2.6 Two-Step Clustering Heuristics

2.6.1 Heuristics for the Clustering Phase

We have seen in Example 3 that it is difficult to trade-off parallelism and communication, even in the presence of unlimited resources. The idea of two-step methods is the following. In the first step, use heuristics to group tasks into clusters. This clustering operation is made assuming unlimited resources, to simplify things. In the second step, clusters will be allocated to available processors, and the final ordering of the tasks will be computed. The basic rule of the game is that all the tasks of a given cluster will be allocated to the same processor. We can think of a virtual processor per cluster in the first phase, while several clusters will be allocated to physical processors in the second phase. Why is clustering a useful heuristic? Sarkar [108] gives the following justification: “If tasks are scheduled on the same processor on the best possible architecture with unbounded number of processors, then they should be scheduled on the same processor in any other architecture”. Sarkar’s argument, although not true in every case, is very intuitive, and clustering techniques have been widely explored.

In this section, we first present various heuristics for the first step, the clustering phase. Then we present heuristics for the second step, namely, allocating and scheduling tasks onto physical processors. We prove no optimality theorem here; please wait until Section 2.7.

2.6.2 Two-Step Clustering Heuristics

We survey three widely studied heuristics for the clustering phase: Kim and Browne’s linear clustering [69], Sarkar’s greedy clustering [108], and (the underlying principle of) Yang and Gerasoulis’s dominant sequence clustering [124].

To be precise in the following discussion, let us restate the definitions of bottom and top levels in the presence of clusters (see Section 1.4 for the definition of $U(u)$ and $B(u)$ for a vertex $u$).

Definition 13 (Clustering, Bottom/Top Level, Estimated Parallel Time)

Let $G = (V, E, \mathcal{C})$ be a DAG.

1. A clustering is a function $\mathcal{C} : V \rightarrow \mathcal{P}$ (each task is assigned a cluster number). We also use $\mathcal{C}$ to denote the induced partitioning of the set of tasks into clusters.

2. Given a clustering $\mathcal{C}$, we define top and bottom levels as follows:

   - For $u \in V$, $U(u, \mathcal{C}) = \max\{u|v\in \text{SUCC}(u), \mathcal{C}(u) = \mathcal{C}(v)\}$
     and $\max\text{SUCC}(u) = \max\{b(u) + c(v)|u \in \text{SUCC}(v), \mathcal{C}(u) = \mathcal{C}(v)\}$.

   - For $u \in V$, $B(u, \mathcal{C}) = \max\{b(u) + u|v\in \text{PRED}(u), \mathcal{C}(u) = \mathcal{C}(v)\}$
     and $\max\text{PRED}(u) = \max\{b(u) + u|v\in \text{PRED}(v), \mathcal{C}(u) = \mathcal{C}(v)\}$.

3. Given a clustering $\mathcal{C}$, we define the estimated parallel time as

   $$EPT(\mathcal{C}) = \max\{EPT(u)|u \in V\}$$

In this definition, bottom and top levels are computed according to the clustering; we add communication costs if the tasks do not belong to the same cluster. The estimated parallel time $EPT$ is the length of the longest path in the graph using this rule. In other words, the current clustering indicates...
which communication costs should be taken into account given already-made decisions: we keep communication costs that link tasks belonging to different clusters so far (and we zero out those that link tasks grouped into the same cluster). Because some clusters will be merged throughout the clustering, this is only an estimation (the current estimation) of the final parallel time. Finally, there remains a problem to solve: what is the ordering of the tasks inside a given cluster? All the tasks that belong to the same cluster are executed by the same processor. But if dependences do not induce a total order within the cluster, we must define such a total ordering. For instance in Example 3, if \( \{T_1, T_2, T_3\} \) becomes a cluster at some point, dependences imply to execute task \( T_1 \) before the other two tasks \( T_2 \) and \( T_3 \), which can be executed in any order. Deciding which is executed first may have a tremendous impact on the computation of longest paths in the graph. We will discuss further the scheduling problem inside clusters when we present Sarkar’s heuristic.

