CHAPTER 5

# Fundamental Heuristics

Equipped with the definitions and conditions established in the previous chapter, this chapter studies the two fundamental heuristics of task scheduling: *list scheduling* and *clustering*. These two heuristics are classes or categories rather than simple algorithms. Most of the algorithms that have been proposed for task scheduling fall into one of these two classes.

Both classes are discussed in general terms, following the expressed intention of this book to focus on common concepts and techniques. These are encountered in many scheduling algorithms from which they are extracted and treated separately. Nevertheless, the discussion is backed up with references to many proposed algorithms.

This chapter starts with list scheduling and a distinction is made between static and dynamic node priorities. Given a processor allocation, list scheduling can also be employed to construct a schedule, which is considered in the subsequent section. The area of clustering can be broken down into a few conceptually different approaches. Those are analyzed, followed by a discussion on how to go from clustering to scheduling.

## 5.1 LIST SCHEDULING

This section is devoted to the dominant heuristic technique encountered in scheduling algorithms, the so-called list scheduling. In its simplest form, the first part of list scheduling sorts the nodes of the task graph to be scheduled according to a priority scheme, while respecting the precedence constraints of the nodes—that is, the resulting node list is in topological order. In the second part, each node of the list is successively scheduled to a processor chosen for the node. Usually, the chosen processor is the one that allows the earliest start time of the node. Algorithm 9 outlines this simplest form of list scheduling.

List scheduling can be considered a heuristic skeleton. An algorithm applying the list scheduling technique has the freedom to define the two, so far unspecified, criteria: the priority scheme for the nodes and the choice criterion for the processor.

*Task Scheduling for Parallel Systems*, by Oliver Sinnen

***Algorithm 9 Simple List Scheduling—Static Priorities ($G = (\mathbf{V}, \mathbf{E}, w, c)$, $\mathbf{P}$)***

1: ▷ *1. Part:*
2: Sort nodes $n \in \mathbf{V}$ into list $L$, according to priority scheme and precedence constraints.
3: ▷ *2. Part:*
4: **for** each $n \in L$ **do**
5: Choose a processor $P \in \mathbf{P}$ for $n$.
6: Schedule $n$ on $P$.
7: **end for**

Section 5.1.3 studies various schemes for the attribution of priorities to nodes, which are mainly based on the task graph related concepts discussed in Section 4.4. Now the concepts behind list scheduling are discussed and a more generic list scheduling heuristic is developed.

***Partial Schedules*** List scheduling creates the final schedule by successively scheduling each node of the list onto a processor of the target system. Thus, each node $n$ is added to the current partial schedule, denoted by $\mathcal{S}_{\text{cur}}$, which consists of the nodes scheduled before $n$. As each node is only scheduled once, that is, the start time and the allocated processor are never changed in a later step of the algorithm, the partial schedules must be feasible in order to achieve a feasible final schedule.

***Free Nodes*** Scheduling a node means to allocate it to a processor and to set its start time. As elaborated in Section 4.1, in order to produce a feasible schedule the start time on the allocated processor must obey Condition 4.1 (Exclusive Processor Allocation) and Condition 4.2 (Precedence Constraint). However, Condition 4.2 can only be verified for a node $n$ if all predecessors—and in consequence all ancestors—of $n$ have already been scheduled: that is, they are part of the current partial schedule. A node for which this is true is called a *free node*.

**Definition 5.1 (Free Node)** *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 $\mathbf{P}$. A node $n \in \mathbf{V}$ is said to be free if $n \notin \mathbf{V}_{\text{cur}}$ and $\mathbf{ance}(n) \subset \mathbf{V}_{\text{cur}}$.*

An essential characteristic of list scheduling is that it guarantees the feasibility of all partial schedules and the final schedule by scheduling only free nodes and choosing an appropriate start time for each node. Every node to be scheduled is free, because the nodes are processed in precedence order (i.e., in topological order). Hence, by definition, at the time a node is scheduled all ancestor nodes have already been processed.

***End Technique*** With the requisite of node $n$ being free, the partial schedule remains feasible if the scheduling of $n$ complies with the two feasibility conditions. The most common way to determine an appropriate start time of $n$ is defined next.

**Definition 5.2 (End Technique)** *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 $\mathbf{P}$. The start time of the free node $n \in \mathbf{V}$, on a given processor $P$, is determined by*

$$t_s(n, P) = \max\{t_{\text{dr}}(n, P), t_f(P)\}. \tag{5.1}$$

This determination of the start time is here called "end technique," as node $n$ is scheduled at the end of all other nodes scheduled on processor $P$. In Section 6.1, another technique is presented—the insertion technique—which can compute earlier start times, yet with a higher complexity. Since $t_s(n, P) \geq t_{\text{dr}}(n, P)$, $t_s(n, P)$ complies with Condition 4.3—that is, it complies with Condition 4.2 for all predecessors of node $n$. Furthermore, as $t_s(n, P) \geq t_f(P)$ (here $t_f(P)$ is the finish time of $P$ in the partial schedule $\mathcal{S}_{\text{cur}}$), $t_s(n, P)$ cannot violate Condition 4.1 either.

An interesting property of the end technique is that it produces a schedule of optimal length if the nodes are scheduled in the "right" order on the "right" processor.

**Theorem 5.1 (End Technique Is Optimal)** *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), the schedule $\mathcal{S}_{\text{end}}$ is created employing the end technique (Definition 5.2), whereby the nodes are scheduled in nondecreasing order of their start times in $\mathcal{S}$ and allocated to the same processors as in $\mathcal{S}$. Then*

$$sl(\mathcal{S}_{\text{end}}) \leq sl(\mathcal{S}). \tag{5.2}$$

*Proof*. First, $\mathcal{S}_{\text{end}}$ is a feasible schedule, since the node order is a topological order according to Lemma 4.1. Without loss of generality, suppose that both schedules start at time unit 0. It then suffices to show that $t_{f,\mathcal{S}_{\text{end}}}(n) \leq t_{f,\mathcal{S}}(n) \,\forall\, n \in \mathbf{V}$ (see Definition 4.10 of schedule length).

Since the processor allocations are identical for both schedules, communications that are remote in one schedule are also remote in the other schedule 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 (i.e., through different start times) as the execution time is identical in both schedules.

By induction, it is shown now that $t_{s,\mathcal{S}_{\text{end}}}(n) \leq t_{s,\mathcal{S}}(n) \,\forall\, n \in \mathbf{V}$. Evidently, this is true for the first node to be scheduled, as it starts in both schedules at time unit 0. Now let $\mathcal{S}_{\text{end,cur}}$ be a partial schedule of the nodes of $\mathbf{V}_{\text{cur}}$ of $G$, for which $t_{s,\mathcal{S}_{\text{end,cur}}}(n) \leq t_{s,\mathcal{S}}(n) \,\forall\, n \in \mathbf{V}_{\text{cur}}$ 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 $t_{\text{dr},\mathcal{S}_{\text{end,cur}}}(n_i, P) \leq t_{\text{dr},\mathcal{S}}(n_i, P)$, 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 $\mathcal{S}$. Thus, $t_{s,\mathcal{S}_{\text{end,cur}}}(n_i, P) \leq t_{s,\mathcal{S}}(n_i, P)$ for node $n_i$. By induction this is true for all nodes of the schedule, in particular, the last node, which proves the theorem. □

This theorem is valid for any given schedule $\mathcal{S}$, in particular, for a schedule of optimal length $\mathcal{S}_{\text{opt}}$. In turn, this means that an optimal schedule for a given task graph and target system is defined by the processor allocation and the nodes' execution order. Given these two inputs, a schedule of optimal length can be constructed with list scheduling and the end technique. In other words, the scheduling problem reduces to allocating the nodes to the processors and to ordering the nodes. This is a very important result as it "simplifies" the scheduling problem. But of course, finding the optimal node order and allocation is still NP-hard.

### 5.1.1 Start Time Minimization

As mentioned at the beginning of this section, the most common processor choice for a node to be scheduled is the processor allowing the earliest start time of the node. This processor is found by simply computing the start time $t_s(n, P)$ for every $P \in \mathbf{P}$ according to Eq. (5.1) and selecting the processor for which the start time is minimal:

$$P_{\min} \in \mathbf{P} : t_s(n, P_{\min}) = \min_{P \in \mathbf{P}} \{\max\{t_{\text{dr}}(n, P), t_f(P)\}\}. \tag{5.3}$$

Algorithm 10 displays the procedure for this processor choice and the subsequent scheduling of the free node $n$ on $P_{\min}$. This procedure is an implementation of lines 5 and 6 of the simple list scheduling (Algorithm 9).

***Algorithm 10 Start Time Minimization: Schedule Free Node n on Earliest-Start-Time Processor***

**Require:** $n$ is a free node
  $t_{\min} \leftarrow \infty$; $P_{\min} \leftarrow NULL$
  **for** each $P \in \mathbf{P}$ **do**
    **if** $t_{\min} > \max\{t_{\text{dr}}(n, P), t_f(P)\}$ **then**
      $t_{\min} \leftarrow \max\{t_{\text{dr}}(n, P), t_f(P)\}$; $P_{\min} \leftarrow P$
    **end if**
  **end for**
  $t_s(n) \leftarrow t_{\min}$; $proc(n) \leftarrow P_{\min}$

In the literature, list scheduling usually implies the above start time minimization method. Many algorithms have been proposed for this kind of list scheduling (Adam et al. [2], Coffman and Graham [38], Graham [80], Hu [93], Kasahara and Nartia [102], Lee et al. [117], Liu et al. [128], Wu and Gajski [207], Yang and Gerasoulis [210]). Some of these publications, especially the earlier ones, are based on restricted scheduling problems as discussed in Section 6.4.

***An Example*** To illustrate, simple list scheduling with start time minimization is applied to the sample task graph (e.g., in Figure 4.1) and three processors. Suppose the first part of list scheduling established the node order $a, b, c, d, e, f, g, i, h, j, k$. Figure 5.1 visualizes the scheduling process described in the following.

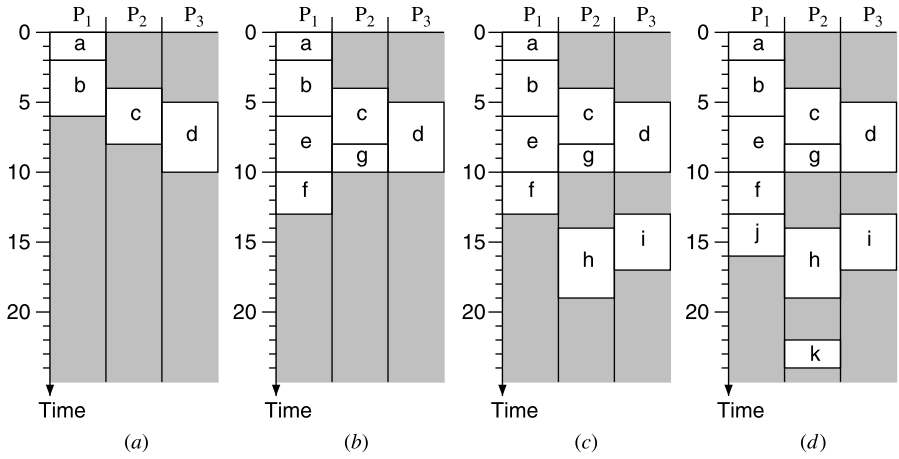



**Figure 5.1.** Example of simple list scheduling with start time minimization: (*a*) to (*c*) are snapshots of partial schedules; (*d*) shows the final schedule. The node order of the sample task graph (e.g., in Figure 4.1) is $a, b, c, d, e, f, g, i, h, j, k$.

It is irrelevant to which processor node $a$ is scheduled and $P_1$ is chosen. The next node, $b$, is also scheduled on $P_1$, as local communication between $a$ and $b$ permits node $b$'s earliest DRT on $P_1$ and thus its earliest start time. Nodes $c$ and $d$ are best scheduled on processors $P_2$ and $P_3$, despite the remote communication with $a$ on these processors; on $P_1$ they had to wait until $b$ finishes, which is later than their start times on $P_2$ and $P_3$. The partial schedule at this point is shown in Figure 5.1($a$). Node $e$ is now best scheduled on $P_1$, due to $P_1$'s early finish time. Node $f$'s start time is identical on all processors and $P_1$ is chosen. Next, node $g$ is scheduled on $P_2$, since its predecessor node $c$ is scheduled on this processor (Figure 5.1($b$)). For node $i$, all processors allow the same earliest start time and the node is scheduled on $P_3$. Node $h$ depends on nodes $d$ and $e$, whereby node $d$ sends the larger message. This determines the earliest start time on the two processors $P_1$ and $P_2$, of which $P_2$ is chosen (Figure 5.1($c$)). The next node $j$ is best scheduled on processor $P_1$ and node $k$'s start must be delayed on every processor in the wait for communications from nodes $h$ and $j$. The final schedule, with $k$ on $P_2$, is depicted in Figure 5.1($d$).

