication e_{cg} is local. A corresponding partial schedule, where g is inserted between c and h, is shown in the Gantt chart of Figure 6.1(b). Of course, g cannot be placed into the space before node c on P_2 or d on P_3 since it depends on c.

It is intuitive that this insertion approach to scheduling has the potential of reducing the schedule length, as nodes might be scheduled earlier and time during which a processor runs idle might be eliminated. Experimental evidence for this intuition is given by Kwok and Ahmad [111] and by Sinnen and Sousa [177].

Clearly, Conditions 4.1 and 4.2 for a feasible schedule also apply when inserting a free node between other already scheduled nodes. This is independent of the scheduling algorithm, as long as the insertion is supposed to preserve the feasibility of

the partial schedule. The subsequently defined condition unites both Conditions 4.1 and 4.2 for a feasible schedule and it also includes the end technique case. Therefore, it is simply called *scheduling condition*.

Condition 6.1 (Scheduling Condition) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph, $\mathbf{P}$ a parallel system, and $\mathcal{S}_{\text{cur}}$ a partial feasible schedule for a subset of nodes $\mathbf{V}_{\text{cur}} \subset \mathbf{V}$ on $\mathbf{P}$. Let $n_{P,1}, n_{P,2}, \ldots, n_{P,l}$ be the nodes scheduled on processor $P \in \mathbf{P}$, that is, $\{n_{P,1}, n_{P,2}, \ldots, n_{P,l}\} = \{n \in \mathbf{V}_{cur} : proc(n) = P\}$, with $t_s(n_{P,i}) < t_s(n_{P,i+1})$ for $i = 1, 2, \ldots, l-1$.*

Two fictive nodes $n_{P,0}$ and $n_{P,l+1}$ are defined for which $t_f(n_{P,0}) = 0$ and $t_s(n_{P,l+1}) = \infty$. A free node $n \in \mathbf{V}$ can be scheduled on P between the nodes $n_{P,i}$ and $n_{P,i+1}$, $0 \leq i \leq l$, if

$$\max\{t_f(n_{P,i}), t_{\text{dr}}(n, P)\} + w(n) \leq t_s(n_{P,i+1}). \tag{6.1}$$

Inequality (6.1) warrants that the time period between two consecutive nodes is large enough so that node n can start its execution after its DRT and the finish time of the first node (term $\max\{t_f(n_{P,i}), t_{\text{dr}}(n, P)\}$), and complete (term $+w(n)$) before the second node starts execution (term $\leq t_s(n_{P,i+1})$). Clearly, Condition 6.1 is also valid for scheduling without communication costs, due to its generic formulation using the DRT. Figure 6.2 illustrates the scheduling condition as it shows the scheduling of a node n between the two consecutive nodes $n_{P,i}$ and $n_{P,i+1}$ on processor P.

The end technique is included in this condition, as the start time of the last, fictive node $n_{P,l+1}$ is $t_s(n_{P,l+1}) = \infty$; in other words, n always fits into the time period starting with the finish time $t_f(P) = t_f(n_{P,l})$ of processor P.

Analogous to the end technique, the start time of a node is set to the earliest time in the first slot that complies with scheduling Condition 6.1.

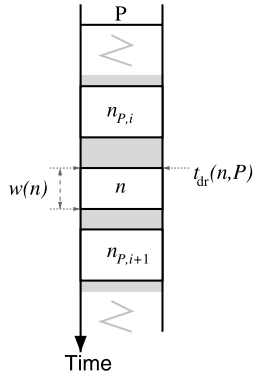


Figure 6.2. Scheduling condition: node n is scheduled in the gap between two consecutive nodes on P.

Definition 6.1 (Insertion Technique) *Let k be the smallest value of i,* $0 \leq i \leq l$*, for which Eq. (6.1) is true. The start time of n on P is then defined as*

$$t_s(n, P) = \max\{t_f(n_{P,k}), t_{\text{dr}}(n, P)\}. \tag{6.2}$$

Again, this definition also applies to scheduling without communication costs.

6.1.1 List Scheduling with Node Insertion

Even though employing the insertion technique in list scheduling with start time minimization has the potential to reduce the schedule length in comparison to the end technique, there is in general no guarantee of improvement. This becomes clear when one realizes that the insertion technique might lead to other processor allocations than the end technique. For instance, with the insertion technique node g is scheduled on P_2 in the example of Figure 6.1; with the end technique it would be allocated to either P_1 or P_2. As a consequence, communications might be remote, when they would be local with the end technique, resulting in a longer schedule. However, with the same node order and processor allocation the insertion technique is at least as good as the end technique. This is formulated in the next theorem, which is an extension of Theorem 5.1 of Section 5.1 with an almost identical proof.

Theorem 6.1 (Insertion Better than End Technique) *Let* $\mathcal{S}$ *be a feasible schedule for task graph* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *on system* $\mathbf{P}$*. Using simple list scheduling (Algorithm 9), with the nodes scheduled in nondecreasing order of their start times in* $\mathcal{S}$ *and allocated to the same processors as in* $\mathcal{S}$*, two schedules are created:* $\mathcal{S}_{\text{end}}$*, employing the end technique (Definition 5.2) and* $\mathcal{S}_{\text{insert}}$ *employing the insertion technique (Definition 6.1). Then,*

$$sl(\mathcal{S}_{\text{insert}}) \leq sl(\mathcal{S}_{\text{end}}) \leq sl(\mathcal{S}). \tag{6.3}$$

Proof. First, $\mathcal{S}_{\text{end}}$ and $\mathcal{S}_{\text{insert}}$ are feasible schedules, since the node order is a topological order according to Lemma 4.1. Following the same argumentation as in the proof of Theorem 5.1, it suffices to shown that $t_{f,\mathcal{S}_{\text{insert}}}(n) \leq t_{f,\mathcal{S}_{\text{end}}}(n) \leq t_{f,\mathcal{S}}(n)$ $\forall\, n \in \mathbf{V}$.

Since the processor allocations are identical for all schedules, communications that are remote in one schedule are also remote in the other schedules and likewise for the local communications. Thus, the DRT $t_{\text{dr}}(n)$ of a node n can only differ between the schedules through different finish times of the predecessors, that is, through different start times, as the execution time is identical in all schedules.

By induction, it is shown now that $t_{s,\mathcal{S}_{\text{insert}}}(n) \leq t_{s,\mathcal{S}_{\text{end}}}(n) \leq t_{s,\mathcal{S}}(n)\, \forall\, n \in \mathbf{V}$. Evidently, this is true for the first node to be scheduled, as it starts in all schedules at time unit 0. Now, let $\mathcal{S}_{\text{insert,cur}}$ and $\mathcal{S}_{\text{end,cur}}$ be two partial schedules of the same nodes of $\mathbf{V}_{\text{cur}}$ of G, for which $t_{s,\mathcal{S}_{\text{insert,cur}}}(n) \leq t_{s,\mathcal{S}_{\text{end,cur}}}(n) \leq t_{s,\mathcal{S}}(n)\, \forall\, n \in \mathbf{V}_{\text{cur}}$ is true. For

the node $n_i \in \mathbf{V}, \notin \mathbf{V}_{\text{cur}}$ to be scheduled next on processor $proc_{\mathcal{S}}(n_i) = P$, it holds that $t_{\text{dr},\mathcal{S}_{\text{insert,cur}}}(n_i, P) \leq t_{\text{dr},\mathcal{S}_{\text{end,cur}}}(n_i, P) \leq t_{\text{dr},\mathcal{S}}(n_i, P)$, with the above argumentation. With the end technique, n_i is scheduled at the end of the last node already scheduled on P. But this cannot be later than in $\mathcal{S}$, because the nodes on P in $\mathcal{S}_{\text{end,cur}}$ are the same nodes that are executed before n_i on P in $\mathcal{S}$, due to the schedule order of the nodes, and, according to the assumption, no node starts later than in $\mathcal{S}$. With the insertion technique, n_i is scheduled in the worst case (i.e., there is no idle period between two nodes already scheduled on P complying with Eq. (6.1)) also at the end of the last node already scheduled on P. Thus, $t_{s,\mathcal{S}_{\text{insert,cur}}}(n_i, P) \leq t_{s,\mathcal{S}_{\text{end,cur}}}(n_i, P) \leq t_{s,\mathcal{S}}(n_i, P)$ for node n_i. By induction this is true for all nodes of the schedule, in particular, the last node, which proves the theorem. □

Theorem 5.1 of Section 5.1 establishes that an optimal schedule is defined by the processor allocation and the nodes' execution order. List scheduling with the end technique can construct an optimal schedule from these inputs.

While this result is included in Theorem 6.1, it also establishes that the insertion technique might improve a given nonoptimal schedule. *Rescheduling* a given schedule with list scheduling and the insertion technique, using the processor allocation and the node order of the original schedule, might improve the schedule length. In particular, schedules produced with the end technique might be improved. What first sounds like a contradiction to Theorem 5.1, after all it states that the end technique is optimal given a processor allocation and the nodes' execution order, becomes clear when one realizes that rescheduling with the insertion technique can only reduce the length of a schedule by reordering the nodes. Inserting a node in an earlier slot changes the node order on the corresponding processor.

Complexity Regarding the complexity of list scheduling with the insertion technique, the second part of the corresponding Algorithm 9 is examined. Determining the data ready time remains $O(\mathbf{PE})$ for all nodes on all processors (see Section 5.1). What changes is the time complexity of the start time calculation. In the worst case, it must be checked for every consecutive node pair on a processor, if the time period between them is large enough to accommodate the node to be scheduled. At the time a node is scheduled, there are at most $O(\mathbf{V})$ nodes scheduled on all processors. So if the start time is determined for every processor—as in the typical case of start time minimization—this amortizes to $O(\max(\mathbf{V}, \mathbf{P}))$, which is of course $O(\mathbf{V})$, because it is meaningless to schedule on more processors than the task graph has nodes. The start time is calculated for every node; thus, the final complexity of the second part of simple list scheduling with start time minimization is $O(\mathbf{V}^2 + \mathbf{PE})$. For comparison, with the end technique it is $O(\mathbf{P}(\mathbf{V} + \mathbf{E}))$ (see Section 5.1).

The insertion technique is employed in various scheduling heuristics, for example, ISH (insertion scheduling heuristic) by Kruatrachue [105], (E)CPFD ((economical) critical path fast duplication) by Ahmad and Kwok [4], BSA (bubble scheduling and allocation) by Kwok and Ahmad [114], and MD (mobility directed) by Wu and Gajski [207].

6.2 NODE DUPLICATION

A major challenge in scheduling is to avoid interprocessor communications. A node often has to wait for its entering communications, while its executing processor runs idle. For instance, in the schedule for the sample task graph visualized by Figure 6.1(a), nodes c and d are delayed due to the communication from node a and both processors P_2 and P_3 run idle during that time. One is tempted to move node a, for example, to P_2 in order to eliminate the cost of communication e_{ac}. However, this move creates communication costs between nodes a and b, as illustrated by the partial schedule in Figure 6.3(a).

A solution that has been exploited to reduce communication costs, while avoiding the above described problem, is node duplication. In this approach, some nodes of a task graph are allocated to more than one processor of the target system. For the previous example, node a is scheduled three times, once on each processor as shown in Figure 6.3(b). The communications from node a are now local on each processor of the target system and nodes b, c, d can start immediately after a finishes. Since nodes c and d start earlier, subsequent nodes can also start earlier, potentially leading to a shorter schedule length. Figure 6.3(c) displays a schedule similar to that of Figure 4.1, yet with the duplication of nodes a and j. The schedule length is reduced from 24 to 22 time units. But the potential of duplication is even higher, as demonstrated by the schedule in Figure 6.3(d), where the duplication of a can reduce the schedule length to 18 time units.

For node duplication, some formal changes must be carried out for the definitions and conditions of Section 4.1. The definition of a schedule (Definition 4.2) changes as follows.

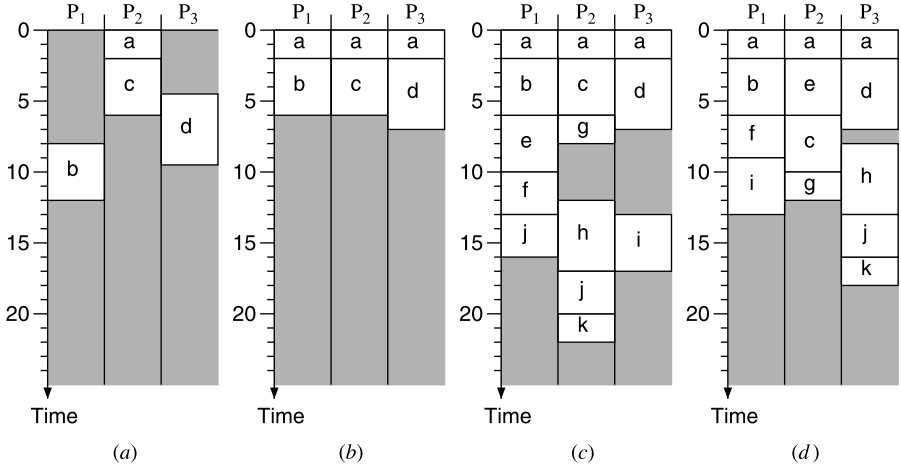


Figure 6.3. Partial (a), (b) and full schedules (c), (d) of the sample task graph (e.g., in Figure 6.1); the schedules of (b), (c), and (d) use node duplication.