The three surveyed heuristics are iterative. The initial clustering \( C_0 \) is one task per cluster (maximum parallelism), and the initial parallel time is \( \text{EPT}(C_0) \). At step \( i \), we refine the clustering \( C_{i-1} \) to build a new clustering \( C_i \) so that \( \text{EPT} \) decreases, or at least does not increase: \( \text{EPT}(C_i) \leq \text{EPT}(C_{i-1}) \). Termination obeys different criteria: processing of all vertices, processing of all edges, or stop when the clustering rule leads to an increase of \( \text{EPT} \).

**Kim and Browne’s Linear Clustering**

The most natural idea to decrease the EPT is to group all the tasks of a longest path into a single cluster. This is the principle of Kim and Browne’s heuristic [69], which can be summarized as follows:

1. Initially, all edges are marked unexamined.

2. Compute top and bottom levels \( h(u, C_0) \) and \( h(v, C_0) \) for each \( v \in V \). Compute \( \text{EPT}(C_0) \). Select a longest path in the graph whose length is \( \text{EPT}(C_0) \) and build \( C_1 \) from \( C_0 \) by grouping all tasks in the selected path into the same cluster. Mark all edges incident to these tasks as examined.

3. While there remain unexamined edges, redo step 2 on the graph of unexamined edges (and incident nodes). Compute \( C_i \) from \( C_{i-1} \) by grouping the tasks of a longest path of \( C_{i-1} \) into a single cluster.

Let us follow Kim and Browne’s heuristic on Example 3. Given \( C_0 \), we have already computed \( \text{EPT}(C_0) = 14 \). \( T_1 \rightarrow T_2 \rightarrow T_7 \) is the single longest path. Therefore we let \( C_1 = \{\{T_1, T_2, T_7\}, \{T_3\}, \{T_4\}, \{T_5\}, \{T_6\}\} \). We are led to the graph of Figure 2.8. As \( \text{EPT}(C_1) = 13.5 \), we accept this clustering. The subgraph under consideration for the second step is circled by the dashed line in Figure 2.8: vertices are reduced to \( \{T_5, T_6, T_7\} \). The longest path now is \( T_1 \rightarrow T_2 \rightarrow T_7 \) therefore \( C_2 = \{\{T_1, T_2, T_7\}, \{T_3\}, \{T_4\}, \{T_5\} \} \). Again, we accept this clustering because \( \text{EPT}(C_2) = 12.5 \). There does not remain any unexamined edge, therefore \( C_2 \) is the final clustering.

**Sarkar’s Greedy Clustering**

Another natural idea to decrease the EPT is to (try to) zero out costly communications. This is the principle of Sarkar’s heuristic [108], which can be...
2.6. Two-Step Clustering Heuristics

We consider the edges one after the other in the sorted list summarized as follows:

1. Sort the edges of the cDAG in descending order of edge (communication) costs.
2. For each edge (in the order above), zero out the edge if the estimated parallel time does not increase. Formally, at step \( i \), let \( e = (a, v) \) be the edge under consideration: we build \( C_i \) by merging the two clusters \( C_{\omega_1}(v) \) and \( C_{\omega_2}(v) \). We do this merging only if \( EPT(C_i) \leq EPT(C_{\omega_1}) \); otherwise we let \( C_i = C_{\omega_1} \).

Let us follow Sarkar’s heuristic on Example 3. We have \( EPT(C_6) = 14 \).

We consider the edges one after the other in the sorted list \( \{(T_1, T_2), (T_3, T_1), (T_3, T_6), (T_2, T_1), (T_1, T_6), (T_6, T_1), (T_5, T_1), (T_5, T_6)\} \).