***Complexity*** The complexity of the simple list scheduling can be broken down into the complexity of the first and the second part (Algorithm 9). As for the first part, its complexity is analyzed in Section 5.1.3, since it depends on the employed priority scheme. The complexity of the second part depends on the way in which a processor is chosen for a node. With start time minimization (Algorithm 10) the complexity is as follows. To calculate the start time of a node $n$ according to Eq. (5.1), the data ready time $t_{\text{dr}}(n)$ is computed, which involves one calculation for every predecessor of $n$. For all nodes, calculating the DRT amortizes to $O(\mathbf{E})$, as the sum of the predecessors of all nodes is $|\mathbf{E}|$. Since this is done for each processor, the total complexity of the DRT calculation is $O(\mathbf{PE})$. The start time of every node is computed on every processor, that

is, $O(\mathbf{PV})$ times; hence, the total complexity is $O(\mathbf{P}(\mathbf{V}+\mathbf{E}))$. It is possible to achieve a slightly lower complexity for list scheduling with static priorities, as analyzed by Radulescu and Gemund [157].

List scheduling with start time minimization belongs to the class of greedy algorithms (Cormen et al. [42]). At each step, the heuristic tries to create a new partial schedule of short length, with the conjecture that this will eventually result in a short final schedule. A mistake regarding communication made in an early step cannot be remedied later. Graham [80] shows that, for task graphs without communication costs, the worst-case length of a schedule produced by list scheduling with start time minimization is twice the optimal length. For task graphs with communication costs, however, no such guarantee on the schedule length exists.

**Theorem 5.2 (No List Schedule Bound)** *For any list scheduling algorithm with start time minimization, no constant $C \in \mathbb{Q}^+$ exists such that*

$$sl(\mathcal{S}) \leq C \times sl(\mathcal{S}_{\text{opt}}) \tag{5.4}$$

*is true for every task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ on every parallel system $\mathbf{P}$. $\mathcal{S}$ is the schedule produced with the list scheduling and $\mathcal{S}_{\text{opt}}$ is a schedule of optimal length.*

*Proof*. The theorem is proved by showing that a task graph can be constructed for any constant $C$, for which the list schedule length is $sl(\mathcal{S}) > C \times sl(\mathcal{S}_{\text{opt}})$. Consider the task graph in Figure 5.2($a$) with $k \in \mathbb{Q}^+$. For $k > 1$, an optimal schedule on two processors, of length 3, is visualized in Figure 5.2($b$); if the nodes are not scheduled all on one processor, at least one communication is sent across the network resulting in $sl(\mathcal{S}) \geq 3k$, for example, as illustrated in Figure 5.2($c$). For list scheduling there are two possible node orders: $A, B, C$ or $B, A, C$. However, both orders result in a schedule where $A$ and $B$ are scheduled on different processors, due to the start time minimization. Independent of which processor node $C$ is scheduled on, the schedule

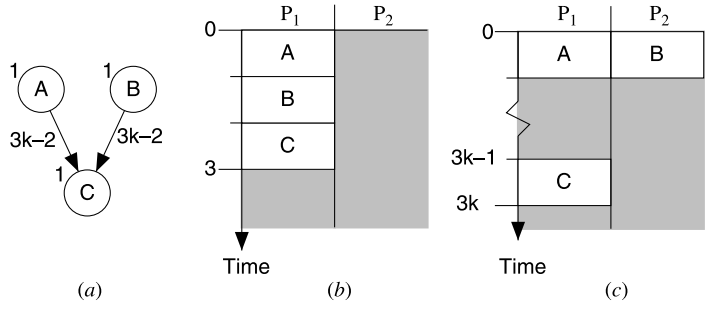



**Figure 5.2.** ($a$) A task graph; ($b$) its optimal schedule ($k > 1$); and ($c$) a list schedule.

length is $3k$ (Figure 5.2($c$)). For any $C$ a $k$ with $k > 1$, $k > C$ can be chosen so that $sl(\mathcal{S}) = 3k > 3C = C \times sl(\mathcal{S}_{\text{opt}})$. □

In particular, list scheduling with start time minimization does not even guarantee a schedule length shorter than the sequential time of a task graph. Still, numerous experiments have demonstrated that list scheduling produces good schedules for task graphs with low to medium communication (e.g., Khan et al. [103], Kwok and Ahmad [111], Sinnen and Sousa [177]).

### 5.1.2 With Dynamic Priorities

The simple list scheduling outlined in Algorithm 9 establishes the schedule order of the nodes before the actual scheduling process. During the node scheduling in the second part, this order remains unaltered, so the node priorities are *static*. To achieve better schedules, it might be beneficial to consider the state of the partial schedule in the schedule order of the nodes. Changing the order of the nodes is equivalent to changing their priorities, hence the node priorities are *dynamic*. It remains that the node order must be compatible with the precedence constraints of the task graph, which is obeyed if only free nodes are scheduled. A list scheduling algorithm for dynamic node priorities is given in Algorithm 11.

***Algorithm 11 List Scheduling—Dynamic Priorities ($G = (\mathbf{V}, \mathbf{E}, w, c)$, P)***

1: Put source nodes $\{n_i \in \mathbf{V} : \mathbf{pred}(n_i) = \emptyset\}$ into set $S$.
2: **while** $S \neq \emptyset$ **do** ▷ *$S$ only contains free nodes*
3: Calculate priorities for nodes $n_i \in S$.
4: Choose a node $n$ from $S$; $S \leftarrow S - n$.
5: Choose a processor $P$ for $n$.
6: Schedule $n$ on $P$.
7: $S \leftarrow S \cup \{n_i \in \mathbf{succ}(n) : \mathbf{pred}(n_i) \subseteq \mathcal{S}_{\mathbf{cur}}\}$
8: **end while**

Instead of iterating over the node list, Algorithm 11 calculates in each step the priority of all free nodes and only then selects a node and a processor for scheduling. After the scheduling of node $n$, the set $S$ of free nodes is updated with the nodes that became free by the scheduling of $n$ (line 7). Obviously, this approach has higher complexity in general than the simple list scheduling presented in Algorithm 9, because the priority of every node is determined several times. Algorithm 11 is a generalization of the simple list scheduling, which becomes apparent by considering line 3. If this line is moved to the beginning of the algorithm and the priorities are calculated for all nodes, not only for the free nodes, the simple list scheduling is obtained. This becomes even more distinct in Section 5.1.3, where the construction of node lists is explained.

***Choosing Node–Processor Pair*** Some algorithms dynamically compute the priorities of the free nodes by evaluating the start time (see Eq. (5.1)) for every free

node on every processor. That node of all free nodes that allows the earliest start time is selected together with the processor on which this time is achieved. The choice of the node and the processor is made simultaneously, based on the state of the partial schedule. Two examples for dynamic list scheduling algorithms employing this technique are ETF (earliest time first) by Hwang et al. [94] and DLS (dynamic level scheduling) by Sih and Lee [169].

In comparison to the simple list scheduling, the complexity increases by a factor of $|\mathbf{V}|$. Evaluating the start time of every free node on every processor is in the worst case $O(\mathbf{P}(\mathbf{V}+\mathbf{E}))$, which is already the complexity of the simple list scheduling. As this step is done $|\mathbf{V}|$ times, the total complexity is $O(\mathbf{P}(\mathbf{V}^2+\mathbf{VE}))$.

### 5.1.3 Node Priorities

Discussion of the list scheduling technique distinguished between static and dynamic node priorities. How these priority schemes are realized is discussed in the next paragraphs.

***Static*** Static priorities are based on the characteristics of the task graph. In the simplest case, the priority metric itself establishes a precedence order among the nodes. It is then sufficient to order the nodes according to their priorities with any sort algorithm, for example, with Mergesort (Cormen et al. [42]), which has a complexity of $O(\mathbf{V}\log\mathbf{V})$. Prominent examples for such priority metrics are the top and the bottom level of a node (Section 4.4.2).

**Lemma 5.1 (Level Orders Are Precedence Orders)** *Let $G=(\mathbf{V},\mathbf{E},w,c)$ be a task graph. The nonincreasing bottom level (bl) order and the nondecreasing top level (tl) order of the nodes $n\in\mathbf{V}$ are precedence orders.*

*Proof*. The lemma follows immediately from the recursive definition of the bottom level and top level in Eqs. (4.43) and (4.44), respectively. A node always has a smaller bottom level than any of its predecessors and a higher top level than any of its successors. □

If the priority metric is not compatible with the precedence constraints of the task graph, a correct order can be established by employing the free node concept. Algorithm 12 shows how to create a priority ordered node list that complies with the precedence constraints. As mentioned before, the general structure is similar to Algorithm 11, but a priority queue ($Q$) is used instead of a set ($S$), because the priorities assigned to the nodes are static.

The simple list scheduling outlined in Algorithm 9 could be rewritten by using the below algorithm and substituting line 5 with lines 5 and 6 from Algorithm 9. However, then the conceptually nice separation of the first and the second part of simple list scheduling is lost.

Suppose the priority queue in Algorithm 12 is a heap (Cormen et al. [42]); the algorithm's complexity is $O(\mathbf{V}\log\mathbf{V}+\mathbf{E})$: every node is inserted once in the heap,

***Algorithm 12 Create Node List***

1: Assign a priority to each $n \in \mathbf{V}$.
2: Put source nodes $n \in \mathbf{V} : \mathbf{pred}(n) = \emptyset$ into priority queue $Q$.
3: **while** $Q \neq \emptyset$ **do** ▷ *$Q$ only contains free nodes*
4: Let $n$ be the node of $Q$ with highest priority; remove $n$ from $Q$.
5: Append $n$ to list $L$.
6: Put $\{n_i \in \mathbf{succ}(n) : \mathbf{pred}(n_i) \subseteq L\}$ into $Q$.
7: **end while**

which costs $O(\log \mathbf{V})$ and the determination of the free nodes amortizes to $O(\mathbf{E})$. Removing the node with the highest priority from the heap is $O(1)$.

An example for a metric that is not compatible with the precedence order is $tl + bl$—a metric that gives preference to the nodes of the critical path. Table 5.1 displays several node orders of the sample task graph obtained with node level based priority metrics. The values of the node levels are given in Table 4.1. As analyzed in Section 4.4, calculating these levels has a complexity of $O(\mathbf{V} + \mathbf{E})$. For all these metrics the first part of the simple list scheduling is thus $O(\mathbf{V} \log \mathbf{V} + \mathbf{E})$ (calculating levels: $O(\mathbf{V} + \mathbf{E})$ + sorting: $O(\mathbf{V} \log \mathbf{V})$ or $O(\mathbf{V} \log \mathbf{V} + \mathbf{E})$).

Already the few examples in Table 5.1 illustrate the large variety of possible node orders. In experiments, the relevance of the node order for the schedule length was demonstrated (e.g., Adam et al. [2], Sinnen and Sousa [177]). Unfortunately, most comparisons of scheduling algorithms (e.g., Ahmad et al. [6], Gerasoulis and Yang [76], Khan et al. [103], Kwok and Ahmad [111], McCreary et al. [136]) analyze entire algorithms: that is, the algorithms differ not only in one but many aspects. This generally impedes one from drawing conclusions on discrete aspects of the algorithms (e.g., on the node order). Still, most algorithms apply orders based on node levels and the bottom level order exhibits good average performance (e.g., Adam et al. [2], Sinnen and Sousa [177]).

*Breaking Ties* Table 5.1 also shows examples where the node order is not fully defined, indicated in the table by parentheses around the respective nodes. Incidentally,

**Table 5.1. Common Priority Metrics and the Corresponding Precedence Compatible Orders of the Nodes of the Sample Task Graph[a]**

| Priority Metric | Node Order |
|---|---|
| $tl$ ($ASAP$) | $a, c, d, e, b, g, i, h, f, j, k$ |
| $bl$ ($ALAP$) | $a, b, d, f, e, c, j, h, g, i, k$ |
| $tl + bl$ | $a, b, f, j, d, e, h, i, c, g, k$ |
| $tl_w$ ($ASAP_w$) | $a, (b, c, d, e), (f, g, i), h, j, k$ |
| $bl_w$ ($ALAP_w$) | $a, (b, d), e, (c, f), h, j, (g, i), k$ |
| $tl_w + bl_w$ | $a, ([b, f, j], d), e, h, ([c, g], i), k$ |

[a]For example, in Figure 4.1. Node levels are given in Table 4.1; order of nodes within parentheses is not defined.

only computation levels are affected here, but this is a general problem that can occur for most level based priority metrics, especially in regular task graphs or for integer task graph weights. Scheduling algorithms therefore use additional criteria to decide the node order. For example, in a bottom level based order, such a tie breaker can be the top level of the nodes. But often ties are broken randomly.

*Example Priority Schemes* Given the enormous number of proposed scheduling algorithms, the following list of priority schemes represents only a small, though important part, of the existing ones. In the HLF (highest level first) (Hu [93]) and HLFET (highest level first with estimated times) (Adem et al. [2]) algorithms, the nodes are ordered according to the computation bottom level $bl_w$. The CP/MISF (critical path/most immediate successors first) algorithm by Kasahara and Nartia [102] is also $bl_w$ based but breaks the ties by giving precedence to the node with the higher number of successors. Wu and Gajski [207] propose the MCP (modified critical path) heuristic, where the nodes are ordered by their $bl$ and for nodes with equal $bl$, the $bl$ values of the successors are considered and so on. A critical path based order is proposed by Ahmad and Kwok [4, 114], where non-CP nodes are scheduled according to their $bl$ and ties are broken with $tl$. Sinnen and Sousa analyze [177] various priority schemes of two different types: (1) node orders according to the (computation) bottom level augmented with metrics based on the entering communication of a node; and (2) critical path based orders as proposed by Ahmad and Kwok [4, 115].