- The function $proc(n)$ for the processor allocation of node n becomes a subset of P, denoted by $\mathbf{proc}(n)$, because with duplication a node can be allocated to more than one processor. Of course, $\mathbf{proc}(n) \subseteq \mathbf{P}$ and $|\mathbf{proc}(n)| \geq 1$.
- The function $t_s(n)$ for the start time assignment to node n is ambiguous with node duplication. Only with specification of a processor is the start time of a node defined. Thus, it is now obligatory to write $t_s(n, P)$ (see Eq. (4.1)) to denote the start time of n on P; t_s becomes a function of the nodes *and* the processors, $t_s : \mathbf{V} \times \mathbf{P} \rightarrow \mathbb{Q}_0^+$.

Node duplication also has an impact on Condition 4.2 regarding the precedence constraints of the task graph.

Condition 6.2 (Precedence Constraint—Node Duplication) *For a schedule $\mathcal{S}_{\text{dup}}$ with node duplication, Eq. (4.5) of Condition 4.2 becomes*

$$t_s(n_j, P) \geq \min_{P_x \in \mathbf{proc}(n_i)} \{t_f(e_{ij}, P_x, P)\}. \tag{6.4}$$

Given the communication e_{ij}, node n_j cannot start until *at least one* instance of the duplicated nodes of n_i has provided the communication e_{ij}. In the schedule of Figure 6.3(c), for example, the communication e_{jk} is received from the duplicated node j on processor P_2; the same communication from the instance of node j on P_1 does not arrive on time on P_2 for the start of k at $t_s(k, P_2) = 20$ ($t_f(e_{jk}, P_1, P_2) = t_f(j, P_1) + c(e_{jk}) = 16 + 6 = 22$).

Following from this altered precedence constraint condition, the definition of the data ready time (Definition 4.8) must be adapted.

Definition 6.2 (Data Ready Time (DRT)—Node Duplication) *For a schedule $\mathcal{S}_{\text{dup}}$ with node duplication, Eq. (4.6) of Definition 4.8 becomes*

$$t_{\text{dr}}(n_j, P) = \max_{n_i \in \mathbf{pred}(n_j)} \{ \min_{P_x \in \mathbf{proc}(n_i)} \{t_f(e_{ij}, P_x, P)\}\}. \tag{6.5}$$

The rest of the definitions and conditions of Section 4.1 remain unmodified, except for the necessary formal adaptation for the altered definition of a schedule (see above). Even though all forgoing conditions and definitions were again formulated in a generic form and could be applied to scheduling without communication costs, it makes no sense to use node duplication when communication has no costs.

Node Order Anomaly A consequence of the modified precedence constraint Condition 6.2 is a possible anomaly in the order of nodes in a schedule. Consider the schedule of a small task graph in Figure 6.4, where node B is duplicated. Looking at the node order on P_1, one observes that node B starts after node C, even though C depends on B. This is valid according to Condition 6.2, because node B is a duplicated node and the communication e_{BC} is provided by the instance of B on P_2. Hence,

Lemma 4.1, stating that the start order of the nodes in a schedule is a precedence order, is in general not valid for schedules with duplicated nodes.

Complexity Node duplication is meaningful for those nodes that have more than one leaving edge, that is, for nodes with an out-degree larger than one:

$$\{n_i \in \mathbf{V} : |\{e_{ij} \in \mathbf{E} : n_j \in \mathbf{V}\}| > 1\}. \tag{6.6}$$

Nodes with an out-degree of one or less can also be duplicated, but their duplication alone cannot reduce communication costs. For example, in Figure 6.4, the duplication of node A cannot be beneficial, as only node C receives its data. Imagine node A were duplicated several times. Only that instance of A that provides its data earliest to C had to be retained; all other duplicated nodes of A could be removed.

However, when a node with out-degree larger than one is duplicated, it might be beneficial to recursively duplicate its ancestors, even if their out-degree is only one. To illustrate this, regard the task graph in Figure 6.5(a). Node B is the only node with an out-degree larger than one. Consequently, it is the primary candidate for duplication and Figure 6.5(b) depicts a schedule on three processors where B is duplicated on each of them. While this is beneficial for the start time of node E on P_2, node D could start earlier on P_3 (at $t_s(D) = 6$ instead of $t_s(D) = 8$) without the duplication of B. Yet, the situation changes if node A is also duplicated together with each instance of B as shown in Figure 6.5(c). The resulting schedule length is even optimal, as it is the length of the computation critical path $cp_w = 11$.

Nodes with an out-degree larger than one are the *primary candidates* for duplication. Furthermore, in each duplicated path of nodes (e.g., $p(A \rightarrow B)$ in Figure 6.5), at least one node must have an out-degree larger than one (i.e., B in $p(A \rightarrow B)$); otherwise the duplication of the path cannot be beneficial.

Papadimitriou and Yannakakis [142] showed that the decision problem of scheduling with node duplication continues to be an NP-complete problem, even with unit

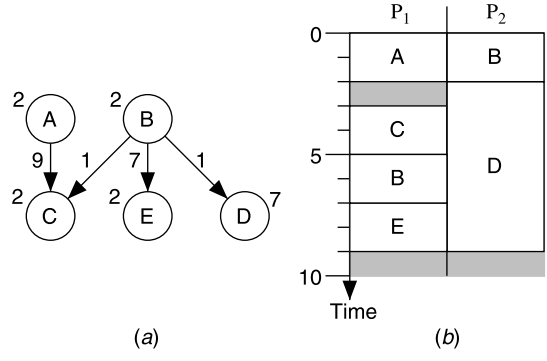


Figure 6.4. A small task graph (a) scheduled on two processors (b) with node duplication: node B is duplicated after node C, even though C depends on B.

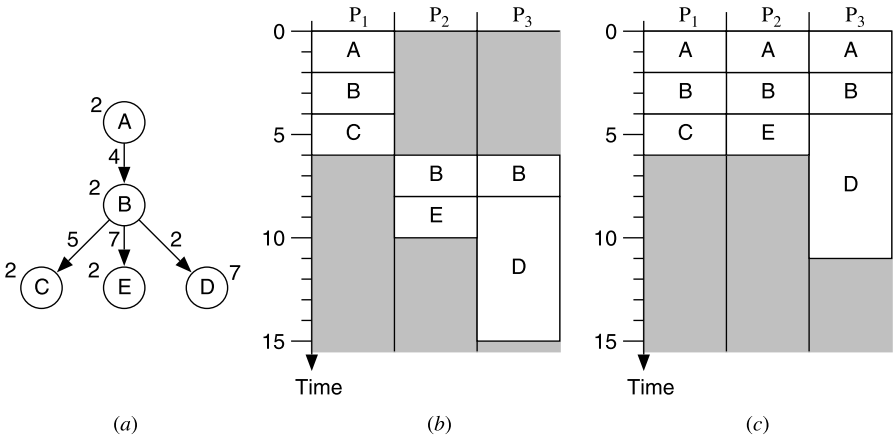


Figure 6.5. A small example task graph (*a*) and two schedules with node duplication; (*b*) only *B* is duplicated; (*c*) *A* and *B* are duplicated.

execution time (UET). Given the above observation, this is not surprising, because task graphs where all nodes have an out-degree of at most one (so called intrees) do not benefit from node duplication and their scheduling is NP-hard. Hence, since this "special case" is NP-hard, the general problem (which includes this case) cannot be easier. More on complexity results for scheduling with node duplication is given in Section 6.4.

Outtrees, on the other hand, strongly benefit from node duplication. In fact, scheduling an outtree on an unlimited number of processors using node duplication is a problem that can be optimally solved in polynomial time (Section 6.4). Hanen and Munier Kordon [86] address the scheduling of outtrees with duplication.

6.2.1 Node Duplication Heuristics

The aim in duplicating a node is to reduce the DRT of its descendant nodes, permitting them to start earlier. So node duplication introduces more computation load into the parallel system in order to decrease the cost of crucial communication. The challenge in scheduling with node duplication is the decision as to which nodes should be duplicated and where.

Node duplication has been employed in a large variety of scheduling approaches. The DSH (duplication scheduling heuristic) by Kruatrachue and Lewis [106] and (E)CPFD by Ahmad and Kwok [4], for instance, are two list scheduling variants with node duplication. Another list scheduling based algorithm with node duplication has been proposed by Hagras and Janeček [83], which is also suitable for heterogeneous processors. Sandnes and Megson [165] use node duplication in a genetic algorithm based heuristic. Clustering algorithms for an unlimited number of processors with node duplication are proposed by Papadimitriou and Yannakakis [142], Liou and Palis [125], and Palis et al. [140]. Another algorithm for unlimited processors has been

reported by Colin and Chrétienne [40]. Darbha and Agrawal [49–51] present a duplication heuristic that obtains optimal schedules for certain granularities. Manoharan [134] studies node duplication for work-greedy heuristics.

Apart from the general scheduling approach, these algorithms also differ over the strategy for duplicating nodes. Some algorithms consider all ancestors of a node for duplication in order to reduce the nodes' DRT, others only consider the predecessors. As nodes are typically duplicated in an existing partial schedule, the heuristics must employ the insertion technique discussed in Section 6.1.

Removal of Redundant Nodes During scheduling with node duplication, it might happen that some nodes are unnecessarily duplicated or that the original node becomes obsolete by a duplicated one. An example for the latter case can be observed in Figure 6.3(c) with node j. The communication e_{jk} provided by node j on P_1 is not used by k on P_2, because j is duplicated on P_2. So j can be removed from P_1, while the schedule remains valid without any effect on its length. For the schedule length to benefit from the idle periods left behind by the removed nodes, a second pass in a scheduling algorithm must be implemented (see Theorem 6.1).

6.3 HETEROGENEOUS PROCESSORS

Up to this point the target system of the scheduling problem always consisted of identical processors. While this is in most cases an accurate modeling of the real parallel system, there are systems that consist of heterogeneous processors. Take, for example, a cluster of PCs (Section 2.1) in which the processor differs from PC to PC. In order to consider the diverse capabilities of the processors in such a system, the scheduling model must be adapted.

Heterogeneity of processors can mean two things:

1. All processors of the system have the same functional capability, but they perform the same task at different speeds.
2. The processors have different functional capabilities; that is, certain tasks can only be performed by certain processors.

This section addresses processor heterogeneity of the first type. The second type of heterogeneity is encountered, for example, in embedded systems. Real time scheduling for embedded systems (Liu [127]) deals with such heterogeneity. The distinction between different services offered by the processors, however, introduces another dimension of conceptual complexity to the scheduling problem. Compromises have to be made for the scheduling algorithms. Introducing heterogeneity in terms of the first type does not change anything fundamental in the scheduling model, as will be seen in the following.

Note that heterogeneity in terms of processing speed also includes processor extensions for certain application domains, such as SSE or AltiVec (Section 2.1). Using such an extension just means that a processor can perform certain tasks faster than

others. A processor that does not possess such an extension can still perform the task, just slower. More on this later.

To include heterogeneous processors in task scheduling, the definition of the target parallel system (Definition 4.3) is modified accordingly, without any alteration of its properties.

Definition 6.3 (Target Parallel System—Heterogeneous Processors) *A target parallel system* $M_{\text{hetero}} = (\mathbf{P}, \omega)$ *consists of a set of processors* $\mathbf{P}$*, whose heterogeneity, in terms of processing speed, is described by the execution time function* ω*. The processors are connected by a communication network with the properties specified in Definition 4.3.*

Due to the processors' heterogeneity, the execution time of a task is no longer given by its mere node weight, but is a function of the processor.

Definition 6.4 (Execution Time) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $M_{\text{hetero}} = (\mathbf{P}, \omega)$ *a heterogeneous parallel system. The execution time of* $n \in \mathbf{V}$ *is the function* $\omega : \mathbf{V} \times \mathbf{P} \to \mathbb{Q}^+$.

Only the finish time of a node is directly affected by the node's execution time. With Definition 6.4, the finish time definition is modified—more precisely is generalized—for heterogeneous systems in the following way.

Definition 6.5 (Node Finish Time—Heterogeneous Processors) *Let* $\mathcal{S}$ *be a schedule for task graph* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *on a heterogeneous parallel system* $M_{\text{hetero}} = (\mathbf{P}, \omega)$*. The finish time of node n on processor* $P \in \mathbf{P}$ *is*

$$t_f(n, P) = t_s(n, P) + \omega(n, P). \tag{6.7}$$

It is important to note that Definition 6.5 is a generalization of the corresponding Definition 4.5 made in Section 4.1. If the execution time is set to be the node weight, $\omega(n, P) = w(n)$, Definition 4.5 is obtained.