1. \( u(T_1, T_2) = 5 \). If we group \( T_1 \) and \( T_2 \) in the same cluster, the EPT decreases from 14 to 13.5 (\( T_1 \rightarrow T_3 \rightarrow T_1 \rightarrow T_6 \rightarrow T_2 \) becomes the longest path). Therefore, we do zero out the edge \( (T_1, T_2) \). In other words,
   \[ C_1 = \{(T_1, T_2), (T_3, T_1), (T_1, T_6), (T_6, T_1), (T_5, T_1), (T_5, T_6)\} \]

2. \( u(T_3, T_2) = 4 \). If we group \( T_3 \) and \( T_2 \) in the same cluster, the EPT decreases from 13.5 to 12.5 (\( \{T_1 \rightarrow T_3 \rightarrow T_5 \rightarrow T_6 \rightarrow T_2\} \) becomes the longest path). Therefore, we do zero out the edge \( (T_3, T_2) \), and
   \[ C_2 = \{(T_1, T_2), (T_3, T_1), (T_1, T_6), (T_6, T_1), (T_5, T_1), (T_5, T_6)\} \]

3. \( u(T_3, T_5) = 3 \). If we group \( T_3 \) and \( T_5 \) in the same cluster, we have a new cluster \( \{T_3, T_1, T_6\} \). How do we schedule it to compute \( EPT(C_3) \)? This cluster is not linear: \( T_4 \) and \( T_5 \) are independent, hence they can be ordered in any way. However, we need an ordering to compute longest paths for the current clustering; the same processor will execute sequentially the three tasks \( \{T_3, T_1, T_6\} \) in the new cluster, and the ordering of these tasks has an impact on the (estimated) parallel time. The idea is to introduce “a virtual arrow” between \( T_4 \) and \( T_5 \). To determine which task should be given priority, we can use the values the bottom levels had at the previous step. Here, because \( \mu(T_1, C_2) = \mu(T_2, C_2) = G_5 \), we break the tie by giving priority to \( T_1 \) and we draw a virtual arrow \( T_4 \rightarrow T_5 \). Now that we have ordered tasks inside the new cluster \( \{T_3, T_1, T_6\} \), we can compute \( EPT(C_3) = 11.5 \). Therefore we do zero out the edge \( (T_3, T_5) \), and
   \[ C_3 = \{(T_1, T_2), (T_3, T_1, T_6), (T_6, T_1)\} \]

4. \( u(T_4, T_2) = 2 \). Merging the two clusters \( \{T_4, T_2\} \) and \( \{T_1, T_6\} \) causes no scheduling problem, as the new cluster remains linear. We obtain \( EPT(C_4) = 11.5 = EPT(C_3) \), and we do accept to zero out the edge \( (T_2, T_1) \). We have
   \[ C_4 = \{(T_1, T_2), (T_3, T_1, T_6), (T_6, T_1)\} \]

5. \( u(T_1, T_6) = 1.5 \). Merging the two clusters \( \{T_1, T_2, T_3\} \) and \( \{T_6\} \) (and scheduling \( T_6 \) in last position) leads to \( EPT(C_5) = 10 \). Therefore
   \[ C_5 = \{(T_1, T_2, T_3), (T_6, T_1)\} \]

6. The edge \( (T_5, T_6) \) is already zeroed out, \( C_6 = C_5 \).

7. Finally, zeroing out either the edge \( (T_1, T_6) \) or \( (T_3, T_5) \) would group all tasks on the same cluster, and EPT would increase to 13. The final clustering \( C_6 \) is represented in Figure 2.11. The corresponding scheduling with two processors is illustrated in Figure 2.12.

Let us emphasize an important point (see the discussion at step 3). When merging two clusters, we have a scheduling problem. If the new cluster is not linear, we have to build an ordering of the tasks inside the cluster. To do so, we use a heuristic within the heuristic. We have chosen to order tasks according to the highest bottom level first heuristic, where bottom levels refer to the values computed at the previous step.

There are \( |E| \) steps in Sarkar’s heuristic, and at each step we traverse the graph to compute new top and bottom levels; hence the complexity of the heuristic is \( O(|E|(|V| + |E|)) \).

---

\(^2\) This variant of Sarkar’s original scheduling strategy is quite natural as it is similar to CP scheduling. It has been proposed by Gerasoulis and Yang [48].
2.6. Two-Step Clustering Heuristics

A third natural idea to decrease the EPT is to zero out one edge of the current longest path, i.e., the dominant sequence in the graph (and to accept this zeroing only when EPT does not increase). The iterative scheme for dominant sequence clusterings is the following:

1. Initially, all edges are marked unexamined. Compute top and bottom levels $\mathcal{U}(v,G_i)$ and $\mathcal{B}(v,G_i)$ for each $v \in V$. Determine the dominant sequence $DS_0$ (a path of length $EPT(G_0)$). Let $i = 0$.

