DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors

Tao Yang, Member, IEEE, and Apostolos Gerasoulis, Member, IEEE

Abstract—We present a low-complexity heuristic, named the dominant sequence clustering algorithm (DSC), for scheduling parallel tasks on an unbounded number of completely connected processors. The performance of DSC is, on average, comparable to, or even better than, other higher-complexity algorithms. We assume no task duplication and non-zero communication overhead between processors. Finding the optimum solution for arbitrary directed acyclic task graphs (DAG’s) is NP-complete. DSC finds optimal schedules for special classes of DAG’s, such as fork, join, coarse-grain trees, and some fine-grain trees. It guarantees a performance within a factor of 2 of the optimum for general coarse-grain DAG’s. We compare DSC with three higher-complexity general scheduling algorithms: the ETF by Hwang, Chow, Anger, and Lee; Sarkar’s clustering algorithm; and the MD by Wu and Gajski. We also give a sample of important practical applications where DSC has been found useful.

Index Terms—Clustering, directed acyclic graph, heuristic algorithm, optimality, parallel processing, scheduling, task precedence

I. INTRODUCTION

We study the scheduling problem of directed acyclic weighted task graphs (DAG’s) on unlimited processor resources and a completely connected interconnection network. The task and edge weights are deterministic. This problem is important in the development of parallel programming tools and compilers for scalable multiple-instruction, multiple-data (MIMD) architectures [17], [19], [21], [23]. Sarkar [17] has proposed a two-step method for scheduling with communication.

1) Schedule an unbounded number of a completely connected architecture. The result of this step will be clusters of tasks, with the constraint that all tasks in a cluster must execute in the same processor.

2) If the number of clusters is larger than the number of processors, then merge the clusters further to the number of physical processors, and also incorporate the network topology in the merging step.

In this paper, we present an efficient algorithm for the first step of Sarkar’s approach. Algorithms for the second step are discussed elsewhere [23]. The objective of scheduling is to allocate tasks onto the processors and then order their execution so that task dependence is satisfied and the length of the schedule, known as the parallel time, is minimized. In the presence of communication, the complexity of the above scheduling problem has been found to be much more difficult than in the classical scheduling problem, where communication is ignored. The general problem is NP-complete, and even for simple graphs, such as fine-grain trees or the concatenation of a fork and a join, the complexity is still NP-complete [3], [15], and [17]. Only for special classes of DAG’s, such as fork, join, fork, and coarse-grain tree, special polynomial algorithms are known [2], [4].

There have been two approaches in the literature addressing the general scheduling problem. The first approach considers heuristics for arbitrary DAG’s, and the second studies optimal algorithms for special classes of DAG’s. When task duplication is allowed, Papadimitriou and Yannakakis [15] have proposed an approximate algorithm for a DAG with equal task weights and equal edge weights, which guarantees a performance within 50% of the optimum. This algorithm has a complexity of $O(v^2(v \log v + e))$, where $v$ is the number of tasks and $e$ is the number of edges. Kruatrachue and Lewis [13] have also given an $O(v^2)$ algorithm for a general DAG based on task duplication. One difficulty in allowing task duplication is that duplicated tasks may require duplicated data among processors, and thus the space complexity could increase when executing parallel programs on real machines.

Without task duplication, many heuristic scheduling algorithms for arbitrary DAG’s have been proposed in the literature (e.g., [12], [17], [19]). A detailed comparison of four heuristic algorithms is given in [9]. One difficulty with most existing algorithms for general DAG’s is their high complexity. As far as we know, no scheduling algorithm exists that works well for arbitrary graphs, finds optimal schedules for special DAG’s, and also has a low complexity. We present one such algorithm in this paper with a complexity of $O((v + e) \log v)$, called the Dominant Sequence Clustering (DSC) algorithm. We compare DSC with ETF algorithm by Hwang, Chow, Anger, and Lee [11], and discuss its similarities and differences with the MD algorithm proposed by Wu and Gajski [19].

The organization is as follows: Section II introduces the basic concepts. Section III describes an initial design of the DSC algorithm and analyzes its weaknesses. Section IV presents an improved version of DSC that takes care of...
the initial weaknesses and analyzes how DSC achieves both low complexity and good performance. Section V gives a performance bound for a general DAG. It shows that the performance of DSC is within 50% of the optimum for coarse-grain DAG's, and it is optimal for join, fork, coarse-grain trees and a class of fine-grain trees. Section VI discusses the related work, presents experimental results and compares the performance of DSC with Sarkar's, ETF, and MD algorithms.