***Dynamic*** Level or critical path based priority schemes turn into dynamic priorities when they are recalculated in each scheduling step based on the current partial schedule. Section 4.4 explained how the allocated path length is calculated given the (partial) processor allocation of the nodes. For instance, the allocated critical path might be recalculated for each partial schedule.

It should be noted that the allocated bottom level $bl(n, \mathcal{S}_{\text{cur}})$ of a node $n$ remains unaltered in list scheduling until $n$ is scheduled; that is, $bl(n, \mathcal{S}_{\text{cur}}) = bl(n)$—the bottom level of $n$ depends only on its descendants (see Eq. (4.43)).

In general, recalculating the levels or the CP in each step of scheduling multiplies the costs for the level or CP determination by a factor of $|\mathbf{V}|$. However, an efficient implementation might only update the levels for those nodes that are affected by the scheduling of a node.

*Start Times* Apart from node orders based on task graph characteristics, priority schemes often use the potential start time of free nodes as a preference criterion. The node priority thereby becomes a function of the processor. Choosing the node–processor pair as discussed in Section 5.1.2 is such a priority scheme. Another technique selects the free node in each step that has the earliest DRT, assuming all communications are remote, sometimes called *ready node* (Yang and Gerasoulis [210]).

Dynamic priorities are often a mixture of task graph based metrics and the earliest start time, whereby the task graph characteristics are only quantified at the beginning of the algorithm and remain unmodified. For instance, the DLS algorithm (Sih and Lee [169]) defines its dynamic level using $bl_w$ and the node's earliest start time on all processors. The ETF algorithm by Hwang et al. [94] also selects the node–processor pair with the earliest start time, breaking ties with $bl_w$.

## 5.2 SCHEDULING WITH GIVEN PROCESSOR ALLOCATION

At the definition of a schedule (Definition 4.2) in Section 4.2, it was stated that scheduling is the spatial and temporal assignment of the tasks to the processors. List scheduling does this assignment in an integrated process, that is, at the same time a node is assigned to a processor and attributed a start time. In fact, when referring to scheduling, normally both the spatial and temporal assignments are meant. Yet, it is a natural idea to split this process into its two fundamental aspects: processor allocation and attribution of start times.

When doing so, there are two alternatives: (1) first processor allocation, then attribution of start times; and (2) first attribution of start times, then processor allocation. For a limited number of processors, the second alternative is extremely difficult, because nodes with overlapping execution times have to be executed on different processors. Furthermore, without knowledge of the processor allocation, it is not known which communications will be local and which remote. Hence, adhering to the processor and precedence constraints (Conditions 4.1 and 4.2) would be very difficult. As a consequence, the two-phase scheduling starts with the allocation (or mapping) of the nodes to the processors. Based on these considerations, the generic structure of a two-phase heuristic is shown in Algorithm 13.

***Algorithm 13 Generic Two-Phase Scheduling Algorithm*** **($G = (\mathbf{V}, \mathbf{E}, w, c)$, $\mathbf{P}$)**

▷ *Processor allocation/mapping*

(1) Assign each node $n \in \mathbf{V}$ to a processor $P \in \mathbf{P}$

▷ *Scheduling/ordering nodes*

(2) Attribute a start time $t_s(n)$ to each node $n \in \mathbf{V}$, adhering to Condition 4.1 (processor constraint) and Condition 4.2 (precedence constraint)

Next it is analyzed how list scheduling can be utilized to handle the second phase of this algorithm. The processor allocation (i.e., the first phase) can be determined in many different ways. One possibility is to extract it from another schedule, as is done, for example, in the proof of Theorem 5.1. This is especially interesting when a more sophisticated scheduling model is used in the second phase, like those discussed in Chapters 7 and 8. With such an approach, the scheduling in the first phase can be considered an estimate, which is refined in the second phase. A genetic algorithm based heuristic for finding a processor allocation is referenced in Section 8.4.2. More on processor allocation heuristics can be found in El-Rewini et al. [65] and Rayward-Smith et al. [159].

### 5.2.1 Phase Two

The second phase of this generic two-phase scheduling algorithm can easily be performed by a list scheduling heuristic. In each iteration of (the second part of) list scheduling (i.e., for each node $n \in \mathbf{V}$), the heuristic first chooses a processor $P$ on which to schedule $n$. This is done in line 5 of both static list scheduling (Algorithm 9) and dynamic list scheduling (Algorithm 11). When the processor allocation $\mathcal{A}$ is already given, this line can simply be omitted, as $proc(n)$ is already determined for all $n \in \mathbf{V}$.

It suffices to schedule each node at the earliest possible start time,

$$t_s(n, proc(n)) = \max\{t_{\mathrm{dr}}(n, proc(n)), t_f(proc(n))\}, \tag{5.5}$$

hence using the end technique. Thus, Eq. (5.5) is an implementation of lines 5 and 6 of the simple list scheduling (Algorithm 9) or the dynamic list scheduling (Algorithm 11). As the selection of the processor is not performed anymore in these algorithms, the complexity of the second part of list scheduling reduces by the factor $|\mathbf{P}|$. The second part of simple list scheduling is then $O(\mathbf{V} + \mathbf{E})$.

From Theorem 5.1 it is known that the end technique is optimal for a given node order and processor allocation. As a result, scheduling with a given processor allocation reduces to finding the best node order.

Both static and dynamic priorities can be employed to order the nodes for their scheduling. However, since the processor allocations are already determined, task graph characteristics, like node levels and the critical path, can be computed using the allocated path length (Section 4.4), that is, the path length based on the known processor allocations. These characteristics do not change during the entire scheduling; hence, dynamic priorities are only sensible when considering the state of the partial schedules, for example, choosing the node among the free nodes that can start earliest, that is, the ready node (Section 5.1.3).

One might wonder whether this scheduling problem with a given preallocation is still NP-hard. After all, it is "only" about finding the best node order. Unfortunately it is still NP-hard, even for task graphs without communication costs, unit execution time, and very simple graph structures, such as forest (Goyal [79]) or chains (Rayward-Smith et al. [159]); see also Hoogeveen et al. [91].

## 5.3 CLUSTERING

As mentioned before in Section 4.2.2, task scheduling under the classic model is a trade-off between minimizing interprocessor communication costs and maximizing the concurrency of the task execution. A natural idea is therefore to determine first—before the actual scheduling—which nodes should always be executed on the same processor. Obvious candidates for grouping are nodes that depend on each other, especially nodes of the critical path.

Clustering is a technique that follows this idea. It is therefore only suitable for scheduling with communication costs. In its core it is a scheduling technique

for an unlimited number of processors. Nevertheless, it is often proposed as an initial step in scheduling for a limited number of processors. To distinguish between the limited number of (physical) processors and the unlimited number of (virtual) processors assumed in the clustering step, the latter are called clusters, hence the term "clustering." In clustering, the nodes of the task graph are mapped *and* scheduled into these clusters.

**Definition 5.3 (Clustering)** *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph. A clustering $\mathcal{C}$ is a schedule of $G$ on an implicit parallel system $\mathbf{C}$ (Definition 4.3) with an "unlimited" number of processors; that is, $|\mathbf{C}| = |\mathbf{V}|$. The processors $C \in \mathbf{C}$ are called clusters.*

Clustering based scheduling algorithms for a limited number of processors consist of several steps, similar to the two-phase scheduling outlined by Algorithm 13 in Section 5.2. Algorithm 13 comprises two phases, where the first phase is the mapping of the nodes and the second is their scheduling. In clustering based algorithms, three steps are necessary: (1) clustering, (2) mapping of the clusters to the (physical) processors, and (3) scheduling of the nodes. Algorithm 14 outlines a generic three-step clustering based scheduling heuristic for a limited number of processors.

***Algorithm 14 Generic Three-Step Clustering Based Scheduling Algorithm ($G = (\mathbf{V}, \mathbf{E}, w, c)$, $\mathbf{P}$)***

$\triangleright$ *Clustering nodes*

(1) Find a clustering $\mathcal{C}$ of $G$

$\triangleright$ *Mapping clusters to processors*

(2) Assign clusters of $\mathbf{C}$ to (physical) processors of $\mathbf{P}$

$\triangleright$ *Scheduling/ordering nodes*

(3) Attribute start time $t_s(n)$ to each node $n \in \mathbf{V}$, adhering to Condition 4.1 (processor constraint) and Condition 4.2 (precedence constraint)

In contrast to the two-phase Algorithm 13, where the first step is the pure processor allocation of the nodes, clustering also includes the scheduling of the nodes in the clusters. As will be seen later, this is done for an accurate estimation of the execution time of the task graph. It also makes clustering a complete scheduling algorithm for an unlimited number of processors.

While the third step is theoretically identical to the second phase of Algorithm 13, its actual implementation may differ, since the partial node orders as established by the clustering $\mathcal{C}$ might be considered in determining the final node order.

In terms of parallel programming terminology, clustering correlates to the step of parallelization designated by orchestration (Culler and Singh [48]) or agglomeration (Foster [69]) as described in Section 2.3.

The often cited motivation for clustering was given by Sarkar [167]: if tasks are best executed in the same processor (cluster) of an ideal system, that is, a system that possesses more processors (clusters) than tasks, they should also be executed on the same processor in any real system. Due to the NP-hardness of scheduling, it cannot be

expected that the foregoing conjecture is always true, yet it is an intuitive reasoning for a heuristic.

In the following, the first step of Algorithm 14, the actual clustering, is studied. Steps 2 and 3 are treated in Section 5.4.

### 5.3.1 Clustering Algorithms

Strictly speaking, the first step of Algorithm 14 can be performed by any scheduling algorithm suitable for an unlimited number of processors. However, in the literature the term clustering designates a certain kind of algorithm, with several characteristic aspects. The following discussion is based on Darte et al. [52], El-Rewini et al. [65], and Gerasoulis and Yang [76].

***Principle of Clustering Algorithms*** Clustering algorithms start with an *initial clustering* $\mathcal{C}_0$ of the task graph $G$. Usually each node $n \in \mathbf{V}$ is allocated to a distinct cluster $C \in \mathbf{C}$.

The clustering algorithm then performs *incremental steps of refinement* going from clustering $\mathcal{C}_{i-1}$ to clustering $\mathcal{C}_i$, in which clusters are *merged*. This means that the nodes of these clusters are merged into one single cluster. If communicating nodes are executed in the same cluster, their communication becomes local and hence its cost is zero according to the target system model (Definition 4.3). This can be beneficial for the total execution time of the graph, as it eliminates communication costs. Normally, a merging of clusters is performed only if the schedule length of the new clustering $\mathcal{C}_i$ decreases or at least remains the same compared to the schedule length of the current clustering $\mathcal{C}_{i-1}$. The steps of refinement are performed until all candidates for merging have been considered. Algorithm 15 summarizes this principle of clustering algorithms.

***Algorithm 15 Principle of Clustering Algorithms*** ($G = (\mathbf{V}, \mathbf{E}, w, c)$)

Create initial clustering $\mathcal{C}_0$: allocate each node $n \in \mathbf{V}$ to a distinct cluster $C \in \mathbf{C}$, $|\mathbf{C}| = |\mathbf{V}|$
$i \leftarrow 0$
**repeat**
  $i \leftarrow i + 1$
  Select candidate clusters for merging
  Create new clustering $\mathcal{C}_i$: merge candidate clusters into one cluster
  **if** $sl(\mathcal{C}_i) > sl(\mathcal{C}_{i-1})$ **then** ▷ *merging increases current schedule length*
    $\mathcal{C}_i \leftarrow \mathcal{C}_{i-1}$ ▷ *reject new clustering* $\mathcal{C}_i$
  **end if**
**until** all candidates for merging have been considered

Figure 5.3 shows some example clusterings of a simple task graph (Figure 5.3($a$)), which has often been used to illustrate clustering (Gerasoulis and Yang [76]). An initial clustering is depicted in Figure 5.3($b$). Each node is allocated to a distinct cluster,

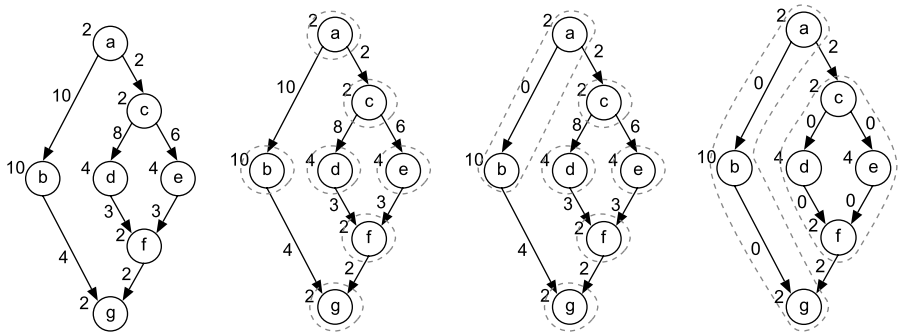



**Figure 5.3.** Clusterings of a simple task graph: (*a*) simple task graph for clustering algorithms (Gerasoulis and Yang [76]); (*b*) initial clustering; (*c*) clustering after clusters of nodes *a* and *b* have been merged; (*d*) clustering with only two clusters.

symbolized by the gray dashed line enclosing each node. Figure 5.3(*c*) depicts the clustering, where the clusters of nodes *a* and *b* of the initial clustering have been merged into one cluster. This might be the clustering produced by an algorithm after the first refinement step. Lastly, Figure 5.3(*d*) illustrates a clustering with only two clusters.