2. While there remains unexamined edges, do
   - Zero an edge in the dominant sequence $DS_i$ if the estimated parallel time does not increase. Mark this edge examined.
   - Increment $i$ and find a new dominant sequence $DS_{i+1}$.

Which edge in the $DS$ can we select for the zeroing? There are several possible variants:

1. Select the edge on the current longest path whose zeroing will decrease EPT as much as possible. This is a very costly strategy, as it implies many computations of top and bottom levels per step.
2. Select the edge on the current longest path whose weight is maximal. This is a greedy strategy that trades off complexity and efficiency. It can be viewed as a variant of Sarkar’s heuristic.
3. Select the first edge of the longest path. This strategy is cheap and very simple (maybe too simple to produce good results?).

We briefly discuss the second variant. If we select the most costly edge on one of the longest paths, we have a heuristic whose complexity will be the same as Sarkar’s heuristic: one traversal per step, hence a worst-case complexity $O(|E|(|V|^+ + |E|))$. For Example 3, we follow the heuristic step by step:

1. With the initial clustering, the longest path is $T_1 \to T_2 \to T_7$. If we zero out the maximum weight edge $(T_1, T_2)$, we decrease the EPT to $13.5$.
2. The longest path is now $T_1 \to T_3 \to T_4 \to T_5 \to T_6 \to T_7$: The maximum weight edge is $(T_3, T_4)$. Zeroing it decreases the EPT to $12.5$.
3. The longest path is now $T_1 \to T_3 \to T_5 \to T_6 \to T_7$: The maximum weight edge is $(T_6, T_7)$. Zeroing it decreases the EPT to $11.5$, as in Sarkar’s heuristic (we add the virtual edge $T_1 \rightarrow T_3$).
4. The longest path is now $T_1 \to T_3 \to T_5 \to T_6 \to T_7$: We zero out the maximum weight edge $(T_6, T_7)$ to obtain a longest path of length $10$.
5. The longest path remains the same. There are two nonzereod edges, $(T_1, T_3)$ and $(T_6, T_7)$, with same weight $1$. Because the EPT would increase, we refuse to zero out $(T_1, T_3)$ ...
6. ... but we do accept to zero out $(T_6, T_7)$ because the EPT decreases to $9$.
7. Finally, we refuse to zero out the last edge $(T_2, T_7)$.

The final clustering is illustrated in Figure 2.13. It turns out that this heuristic is optimal for our example; we obtain the scheduling of Figure 2.2 with two processors. Of course we make the same warning as before. No definitive conclusion can be drawn from this single comparison.

Figure 2.11: Final clustering in Sarkar’s heuristic.

Figure 2.12: Scheduling Sarkar’s clustering with two processors.
Two-Step Clustering Heuristics

Yang and Gerasoulis’s DSC

First we give a short overview of Yang and Gerasoulis’s DSC heuristic. Then we present the heuristic in more detail (the reader may want to skip this presentation because it is rather technical).

Overview. As stated earlier, a dominant sequence clustering will cost at most \(O(|E|(|V|-|E|))\) operations, which is the same cost as Sarkar’s heuristic. For large graphs this is too high a cost. This is why Yang and Gerasoulis [124] have advocated a variant where some edges that do not belong to a cluster are still zeroed. Their heuristic cannot be easily described. In a word, the key idea is to relax the zeroing condition: a selected edge is zeroed if the top level of its head decreases (rather than asking for a decrease of EPT). A heap of “free” tasks is maintained to allow us to schedule the highest priority tasks in logarithmic time. Overall, the total complexity of the heuristic is \(O(|V| + |E| + k|E| |V|)\), which makes it very attractive in large-scale applications.

Detailing the Heuristic. Our description of Yang and Gerasoulis’s DSC is mainly borrowed from Chrétienne and Picouleau [23]. As with all the previous heuristics, Yang and Gerasoulis’s DSC is iterative:

1. The initial clustering \(C_0\) assigns one task per cluster (maximum parallelism), and the initial parallel time is EPT(\(C_0\)). Initially, all tasks are marked unexamined.