Furthermore, the definition of the node weight (i.e., the computation cost) also needs an adjustment. Again, it is generalized from Definition 4.4 by representing now the average execution time.

Definition 6.6 (Computation Cost—Heterogeneous Processors) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $M_{\text{hetero}} = (\mathbf{P}, \omega)$ *a heterogeneous parallel system. The computation cost* $w(n)$ *of node n is the* average *time the task represented by n occupies a processor of* $\mathbf{P}$ *for its execution.*

Since the task graph maintains a node weight also for heterogeneous systems, metrics as node levels or the critical path (Section 4.4) can still be utilized for its analysis. However, since the node weight now reflects the average cost, the interpretation of these metrics is bound to the average case. In general, no safe conclusions can be

made on the worst or best case, for example, as done in Lemma 4.4 (CP bound on schedule length) and Lemma 4.6 (level bounds on start time).

A heterogeneous system is said to be *consistent*, or *uniform*, if the relation of the execution times on the different processors is independent of the task. For instance, if processor P executes a task twice as fast as processor Q, it does so for all tasks. In this case, a relative speed $s(P)$, with the average speed set to 1, can be assigned to every processor, so that the execution time of node n on processor P is

$$\omega(n, P) = \frac{w(n)}{s(P)}. \tag{6.8}$$

In an *inconsistent*, or *unrelated*, system the above conclusion is not valid. Processor P might be faster than processor Q with some tasks, but slower with others. In this case, the execution time is given by the corresponding entry in a $|\mathbf{V}| \times |\mathbf{P}|$ cost matrix W:

$$\omega(n_i, P_k) = w_{ik}. \tag{6.9}$$

Once more, the inconsistent model is a generalization including the consistent model. As mentioned before, inconsistency can arise when some of the processors are equipped with acceleration extensions, like SSE or AltiVec, for certain application types (e.g., multimedia). In that case, two processors P and Q, where only P is equipped with a special execution unit, might execute most tasks at the same speed, but some tasks that can benefit from the special unit, are executed faster by P.

In both types of system, consistent or inconsistent, the execution time ω of a node is a function of the node and the processor. For generality, it will be represented as such in the following discussions without any further specification. Of course, in practice, it is often more convenient to regard a heterogeneous system as consistent and to measure the processor speeds with some benchmark.

NP-Completeness Essentially, task scheduling for heterogeneous systems is a straightforward generalization of task scheduling for homogeneous processors, which relates to the execution time of the nodes. Therefore, it is not surprising that scheduling for heterogeneous systems is also an NP-complete problem.

Theorem 6.2 (NP-Completeness—Heterogeneous Processors) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $M_{\text{hetero}} = (\mathbf{P}, \omega)$ *a heterogeneous parallel system. The decision problem* H-SCHED (G, M_{hetero}) *associated with the scheduling problem is as follows. Is there a schedule* $\mathcal{S}$ *for* G *on* M_{hetero} *with length* $sl(\mathcal{S}) \leq T, T \in \mathbb{Q}^+$? H-SCHED ($G$, M_{hetero}) *is NP-complete.*

Proof. First, it is argued that H-SCHED (G, M_{hetero}) belongs to NP; then it is shown that H-SCHED (G, M_{hetero}) is NP-hard by reducing SCHED ($G_{\text{SCHED}}, \mathbf{P}_{\text{SCHED}}$) (Theorem 4.1) in polynomial time to H-SCHED (G, M_{hetero}).

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

For any instance of SCHED (G_{SCHED}, $\mathbf{P}_{\text{SCHED}}$) an instance of H-SCHED (G, M_{hetero}) is constructed by simply setting $G = G_{\text{SCHED}}$, $\mathbf{P} = \mathbf{P}_{\text{SCHED}}$, $\omega = w$, and $T = T_{\text{SCHED}}$; thus all processors of $\mathbf{P}$ are identical and the node execution time is only a function of the node n: $\omega(n, P) = w(n) \,\forall\, P \in \mathbf{P}$. Obviously, this construction is polynomial in the size of the instance of SCHED (G_{SCHED}, $\mathbf{P}_{\text{SCHED}}$). Furthermore, if and only if there is a schedule for the instance of H-SCHED (G, $\mathbf{P}$) that meets the bound T, is there a schedule for the instance SCHED (G_{SCHED}, $\mathbf{P}_{\text{SCHED}}$) meeting the bound T_{SCHED}. □

6.3.1 Scheduling

As the modification carried out for heterogeneous systems only affects the finish time calculation of a node, virtually all scheduling heuristics designed for homogeneous systems can be used for heterogeneous systems.

However, in order to produce efficient schedules, a scheduling heuristic should be aware of the heterogeneity of the processors. Yet again, this can often be achieved with a simple generalization. Instead of making decisions based on the start time of a node, the finish time naturally includes the processing capacity of the processor. For example, the possible start time of a node n might be earlier on processor P than on processor Q, but n might finish earlier on Q, due to Q's faster execution. So both the state of the current partial schedule and the heterogeneity of the processors are considered in decisions based on the finish time.

Finish Time Minimization With the above argumentation, the common start time minimization in list scheduling (Section 5.1.1) can be substituted by finish time minimization. The processor selected in each step (see Eq. (5.3)) is then

$$P_{\min} \in \mathbf{P} : t_f(n, P_{\min}) = \min_{P \in \mathbf{P}} \{\max\{t_{\text{dr}}(n, P), t_f(P)\} + \omega(n, P)\}. \tag{6.10}$$

With an accordingly modified procedure for the processor choice in Algorithm 10, a heterogeneous list scheduling heuristic is complete. The same node list construction that is used for homogeneous processors can be employed, because the node weights represent the average costs. Such a list scheduling strategy maintains the greedy characteristic and the same complexity. Moreover, by employing finish time minimization on homogeneous processors, one selects the same processor as one would with start time minimization; hence, it can be used in either case.

Heuristics for Heterogeneous Processors Some scheduling algorithms for heterogeneous processors have been proposed in the past. Heuristics based on the list scheduling approach are reported in Beaumont et al. [18], Liu et al. [126], Menascé et al. [137], Oh and Ha [139], Sinnen and Sousa [177], and Topcuoglu et al. [187, 188].

Also, the DLS algorithm (Sih and Lee [169]) is a dynamic list scheduling heuristic for heterogeneous processors. BSA by Kwok and Ahmad [114] is aware of heterogeneous processors, too. Finally, genetic scheduling algorithms are often designed for heterogeneous processors (e.g., Singh and Youssef [170], Wang et al. [198]) and Woo et al. [205]).

6.4 COMPLEXITY RESULTS

In Chapter 4 the scheduling problem, as stated in Definition 4.13, was defined in quite general terms. One "special case"—scheduling without communication costs—was also studied in Section 4.3. As was mentioned at the time, various other scheduling problems arise, many of which can be treated as special cases derived from the general scheduling problem. In the meantime, the concept of heterogeneous processors was introduced in the previous section. This section surveys and analyzes the NP-hardness of the scheduling problems that are obtained by relaxing, limiting, or generalizing one or more of the general problem's parameters, namely:

- *Processors.* As seen in Section 4.3.2, an unlimited number of processors can make scheduling polynomial-time solvable. Another interesting case to study is whether scheduling a task graph onto a fixed number of processors (in particular, two) is NP-hard.
- *Task Graph Structure.* Some task graph structures are easier to schedule than others. Analyzing the scheduling problem for restricted task graph structures might be valuable for associated real world problems. Moreover, it can give more insights into the general scheduling problem and help with the design of scheduling algorithms.
- *Costs.* Without communication costs, the scheduling problem seems to be less difficult, in particular, for an unlimited number of processors (Section 4.3.2). Other cases of interest relate to the granularity of the task graph or the unity of computation and/or communication costs.

As will be seen, most of these special problems remain NP-hard. The problems are studied in four groups: scheduling *without* communication costs, scheduling *with* communication costs, scheduling with node duplication, and scheduling on heterogeneous processors.

The virtually unlimited number of different scheduling problems makes a classification and designation scheme indispensable in order to maintain an overview and to reveal relations between problems.

6.4.1 $\alpha|\beta|\gamma$ Classification

This section introduces the alpha-beta-gamma ($\alpha|\beta|\gamma$) classification scheme of scheduling problems. It is a simple and extendible scheme that covers more scheduling problems than are discussed in the scope of this book. Originally it was presented

by Graham et al. [81] and later extended by Veltman et al. [196], in particular, for scheduling with communication delays. The scheme is typically found in surveys and comparisons of scheduling problems (e.g., Brucker [27], Chrétienne et al. [34], Leung [121]) each of which does its own necessary extension and adaptation.

Each of the three fields specifies one aspect of the scheduling problem. The α field specifies the processor environment, the β field the task characteristics, and the γ field the optimality criterion. The following definitions of the subfields of $\alpha|\beta|\gamma$ are based on the ones found in Brucker [27], Chrétienne et al. [34], Leung [121], and Veltman et al. [196]. Only those subfields are listed here that are relevant for this text and some adaptations and extensions are made, especially for scheduling with communication costs. Many more exist for other scheduling problems, for instance, scheduling with deadlines or with preemption. The interested reader should refer to the above mentioned publications.

Some of the designations used in the scheme are different from those used in this book. For comparability, the scheme's field designations, as found in the above mentioned literature, are utilized in the following. The accompanying explanations should make it clear what is meant in relation to the framework used in this text.

Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be the task graph to be scheduled and $\circ$ the symbol for an empty field.

α—Processor Environment The first field specifies the processor environment and it can have two subfields:

1. *Processor Type.*
   - P. The target system $\mathbf{P}$ consists of identical processors; that is, the execution time (Definition 6.4) of node $n \in \mathbf{V}$ is identical on all processors, $\omega(n, P) = w(n), \forall P \in \mathbf{P}$.
   - Q. The target system $M_{\text{hetero}} = (\mathbf{P}, \omega)$ consists of uniform heterogeneous processors; that is, the execution time of node $n \in \mathbf{V}$ is consistent with the processor speed $s(P)$, $\omega(n, P) = w(n)/s(P), \forall P \in \mathbf{P}$.
2. *Number of Processors.*
   - $\circ$. The number of processors is a variable and is specified as part of the problem instance.
   - m (*positive integer number*). Denotes a fixed number of m processors. Hence, the number of processors m is part of the problem type.
   - ∞. Denotes an unlimited number of processors. Since preemption is not allowed in any of the here discussed problems, this is equivalent to $|\mathbf{P}| \geq |\mathbf{V}|$; that is, there are at least as many processors as tasks.

β—Task Characteristics This field specifies the characteristics of the tasks and it can have various subfields, separated by a comma, all of which are optional. You might find different orders of these subfields in other texts, as the order is not consistent across the literature.

1. *Precedence Relations.* This subfield describes the precedence relations between tasks. In other words, it specifies the structure of the task graph G. In can take the following values:
   - $\circ$. The tasks are independent; that is, $\mathbf{E} = \emptyset$.
   - *prec*. There are precedence constraints between tasks; hence, $|\mathbf{E}| \geq 1$.
   - {*outtree*, *intree*, *tree*}. The task graph G is a rooted tree (*tree*). A rooted tree is a graph with either an in-degree of at most one for each node $n \in \mathbf{V}$ (*outtree*, e.g., Figure 6.6(a)) or an out-degree of at most one for each node $n \in \mathbf{V}$ (*intree*, e.g., Figure 6.6(b)). Note that this definition of tree is general in that a tree can have multiple root nodes, which is usually referred to as forest (Cormen et al. [42]).
   - *op-forest*. The task graph G is an opposing forest, consisting of intrees *and* outtrees.
   - {*fork*, *join*}. The task graph G is a *fork* or *join* graph. A fork (join) graph is an outtree (intree) with one source (sink) node n_{root}, the root node, where all other nodes are successors (predecessors) of n_{root}, for example, Figure 6.7.
   - *chains.* The task graph G is a tree, where all nodes have an out-degree *and* an in-degree of at most one (e.g., Figure 6.8). That is, the task graph consists of disjoint chains of nodes. Note that it is denoted by *chains* not *chain*, as most scheduling problems are trivial when there is only one chain of nodes.

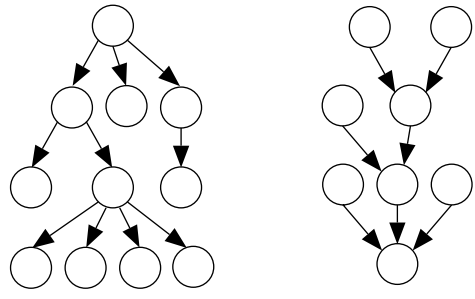


Figure 6.6. Examples of (a) outtree and (b) intree.

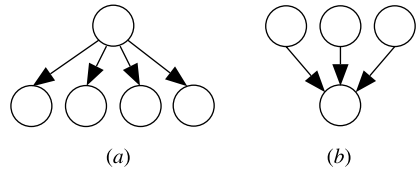


Figure 6.7. Examples of (a) fork graph and (b) join graph.