***Implicit Schedule*** An essential characteristic of clustering algorithms is that the clustering obtained at each step is a feasible schedule. Otherwise it would not be possible to determine the current schedule length, on which clustering decisions are based.

In clustering algorithms, the scheduling of the nodes is often given implicitly by the allocation of the nodes to the clusters. The first example for this is the initial clustering, where each node starts as-soon-as-possible, that is, at its top level (Section 4.4.2; see also Section 4.3.2):

$$t_s(n) = tl(n) \quad \forall n \in \mathbf{V}. \tag{5.6}$$

The resulting schedule is shown in Figure 5.4(*a*), which is the implicit schedule of the clustering in Figure 5.3(*b*).

In general, the starting times of the nodes (i.e., their scheduling) are implicitly given whenever the node order within each cluster is well defined. This is the case when there are no independent nodes in the same cluster. If this criterion is met, the start time of each node in clustering $\mathcal{C}_i$ is its *allocated* top level (Section 4.4):

$$t_s(n) = tl(n, \mathcal{C}_i) \quad \forall n \in \mathbf{V}. \tag{5.7}$$

Figure 5.4(*b*) displays the implicit schedule for the clustering of Figure 5.3(*c*). It is not clear at this point what the implicit schedule of the clustering shown in

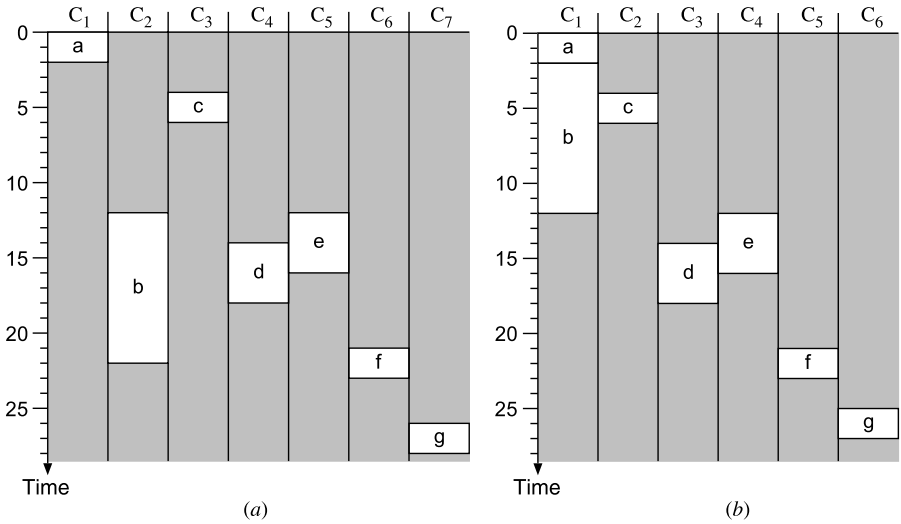



**Figure 5.4.** Implicit schedules of (*a*) the initial clustering as shown in Figure 5.3(*b*) and (*b*) the clustering in Figure 5.3(*c*).

Figure 5.3(*d*) is, since nodes *d* and *e* are independent and in the same cluster. A clustering algorithm must establish a rule on how to order (i.e., schedule) such nodes.

In order to keep the complexity of clustering algorithms as low as possible, these algorithms try to be smart about the scheduling of the nodes in each incremental step. Actually, it is not necessary to generate a schedule at each step; it is merely necessary to determine the current schedule length. For this reason, the term *estimated parallel time* (EPT) was introduced for clustering algorithms (Gerasoulis and Yang [76]), which simply is the length of the (implicit) schedule of the present clustering. The aim of the algorithms is to obtain the EPT with as little effort as possible. Different clustering algorithms pursue different strategies and techniques for the ordering (i.e., the scheduling) of the nodes, as will be seen later.

***Edge Zeroing*** It is important to realize that merging two clusters $C_x$ and $C_y$ can only be beneficial if there is at least one edge $e_{ij}$ going from a node $n_i$ in cluster $C_x$ to a node $n_j$ in cluster $C_y$ (or vice versa). Such a merging will turn the remote communication associated with $e_{ij}$ into a local communication. For this reason, the merging of the clusters is also called *edge zeroing*, as the communication costs of the corresponding edges become zero. Merging clusters that do not result in edge zeroing cannot improve the schedule length (i.e., the EPT) of the clustering.

The attentive reader might have observed that edge zeroing is already reflected in Figure 5.3. The edge weights are 0, whenever the incident nodes are in the same cluster. For example, in Figure 5.3(*c*) the edge weight $c(e_{ab})$ is zero, since both *a* and *b* are in the same cluster, while it is $c(e_{ab}) = 10$ in the initial clustering of Figure 5.3(*b*). In Figure 5.3(*d*) most edges are zero, because there are only two clusters resulting in mainly local communication.

As a side comment, note that clustering is meaningless for scheduling without communication costs (Section 4.3) for the above described reasons. In fact, without communication costs, allocating each node to a distinct cluster (processor) results already in an optimal schedule (Theorem 4.4).

***Nonbacktracking*** Clustering algorithms are nonbacktracking algorithms. In other words, clusters that have been merged in some step of the algorithm are never unmerged at a later time. This approach simplifies the clustering heuristics and significantly reduces their complexity.

***Multiplicity of Edge Zeroing*** One of the main points of distinction between clustering algorithms is the multiplicity of edge zeroing. As was mentioned earlier, merging clusters corresponds to zeroing edges. Clustering algorithms differ in how many edges are zeroed in a single step of the clustering. Three types can be distinguished:

1. *Path.* All edges of a path in the task graph are zeroed when beneficial.
2. *Edge.* One single edge is zeroed when beneficial.
3. *Node.* One or more incident edges of a single node are zeroed when beneficial.

The above distinction only applies to the edges that are the primary candidates for edge zeroing. In a heuristic, the zeroing of an edge can trigger the implicit zeroing of other edges that are incident on the same node. Heuristics based on each of these three types of multiplicity will be studied next.

### 5.3.2 Linear Clustering

Linear clustering is a special class of clustering. In linear clustering only dependent nodes are grouped into one cluster. If two nodes in one cluster are independent, this cluster, and with it the entire clustering, is nonlinear.

**Definition 5.4 (Linear Cluster and Clustering)** *Let $C \in \mathbf{C}$ be a cluster of a clustering $\mathcal{C}$ of task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$. Let $n_1, n_2, \ldots, n_m$ be all the nodes allocated to $C$, that is, $\{n_1, n_2, \ldots, n_m\} = \{n \in \mathbf{V} : proc(n) = C\}$, arranged in topological order. The cluster is linear if and only if there is a path $p(n_1 \rightarrow n_m)$ from $n_1$ to $n_m$ to which all nodes allocated to $C$ belong, that is, $\{n_1, n_2, \ldots, n_m\} \subseteq \{n \in \mathbf{V} : n \in p(n_1 \rightarrow n_m)\}$. A clustering $\mathcal{C}$ is said to be* linear *if and only if all of its clusters $\mathbf{C}$ are linear; otherwise it is* nonlinear.

For example, in Figure 5.3 the clusterings (*b*) and (*c*) are linear, while the clustering of (*d*) is nonlinear (*d* and *e* are independent and in the same cluster). In general, initial clusterings with one node per cluster are always linear.

From the considerations of implicit schedules, it is clear that linear clusterings have well-defined implicit schedules. In any linear clustering $\mathcal{C}_{\text{linear}}$ of a task graph

$G = (\mathbf{V}, \mathbf{E}, w, c)$, each node starts execution at its allocated top level, that is, at the earliest possible time, according to Eq. (5.7). As a result, the schedule length of $\mathcal{C}_{\text{linear}}$ is the *allocated* length of the *allocated* critical path $cp(\mathcal{C}_{\text{linear}})$,

$$\begin{aligned} sl(\mathcal{C}_{\text{linear}}) &= len(cp(\mathcal{C}_{\text{linear}}), \mathcal{C}_{\text{linear}}) \\ &= tl(n_{cp(\mathcal{C}_{\text{linear}})}, \mathcal{C}_{\text{linear}}) + bl(n_{cp(\mathcal{C}_{\text{linear}})}, \mathcal{C}_{\text{linear}}), \end{aligned} \tag{5.8}$$

with $n_{cp(\mathcal{C}_{\text{linear}})} \in cp(\mathcal{C}_{\text{linear}})$ being a node of the allocated critical path. A direct consequence of this is formulated in Lemma 5.2.

**Lemma 5.2 (Linear Clustering: Schedule Length and Zeroed Edges)** *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph, and $\mathcal{C}_{\text{linear}}$ and $\mathcal{C}_{\text{linear},+}$ two linear clusterings of $G$. Let $\mathbf{E}_{\text{zero}}$ be the set of zeroed edges of $\mathcal{C}_{\text{linear}}$, that is, $\mathbf{E}_{\text{zero}} = \{e_{ij} \in \mathbf{E} : proc(n_i) = proc(n_j)\}$, and $\mathbf{E}_{\text{zero},+}$ the set of zeroed edges of $\mathcal{C}_{\text{linear},+}$, with $|\mathbf{E}_{\text{zero},+}| > |\mathbf{E}_{\text{zero}}|$ and $\mathbf{E}_{\text{zero}} \subset \mathbf{E}_{\text{zero},+}$. For the schedule lengths of the clusterings it holds that*

$$sl(\mathcal{C}_{\text{linear},+}) \leq sl(\mathcal{C}_{\text{linear}}). \tag{5.9}$$

*Proof*. Since in $\mathcal{C}_{\text{linear},+}$ at least one more edge is zeroed than in $\mathcal{C}_{\text{linear}}$, it holds for the allocated top and bottom levels that

$$\begin{aligned} tl(n, \mathcal{C}_{\text{linear},+}) &\leq tl(n, \mathcal{C}_{\text{linear}}) \\ bl(n, \mathcal{C}_{\text{linear},+}) &\leq bl(n, \mathcal{C}_{\text{linear}}) \end{aligned} \quad \forall n \in \mathbf{V}. \tag{5.10}$$

Since the schedule length of a clustering is the sum of the allocated top and bottom levels of an allocated critical path node, Eq. (5.8), this proves the lemma. □

In other words, given a linear clustering, zeroing more edges, while maintaining the linear property, does not increase the schedule length. Based on these observations, it is easy to establish a bound on the schedule length of a linear clustering relative to the granularity of the task graph.

**Theorem 5.3 (Bound on Linear Clustering)** *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph, $\mathcal{C}_{\text{linear}}$ a linear clustering of $G$, and $\mathcal{C}_{\text{opt}}$ a clustering of $G$ with optimal schedule length. It holds that*

$$sl(\mathcal{C}_{\text{linear}}) \leq \left(1 + \frac{1}{g_{\text{weak}}(G)}\right) sl(\mathcal{C}_{\text{opt}}). \tag{5.11}$$

*Proof*. Let $\mathcal{C}_0$ be an initial clustering, where each task $n \in \mathbf{V}$ is allocated to a different cluster $C \in \mathbf{C}$. Consequently, the allocated length of the allocated critical path $cp(\mathcal{C}_0)$, that is, $\mathcal{C}_0$'s schedule length (see Eq. (5.8)), is identical to the length of the critical path $cp$ of $G$

$$sl(\mathcal{C}_0) = len(cp(\mathcal{C}_0), \mathcal{C}_0) = len(cp). \tag{5.12}$$

From Lemma 5.2 it is clear that

$$sl(\mathcal{C}_{\text{linear}}) \leq sl(\mathcal{C}_0) \tag{5.13}$$

for any linear clustering $\mathcal{C}_{\text{linear}}$. Theorem 4.6 states that

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

from which it follows that

$$sl(\mathcal{C}_{\text{linear}}) \leq sl(\mathcal{C}_0) = len(cp) \leq \left(1 + \frac{1}{g_{\text{weak}}(G)}\right) len_w(cp_w). \tag{5.14}$$

Finally, with $len_w(cp_w) \leq sl(\mathcal{S})$ for any schedule $\mathcal{S}$ (Lemma 4.4), in particular, $len_w(cp_w) \leq sl(\mathcal{S}_{\text{opt}})$, one gets

$$sl(\mathcal{C}_{\text{linear}}) \leq \left(1 + \frac{1}{g_{\text{weak}}(G)}\right) len_w(cp_w) \leq \left(1 + \frac{1}{g_{\text{weak}}(G)}\right) sl(\mathcal{C}_{\text{opt}}). \tag{5.15}$$

□

From Theorem 5.3 it follows that the larger the weak granularity, the smaller is the possible difference between the schedule length of the linear clustering and the optimal schedule length:

$$g_{\text{weak}}(G) \to \infty \Rightarrow sl(\mathcal{C}_{\text{linear}}) \to sl(\mathcal{C}_{\text{opt}}). \tag{5.16}$$

This observation is no surprise, given that Theorem 4.4 establishes that an optimal schedule can be created in polynomial time for scheduling without communication costs on an unlimited number of processors. Only the minimization of the number of used processors is NP-hard. While clustering in general is not a technique useful for scheduling without communication costs, linear clustering can be employed as a heuristic for the reduction of the processor number. Starting with an initial clustering, which is optimal for scheduling without communication costs (Theorem 4.4), Lemma 5.2 guarantees that any linear clustering is still optimal for this scheduling problem.