2. At each step, we have two different types of clusters in the current clustering: the examined clusters, whose content may only increase, and the nonexamined clusters, which are singletons that may be later merged into an examined cluster. Each task of an examined cluster is said to be inserted.

3. In the initial clustering \(C_0\), clusters (singletons) that contain an entry task (a task without predecessor) are marked examined, otherwise they are marked nonexamined.

4. At each step, a task may be inserted only if all its predecessors have already been inserted.

5. The algorithm ends when all tasks have been inserted.

Consider the clustering \(C_i\) at step \(i\), and let \(T\) be a noninserted task:

- \(T\) is said to be free if all its predecessors have been inserted. The priority of a free task is the sum of its bottom level \(bl(t, C_i)\) and its top level \(tl(t, C_i)\).
- \(T\) is said to be partially free if at least one of its predecessors has been inserted and at least one other has not. The priority of a partially free task is the sum of its bottom level \(bl(t, C_i)\) and its “examined” top level \(cl(t, C_i)\) where \(cl(t, C_i)\) is defined as the maximum weight of a path leading to \(T\) and is composed only of inserted tasks (except \(T\)).

As specified earlier, only free tasks can be inserted. We use the previous priorities to decide which free task will be inserted. We determine the following two tasks:

- the free task \(T_f\) with highest priority \(\alpha\)
- and the partially free task \(T_{pf}\) with highest priority \(\beta\)

Two cases can occur:

1. \(\alpha \geq \beta\). The heuristic explores all (inserted) predecessors of \(T_f\) until it finds one of them, say \(T_{prf}\) such that adding \(T_f\) as the last task of the (examined) cluster that contains \(T_{prf}\) allows us to schedule \(T_f\) strictly sooner in the resulting new clustering (in other words if the top level of \(T_f\) decreases with this insertion). If one of these tests is successful, we adopt the new clustering, otherwise we leave \(T_f\) in its singleton cluster, which becomes examined.
2.6. Two-Step Clustering Heuristics

2. \( \alpha < \beta \). The heuristic explores all inserted predecessors of \( T_{pf} \) until it finds one of them, say \( T_{pre} \), such that adding \( T_{pf} \) as the last task of the (examined) cluster that contains \( T_{pre} \) allows us to decrease the examined top level of \( T_{pf} \). If one of these tests is successful, we “freeze” the cluster containing \( T_{pre} \), becomes free. Using the same procedure as before, except that the cluster containing \( T_{pre} \) is no longer a candidate, we try to insert \( T_f \) as the last task of an examined cluster.

We do not go further into the description of the heuristic. In fact, rather than choosing any predecessor, Yang and Gerasoulis choose the one that leads to the largest diminution of the current top level. They use a procedure similar to that of Exercise 2.8. The use of the estimated top level is a key point to decrease the complexity of the heuristic. We refer the reader to [124] for a detailed presentation, including several variants and comparisons.

2.6.2 From Clustering to Scheduling with Limited Resources

When we have a clustering, we are not done yet. Clustering is the first step of a two-step process:

1. Cluster the task graph first. All tasks in a cluster will execute in the same processor.
2. (i) Assign the clusters to physical processors.
   (ii) Order the execution of the tasks in each (physical) processor.

We have just dealt with the first step, the clustering. To tackle the second step, the cluster assignment and the final ordering of tasks, we rely on heuristics! Indeed, the second step itself is NP-complete too, even if there are as many physical processors as clusters (Exercise 2.4 asks you to show this).

Cluster Assignment

When using a clustering heuristic, we assume the availability of unlimited resource. There is no way to predict the final number of clusters. Most likely, we end up with more clusters than available physical processors. Hence, we have a need to assign many clusters to the same physical processor. A simple strategy is the following:

1. Compute the computational load (the sum of the task weights) for each cluster.
2. Sort the clusters in an increasing order of their loads.