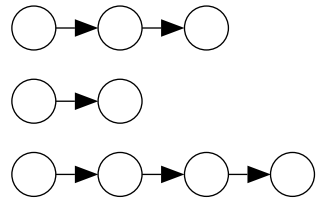


Figure 6.8. Example of chains graph.

- {*harpoon*, *in-harpoon*, *out-harpoon*}. The task graph G is a tree, with a harpoon-like structure. Such a graph is very similar to a fork or join graph. It consists of one root node n_{root} and x chains. In an *out-harpoon* there is an edge from n_{root} to each source node of the x chains, while in an *in-harpoon* there is an edge from each sink node of the x chains to n_{root}. Figure 6.9 illustrates an out-harpoon, where each chain consists of two nodes.
- *fork-join.* The task graph G is a fork-join graph. In a fork-join graph there is one source node n_s and one sink node n_t. All other nodes are successors of n_s and predecessors of n_t and are independent of each other, for example, in Figure 6.10.
- *sp-graph.* The task graph G is a series–parallel graph. A series–parallel graph can be constructed recursively using three basic graphs: (1) a graph consisting of a single node; (2) a single chain graph, that is, $\mathbf{V} = \{n_1, n_2, \ldots, n_l\}$ and $\mathbf{E} = \bigcup_{i=1}^{l-1} e_{i,i+1}$ (the "series" graph); and (3) a fork-join graph (the "parallel" graph), see above. Each of these graphs is a series–parallel graph in itself. Let $G_1 = (\mathbf{V}_1, \mathbf{E}_1)$ and $G_2 = (\mathbf{V}_2, \mathbf{E}_2)$ be two series–parallel graphs, with n_s and

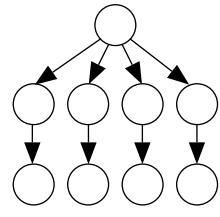


Figure 6.9. Example of out-harpoon graph.

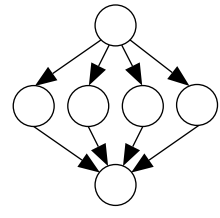


Figure 6.10. Example of fork-join graph.

n_t being the sink and source nodes of G_2, respectively. A new series–parallel graph G can be constructed from G_1 by substituting any node $n_i \in \mathbf{V}_1$ with the complete G_2. That is, n_i and all edges that are incident on n_i are removed from G_1. New edges are created from the predecessors of n_i, $\mathbf{pred}(n_i)$, to n_s and from n_t to the successors of n_i, $\mathbf{succ}(n_i)$. Note that any series–parallel graph has exactly one source node and one sink node. An example for a series–parallel graph is depicted in Figure 6.11.

- *bipartie.* The task graph G is a bipartie graph. That means $\mathbf{V}$ can be partitioned into two subsets $\mathbf{V}_1$ and $\mathbf{V}_2$ such that $e_{ij} \in \mathbf{E}$ implies $n_i \in \mathbf{V}_1$ and $n_j \in \mathbf{V}_2$. In other words, all edges go from the nodes of $\mathbf{V}_1$ to the nodes of $\mathbf{V}_2$ (e.g., Figure 6.12).
- *int-ordered.* The task graph G is interval-ordered. Let each node $n \in \mathbf{V}$ be mapped to an interval $[l(n), r(n)]$ on the real line. A task graph is said to be interval-ordered if and only if there exists a node-to-interval mapping with the following property for any two nodes $n_i, n_j \in \mathbf{V}$:

$$r(n_i) \leq l(n_j) \Longleftrightarrow n_j \in \mathbf{desc}(n_i) \tag{6.11}$$

This means that the intervals of any two nodes do not overlap if and only if one node is the descendant of the other (El-Rewini et al. [65], Kwok and Ahmad [113]).

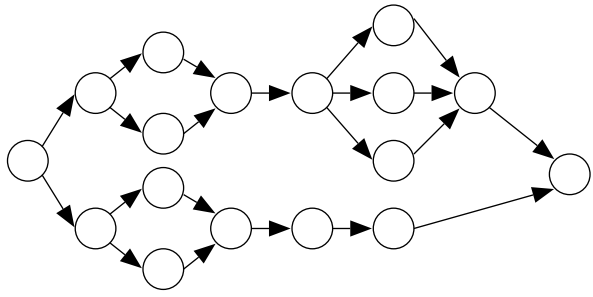


Figure 6.11. Example of series–parallel graph.

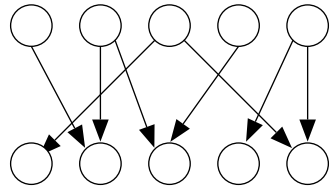


Figure 6.12. Example of bipartie graph.

2. *Computation Costs.* This subfield relates to the computation costs of the tasks.
   - ∘. Each node $n_i \in \mathbf{V}$ is associated with a computation cost $w(n_i)$.
   - p_i. If symbol p_i is present, the computation costs of the nodes are restricted in some form. p_i stands for processing requirement (time) of task i; hence, it corresponds directly to $w(n_i)$. Typical restrictions are:
     - $p_i = p$. Each node has the same computation cost, $w(n_i) = w, \forall n_i \in \mathbf{V}$.
     - $p_i = 1$. Each node has unit computation cost, that is, $w(n_i) = 1$, $\forall n_i \in \mathbf{V}$. This is also known as *UET* (*unit execution time*). If $p_i = \{1, 2\}$ the computation cost of each node is either 1 or 2.
3. *Communication Costs.* This subfield relates to the communication costs associated with the edges of the task graph.
   - ∘. No communication delays occur. There are no weights associated with the edges of G; that is, $G = (\mathbf{V}, \mathbf{E}, w)$ or, for $G = (\mathbf{V}, \mathbf{E}, w, c)$, $c(e_{ij}) = 0 \, \forall e_{ij} \in \mathbf{E}$.
   - c_{ij}. Each edge $e_{ij} \in \mathbf{E}$ is associated with a communication cost $c(e_{ij})$. Restrictions might be specified for the values of $c(e_{ij})$, for instance, as in the next point.
   - $c_{ij} \leq p_{\min}$. Each edge has a communication cost that is smaller than or equal to the minimum computation cost of the nodes, $c(e_{ij}) \leq \min_{n \in \mathbf{V}} w(n)$, $\forall e_{ij} \in \mathbf{E}$. In Chrétienne et al. [34] this is referred to as *SCT* (*small communication time*). It implies $\min_{n \in \mathbf{V}} w(n) / \max_{e_{ij} \in \mathbf{E}} c(e_{ij}) \geq 1$, which is of course the definition of coarse granularity, $\min_{n \in \mathbf{V}} w(n) / \max_{e_{ij} \in \mathbf{E}} c(e_{ij}) = g(G) \geq 1$ (Section 4.4.3, Definition 4.20). Coarser granularity is specified with $c_{ij} \leq r \cdot p_{\min}$, where r is a constant and $0 < r \leq 1$ (designated r-SCT in Chrétienne et al. [34]).
   - c. Each edge has the same communication cost, $c(e_{ij}) = c$, $\forall e_{ij} \in \mathbf{E}$. If the field is $c = 1$, each edge has *unit communication time* (*UCT*), that is, $c(e_{ij}) = 1, \forall e_{ij} \in \mathbf{E}$.
4. *Task Duplication.*
   - ∘. Task duplication is not allowed.
   - *dup*. Task duplication is allowed.

***γ*—Optimality Criterion** The γ field specifies the optimality criterion for the scheduling problem and there are no subfields. All scheduling problems discussed in this book are about finding the shortest schedule length *sl*, which is represented by the symbol $C_{\max}$ in the γ field. Here C_i stands for the completion time of task n_i; that is, $C_i = t_f(n_i)$, with $C_{\max} = \max_{n_i \in \mathbf{V}} C_i$ being the maximum completion time of all tasks, which is the definition of the schedule length (Definition 4.10).

Examples The following examples illustrate the utilization of the $\alpha|\beta|\gamma$ notation. A short $\alpha|\beta|\gamma$ notation without any optional field, $P||C_{\max}$, refers to the problem of scheduling independent tasks with arbitrary computation costs on $|\mathbf{P}|$ identical processors such that the schedule length is minimized. $P2|prec, p_i = 1|C_{\max}$ is the

problem of scheduling tasks with precedence constraints and unit computation costs on two identical processors such that the schedule length is minimized. If the two processors are uniformly heterogeneous, this changes to $Q2|prec, p_i = 1|C_{\max}$. Finally, $P\infty|tree, c_{ij} \leq p_{\min}, dup|C_{\max}$ denotes the problem of scheduling a task graph with tree structure, arbitrary computation costs, and communication costs that are smaller than or equal to the minimum computation cost (i.e., the graph is coarse grained) on an unlimited number of identical processors. Node duplication is allowed and the objective is again to minimize the schedule length.

Generalizations and Reductions The theory of NP-completeness implies that a general problem cannot be easier, complexity-wise, than any of its special cases. Many of the previously defined subfields restrict the parameters of the task scheduling in some way, making it a subproblem of a general one. In particular, many specific graph structures are listed, which are also related to each other. A fork graph, for instance, is a special form of a tree. If it is proved that a particular scheduling problem is NP-complete for tree structured task graphs, then the scheduling problem is also NP-complete for an arbitrary graph structure. In face of this observation, it is interesting to realize the relations of the different graph structures defined for the β field. Figure 6.13 visualizes these relations. Each arrow points in the direction of the more general structure: that is, the graph at its tail is a special case of the graph at its head.

The above observation is reflected in the way NP-completeness of a problem is usually proved. A common step in such proofs is to reduce a known NP-complete problem in polynomial time to that problem, as is done in all NP-completeness proofs of this text. This is particularly easy if the known problem is a special case, as, for example, in Theorem 6.2. Thus, if NP-completeness is shown for a special case, the NP-completeness of all more general cases can be proved using such reductions. As a consequence, Figure 6.13 can also be interpreted as a reduction graph (Brucker and Knust [29]), although not all task graph structures lead necessarily to an NP-complete scheduling problem.

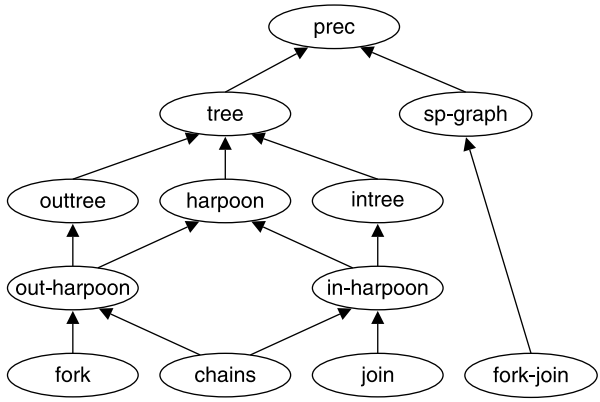


Figure 6.13. Relations between fundamental graph structures.

6.4.2 Without Communication Costs

This section is dedicated to scheduling problems that do not consider communication costs. They are all special cases of the scheduling problem analyzed in Sections 4.3 and 4.3.2. Table 6.1 lists the various scheduling problems, having six columns: (1) $\alpha|\beta|\gamma$ notation of the scheduling problem, (2) number of processors, (3) restrictions on computation costs (in next sections also communication costs), (4) structure of the task graph, (5) NP-hardness or complexity, and (6) reference to corresponding publication and/or section in this book.

As is established in Theorem 4.4, scheduling with an unlimited number of processors, $P\infty|prec|C_{\max}$, is polynomial-time solvable.

With a limited number of processors, $P|prec|C_{\max}$, the problem is NP-hard (Theorem 4.3) and remains so even for independent tasks $P||C_{\max}$, which is closely related to BINPACKING (Garey and Johnson [73]). Also, to schedule task graphs with unit execution time (UET) on a limited number of processors, $P|prec, p_i = 1|C_{\max}$, is an NP-hard problem.

However, scheduling without considering communication costs becomes more tractable with UET, as some of the problems with special graph structures are now polynomial-time solvable, for example, $P|p_i = 1|C_{\max}$, $P|tree, p_i = 1|C_{\max}$. So restricting the graph structure *or* the costs does not help to escape NP-hardness, but doing both makes the problem tractable in some cases.

Scheduling on only two processors proves to be more difficult than one might imagine, as such seemingly simple problems like $P2||C_{\max}$ and $P2|chains|C_{\max}$ are NP-hard. Nevertheless, scheduling with UET on two processors is polynomial-time solvable for an arbitrarily structured task graph, $P2|prec, p_i = 1|C_{\max}$.

The complexity for other fixed numbers of processors m, $Pm|prec|C_{\max}$ and also $Pm|prec, p_i = 1|C_{\max}$, is yet unknown.

6.4.3 With Communication Costs

This section comes back to scheduling with communication costs—the general scheduling problem of this text. As it encloses the case without communication costs (it is the case where $c(e) = 0, \forall e \in \mathbf{E}$), it can only be more difficult.

Again, Table 6.2 lists the various scheduling problems with the same six column types as in Table 6.1, with the only exception that the costs column now also includes restrictions on the communication costs. As defined in Section 4.4.3, $g(G)$ stands for the granularity of the task graph G, with $g(G) \geq 1$ indicating coarse granularity. In some cases the result remains valid also for weak granularity $g_{\text{weak}}(G)$, but this is not distinguished here.