For coarse grained task graphs $G$, that is, $g_{\text{weak}}(G) \geq 1$, Theorem 5.3 establishes a 2-optimal bound

$$sl(\mathcal{C}_{\text{linear}}) \leq 2sl(\mathcal{C}_{\text{opt}}). \tag{5.17}$$

For coarse grained task graphs, Gerasoulis and Yang [77] even demonstrated that there is at least one linear clustering that is optimal. Unfortunately, the algorithm to obtain such a clustering in polynomial time is unknown.

***Algorithm*** A linear clustering can be achieved with a simple and intuitive algorithm proposed by Kim and Browne [104]. Consider as a basis the generic clustering Algorithm 15: in each refinement step a new longest path $p$, consisting of previously unexamined edges, is selected in task graph $G$. The nodes of $p$ are merged into one cluster and the edges incident on these nodes are marked examined. This is repeated until all edges of $G$ have been examined. Clearly, such an algorithm belongs to the path type in terms of the multiplicity of edge zeroing.

Algorithm 16 outlines the above described procedure. $\mathbf{E}_{\text{unex}} \subseteq \mathbf{E}$ is the set of unexamined edges and $G_{\text{unex}} = (\mathbf{V}_{\text{unex}}, \mathbf{E}_{\text{unex}})$ is the task graph spawned by these edges and the nodes $\mathbf{V}_{\text{unex}} \subseteq \mathbf{V}$, which are incident on these edges. Hence, each $G_{\text{unex}}$ is a subgraph of $G$. In each step the critical path $cp$ of the current unexamined graph $G_{\text{unex}}$ is determined and its edges are merged into one cluster. Observe that all edges incident on the nodes of $cp$, $\{e_{ij} \in \mathbf{E}_{\text{unex}} : n_i \in \mathbf{V}_{cp} \vee n_j \in \mathbf{V}_{cp}\}$, are marked as examined at the end of each step, and not only those edges $e_{ij} \in \mathbf{E}_{cp}$ that are part of $cp$. This is important, because otherwise, in a later step, one of the edges $e_{ij} \notin \mathbf{E}_{cp}$ not being part of $cp$ could be zeroed. This would mean that the edge's two incident nodes $n_i$ and $n_j$, of which at least one is part of a cluster with more than one node, are merged into the same cluster, potentially resulting in a nonlinear cluster.

***Algorithm 16 Linear Clustering Algorithm*** $(G = (\mathbf{V}, \mathbf{E}, w, c))$

Create initial clustering $\mathcal{C}_0$: allocate each node $n \in \mathbf{V}$ to a distinct cluster $C \in \mathbf{C}$, $|\mathbf{C}| = |\mathbf{V}|$
$\mathbf{E}_{\text{unex}} \leftarrow \mathbf{E}$ ▷ $\mathbf{E}_{\text{unex}}$*: set of unexamined edges,* $\mathbf{E}_{\text{unex}} \subseteq \mathbf{E}$
**while** $\mathbf{E}_{\text{unex}} \neq \emptyset$ **do** ▷ *there is at least one unexamined edge*
  Find a critical path $cp$ of graph $G_{\text{unex}} = (\mathbf{V}_{\text{unex}}, \mathbf{E}_{\text{unex}})$ ▷ $\mathbf{V}_{\text{unex}}$*: set of nodes* $n \in \mathbf{V}$ *on which edges* $\mathbf{E}_{\text{unex}}$ *are incident*
  Merge all nodes $\mathbf{V}_{cp}$ of $cp$ into one cluster
  $\mathbf{E}_{\text{unex}} \leftarrow \mathbf{E}_{\text{unex}} - \{e_{ij} \in \mathbf{E}_{\text{unex}} : n_i \in \mathbf{V}_{cp} \vee n_j \in \mathbf{V}_{cp}\}$ ▷ *Mark edges that are incident on nodes of* $\mathbf{V}_{cp}$ *as examined*
**end while**

In comparison to the general structure of a clustering algorithm, as outlined in Algorithm 15, there is no check of whether the schedule length of the new clustering is not worse than the current one. The check is unnecessary, as Lemma 5.2 guarantees that this is the case.

Finding the critical path of the task graph $G_{\text{unex}}$ in each step is essentially the determination of the nodes' top and bottom levels. This can be performed using the algorithms presented in Section 4.4.2, with a complexity of $O(\mathbf{V} + \mathbf{E})$. Merging the nodes and marking the corresponding nodes and edges as examined has lower complexity than $O(\mathbf{V} + \mathbf{E})$. As there are at most $O(\mathbf{V})$ steps, Algorithm 16's total complexity is thus $O(\mathbf{V}(\mathbf{V} + \mathbf{E}))$.

As a last comment on linear clustering, note the similarity between linear projections of dependence graphs as analyzed in Section 3.5.1 and linear clustering. The linear projection of DGs for example, is used in VLSI array processor design (Kung [109]).

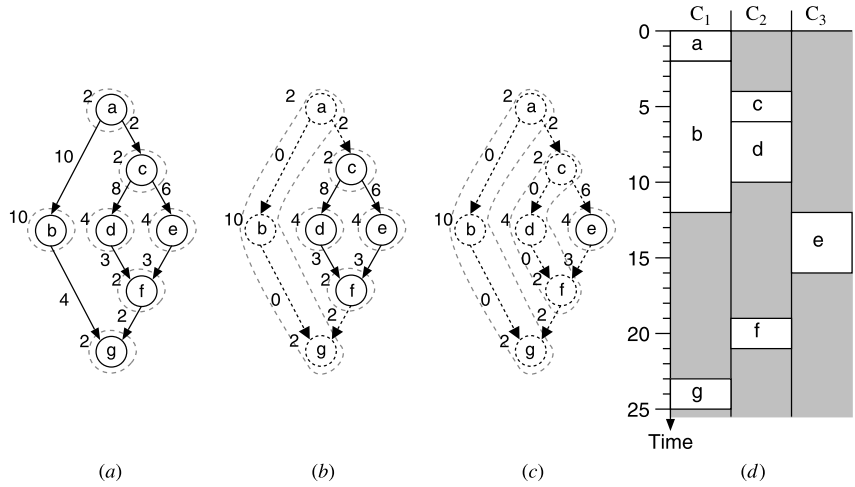



**Figure 5.5.** Linear clusterings of a simple task graph (Figure 5.3(*a*)): (*a*) initial clustering; (*b*) clustering after first step; (*c*) final clustering; (*d*) schedule of final clustering. (Examined elements are dotted.)

***Example*** As an example, the linear clustering algorithm is applied to the task graph of Figure 5.3(*a*). The usual initial clustering is depicted in Figure 5.5(*a*).

- All edges are unexamined at this stage and the critical path is $cp = \langle a, b, g \rangle$ with a length of $len(cp) = 28$. These nodes are merged into one cluster and they are marked examined as are their incident edges, $e_{ab}, e_{bg}, e_{ac}, e_{fg}$. The resulting clustering is illustrated in Figure 5.5(*b*), where the examined nodes and edges are drawn with dotted lines.
- The current unexamined graph $G_{\text{unex}}$ consists of the nodes $\mathbf{V}_{\text{unex}} = \{c, d, e, f\}$ and the edges $\mathbf{E}_{\text{unex}} = \{e_{cd}, e_{ce}, e_{df}, e_{ef}\}$. Its critical path is $cp = \langle c, d, f \rangle$ with a length of $len(cp) = 19$. These nodes are merged into one cluster and they are marked examined as are their incident edges, $e_{cd}, e_{ce}, e_{df}, e_{ef}$. The resulting clustering is illustrated in Figure 5.5(*c*).
- There are no unexamined edges left, $\mathbf{E}_{\text{unex}} = \emptyset$, and the algorithm terminates. Therefore, the final clustering $\mathcal{C}_{\text{final}}$ is the one given in Figure 5.5(*c*), with a schedule length of

$$sl(\mathcal{C}_{\text{final}}) = len(cp(\mathcal{C}_{\text{final}}), \mathcal{C}_{\text{final}}) = len(\langle a, c, e, f, g \rangle, \mathcal{C}_{\text{final}}) = 25.$$

  The resulting implicit schedule is displayed in Figure 5.5(*d*).

### 5.3.3 Single Edge Clustering

The second approach to clustering discussed here considers one single edge at a time for zeroing. A simple algorithm based on this approach was proposed by Sarkar

[167]. At the beginning of the algorithm the edges of the task graph are assigned a priority, which determines the order in which they are considered. The main part of the algorithm iterates over all edges **E** and checks for each edge if its zeroing results in a schedule length that is smaller than or equal to the schedule length of the present clustering. If yes, the zeroing is accepted: that is, the clusters of the incident nodes of the edge are merged. This procedure is repeated until all edges have been considered. Algorithm 17 outlines a generic version of the above described algorithm.

***Algorithm 17 Single Edge Clustering Algorithm*** **($G = (V, E, w, c)$)**

Create initial clustering $\mathcal{C}_0$: allocate each node $n \in \mathbf{V}$ to a distinct cluster $C \in \mathbf{C}$, $|\mathbf{C}| = |\mathbf{V}|$
$i \leftarrow 0$
$\mathbf{E}_{\text{unex}} \leftarrow \mathbf{E}$ ▷ $\mathbf{E}_{\text{unex}}$*: set of unexamined edges,* $\mathbf{E}_{\text{unex}} \subseteq \mathbf{E}$
**while** $\mathbf{E}_{\text{unex}} \neq \emptyset$ **do** ▷ *there is at least one unexamined edge*
  $i \leftarrow i + 1$
  Let $e_{ij}$ be the edge of $\mathbf{E}_{\text{unex}}$ with highest priority
  **if** $proc(n_i) = proc(n_j)$ **then** ▷ *incident nodes are in same cluster*
    $\mathcal{C}_i \leftarrow \mathcal{C}_{i-1}$ ▷ *nothing to do*
  **else**
    Create new clustering $\mathcal{C}_i$: merge clusters $proc(n_i)$ and $proc(n_j)$ into one cluster
    Schedule $\mathcal{C}_i$ using heuristic
    **if** $sl(\mathcal{C}_i) > sl(\mathcal{C}_{i-1})$ **then** ▷ *merging increases current schedule length*
      $\mathcal{C}_i \leftarrow \mathcal{C}_{i-1}$ ▷ *reject new clustering* $\mathcal{C}_i$
    **end if**
  **end if**
  $\mathbf{E}_{\text{unex}} \leftarrow \mathbf{E}_{\text{unex}} - \{e_{ij}\}$ ▷ *Mark edge* $e_{ij}$ *as examined*
**end while**

Observe the `if` statement that checks whether the incident nodes $n_i$ and $n_j$ of the currently considered edge are already in the same cluster ($proc(n_i) = proc(n_j)$). Even though all nodes are initially allocated to a distinct cluster, this situation can in fact occur due to the zeroing of previous edges in the list. To illustrate this, regard Figure 5.6, where a three-node task graph is clustered by zeroing one edge at a time. The edges are considered in the order $e_{ab}, e_{ac}, e_{bc}$. Figure 5.6($a$) shows the initial clustering, Figure 5.6($b$) the clustering after the zeroing of $e_{ab}$, and Figure 5.6($c$) after the zeroing of $e_{ac}$. The zeroing of $e_{ac}$ merges all three nodes $a, b, c$ into one cluster, which implicitly zeros edge $e_{bc}$ as its communication is now also local. Therefore, the correct clustering after the zeroing of $e_{ac}$ is the one shown in Figure 5.6($d$).

A key characteristic of this approach to clustering is that there is no clear implicit schedule. This is always a problem, irrespective of the order in which the edges are considered. Independent nodes might be allocated to the same cluster. Consequently, Algorithm 17 contains a statement for the determination of the schedule of each new clustering $\mathcal{C}_i$. In essence, this is scheduling with a given processor allocation as discussed in Section 5.2, where the processor allocation is given by the clustering $\mathcal{C}_i$ of the nodes. As a consequence, the determination of the schedule reduces to the

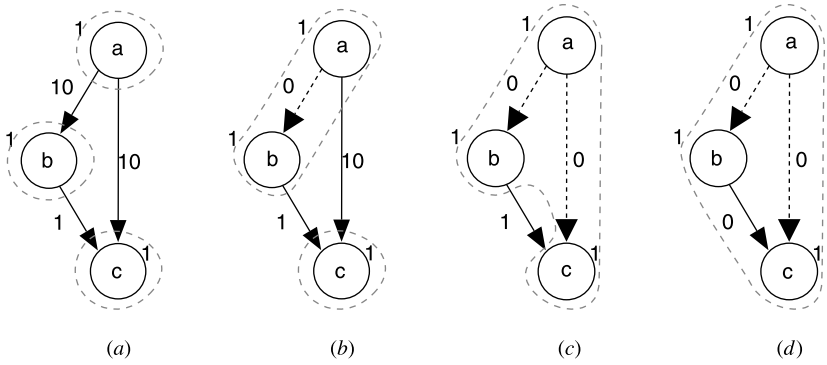



**Figure 5.6.** Implicit zeroing of edges: (*a*) initial clustering; (*b*) clustering after zeroing $e_{ab}$; (c) clustering after zeroing $e_{ab}$ and $e_{ac}$; (*d*) $e_{bc}$ was implicitly zeroed in (*c*).

ordering of the nodes. Any priority scheme for ordering the nodes, as discussed in Section 5.1.3, can be employed. Not surprisingly, Gerasoulis and Yang [76] suggest employing the allocated bottom level (Definition 4.19) of the current clustering $\mathcal{C}_{i-1}$.