3. Use a load-balancing algorithm so that each processor has approximately the same load. A first algorithm is the wrap mapping\(^3\) of clusters to processors. An alternative suggested by Gerasoulis, Jiao, and Yang [47] is to map all clusters with a load greater than the average load on separate processors, while using a wrap mapping for the remaining clusters and processors.

Another approach is used by Sarkar [108]. The idea is to scan the tasks according to a priority list (bottom levels). At each step, a task (along with all the other tasks of the cluster that it belongs to) is assigned to the processor that minimizes the increase in the estimated parallel time (EPT). The EPT is determined by allocating the already scheduled clusters on the physical processors and the yet-unscheduled clusters on extra (virtual) processors.

Final Task Scheduling

Finally, we have to schedule all tasks. Note that even if we want to keep a specified ordering inside each cluster (because the cluster is linear or because we computed an ordering during the clustering step), we still have to compute a scheduling, because there are many clusters per processor.

But now each task is assigned a processor number, so those edges that require a communication cost are perfectly known. To order the execution of tasks within each processor such that the total parallel time is minimized, we can use a list scheduling solution. Each processor has a priority list of tasks based on bottom levels (critical paths). Of course in the computation of bottom levels we include only the communication costs that are not internalized. There are two natural strategies:

- **Free list scheduling.** At each step, we schedule the tasks with highest priority among the free tasks. Remember that a task is free when all its predecessors have been executed.
- **Ready list scheduling.** At each step, we schedule the highest priority tasks among the ready tasks. A task is ready when it can start execution immediately. More precisely, a task is ready at step \( t \) if it is free, and if \( \alpha(T_i^*) + \underbrace{w(T) + c(T_i^*, T)}_{p_i} \leq t \) for each predecessor \( T_i^* \) of \( T \) that has not been allocated to the same processor as \( T \).

According to Yang and Gerasoulis [123], ready list scheduling “performs better” than free list scheduling. Few optimality results are known,

\(^3\)This means that the cluster ranked in position \( i \) is assigned to processor number \( i \cdot n \cdot n \) where \( n \) is the total number of physical processors.
2.7. Linear Clustering

However, for example, we have to slightly change the priority rule (use a modified bottom level computation where the weight of the task is not included) to prove the optimality of ready list scheduling for fork and join graphs [123].

2.6.3 Clustering Epilogue

At the end of this presentation of clustering techniques, we feel embarrassed. For each step, we have to choose a heuristic. Inside each heuristic, we have to select several subheuristics. Given a task graph, there are very few arguments to help us make the right choice. For the reader who likes theorems and bounds, there are some in the next section.

2.7 Linear Clustering

In this section, we come back to linear clustering (which we informally introduced to describe Kim and Browne’s heuristic). We have seen that linear clusters make scheduling much easier. From a theoretical point of view, it is nice to know that there always exists a linear optimal clustering when the task graph is coarse-grain, i.e., when any computation costs more than any communication. This important result is due to Gerasoulis and Yang [49]. We start with the definition of a linear cluster.

Definition 14 (Linear cluster) Let $G = (V, E, w_c)$ be a cDAG. A linear cluster is a set of tasks $C$ that belong to a dependence path of $G$: $C = \{T_1, T_2, \ldots, T_m\}$ where $(T_i, T_{i+1}) \in E$ for $1 \leq i < m$. A linear clustering of $G$ is a partition of $V$ into linear clusters.

Remember that the granularity of a cDAG $G = (V, E, w_c)$ is the computation-to-communication ratio $g(G) = \frac{\text{number of nodes}}{\text{number of edges}}$. $G$ is said to be coarse-grain if $g(G) \geq 1$ (see Definition 10). We start with the optimality of linear clusterings for coarse-grain graphs, and we deduce a trivial bound for arbitrary graphs.

Theorem 9 Let $G = (V, E, w_c)$ be a coarse-grain cDAG. There exists a linear clustering for $G$ that is optimal.

Proof The proof is iterative. Given an optimal clustering, we make it more and more linear, so to speak, without increasing the makespan. At each step, we take a nonlinear cluster $NLC$ and extract some tasks out of it. The extracted tasks will belong to a dependence path of $G$. The cluster is