That scheduling with communication costs is more difficult than without is immediately apparent from the first entry. $P\infty|prec, c_{ij}|C_{\max}$, that is, the scheduling of an arbitrary task graph on an unlimited number of processors, is NP-hard, while $P\infty|prec|C_{\max}$ is polynomial-time solvable. Even with restricted task graph structures, $P\infty|\{tree, harpoon, fork\text{-}join\}, c_{ij}|C_{\max}$, scheduling remains NP-hard, with the only exception being the primitive fork and join graphs $P\infty|\{fork, join\}, c_{ij}|C_{\max}$,

Table 6.1. Complexity Results for Task Scheduling Without Considering Communication Costs

$\alpha\|\beta\|\gamma$	Processors	Costs	Structure	Complexity	Reference
$P\infty\|prec\|C_{\max}$	Unlimited			$O(\mathbf{V}+\mathbf{E})$	Theorem 4.4 (Section 4.3.2)
$P\|prec\|C_{\max}$	$\|\mathbf{P}\|$			NP-hard	Karp [99], Theorem 4.3 (Section 4.3.2)
$P\|\|C_{\max}$			Independent	NP-hard	Garey and Johnson [72]
$P\|int\text{-}ord\|C_{\max}$			Interval–ordered	NP-hard	Papadimitriou and Yannakakis [141]
$P\|prec, p_i = 1\|C_{\max}$		$w(n) = 1$		NP-hard	Ullman [192]
$P\|p_i = 1\|C_{\max}$			Independent	$O(\mathbf{V})$	Round robin
$P\|tree, p_i = 1\|C_{\max}$			Tree	$O(\mathbf{V})$	Hu [93]
$P\|op\text{-}forest, p_i = 1\|C_{\max}$			Opposing forest	NP-hard	Garey et al. [74]
$P\|int\text{-}ord, p_i = 1\|C_{\max}$			Interval–ordered	$O(\mathbf{V}+\mathbf{E})$	Papadimitriou and Yannakakis [141]
$P2\|prec\|C_{\max}$	2			NP-hard	
$P2\|\|C_{\max}$			Independent	NP-hard	Lenstra et al. [119]
$P2\|chains\|C_{\max}$			Chains	NP-hard	Du et al. [56]
$P2\|prec, p_i \in \{1,2\}\|C_{\max}$		$w(n) \in \{1,2\}$		NP-hard	Ullman [192]
$P2\|prec, p_i = 1\|C_{\max}$		$w(n) = 1$		$O(\mathbf{V}^2)$	Coffman and Graham [38], Fujii et al. [70], Gabow [71], Sethi [168]

Table 6.2. Complexity Results for Task Scheduling with Considering Communication Costs

$\alpha\|\beta\|\gamma$	Processors	Costs	Structure	Complexity	Reference
$P\infty\|prec, c_{ij}\|C_{\max}$	Unlimited			NP-hard	Exercise 4.4.3
$P\infty\|tree, c_{ij}\|C_{\max}$			Tree	NP-hard	Chrètienne [33]
$P\infty\|harpoon, c_{ij}\|C_{\max}$			Harpoon	NP-hard	Chrètienne [33]
$P\infty\|fork\text{-}join, c_{ij}\|C_{\max}$			Fork-join	NP-hard	Chrètienne [32], Exercise 4.4.3
$P\infty\|\{fork, join\}, c_{ij}\|C_{\max}$			Fork *or* join	Polynomial	Chrètienne [31]
$P\infty\|prec, c_{ij} \leq r \cdot p_{\min}\|C_{\max}$		$g(G) \geq 1/r$, $0 < r \leq 1$		NP-hard	Picouleau [149, 150]
$P\infty\|tree, c_{ij} \leq p_{\min}\|C_{\max}$		$g(G) \geq 1$	Tree	$O(\mathbf{V})$	Chrètienne [30]
$P\infty\|bipartie, c_{ij} \leq p_{\min}\|C_{\max}$			Bipartie	$O(\mathbf{E} \log \mathbf{E})$	Chrètienne and Picouleau [35]
$P\infty\|sp\text{-}graph, c_{ij} \leq p_{\min}\|C_{\max}$			Series-parallel	Polynomial	Chrètienne and Picouleau [35]
$P\infty\|prec, p_i = 1, c \geq 1\|C_{\max}$		$w(n) = 1$, $c(e_{ij}) = c \geq 1$		NP-hard	Papadimitriou and Yannakakis [142]
$P\infty\|prec, p_i = 1, c = 1\|C_{\max}$		$w(n) = 1$, $c(e_{ij}) = 1$		NP-hard	Papadimitriou and Yannakakis [142], Picouleau [149, 150]
$P\|prec, c_{ij}\|C_{\max}$	$\|\mathbf{P}\|$			NP-hard	Theorem 4.1 (Section 4.2.2)
$P\|prec, p_i = 1, c = 1\|C_{\max}$		$w(n) = 1$, $c(e_{ij}) = 1$		NP-hard	Rayward-Smith [158]
$P\|tree, p_i = 1, c = 1\|C_{\max}$			Tree	NP-hard	Veltman [195], Lenstra et al. [120]
$P\|bipartie, p_i = 1, c = 1\|C_{\max}$			Bipartie	NP-hard	Hoogeveen et al. [90]
$P\|int\text{-}ord, p_i = 1, c = 1\|C_{\max}$			Interval–ordered	$O(\mathbf{E} + \mathbf{VP})$	Ali and El-Rewini [10], El-Rewini and Ali [62]
$Pm\|tree, p_i = 1, c = 1\|C_{\max}$	m (fixed)	$w(n) = 1$, $c(e_{ij}) = 1$	Tree	Polynomial	Varvarigou et al. [194]
$P2\|tree, p_i = 1, c = 1\|C_{\max}$	2	$w(n) = 1$, $c(e_{ij}) = 1$	Tree	Polynomial	El-Rewini and Ali [61], Lenstra et al. [120] (intree)

which can be scheduled optimally in polynomial time on an unlimited number of processors. Interestingly, the harpoon graph, $P\infty|harpoon, c_{ij}|C_{\max}$, which is closely related to the fork and join graphs and only slightly more elaborated, makes scheduling NP-hard.

While any form of coarse granularity, $g(G) \geq 1/r$, $0 < r \leq 1$, of the task graph does not make the problem tractable, $P\infty|prec, c_{ij} \leq r \bullet p_{\min}|C_{\max}$, the combination of coarse granularity and special graph structures does: $P\infty|\{tree, bipartie, sp\text{-}graph\}, c_{ij} \leq p_{\min}|C_{\max}$. UET and UCT alone, however, do not make scheduling polynomial-time solvable, $P\infty|prec, p_i = 1, c \geq 1|C_{\max}$. This situation is similar to scheduling without communication costs: restricting the graph structure *or* the costs does not help, but doing both makes the problem tractable in some cases.

Of course, scheduling with communication costs does not become easier when the number of processors is limited. For example, $P|tree, p_i = 1, c = 1|C_{\max}$ is NP-hard, while $P\infty|tree, c_{ij} \leq p_{\min}|C_{\max}$ (which includes $P\infty|tree, p_i = 1, c = 1|C_{\max}$) is not. Only interval-ordered task graphs with UET and UCT can be scheduled optimally on a limited number of processors, $P|int\text{-}ord, p_i = 1, c = 1|C_{\max}$, in polynomial time.

When the number of processors is fixed, scheduling a tree structured task graph with UET and UCT is tractable, for example, $P2|tree, p_i = 1, c = 1|C_{\max}$.

6.4.4 With Node Duplication

Intuitively, node duplication makes scheduling easier, because the costs of communication can be eliminated by making it local. Hence, the problem comes closer to scheduling without communication costs. Table 6.3 lists various scheduling problems where node duplication is allowed. Of course, node duplication implies communication costs, since duplicating a node is meaningless otherwise.

Needless to say that scheduling with task duplication is NP-hard even on an unlimited number of processors $P\infty|prec, c_{ij}, dup|C_{\max}$. On the one hand, task duplication makes the scheduling of an outtree on an unlimited number of processors tractable, $P\infty|outtree, c_{ij}, dup|C_{\max}$. On the other hand, it does not help to bring the complexity of scheduling intrees down, $P\infty|intree, c_{ij}, dup|C_{\max}$, which consequently remains NP-hard (see also Section 6.2).

Nevertheless, scheduling of coarse grained task graphs on an unlimited number of processors is polynomial-time solvable with task duplication, $P\infty|prec, c_{ij} \leq p_{\min}, dup|C_{\max}$. One might wonder how this is possible, given that without task duplication, $P\infty|prec, c_{ij} \leq p_{\min}|C_{\max}$, the problem is NP-hard and that duplication is not beneficial in all cases, in particular, for intrees. Looking back at Section 6.4.3, it can be observed that $P\infty|tree, c_{ij} \leq p_{\min}|C_{\max}$ is in fact tractable, which includes the scheduling of coarse grained intrees. Hence, the case for which task duplication is not beneficial is already tractable without duplicating nodes.

The scheduling of non-coarse grained task graphs, even with UET and UCT, where UET < UCT, $P\infty|prec, p_i = 1, c > 1, dup|C_{\max}$, belongs to the class of NP-hard problems.

Table 6.3. Complexity Results for Task Scheduling with Node Duplication

$\alpha\|\beta\|\gamma$	Processors	Costs	Structure	Complexity	Reference
$P\infty\|prec, c_{ij}, dup\|C_{\max}$	Unlimited			NP-hard	
$P\infty\|outtree, c_{ij}, dup\|C_{\max}$			Outtree	Polynomial	Chrètienne [33]
$P\infty\|intree, c_{ij}, dup\|C_{\max}$			Intree	NP-hard	Chrètienne [33]
$P\infty\|in\text{-}harpoon, c_{ij}, dup\|C_{\max}$			In-harpoon	NP-hard	Chrètienne [33]
$P\infty\|prec, c_{ij} \leq p_{\min}, dup\|C_{\max}$		$g(G) \geq 1$		Polynomial	Colin and Chrètienne [40]
$P\infty\|prec, p_i = 1, c > 1, dup\|C_{\max}$		$w(n) = 1,$ $c(e_{ij}) = c > 1$		NP-hard	Papadimitriou and Yannakakis [142]
$P\infty\|prec, p_i = 1, c > 0, dup\|C_{\max}$		$w(n) = 1,$ $c(e_{ij}) = c > 0$		$O(\mathbf{V}^{c+1})$	Jung et al. [97]
$P\|prec, c_{ij}, dup\|C_{\max}$	$\|\mathbf{P}\|$			NP-hard	
$P\|prec, p_i = 1, c = 1, dup\|C_{\max}$		$w(n) = 1,$ $c(e_{ij}) = 1$		NP-hard	Chrètienne and Picouleau [35]

Table 6.4. Complexity Results for Scheduling on Heterogeneous Processors

$\alpha\|\beta\|\gamma$	Processors	Costs	Structure	Complexity	Reference
$Q\|prec, c_{ij}\|C_{\max}$	$\|\mathbf{P}\|$	$c(e_{ij})$		NP-hard	Theorem 6.2 (Section 6.3)
$Q\|prec\|C_{\max}$				NP-hard	
$Q\|chains, p_i = 1\|C_{\max}$		$w(n) = 1$	Chains	NP-hard	Kubiak [107]
$Q2\|chains, p_i = 1\|C_{\max}$	2	$w(n) = 1$	Chains	$O(\mathbf{V})$	Brucker et al. [28]

With a limited number of processors, task duplication does not change the fate of scheduling—it remains NP-hard, $P|prec, c_{ij}, dup|C_{\max}$, even with UET and UCT, $P|prec, p_i = 1, c = 1, dup|C_{\max}$.

6.4.5 Heterogeneous Processors

There is not much to report for task scheduling on heterogeneous processors. As a generalization of the scheduling problem, it can only be harder. Very few results are known and Table 6.4 lists some of them.

Both scheduling with and without communication costs is NP-hard on a limited number of consistent heterogeneous processors, $Q|prec, c_{ij}|C_{\max}$ and $Q|prec|C_{\max}$.

With heterogeneous processors, even extremely simple task graphs consisting of chains with UET and no communication costs, $Q|chains, p_i = 1|C_{\max}$, cannot be optimally scheduled in polynomial time. On homogeneous processors this is even possible for trees, $P|tree, p_i = 1|C_{\max}$. Only when the number of processors is limited to two, is the problem tractable, $Q2|chains, p_i = 1|C_{\max}$.

It can be concluded that task scheduling remains a difficult problem even for relaxed parameters. Some seemingly simple problems are NP-hard, for example, the scheduling of independent tasks. Optimal solutions in polynomial time are only known for a very few, specific problems. More on scheduling problems, not only limited to task scheduling, and their NP-completeness can be found in Brucker [27], Chrétienne et al. [34], Coffman [37], and Leung [121].