The last issue to address in this clustering approach is the order in which edges are considered. There is no constraint on the possible orders; however, an argument for a good heuristic is to zero "important" edges first. This is the same argument as in list scheduling, where "important" nodes should be considered first. Another analogy to list scheduling is that the priority of the edges can be static or dynamic. Algorithm 17 is formulated in a generic way that allows one to handle both static and dynamic priorities.

***Static Edge Order*** In case static priorities are used, the edges can be sorted into a list before the main part of the algorithm starts, just like in static list scheduling (Algorithm 9). An idea that comes to mind is to sort the edges in decreasing order of their weights (as suggested by Sarkar [167]); that is, edges with high costs are considered earlier than those with low costs. Such an ordering will substantially reduce the total communication cost of the resulting clustering, with the implicit assumption that this also reduces the schedule length. As with the ordering of the nodes in, for example, list scheduling (Section 5.1.3), many other orderings are conceivable. Exercise 5.7 asks you to suggest priority schemes to order edges.

The complexity of single edge clustering with a static edge order is as follows. In each step one edge is considered for zeroing; that is, there are $|\mathbf{E}|$ steps. When an edge is zeroed, the node order must be determined, which usually is $O(\mathbf{V}+\mathbf{E})$ (e.g., using the allocated bottom level as the node priority; see above). Then the schedule length is determined, which implies in the worst case a construction of a schedule, which is also $O(\mathbf{V}+\mathbf{E})$ (see Section 5.2). It results in a complexity of $O(\mathbf{V}+\mathbf{E})$ for each step and a total complexity of $O(\mathbf{E}(\mathbf{V}+\mathbf{E}))$. This complexity result implies that the ordering of the edges has lower or equal complexity. The total complexity might increase for the case where higher complexity scheduling algorithms are used in each step.

***Example*** To illustrate Algorithm 17 with a static edge order, it is applied to the simple graph of Figure 5.3($a$). The edges are sorted in decreasing order of their weights, resulting in the order $e_{ab}, e_{cd}, e_{ce}, e_{bg}, e_{df}, e_{ef}, e_{ac}, e_{fg}$. Note that edges $e_{df}, e_{ef}$ and $e_{ac}, e_{fg}$ have the same weight value, and therefore their order was chosen at random. Independent nodes are scheduled in decreasing order of their allocated bottom level of the current clustering $\mathcal{C}_{i-1}$. The usual initial clustering is depicted in Figure 5.7($a$).

- $e_{ab}, e_{cd}$. The zeroing of the first two edges $e_{ab}$ and $e_{cd}$ is straightforward as only dependent nodes are merged. The resulting clusterings are linear and the schedule length decreases. Figure 5.7($b$) depicts the clustering $\mathcal{C}_2$ after these two steps, with an implicit schedule length of $sl(\mathcal{C}_2) = 25$.
- $e_{ce}$. Zeroing the edge $e_{ce}$ results in a nonlinear cluster as it merges the nodes $c, d, e$ into a single cluster, and $d$ and $e$ are independent. The allocated bottom levels of both are $bl(d, \mathcal{C}_2) = bl(e, \mathcal{C}_2) = 13$, and priority is given here at random to $d$. So the nodes are scheduled in order $c, d, e$ in their cluster, with the start times $t_s(c) = 4$, $t_s(d) = 6$, $t_s(e) = 10$. This results in a schedule length of $sl(\mathcal{C}_3) = 23$, which is less than the current schedule length $sl(\mathcal{C}_2) = 25$. Hence, the new clustering $\mathcal{C}_3$, shown in Figure 5.7($c$), is accepted.
- $e_{bg}$. By zeroing edge $e_{bg}$, the nodes $a, b, g$ are merged into one cluster, which is linear. This does not decrease the schedule length, but it also does not increase it; hence, the clustering $\mathcal{C}_4$ is accepted (Figure 5.7($d$)).
- $e_{df}$. The zeroing of edge $e_{df}$ merges the nodes $c, d, e, f$ into one cluster. As node $f$ depends on $d$ and $e$, it is scheduled after them, resulting in a schedule length of $sl(\mathcal{C}_5) = 20$ for this clustering $\mathcal{C}_5$ (Figure 5.7($e$)).
- $e_{ef}$. Edge $e_{ef}$ has been zeroed implicitly (its weight is 0 in Figure 5.7($e$)), when edge $e_{df}$ was zeroed, so nothing is done in this step.
- $e_{ac}, e_{fg}$. The zeroing of either $e_{ac}$ or $e_{fg}$, or both, would merge all nodes into one single cluster, increasing the schedule length to 26. Hence, they are not accepted, which makes the clustering of Figure 5.7($e$) the final clustering, with the schedule length $sl(\mathcal{C}_{\text{final}}) = 20$.

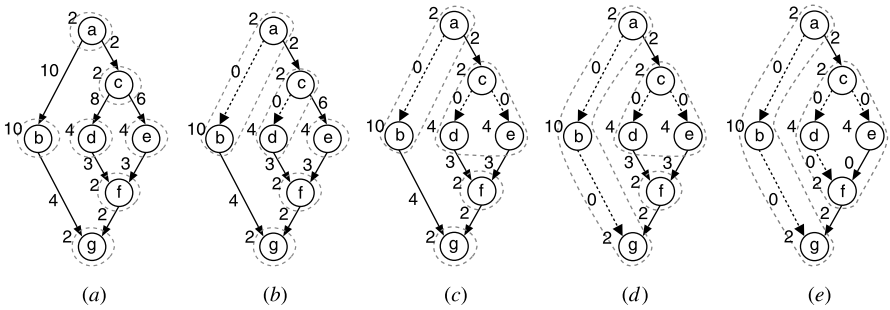



**Figure 5.7.** Single edge clustering of a simple task graph (Figure 5.3($a$)): ($a$) initial clustering; ($b$) clustering after zeroing $e_{ab}, e_{cd}$; ($c$) after zeroing $e_{ce}$; ($d$) after zeroing $e_{bg}$; (e) after zeroing $e_{df}$, also final clustering. (Examined edges are dotted.)

***Dynamic Edge Order*** In dynamic list scheduling, the priority of all free nodes is recalculated in each step of the algorithm (Section 5.1.2). The same principle applies here for dynamic edge orders. In each step of the clustering, the priorities of the unexamined edges are recalculated and the edge with the highest priority is considered for zeroing.

In terms of priority metrics, a natural idea is again to attribute higher priority to the edges of an allocated critical path of the current clustering $\mathcal{C}_{i-1}$, which naturally changes in each step and therefore requires a recalculation. While this seems similar to the approach of the linear clustering algorithm presented earlier in this section (Algorithm 16), it differs in two substantial points. First, only one edge of an allocated CP is zeroed at each step, and a heuristic has to be employed to determine which (as discussed later). Second, by only zeroing one edge of the current allocated CP, the allocated CP of the next clustering $\mathcal{C}_i$ might already be different. As a consequence, the clusterings produced by this algorithm are generally not linear.

Since computing an allocated CP (and the allocated node levels for that matter) has a complexity of $O(\mathbf{V} + \mathbf{E})$, the runtime complexity of the clustering algorithm is usually not increased by using dynamic priorities in comparison to using static.

Zeroing an edge of the allocated CP, even if it is unique, does not necessarily mean that the schedule length of the resulting clustering $\mathcal{C}_i$ can be decreased. The reason lies in the fact that the schedule length of $\mathcal{C}_{i-1}$ might be longer than the allocated length of an allocated CP of the task graph,

$$sl(\mathcal{C}_{i-1}) \geq len(cp(\mathcal{C}_{i-1}), \mathcal{C}_{i-1}). \tag{5.18}$$

Only in linear clusterings does the scheduling length equal the allocated length of the allocated critical path, Eq. (5.8). The ordering of independent nodes allocated to the same cluster has an impact on the schedule length of $\mathcal{C}_{i-1}$, without having any influence on the allocated CP. To support this reasoning, regard the scheduling with a given processor allocation discussed in Section 5.2. The allocated CP is the same for all possible schedules, while finding a schedule with optimal length is NP-hard.

***Dominant Sequence*** The foregoing thoughts lead to the introduction of a new concept, the *dominant sequence* (Gerasoulis and Yang [76, 77]). The dominant sequence is again a path of maximal length; however, it integrates the scheduling order of independent nodes into the calculation of the path length. This is accomplished by introducing the *scheduled* task graph, which is the task graph $G$ extended with virtual edges.

**Definition 5.5 (Scheduled Task Graph, Virtual Edge, Dominant Sequence)** *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph and $\mathcal{C}$ a clustering of $G$. $G_{\text{scheduled}} = (\mathbf{V}, \mathbf{E}_{\text{scheduled}}, w, c)$ is the* scheduled task graph *of which $G$ is a subgraph, $G \subseteq G_{\text{scheduled}}$, having the same nodes as $G$, but additional edges, $\mathbf{E} \subseteq \mathbf{E}_{\text{scheduled}}$. There is a* virtual edge *$e_{ij} \in \mathbf{E}_{\text{scheduled}}, \notin \mathbf{E}$ for each independent node pair $n_i, n_j \in \mathbf{V}$ (i.e., there is no path $p(n_i \to n_j)$ or $p(n_j \to n_i)$ in $G$) that is allocated to the same cluster $C = proc(n_i) = proc(n_j)$, and where $n_j$ is scheduled directly after node $n_i$ on*

*C (i.e., there is no node scheduled between them). The weight of* $e_{ij}$ *is* $c(e_{ij}) = 0$. *Given* $G_{\text{scheduled}}$, *the* dominant sequence *(DS) of G and clustering* $\mathcal{C}$ *is defined as the allocated critical path of* $G_{\text{scheduled}}$ *and clustering* $\mathcal{C}$:

$$ds(\mathcal{C}) = cp(G_{\text{scheduled}}, \mathcal{C}). \tag{5.19}$$

To illustrate the concept of the scheduled task graph and its virtual edges, return to the clustering example of Figure 5.7. In Figure 5.7(*e*), two independent nodes, *d* and *e*, are allocated to the same cluster. During the application of the single edge clustering algorithm, it was established that in the execution order *d* precedes *e*. Using the concept of virtual edges, the edge $e_{de}$ with weight $c(e_{de}) = 0$ is therefore inserted into the scheduled task graph, as depicted in Figure 5.8. The allocated critical path of this scheduled task graph is $\langle a, c, d, e, f, g\rangle$ with length 20, which is the dominant sequence $ds(\mathcal{C}_{\text{final}})$ of this clustering $\mathcal{C}_{\text{final}}$.

So the length of the dominant sequence $ds(\mathcal{C}_{\text{final}})$ of $\mathcal{C}_{\text{final}}$ is equal to its schedule length $sl(\mathcal{C}_{\text{final}}) = 20$. This reminds one of Eq. (5.8), which establishes for linear clusterings that the schedule length is identical to the allocated CP. This is no coincidence, as Lemma 5.3 demonstrates.

**Lemma 5.3 (Clustering Is Linear for Scheduled Task Graph)** *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph,* $\mathcal{C}$ *a clustering of G, and* $G_{\text{scheduled}} = (\mathbf{V}, \mathbf{E}_{\text{scheduled}}, w, c)$ *the corresponding scheduled task graph. Then* $\mathcal{C}$ *is a linear clustering of* $G_{\text{scheduled}}$.

*Proof*. The proof follows from the definition of the scheduled task graph. Let $n_1, n_2, \ldots, n_m$ be all the nodes allocated to any cluster $C$, that is, $\{n_1, n_2, \ldots, n_m\} = \{n \in \mathbf{V} : proc(n) = C\}$, arranged in topological order of $G_{\text{scheduled}}$. For any node pair $n_i, n_{i+1}$, $i = 1, m-1$, there is a path in $G_{\text{scheduled}}$ from $n_i$ to $n_{i+1}$. Either this path already exists in $G$ or, if it does not, there is a virtual edge $e_{i,i+1}$ representing this path,

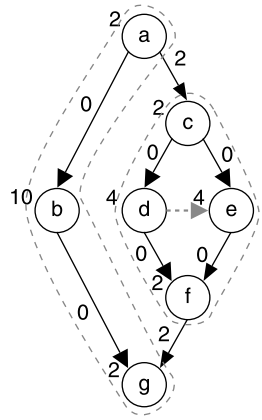



**Figure 5.8.** Scheduled task graph of clustering $\mathcal{C}_{\text{final}}$ from Figure 5.7(*e*): virtual edge $e_{de}$ (gray and dashes) indicates scheduling order between nodes *d* and *e*.

as per definition of $G_{\text{scheduled}}$. By induction, there is a path $p(n_1 \rightarrow n_m)$ from $n_1$ to $n_m$ in $G_{\text{scheduled}}$ to which all nodes allocated to $C$ belong, that is, $\{n_1, n_2, \ldots, n_m\} \subseteq \{n \in \mathbf{V} : n \in p(n_1 \rightarrow n_m) \text{ in } G_{\text{scheduled}}\}$. Per Definition 5.4, this makes $C$ a linear cluster and the clustering $\mathcal{C}$ linear, because it holds for any of its clusters. □

A direct consequence of this lemma is that the relation between the schedule length and the length of the DS, as observed earlier, is generally true:

$$sl(\mathcal{C}) = len(ds(\mathcal{C}), \mathcal{C}). \tag{5.20}$$

So what the allocated CP is to a linear clustering the DS is to a nonlinear one. Note that this is valid, because virtual edges have zero weight and thus add nothing to the length of a path.