6.5 GENETIC ALGORITHMS

NP-hard problems are often tackled with the application of stochastic search algorithms like simulated annealing or genetic algorithms. Task scheduling is no exception in this respect, and the genetic algorithm (GA) method especially has gained considerable popularity. A GA is a search algorithm that is based on the principles of evolution and natural genetics. A pool of solutions to the problem, the population, creates new generations by breeding and mutation. According to evolutionary theory, only the most suited elements of the population are likely to survive and generate

offspring, transmitting their biological inheritance to the next generation (Davis [54], Goldberg [78], Holland [89], Man et al. [133], Reeves and Rowe [161]).

This section studies how a GA can be applied to the task scheduling problem. As the area of genetic algorithms is very broad, only a very brief introduction to its principles can be given. For more information please refer to the corresponding literature, for example, the references just given. The rest of the section then concentrates on how the GA method can be applied to the scheduling problem.

6.5.1 Basics

A GA operates through a simple cycle of stages: creation of a population, evaluation of the elements of the population, selection of the best elements, and reproduction to create a new population. Algorithm 20 outlines a simple GA and the fundamental components are described in the following.

- *Chromosome.* A chromosome represents one solution to the problem, encoded as a problem-specific string.
- *Population.* A pool of chromosomes is referred to as the population. The initial population is typically created randomly, sometimes guided by heuristics.
- *Evaluation.* At each iteration, the quality of each chromosome is evaluated with a *fitness function*.
- *Selection.* Through selection, the GA implements the evolutionary principle "survival of the fittest." The selection operator chooses chromosomes based on their fitness value computed during evaluation.
- *Crossover.* Crossover is the main operator for the creation of new chromosomes. Two "parent" chromosomes are combined to build a new chromosome, which thereby inherits the genetic material of its ancestors.
- *Mutation.* The mutation operator randomly changes parts of a chromosome. It is applied with low probability and mainly serves as a safeguard to avoid the convergence to a locally best solution.

A GA terminates after a specified number of generations, when a time limit is exceeded or when a solution with sufficient quality is found.

Algorithm 20 Outline of a Simple GA

```
Create initial population
Evaluation
while termination criterion not met do
    Selection
    Crossover
    Mutation
    Evaluation
end while
Return best solution
```

6.5.2 Chromosomes

The genetic formulation of a problem begins with the definition of an appropriate chromosome encoding. In order to achieve good performance, the chromosome should be simple, because this permits one to employ simple and fast operators.

For task scheduling, a chromosome represents a solution to the scheduling problem—in other words a schedule (Definition 4.2). A schedule consists of the processor allocation and the start time of each node of the task graph. Theorem 5.1 establishes that it suffices to know the processor allocation and the execution order of the nodes in order to construct a schedule (using the end technique) that is not longer than any other schedule with the same processor allocation and node execution order. Given this observation, two types of chromosome representations for task scheduling can be distinguished (Rebreyend et al. [160]).

Indirect Representation In the indirect representation, the chromosome string holds information that serves as the input for a heuristic to create a schedule. Two approaches are obvious for the encoding of the chromosome:

- *Processor Allocation.* The chromosome encodes the processor allocation of the nodes. Every chromosome consists of $|\mathbf{V}|$ genes, each of which represents one node of the task graph. Assuming a consecutive numbering of the nodes and processors, starting with 1, gene i corresponds to node $n_i \in \mathbf{V}$. The value of the gene corresponds to the index of the processor, 1 to $|\mathbf{P}|$, to which the node is allocated (Figure 6.14). In GA terminology this is called *value encoding*. Such an encoding is employed, for example, by Benten and Sait [20]. In order to construct a schedule for each chromosome, the node order must be determined by a separate heuristic. Naturally, the techniques for the first phase of list scheduling can be employed (Section 5.1.3). In turn, this means that the same node order can be used for all chromosomes, or that a particular node order is determined for each chromosome, based on the processor allocation, for instance, a node priority based on allocated levels (Section 4.4.2).
- *Node List.* The chromosome encodes a node list. Every chromosome is a permutation of the $|\mathbf{V}|$ nodes of the task graph. Each gene represents a position in the node list and the value of the gene is the index of the node at that position (Figure 6.15). In GA terminology this is called *permutation encoding*. Depending on the operators used in the GA (see below), the node list can be the execution order of the nodes or merely a priority ordering. In the former case, the node list must adhere to the precedence constraints of the nodes. If this holds, a chromosome essentially establishes the first part of list scheduling (Algorithm 9). In the latter case, the position of a node in the list merely specifies its priority. An ordering that adheres to the precedence constraints and the priorities is computed using Algorithm 12. As the chromosome only establishes (directly or indirectly) the node order, the processor allocation must be

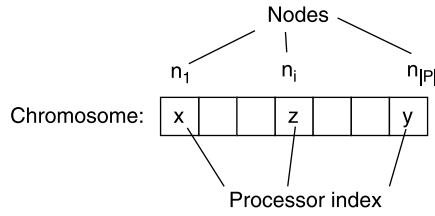


Figure 6.14. Chromosome that encodes processor allocation.

determined by a heuristic. Again, this can be done globally (i.e., identically for all chromosomes) or individually for each chromosome, which naturally can be done with list scheduling (Algorithm 9). Kwok and Ahmad [110] propose to encode the node list in the chromosome, with the nodes always being in precedence order.

Some authors proposed value encoding for the node priorities (e.g., Ahmad and Dhodhi [3], and Dhodi et al. [55], Sandnes and Megson [164, 165]). The chromosome has then the same structure as the processor allocation chromosome—that is, each gene represents a node—while each gene's value is the corresponding node's priority. The actual node order is determined for each chromosome with a procedure such as Algorithm 12. A disadvantage of such an encoding is that nodes can have the same priority.

Figure 6.16 gives examples for the two different encoding approaches. The chromosomes are for the scheduling of the sample task graph (Figure 6.16(*a*)) on three processors.

Direct Representation In the direct representation, the chromosome encodes the processor allocation *and* the execution order of the nodes. Consequently, the chromosome consists of two parts—the two chromosome approaches used in the indirect representation as discussed earlier. With this information, the nodes' start times are readily computed, using a simple list scheduling (see Theorem 5.1). No heuristic decisions are taken in the construction of the schedule. Wang et al. [198] use such a two-part chromosome—one for the processor allocation and the other for a global node list.

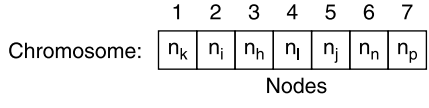


Figure 6.15. Chromosome that encodes node list.

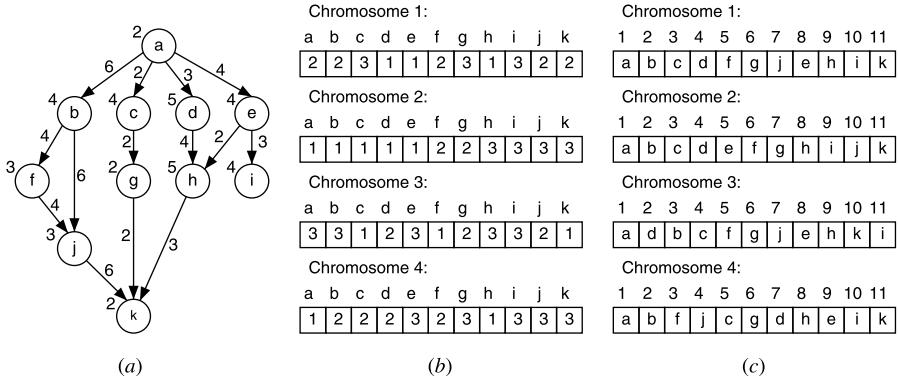


Figure 6.16. Four examples of chromosomes that encode processor allocations (*b*) or node lists (*c*) for the scheduling of the the sample task graph (*a*) on three processors.

Figure 6.17 shows two chromosomes consisting of two parts, one for the processor allocation and one for the global node list. They encode the scheduling of the sample task graph (e.g., Figure 6.16(*a*)) on three processors. In this case, the node lists are in precedence order. Beneath the chromosomes, the corresponding schedules are depicted.

It goes without saying that many other chromosome encodings are possible. Strictly speaking, it is not necessary to know the absolute order of the nodes. It suffices to have partial orders of the nodes allocated to each processor. Such an encoding is reported by Hou et al. [92], which is further improved by Correa et al. [43, 44]. Zomaya et al. [213] employ a similar encoding. Wu et al. [206], use lists of node–processor pairs. The order of the pairs establishes the node order. This encoding can be very useful in combination with node duplication, as a node can be allocated to more than one processor. See also Exercise 6.10, which asks you to suggest a chromosome encoding for task scheduling with node duplication.

Comparison of Representation Types An essential difference between the direct and indirect representation types of chromosomes is their ability to represent an optimal schedule. As mentioned before, from Theorem 5.1 it is known that the processor allocation and the node order of an optimal schedule are enough to reconstruct a schedule of optimal length. Therefore, it is guaranteed that the optimal solution is situated in the search space of GAs with direct representation.

In contrast, from Section 5.2 it is known that scheduling with a given processor allocation is still NP-hard. So if the chromosome encodes only the processor allocation, the GA based scheduling algorithm is still faced with an NP-hard problem for each chromosome. Heuristic decisions taken for ordering the nodes cannot guarantee that an optimal schedule will be obtained, even if the processor allocation is optimal. Equally, finding the optimal processor allocation for a node ordering given by a chromosome is also NP-hard. To realize this, just consider the

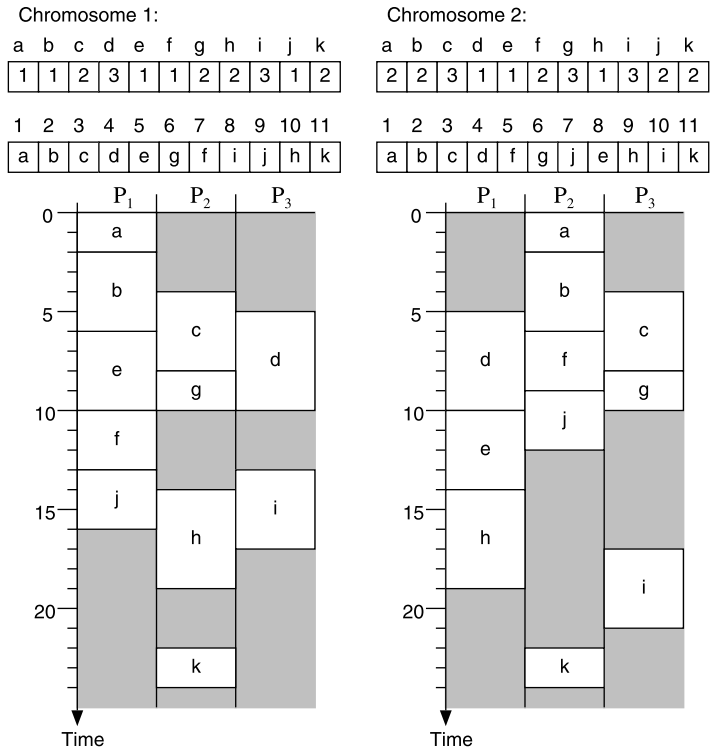


Figure 6.17. Two examples of chromosomes (top) that encode both processor allocation and node list for the scheduling of the sample task graph (Figure 6.16(*a*)) on three processors, and the schedules they represent (bottom).

scheduling of independent nodes, $P||C_{\max}$, which is NP-hard, although the node order is irrelevant. Once again, heuristic decisions taken to allocate the nodes cannot guarantee that an optimal schedule will be obtained, even if the node order is optimal.

It follows that it is not guaranteed for GAs with indirect representation that the optimal solution lies in their search space. In other words, the direct representation permits one to reflect an optimal solution, while the indirect representation sometimes cannot.

This advantage of the direct representation comes with a price, however. First, by having two very different parts, the simplicity of the chromosome is lost. Second, as a direct consequence, more complex crossover and mutation operators must be employed. Essentially, a separate operator must be used for each part, because they are of different type—one is value encoded, the other is permutation encoded. Last but not least, the search space of the GA increases dramatically in comparison to the simpler indirect representation chromosomes. The total number of possible values is the product of the possible values of each part.

Initial Population As is typical for genetic algorithms, most scheduling heuristics generate the initial population randomly, with the necessary care on feasible solutions. The typical population size N ranges from less than ten chromosomes to about a hundred. With the above presented chromosome encodings, the random generation is straightforward:

- *Processor Allocation Chromosomes.* Processor allocation chromosomes are easily constructed by randomly assigning a number from 1 to $|\mathbf{P}|$ to each gene of the chromosome.
- *Node List Chromosomes.* Node list chromosomes are similarly easy to construct. For each chromosome the nodes of the task graph are randomly shuffled. If the nodes are supposed to be in precedence order, Algorithm 12 can bring them into such an order, using the randomly generated order as a node priority.

To enrich the initial seed, and in turn to potentially speed up the GA and/or improve the quality of the result, it is not uncommon to complement the initial population with nonrandomly generated chromosomes. For example, this can be schedules produced by other fast heuristics such as list scheduling (Kwok and Ahmad [110]). Other conceivable seed chromosomes are those that represent extreme situations. For example, one initial chromosome can represent a sequential schedule, where all tasks are assigned to the same processor. In combination with elitism (to be discussed later in the context of selection), this can lead to certain guarantees on the quality of the result.

The downside of such seed chromosomes is that they might bias the genetic search toward local minima. Hence, they should be used with caution and only a small fraction of the initial population should be nonrandom.

Fitness Function As the objective in task scheduling is to find the shortest possible schedule, the fitness of a chromosome is directly related to the length of the associated schedule (Hou et al. [92]).

Definition 6.7 (Fitness Function) *Let c be a chromosome and $\mathcal{S}_c$ the schedule associated with c. The fitness of c is defined as*

$$F(c) = sl_{\max} - sl(\mathcal{S}_c), \tag{6.12}$$

where $sl_{\max}$ is the maximum schedule length among all chromosomes of the population.

Hence, to evaluate the fitness of a chromosome c, the corresponding schedule $\mathcal{S}_c$ must be constructed. How this is performed depends on the representation type of the chromosome. In case of an indirect representation, this involves the application of a heuristic.

Variations to this definition of the fitness function have been proposed. For instance, in Dhodhi et al. [55] the fitness value is normalized with the sum of all fitness values of the population. In Zomaya et al. [213], a small constant value is added to impede

a fitness of $F(c) = 0$, as this is problematic for some selection operators, as will be seen later.

6.5.3 Reproduction

The operators are the instruments with which a GA explores the search space of a problem. Crossover and mutation are reproduction operators that create new chromosomes from existing ones. Their design is closely tied to the chosen chromosome representation. Since a solution represented by a chromosome should be feasible—otherwise the search space becomes even larger—operators must be defined correspondingly.

Crossover The crossover operator is the more significant one of the two. It implements the principle of evolution. New chromosomes are created with this operator by combining two randomly selected parent chromosomes, thereby inheriting the genetic material of its ancestors. How this is done depends on the encoding of the chromosome. In Section 6.5.2 two indirect chromosome encodings are presented, whereby the direct representation is a combination of the two. Since the two encodings differ significantly, two different operators are necessary.

- *Processor Allocation.* For the chromosome encoding of the processor allocation, quite simple crossover operators can be employed. The processor allocation chromosome is a value encoded chromosome, where each gene can assume the same range of values (1 to $|\mathbf{P}|$). Furthermore, the value of one gene has no impact on the possible values of the other genes. Therefore, a simple two-point crossover operator works as follows.

  Given two randomly chosen chromosomes c_1 and c_2, two new chromosomes c_3 and c_4 are generated by swapping a randomly determined gene interval. Let the interval range from i to j. Outside this interval, that is, $[1, i-1]$ and $[j+1, |\mathbf{V}|]$, the genes of c_3 have the values of c_1 and inside those of c_2. For c_4 it is the converse.

  Figure 6.18 illustrates this crossover operator. The simpler and more common single-point crossover operator is a special case of this, achieved by setting $j = |\mathbf{V}|$. Figure 6.19 visualizes a single-point crossover operator. The extension of this concept to multiple crossover points is straightforward. Note that the generated new chromosomes are always valid chromosomes; that is, they represent a valid processor allocation of the tasks $\mathbf{V}$.

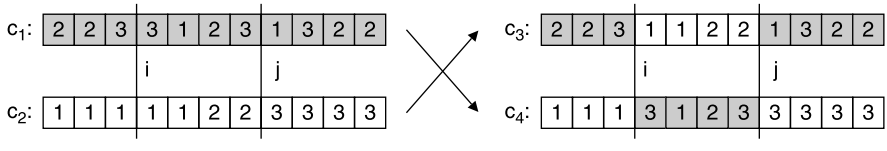


Figure 6.18. Illustration of two-point crossover of processor allocation chromosomes ($i = 4$, $j = 7$).

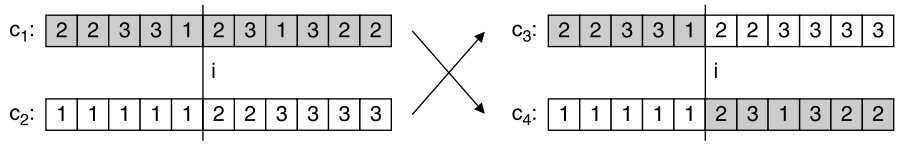


Figure 6.19. Illustration of single-point crossover of processor allocation chromosomes ($i = 6$).

- *Node List.* The chromosome encoding of a node list is a permutation encoding of the node set $\mathbf{V}$. Each element of the permutation, that is, each node $n \in \mathbf{V}$, can appear only once in the chromosome. Thus, the simple crossover operator presented for the processor allocation encoding cannot be employed. To understand this, image two chromosomes c_1 and c_2 in which node f is at position 5 in c_1 and position 6 in c_2. In the case where a single-point crossover operator divides the chromosomes at, for example, position 6, the resulting chromosome c_3 will have node f twice, at position 5 and at position 6, while there will be no f in chromosome c_4. This is depicted in Figure 6.20. The problem is overcome with the following single-point permutation crossover operator.

  Given two randomly chosen chromosomes c_1 and c_2, a crossover point i, $1 \leq i < |\mathbf{V}|$, is selected randomly. The genes $[1, i]$ of c_1 and c_2 are copied to the genes $[1, i]$ of the new chromosomes c_3 and c_4, respectively. To fill the remaining genes $[i + 1, |\mathbf{V}|]$ of c_3 (c_4), chromosome c_2 (c_1) is scanned from the first to the last gene and each node that is not yet in c_3 (c_4) is added to the next empty position of c_3 (c_4) in the order that it is discovered. Figure 6.21 illustrates the procedure of this operator.

  Under the condition that the node lists of chromosomes c_1 and c_2 are in precedence order, this operator even guarantees that the node lists of c_3 and c_4 also are. It is easy to see this for the genes $[1, i]$ of both c_3 and c_4 as they are only copied from c_1 and c_2. The remaining genes of c_3 and c_4 are filled in the same relative order in which they appear in c_2 and c_1, respectively. Hence, among themselves, these remaining nodes must also be in precedence order. Furthermore, there cannot be a precedence conflict between the nodes on the left side of the crossover point with those on the right side of the crossover point,

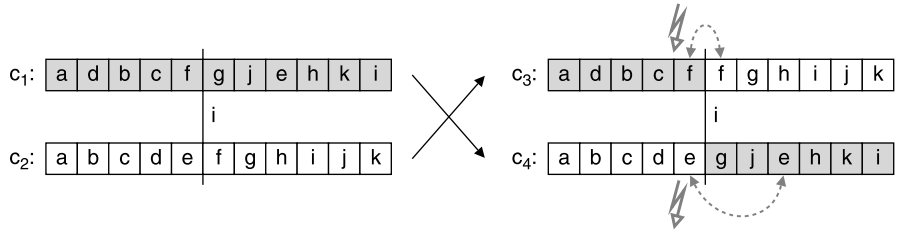


Figure 6.20. Simple single-point crossover as in Figure 6.19 *fails* for node list chromosomes ($i = 6$).

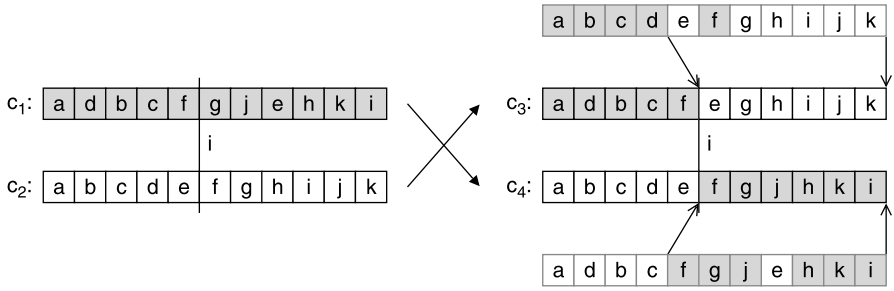


Figure 6.21. Illustration of single-point crossover of node list chromosomes ($i = 6$).

because this separation of the nodes into two groups has not changed from c_1 to c_3 neither from c_2 to c_4 and it adheres to the precedence constraints in c_1 and c_2.

Crossover is usually performed with a probability of 0.5 to 1. If it is not performed, the selected chromosomes c_1 and c_2 are simply copied to the population of the next generation.

Mutation The mutation operator is applied with a much lower probability (about 0.1 or less) than the crossover operator. Its main purpose is to serve as a safeguard to avoid the convergence of the state search to a locally best solution. A new chromosome c_2 is created by copying a randomly picked chromosome c_1 and randomly changing parts of it. Again, a distinction is made between the operators for processor allocation and node list chromosomes.

- *Processor Allocation.* Being a value encoded chromosome, the mutation operator can simply change the values of randomly picked genes. This can be done with any procedure, like increasing or decreasing the current values, or by assigning a randomly determined value. The only condition is that the resulting values are in the range 1–|**P**|. Figure 6.22 illustrates such a simple mutation operator, where two genes are "mutated."

  Another alternative is to swap the values of two randomly picked genes as visualized in Figure 6.23. In terms of processor allocation, this has the effect that the number of nodes on each processor remains constant. While this is good for

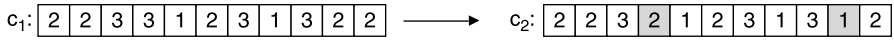


Figure 6.22. Illustration of simple mutation operator for processor allocation chromosome.

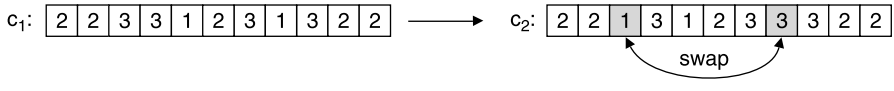


Figure 6.23. Illustration of swapping mutation operator for processor allocation chromosome.

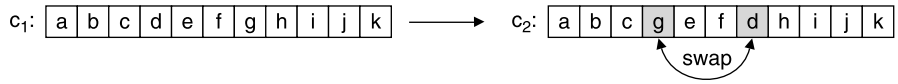


Figure 6.24. Illustration of swapping mutation operator for node list chromosome.

maintaining a certain load balance in a processor allocation, it does not explore the entire solution space. Especially for task graphs with high communication costs (i.e., high CCRs (Definition 4.23)), reducing communication costs is often crucial. The swapping mutation operator does not support this.

- *Node List.* For chromosomes with permutation encoding, a swapping mutation operator, as just described, is very suitable. The purpose of the mutation operator is to randomly change the node order at some places—swapping two nodes does exactly that (Figure 6.24). In contrast to the previously discussed crossover operator, however, such swapping can bring the nodes out of precedence order. If the node list chromosomes are supposed to be in precedence order, a correcting procedure must be applied after the mutation. As mentioned before, Algorithm 12 can serve this purpose, using the position of the nodes in the mutated chromosome as the node priority.

The direct representation type (as discussed in Section 6.5.2) combines the processor allocation and the node list in one chromosome. Since these parts differ strongly, the simple solution for the operators is to apply the above described ones separately to each part.

Other operators, in combination with appropriate chromosome encodings, have been suggested, which attempt to avoid the creation of invalid chromosomes, especially regarding the node order (e.g., Hou et al. [92], Zomaya et al. [213]).

6.5.4 Selection, Complexity, and Flexibility

Selection The selection operator of the GA mimics the "survival of the fittest" of evolution. Consequently, it is not simply the selection of the fittest chromosomes from the pool of chromosomes, although fitter chromosomes should have a higher probability of survival. A stochastic approach is needed for the process.

Many selection operators have been suggested for GAs in the literature. As the scheduling problem has a clear and appropriate definition of the fitness function, standard selection methods can be utilized. The following two have been used before.

Tournament In tournament selection, two chromosomes are randomly picked from the pool. The fitter of the two goes into the population of the next generation, while the losing one is either discarded or put back into the pool. This process is repeated until the initial population size has been reached for the next generation, that is, N tournaments are performed. Such a selection is employed in Sinnen et al. [180].

Roulette Wheel This selection method simulates the behavior of a roulette wheel, where each chromosome has a slot on the wheel. The size of each slot is proportional to the fitness of the corresponding chromosome. In each iteration a position on the roulette wheel is randomly determined and the corresponding chromosome is selected. This process is repeated N times until all chromosomes for the next generation have been selected. In roulette wheel selection, a chromosome c with fitness $F(c) = 0$ has no chance of getting selected. This is not desirable in a GA, especially if such chromosomes are not extremely rare. With Definition 6.7 of the fitness function this can happen though. A simple solution is to add a small value to the fitness value of each chromosome, for example, a very small percentage of the maximum encountered schedule length $sl_{\max}$ (Zomaya et al. [213]). Examples for GA based scheduling algorithms that employ this operator are given by Dhodhi et al. [55], Hou et al. [92], and Zomaya et al. [213].