As the dominant sequence determines the schedule length of a clustering $\mathcal{C}_{i-1}$, zeroing one of its edges $e \in ds(\mathcal{C}_{i-1})$ can reduce the schedule length $sl(\mathcal{C}_{i-1})$. Or considered the other way around, if an edge is not part of a DS, zeroing it cannot reduce the schedule length.[1] In particular, if an edge belongs to an allocated CP, but not to a DS, its zeroing cannot reduce the schedule length of the current clustering $\mathcal{C}_{i-1}$.

Unfortunately, there is no guarantee that the schedule length $sl(\mathcal{C}_{i-1})$ of the current clustering $\mathcal{C}_{i-1}$ decreases or at least remains the same if an edge $e \in ds(\mathcal{C}_{i-1})$ of the DS is zeroed. Zeroing an edge of the DS can result in the insertion of one or more new virtual edges into the scheduled task graph $G_{\text{scheduled}}$. These edges might become part of the DS of the new scheduled task graph, potentially increasing its length. It also means that the schedule length of a clustering $\mathcal{C}_{i-1}$ is reduced, if no new virtual edges are inserted into the scheduled task graph $G_{\text{scheduled}}$ of $\mathcal{C}_{i-1}$ and the DS was unique.

Given the above reasoning, it seems very appropriate for a dynamic single edge clustering heuristic to aim at zeroing an edge of the current dominant sequence. Gerasoulis and Yang [76] showed analytically and experimentally that doing so often leads to better clusterings when compared to other heuristics. The complexity of determining the DS is identical to computing the allocated CP, namely, $O(\mathbf{V} + \mathbf{E})$ for each step. Hence, it does not increase the complexity of single edge clustering.

***Selecting Edge in Path*** Single edge clustering heuristics that dynamically order the edges based on a path-based metric, be it the allocated CP, the DS, or something else, still have to choose among the multiple edges of the path $p$. Darte et al. [52] see three alternatives:

1. Select the edge of the path, whose zeroing will reduce the schedule length of the resulting clustering most.
2. Select the edge with the maximum weight, that is, $e_{\max} \in p$ with $c(e_{\max}) = \max_{e \in p} c(e)$.
3. Select the first edge of the path $p$.

[1] Unless an edge of the dominant sequence is implicitly zeroed as a consequence.

Obviously, only nonzero edges are considered in all cases. The first alternative is of course very costly, as the schedule length must be determined for the zeroing of each edge of the path, which generally is $O(\mathbf{V} + \mathbf{E})$. In the worst case, there are $O(\mathbf{E})$ edges in the path; thus, the total complexity of single edge clustering increases by the same factor, resulting in $O(\mathbf{E}^2(\mathbf{V} + \mathbf{E}))$. The second alternative is similar to statically ordering the edges by their weights, as proposed by Sarkar [167] (see earlier example). Interestingly, the total complexity is the same in both cases, dynamic and static, namely, $O(\mathbf{E}(\mathbf{V} + \mathbf{E}))$. The third alternative is simple and cheap, but it does not have a lower total complexity.

### 5.3.4 List Scheduling as Clustering

Regarding the multiplicity of edge zeroing, two of the three types (Section 5.3.1) have been discussed so far. Linear clustering belongs to the path type, whereas single edge clustering considers only one edge at a time. In this section, a heuristic approach with node multiplicity is analyzed, where in each step of the heuristic the edges incident on a given node are considered for zeroing.

Essentially, the approach analyzed in the following is an adapted list scheduling. This might come as a surprise, given that this chapter makes a clear distinction between both approaches. However, it will be seen that list scheduling can in fact be interpreted as a clustering heuristic if certain conditions are met.

***Number of Processors*** Clustering algorithms have no limit on the number of processors, while list scheduling generally has. However, by making the target system consist of $|\mathbf{V}|$ processors in a list scheduling algorithm, there is virtually an unlimited number of processors (see Section 4.2.2).

***Edge Zeroing*** A distinct characteristic of clustering is that nodes are only merged into the same cluster if an edge is zeroed as a consequence (Section 5.3.1). In contrast, list scheduling might in general allocate a node $n$ to a processor $P$, even if none of $n$'s predecessors is on $P$. So $n$ is merged with the nodes already on $P$, $\{n_i \in \mathbf{V} : proc(n_i) = P\}$, yet no edge can be zeroed, as there are no edges between $n$ and these nodes. Using start time minimization (Section 5.1.1), the starting time of $n$ on such a processor $P$ must be (Condition 4.2)

$$t_s(n, P) \geq t_{\mathrm{dr}}(n, P) = t_{\mathrm{dr}}(n), \tag{5.21}$$

where $t_{\mathrm{dr}}(n)$ is the data ready time of $n$, assuming all of its entering communications are remote. If instead node $n$ is scheduled on an empty processor $P_{\mathrm{empty}}$, $\{n_i \in \mathbf{V} : proc(n_i) = P_{\mathrm{empty}}\} = \emptyset$, its start time is not later

$$t_s(n, P_{\mathrm{empty}}) = \max\{t_{\mathrm{dr}}(n, P_{\mathrm{empty}}), t_f(P)\} = t_{\mathrm{dr}}(n). \tag{5.22}$$

Consequently, scheduling node $n$ on an empty processor $P_{\mathrm{empty}}$ instead of on processor $P$ does not change the heuristic approach of start time minimization. Furthermore,

when $|\mathbf{V}|$ processors are used in list scheduling (as proposed earlier), there will always be at least one empty processor.

So list scheduling with start time minimization can be employed as a clustering heuristic by merely adding a tie breaker. If the start time of $t_s(n) = t_{\mathrm{dr}}(n)$ or less cannot be achieved on any of the processors to which $n$'s predecessors are scheduled (which would at least zero one edge), schedule $n$ onto an empty processor. There might be other processors that could achieve the same start time of $t_s(n) = t_{\mathrm{dr}}(n)$, but the tie is broken by always using an empty processor.

Algorithm 18 shows the modified start time minimization of list scheduling. Compared to the original Algorithm 10, only two points differ: (1) $P_{\mathrm{empty}}$ is made the default processor and (2) the `for` loop only iterates over the processors of the predecessors of $n$, $\bigcup_{n_i \in \mathbf{pred}(n)} proc(n_i)$, and not over all processors.

***Algorithm 18 List Scheduling's Start Time Minimization for Clustering (See Algorithm 10)***

**Require:** $n$ is a free node
  $t_{\min} \leftarrow t_{\mathrm{dr}}(n)$; $P_{\min} \leftarrow P_{\mathrm{empty}}$
  **for** each $P \in \bigcup_{n_i \in \mathbf{pred}(n)} proc(n_i)$ **do**
    **if** $t_{\min} \geq \max\{t_{\mathrm{dr}}(n, P), t_f(P)\}$ **then**
      $t_{\min} \leftarrow \max\{t_{\mathrm{dr}}(n, P), t_f(P)\}$; $P_{\min} \leftarrow P$
    **end if**
  **end for**
  $t_s(n) \leftarrow t_{\min}$; $proc(n) \leftarrow P_{\min}$

In each step of list scheduling one node $n_j$ is scheduled on a processor. From a clustering point of view, this means that the incoming edges of $n_j$, $\{e_{ij} \in \mathbf{E} : n_i \in \mathbf{pred}(n_j)\}$, are considered for zeroing. If $n_j$ is scheduled on an empty processor, none of these edges is zeroed.

***Implicit Schedules*** List scheduling was presented in Section 5.1 as an algorithm that incrementally builds a schedule by extending the current partial schedule with one node at a time. Only the final schedule is a complete and feasible schedule. In contrast, clustering heuristics start with an initial clustering and construct intermediate clusterings that are all feasible. This conflict can be solved by making a similar assumption for list scheduling: simply assume that each node is initially allocated to a distinct processor, with the corresponding implicit schedule. In each step of list scheduling, one node is considered for merging with its predecessors. If it is scheduled on an empty processor, that can be interpreted as rejecting the merge and leaving it on the processor it was (implicitly) scheduled on.

***Complexity*** List scheduling has a significantly lower complexity than the clustering algorithms presented so far. As stated in Section 5.1, the complexity of the second part of list scheduling with start time minimization (Algorithm 10) is usually $O(\mathbf{P}(\mathbf{V} + \mathbf{E}))$. With the proposed start time minimization (Algorithm 18), this changes

slightly. The start time of a node $n$ is not evaluated for all processors as in Algorithm 10, but only for the processors holding at least one of $n$ predecessors, that is, $O(\mathbf{pred}(n))$ times. Since determining $n$'s start time involves the calculation of the DRT on each processor, which is $O(\mathbf{pred}(n))$, this step has a complexity of $O(\mathbf{pred}(n)\,\mathbf{pred}(n))$. However, an efficient implementation can bring this down to $O(\mathbf{pred}(n)\log\mathbf{pred}(n))$, using a priority queue (e.g., heap, Cormen et al. [42]) for the arrival times of the incoming communications. The data arrival time only changes between processors if it becomes local; otherwise it remains the same for all processors. For all nodes, this amortizes to $O(\mathbf{E}\log\mathbf{V})$ (with $O(\log\mathbf{E}) = O(\log\mathbf{V})$). Since there are $|\mathbf{V}|$ steps, one for each node, the total complexity of the second part of this list scheduling is $O(\mathbf{V}+\mathbf{E}\log\mathbf{V})$. Assuming a level-based node order calculated in the first part of list scheduling, which is $O(\mathbf{V}\log\mathbf{V}+\mathbf{E})$ (Section 5.1.3), the resulting list scheduling complexity is $O((\mathbf{V}+\mathbf{E})\log\mathbf{V})$. In comparison, linear clustering is $O(\mathbf{V}(\mathbf{V}+\mathbf{E}))$, Algorithm 16, and single edge clustering is at least $O(\mathbf{E}(\mathbf{V}+\mathbf{E}))$, Algorithm 17.

One of the reasons for the higher complexity of many clustering algorithms is the fact that nodes might need to be reordered and rescheduled in each step after merging. This is not the case in list scheduling (with the end technique), because a node is added to the end of the nodes already allocated to a processor. Another contributing factor can also be the recalculation of the node levels in each step.

***Dynamic Sequence Clustering*** Gerasoulis and Yang [76] classified the MCP (modified critical path) (Wu and Gajski [207]) list scheduling heuristic as a clustering heuristic (its node priority scheme is mentioned in Section 5.1.3). Another heuristic analyzed in their comparison of clustering algorithms is the DSC (dynamic sequence clustering) (Yang and Gerasoulis [211]). Like MCP, it can be interpreted as a list scheduling heuristic.

A very special feature of DSC is that it zeros the edges of the dynamic sequence, as its name suggests. In experiments, it was shown to outperform the other compared algorithms. Moreover, it produces optimal results for task graphs with fork or join structure. Despite the good performance, it only has a complexity of $O((\mathbf{V}+\mathbf{E})\log\mathbf{V})$. This performance and complexity are achieved through several key points:

- The algorithm concentrates on the next node of the dynamic sequence $n_{\mathrm{DS}}$, even if it is only partially free. The free nodes that are scheduled until $n_{\mathrm{DS}}$ becomes free are not allowed to have a negative effect on the scheduling of $n_{\mathrm{DS}}$. Hence, DSC implements a list scheduling with a lookahead technique.
- The node priority is the sum of the allocated bottom level and allocated top level of the *scheduled* task graph, $tl(n,\mathcal{C}_{i-1}) + bl(n,\mathcal{C}_{i-1})$ in $G_{\mathrm{scheduled}}$. Note that the bottom level of a node does not change during list scheduling (Section 5.1.3), that is, $(bl(n,\mathcal{C}_{i-1})$ in $G_{\mathrm{scheduled}}) = (bl(n)$ in $G)$.
- To reduce complexity, the top level is estimated in a form that does not change the behavior of the algorithm, but does not require a complete traversal of the task graph.

For a detailed description of DSC and its theoretical properties please refer to Yang and Gerasoulis [211].

### 5.3.5 Other Algorithms

As mentioned at the beginning of this clustering section, virtually any scheduling algorithm that is suitable for an unlimited number of processors can be employed for the first phase of the three-step clustering based scheduling Algorithm 14.

For example, Hanen and Munier [86] proposed an integer linear program (ILP) formulation of scheduling a task graph on an unlimited number of processors. For coarse grained task graphs, that is, $g(G) \geq 1$, their relaxed ILP heuristic produces clusterings with a length that is within a factor of $\frac{4}{3}$ of the optimal solution, $sl(\mathcal{C}) \leq \frac{4}{3} sl(\mathcal{C}_{\text{opt}})$.

Apart from the above mentioned MCP algorithm, Wu and Gajski [207] proposed another algorithm called MD (mobility directed), which is a list scheduling algorithm for an unlimited number of processors with dynamic node priorities, using a node insertion technique (Section 6.1).

A clustering algorithm for heterogeneous processors was proposed by Cirou and Jeannot [36], using the concept of node triplets.

In a survey paper, Kwok and Ahmad [113] go as far as to classify scheduling algorithms that are designed for an unlimited number of processors into one class.

***Grain Packing*** Another problem related to task graphs, the so-called grain packing problem (El-Rewini et al. [65]), is essentially the same as the clustering problem. Grain packing addresses the problem of how to partition a program into subtasks (grains) in order to obtain the shortest possible execution time. Generally, this is the subtask decomposition step of parallel programming (Section 2.4). However, when subtasks are grouped together (i.e., packed), grain packing rather relates, like clustering, to orchestration (Culler and Singh [48]) or agglomeration (Foster [69]) as described in Section 2.3.

Given a task graph, grain packing algorithms determine how to group the nodes together into larger nodes so that the program can run in minimum time. Well, that is exactly what clustering algorithms are about. Some algorithms that were proposed for the grain packing problem are in fact scheduling algorithms, for example, the ISH (insertion scheduling heuristic) and the DSH (duplication scheduling heuristic) by Kruatrachue and Lewis [105, 106].

A noteworthy approach in this context is a grain packing algorithm proposed by McCreary and Gill [135]. Nodes are grouped into so-called clans, done by a graph partitioning algorithm (i.e., a pure graph theoretic approach). As a consequence, there is no implicit schedule associated with the resulting node packing. Hence, this is a grain packing algorithm, which is indeed different from clustering. In effect, the algorithm changes the granularity of the task graph before it is scheduled.

***Clustering with Node Duplication*** One of the objectives of clustering algorithms is the reduction of communication costs. Another scheduling technique, whose

only objective is the elimination of interprocessor communication, is node duplication. This technique is studied in Section 6.2. The combination of clustering and node duplication is very powerful in reducing communication costs and can lead to even better results than pure clustering. Some algorithms integrating both techniques have been proposed (e.g., Liou and Palis [125], Palis et al. [140]).

## 5.4 FROM CLUSTERING TO SCHEDULING

The previous section discussed algorithms for the first phase (i.e., the clustering phase) of Algorithm 14. For a complete scheduling algorithm for an arbitrary number of processors, it remains to study the second and third phases of such a clustering based scheduling algorithm.

In the second phase, the clusters $\mathbf{C}$ of the obtained clustering $\mathcal{C}$ are mapped onto the available physical processors $\mathbf{P}$. If the number of clusters $|\mathbf{C}|$ is less than or equal to the number of physical processors $|\mathbf{P}|$, $|\mathbf{C}| \leq |\mathbf{P}|$, this is trivial. As per Definition 4.3, all processors are identical and fully connected, which means that it does not matter how the clusters are mapped onto the processors. Moreover, even the scheduling of phase three is unnecessary, as the scheduling is given implicitly by the clustering $\mathcal{C}$.

This is of course different, in the more likely case that there are more clusters than processors, $|\mathbf{C}| > |\mathbf{P}|$, which is assumed in the following. Note that all presented clustering algorithms do not try to minimize the number of clusters.

### 5.4.1 Assigning Clusters to Processors

As was mentioned before, scheduling is a trade-off between minimizing the communication costs and maximizing the concurrency of the task execution. The clustering phase is all about minimizing the communication costs; hence, it is intuitive to maximize the concurrency of the task execution in the cluster to processor mapping. Achieving high concurrency is equivalent to balancing the load across the processors.

This reminds one of a variant of the classic bin packing problem (Cormen et al. [42], Garey and Johnson [73]): given $n$ objects, with $s_i$ the size of object $i$, and $m$ bins, the problem is to find a distribution of the objects over the $m$ bins so that the maximum fill height of the bins is minimal. While bin packing is also an NP-hard problem (Garey and Johnson [73]), there exist many heuristics for its near optimal solution.

In order to benefit from this wealth of heuristics, it is only necessary to cast the cluster mapping problem as a bin packing problem. This is straightforward:

- The clusters are the objects.
- The size $s_i$ of a cluster $C_i$ is the total computation load of its nodes,

$$w(C_i) = \sum_{n \in \mathbf{V}: proc(n) = C_i} w(n). \tag{5.23}$$

- The processors are the bins.

With this formulation, bin packing heuristics can be employed for the cluster to processor mappings. Consider the two examples given in the following.

***Wrap Mapping*** First, the clusters are ordered in decreasing order of their total weight $w(C)$. Then, starting with the first, each cluster is assigned to a distinct processor. When all processors have already been assigned one cluster, the algorithm returns to the first processor (i.e., it wraps around) and repeats the process until all clusters have been assigned (Darte et al. [52]). Algorithm 19 outlines this approach. Such a load balancing approach is also referred to as round robin. Its complexity is $O(\mathbf{V} + \mathbf{C} \log \mathbf{C})$, where calculating the total weights for all clusters is $O(\mathbf{V})$, sorting the clusters is $O(\mathbf{C} \log \mathbf{C})$ (e.g., Mergesort, Cormen et al., [42]), and the actual cluster to processor assignment is $O(\mathbf{C})$.

***Algorithm 19 Wrap Mapping of Clusters* C *onto Processors* P**

**Require:** Let $\{P_1, P_2, \ldots, P_{|\mathbf{P}|}\}$ be set of processors $\mathbf{P}$

Sort clusters $\mathbf{C}$ in decreasing order of their total computation weight $w(C)$; let $C_1, C_2, \ldots, C_{|\mathbf{C}|}$ be the final order

**for** $i = 1$ to $|\mathbf{C}|$ **do**

Map cluster $C_i$ onto processor $P_{i \bmod |\mathbf{P}|}$

**end for**

***Load Minimization Mapping*** Essentially, this algorithm is just a special case of list scheduling with start time minimization (Section 5.1). To apply list scheduling to this problem, it suffices to consider each cluster as an independent node. In the first phase they are ordered in decreasing order of their total weight $w(C)$, identical to the wrap mapping. In the second phase they are assigned to the processor with the least load, that is, earliest finish time in list scheduling terminology. In comparison to wrap mapping, the actual assignment of the clusters to the processors increases to $O(\mathbf{C} \log \mathbf{P})$, using a priority queue for the processors (e.g., heap, Cormen et al. [42]). With the above conjecture that $|\mathbf{C}| > |\mathbf{P}|$, however, the total complexity is not affected and remains $O(\mathbf{V} + \mathbf{C} \log \mathbf{C})$.

Gerasoulis et al. [75], proposed a variant of wrap mapping, where the clusters with above average total weight are separated from the other clusters.

A quite different technique was proposed by Sarkar [167]. Essentially, he suggested to use list scheduling to assign the clusters to the processors. As usual, the nodes are first ordered into a priority list. While iterating over the list, each node is assigned to the processor that results in the smallest increase of the schedule length. The crucial difference from ordinary list scheduling is that as soon as one node of a cluster $C$ is assigned to a processor $P$, all other nodes belonging to the same cluster $C$ are also assigned to processor $P$. A major difference from the load balancing heuristics discussed earlier stems from the fact that communication costs are also taken into account, namely, when determining the schedule length increase.

### 5.4.2 Scheduling on Processors

The third phase of Algorithm 14 is algorithmically identical to the second phase of scheduling with a given processor allocation, Algorithm 13, Section 5.2. At this point each node has already been allocated to a physical processor. It only remains to order the nodes, for which any node priority scheme can be utilized.

Potentially, the node ordering can make use of the already existing partial node orders as established by the clustering $\mathcal{C}$. However, to the author's best knowledge, no such scheme has yet been proposed.

***Performance*** A question that naturally imposes itself is how clustering based scheduling algorithms, as outlined by Algorithm 14, perform in comparison to single-phase algorithms like list scheduling. The striking answer is that it is unknown. To the author's best knowledge, no direct experimental comparison has been performed for such algorithms and a limited number of target system processors.

While there are many comparisons of scheduling algorithms (e.g., Ahmad et al. [5, 6] Gerasoulis and Yang [76], Khan et al. [103], Kwok and Ahmad [111, 112] McCreary et al. [136]), heuristics are compared in classes. Clustering algorithms are only compared against other clustering algorithms or against algorithms for an unlimited number of processors. They are not compared as being the first phase of multiphase scheduling algorithms for a limited number of processors.

As discussed at the beginning of Section 5.3, there are good intuitive arguments that multiphase scheduling algorithms can produce better schedules than single-phase algorithms; yet, to the author's best knowledge, there is no experimental evidence. Hence, the reader is impudently asked for help with Exercise 5.8, which requests such a comparison.

## 5.5 CONCLUDING REMARKS

In this chapter, the two major classes of scheduling heuristics have been analyzed, namely, list scheduling and clustering. A myriad of algorithms belonging to either one of these two classes (or to both as discussed in Section 5.3.4) has been proposed in the past. In face of this, the discussion concentrated on the essential and common concepts and techniques encountered in these algorithms. In Section 6.3, Chapter 7, and Chapter 8, it will be seen how these fundamental heuristics, especially list scheduling, can be employed under more sophisticated system models.

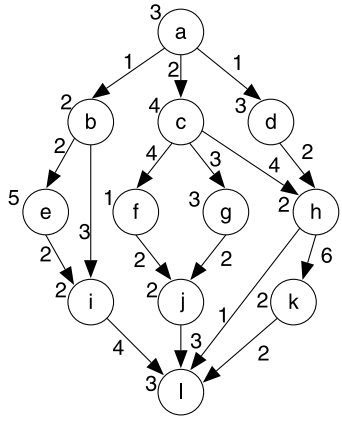
Chapter 6 looks at more advanced scheduling techniques, which can be used in combination with the fundamental heuristics. Furthermore, it returns to the more theoretic aspects of task scheduling, like integrating heterogeneous processors and studying the complexity of variations of the general scheduling problem.

This chapter concludes with some bibliographic notes. A comprehensive survey of task scheduling algorithms can be found in Kwok and Ahmad [113]. Comparisons of algorithms are often a good starting and orientation point for studying task scheduling; for example, see Ahmad et al. [6], Gerasoulis and Yang [76], Khan et al. [103], Kwok and Ahmad [111], and McCreary et al. [136]. Other comprehensive publications that (at least in part) study task scheduling algorithms and its theory are by Chrétienne et al. [34], Coffman [37], Cosnard and Trystram [45], Darte et al. [52], El-Rewini and Abd-El-Barr [60], El-Rewini and Lewis [64], and El-Rewini, Lewis, and Ali [65].

## 5.6 EXERCISES

**5.1** When nodes are ordered according to a priority scheme (e.g., bottom level), it is possible that some nodes have the same priority. Propose tie breaking metrics for these cases. (Suggestion: Consider the edges that are incident on the nodes in question.)

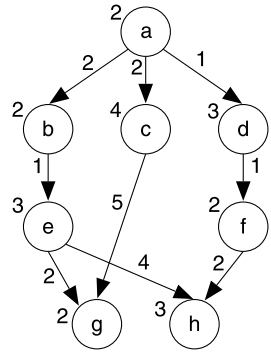
**5.2** Use list scheduling with start time minimization to schedule the following task graph on four processors:




**(a)** The nodes shall be ordered in alphabetical order. What is the resulting schedule length?

**(b)** Now order the nodes according to their bottom levels and repeat the scheduling. What is the resulting schedule length?

**5.3** Use list scheduling with dynamic priorities to schedule the following task graph on three processors:




Dynamically compute the priorities of the free nodes by evaluating the start time for every free node on every processor. That node of all free nodes that allows the earliest start time is selected together with the processor on which this time is achieved. What is the resulting schedule length?

**5.4** Create a linear clustering for the task graph of Exercise 5.2. Draw the *scheduled* task graph (Definition 5.5) for the final clustering. What is its dominant sequence?

**5.5** The following questions are about linear clustering (Section 5.3.2):

**(a)** What is the general advantage of linear clustering in comparison with other clustering approaches?

**(b)** What is the practical advantage of merging in each step the critical path of the unexamined graph?

**(c)** After merging the nodes of a critical path *cp* into one cluster, why is it necessary to zero all edges incident on the nodes, and not only the ones that are part of the path?

**5.6** Use single edge clustering to cluster the task graph of Exercise 5.2. Consider the edges in decreasing order of their weights. Break ties between equally weighted edges by giving preference to the edge whose origin node identifier comes earlier in the alphabet. Break a further tie by considering the target nodes in the same way.

Draw the *scheduled* task graph (Definition 5.5) for the final clustering. What is its dominant sequence?

**5.7** In single edge clustering, the edges are ordered according to a priority scheme, which can be static or dynamic. As a static scheme, sorting the edges in decreasing order of their weights is intuitive and simple. Suggest other static edge orderings or priority schemes. Justify your suggestions.

You might want to review how nodes are ordered; see Sections 4.4 and 5.1.3.

**5.8** Research: Experimentally compare single-phase with multiple-phase scheduling heuristics for a limited number of processors. For single-phase heuristics, considere variants of list scheduling with different priority schemes, static as well as dynamic. For multiphase scheduling, consider the three-phase clustering based algorithm as discussed in Section 5.3.1. Employ different clustering heuristics in the first phase, as well as different cluster-to-processor mappings in the second phase.

**(a)** Implement algorithms.

**(b)** Experimentally compare them by scheduling a large set of different task graphs.

**(c)** Document results in an article.

**(d)** Publish article in peer-reviewed conference or journal.