Elitism Elitism is an extension to the selection process that guarantees the survival of the fittest chromosome. With many selection operators (e.g., roulette wheel), the survival of the fittest chromosome is probable, but not guaranteed. With elitism, the simple solution is to just copy the fittest chromosome(s) from one generation to the next.

For GA based task scheduling algorithms, this means that the schedule length of the fittest chromosome upon termination is shorter than or equal to the schedule lengths of the initial chromosomes. By putting a chromosome representing a sequential schedule into the initial population (Section 6.5.2), it is guaranteed that the GA will not produce a solution that is worse than a sequential schedule. This is not as natural as it sounds, given that list scheduling with start time minimization (Theorem 5.2) has no such guarantee. Furthermore, using elitism, GA based task scheduling algorithms can improve schedules produced by other heuristics. It suffices to put chromosomes representing such schedules into the initial population. Elitism then guarantees that the final result will not be worse.

Complexity Usually, the designer of a GA for solving scheduling problems attempts to create operators of low complexity, in particular, lower than the complexity of the evaluation of a chromosome, which is essentially the construction of a schedule. As most GAs use an algorithm based on list scheduling, this complexity is at least $O(\mathbf{V} + \mathbf{E})$, for example, when the processor allocation is encoded in the chromosome. The runtime of a GA is then determined by the population size, N, and the number of generations Gen. Thus, GAs have a complexity of at least $O(N \times Gen \times (\mathbf{V} + \mathbf{E}))$. Typically, the values for the population size and the number of generations are up to a few hundred.

Flexibility GAs are a very flexible instrument for task scheduling. It is quite simple to adapt a GA to scheduling in a special environment, for example, in systems with heterogeneous processors (Singh and Youssef [170], Wang et al. [198], Woo et al. [205]), or in systems with only partially connected processors (Sandnes and Megson [164]). Moreover, scheduling techniques such as node duplication can be included (Sandnes and Megson [165], Tsuchiya et al. [191], Wu et al. [206]). In Dhodhi et al. [55],

scheduling of the communications is included in the GA based scheduling algorithm. All this is achieved by correspondingly enhanced chromosomes and the combination of GA with other scheduling heuristics.

The flexibility of GAs is also their shortcoming. GAs have a myriad of parameters which impact on the quality of the solution and the performance. The chromosome representation, the size of the initial population, the probability of crossover and mutation—just to name a few. As a consequence, finding a good GA heuristic involves many cycles of experiment and refinement.

6.6 CONCLUDING REMARKS

This chapter studied advanced aspects of task scheduling. The first sections presented scheduling techniques that are utilized in scheduling algorithms to improve the quality of the produced schedules. The insertion technique tries to use idle slots between already scheduled nodes to improve the processor utilization efficiency. Node duplication, on the other hand, aims at reducing the communication costs of schedules. Following the framework approach of this text, they were analyzed detached from concrete algorithms in a general manner.

Up to that point, everything was based on the scheduling model as introduced in Section 4.1. In Section 6.3 it was extended toward heterogeneous processors. The extension is simple but powerful and most of the discussed scheduling techniques can be employed with little or no modification.

After this, the chapter returned to a topic that had been more or less skipped in Chapter 4—the analysis of special cases of the scheduling problem. Unfortunately, most of them remain NP-hard, as could be seen in the extensive survey of scheduling problems in Section 6.4. The survey employed the $\alpha|\beta|\gamma$ notation for the classification of the various scheduling problems.

As the scheduling problem is in general NP-hard, genetic algorithms have been employed for its solution. This chapter discussed how genetic algorithms can be applied to task scheduling and the implications of different approaches.

Genetic algorithms are employed in many other areas apart from task scheduling. But the GA approach is not the only generic technique that has been applied to task scheduling. Integer linear programming (ILP) is a technique that is utilized for all sorts of scheduling problems. Hanen and Munier [85] formulate an integer linear programming (ILP) based algorithm for task scheduling. Another universal technique, which has been applied recently to task scheduling (Kwok and Ahmad [115]) is A* search (Berman and Paul [22]). In contrast to stochastic search algorithms, A* explores the entire search space systematically and therefore obtains the optimal result. It tries to reduce the runtime by pruning of unpromising state subtrees. For task scheduling, its runtime is nevertheless exponential; therefore, it can only be applied to relatively small graphs.

It goes without saying that there are many other interesting aspects of scheduling. One last example shall be scheduling tools. Many task scheduling algorithms have been implemented in graph based parallelization tools. In the parallelization process of such tools, a graph is generated from an initial program specification. This

specification usually has the form of annotated code, where the annotations specify the task graph structure. The graph is then scheduled onto the target parallel system and code is generated according to the schedule and the initial program specification (see also Section 2.3). Examples of such tools are Task Grapher (El-Rewini and Lewis [63]), Parallax (Lewis and El-Rewini [124]), Hypertool (Wu and Gajski [207]), OREGAMI (Lo et al. [130]), CASCH (Ahmad et al. [7]), Meander (Wirtz [201], Sinnen and Sousa [175], Trichina and Oinonen [190]), and PYRROS (Yang and Gerasoulis [209]).

While there are more aspects of task scheduling worth studying, this and the previous chapters covered the fundamentals of task scheduling, basic techniques, and a variety of algorithmic approaches. The next chapters are going to concentrate on an aspect that has been neglected in the literature until recently—the scheduling model. This chapter saw the generalization of the task scheduling model toward heterogeneous processors. This is a generalization of the scheduling model that makes it more realistic for some systems. However, there are other aspects of the scheduling model that do not accurately reflect real systems. As a consequence, schedules can be inaccurate and inefficient on real systems, even if they are optimal under the scheduling model. Therefore, Chapter 7 is going to address the awareness of contention for communication resources in the scheduling model and Chapter 8 will analyze the involvement of the processor in the communication.

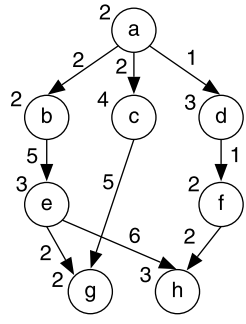
6.7 EXERCISES

6.1 In Figure 6.25(*b*) a partial schedule of the sample task graph (Figure 6.25(*a*)) is given. Use list scheduling with start time minimization and the *insertion technique* to schedule the remaining nodes in order c, d, g, i, h, k. Repeat this using the end technique and compare the results.

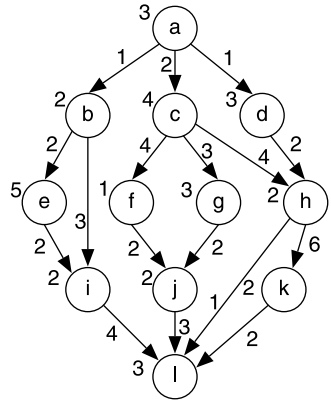
6.2 Schedule the following task graph on three processors using *node duplication*.

(a) Which nodes are the primary candidates for duplication?

(b) Which nodes might be also duplicated as a consequence?



6.3 Repeat Exercise 6.2 for the following task graph to be scheduled on four processors.



6.4 Schedule the sample task graph (e.g., in Figure 6.25) on three *heterogeneous* processors using list scheduling with *finish* time minimization. The heterogeneity of the processors is consistent and they have the following relative speed values: $s(P_1) = 1$, $s(P_2) = 1.5$, $s(P_3) = 2$. Let the node and edge weights in Figure 6.25 be the average values. Process the nodes in decreasing order of their bottom levels.

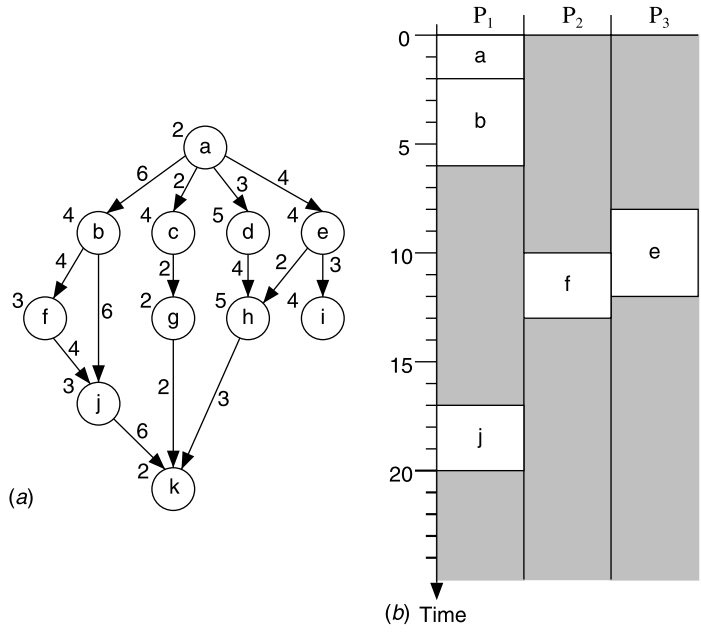


Figure 6.25. Partial schedule (*b*) of sample task graph (*a*).

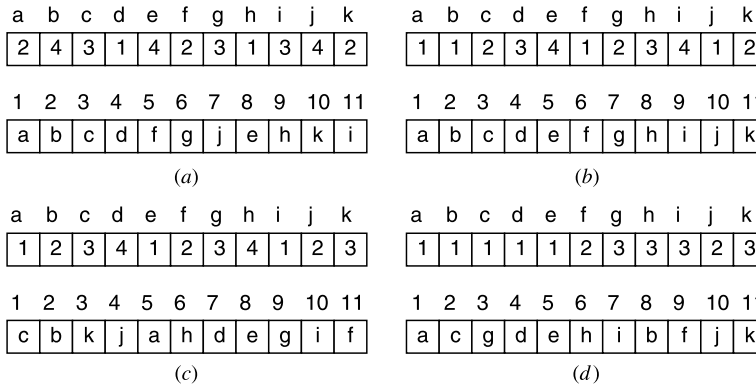


Figure 6.26. Four chromosomes for scheduling the sample task graph (e.g., Figure 6.25) on four processors.

6.5 A given genetic algorithm for task scheduling encodes in each chromosome the processor allocation and the node list (direct representation, Section 6.5.2). Construct the schedules represented by the four chromosomes of Figure 6.26 for scheduling the sample task graph (e.g., Figure 6.25) on four processors. It is not required that the node list is in precedence order; hence, the node order reflects the node priorities.

What is the fitness value of each chromosome (Definition 6.7)?

6.6 Genetic algorithms are very flexible. An example is the fact that the same chromosome encoding as used in Figure 6.26 can be employed for heterogeneous processors. Repeat Exercise 6.5 for four heterogeneous processors. The heterogeneity of the processors is consistent and they have the following relative speed values: $s(P_1) = 1$, $s(P_2) = 2$, $s(P_3) = 2$, $s(P_4) = 1$. Order the chromosomes according to their fitness values. Is this the same order as in Exercise 6.5?

6.7 Figure 6.27 depicts two schedules of Exercise 6.2's task graph on three processors. Specify two chromosomes in direct representation (as in Figure 6.26) that represent these schedules.

6.8 Apply a single-point crossover operator to the chromosomes of Figure 6.26. Perform the crossover between the pairs:

(a) (a,c) with crossover point $i = 6$.

(b) (b,d) with crossover point $i = 5$.

Construct the corresponding schedules of the resulting chromosomes.

6.9 Apply a simple mutation operator to the chromosomes of Figure 6.26. For the processor allocation part of the chromosome, randomly increase or decrease the processor index of one gene, modulo the number of processors $|\mathbf{P}|$. For the node list, swap two randomly determined values.

Construct the corresponding schedules of the resulting chromosomes.

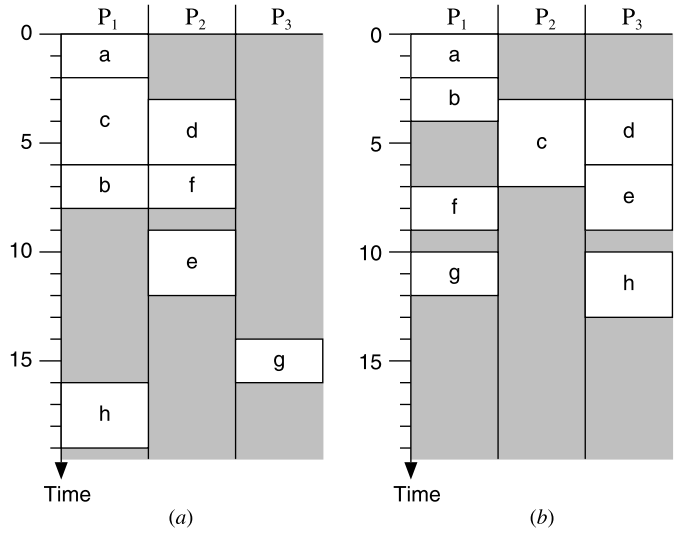


Figure 6.27. Two schedules of Exercise 6.2's task graph on three processors.

6.10 For genetic algorithms and node duplication:

(a) Why is it more difficult to find an efficient chromosome encoding for task scheduling when node duplication is allowed?

(b) Suggest an encoding and discuss appropriate operators.