II. PRELIMINARIES

A directed acyclic task graph (DAG) is defined by a tuple $G = (V, E, C, T)$ where $V$ is the set of task nodes and $v = |V|$ is the number of nodes, $E$ is the set of communication edges, $e = |E|$ is the number of edges, $C$ is the set of edge communication costs, and $T$ is the set of node computation costs. The value $c_{i,j} \in C$ is the communication cost incurred along the edge $(n_i, n_j) \in E$, which is zero if both nodes are mapped in the same processor. The value $t_i \in T$ is the execution time of node $n_i \in V$. PRED($n_x$) is the set of immediate predecessors of $n_x$, and SUCC($n_x$) is the set of immediate successors of $n_x$. An example of DAG is shown in Fig. 1(a), with seven tasks $n_1, n_2, \ldots, n_7$. Their execution times are on the right side of the bullets, and edge weights are written on the edges.

The task execution model is the compile time macro-dataflow model. A task receives all input before starting execution in parallel, executes to completion without interruption, and immediately sends the output to all successor tasks in parallel. (See Wu and Gajski [19] and Sarkar [17].) Duplication of the same task in separate processors is not allowed.

Given a DAG and an unbounded number of completely connected processors, the scheduling problem consists of two parts: the task to processor assignment, called clustering in this paper, and the task execution ordering for each processor. A DAG with a given clustering, but without the task execution ordering, is called a clustered graph. A communication edge weight in a clustered graph becomes zero if the start and end nodes of this edge are in the same cluster. Fig. 1(c) shows a clustering of the DAG in Fig. 1(a), excluding the dashed edges for ordering. The tasks $n_1$ and $n_2$ constitute one cluster, and the rest of the tasks constitute another. Fig. 1(b) shows a Gantt chart of a scheduling in which the processor assignment for each task and the starting and completion times are defined. An equivalent way of representing a schedule is shown in Fig. 1(c), called a scheduled DAG. A scheduled DAG contains both clustering and task execution ordering information. The dashed pseudo edges between $n_1$ and $n_4$ in Fig. 1(c) represent the execution ordering imposed by the schedule. The task starting times are not explicitly given in a scheduled DAG, but can be computed by traversing this DAG.

CLUST($n_x$) stands for the cluster of node $n_x$. If a cluster contains only one task, it is called a unit cluster. We distinguish between two types of clusters: the linear and the nonlinear. Two tasks are called independent if there are no dependence paths between them. A cluster is called nonlinear if there are two independent tasks in the same cluster; otherwise, it is called linear. In Fig. 1(c), there are two clusters: The cluster that contains $n_1$ and $n_2$ is linear; the other cluster is nonlinear, because $n_3$ and $n_4$ are independent tasks in Fig. 1(a). A schedule imposes an ordering of tasks in nonlinear clusters. Thus, the nonlinear clusters of a DAG can be thought of as linear clusters in the scheduled DAG if the execution orders between independent tasks are counted as edges. A linear clustering preserves the parallelism present in a DAG, and nonlinear clustering reduces parallelism by sequentializing parallel tasks. A further discussion of this issue can be found in Gerasoulis and Yang [8].

The critical path of a clustered graph is the longest path in that graph, including both nonzero communication edge cost and task weights in that path. The parallel time in executing a clustered DAG is determined by the critical path of the scheduled DAG, not by the critical path of the clustered DAG. We call the critical path of a scheduled DAG the dominant sequence (DS), to distinguish it from the critical path of the clustered DAG. For Fig. 1(c), the critical path of the clustered graph is $(n_1, n_2, n_7)$, and the DS is still that path. If the weight of $n_5$ is changed from 2 to 6, then the critical path of the clustered graph remains the same, $(n_1, n_2, n_7)$, but the DS changes to $(n_5, n_3, n_4, n_6, n_7)$. A path that is not a DS is called a SubDS.

Let $\text{level}(n_x)$ be the length of the longest path from an entry (top) node to $n_x$, excluding the weight of $n_x$ in a DAG. Symmetrically, let $\text{blevel}(n_x)$ be the length of the longest path from $n_x$ to an exit (bottom) node. For example, in Fig. 1(c), $\text{level}(n_1) = 0$, $\text{blevel}(n_1) = 10$, $\text{level}(n_3) = 2$, $\text{blevel}(n_3) = 4$. The following formula can be used to determine the parallel time from a scheduled graph:

$$\text{PT} = \max_{n_x \in V} \{ \text{level}(n_x) + \text{blevel}(n_x) \}.$$  \hspace{1cm} (1)
A. Scheduling as Successive Clustering Refinements

Our approach for solving the scheduling problem with unlimited resources is to consider a scheduling algorithm as performing a sequence of clustering refinement steps. As a matter of fact, most of the existing algorithms can be characterized by using such a framework [9]. The initial step assumes that each node is mapped in a unit cluster. At each step, the algorithm tries to improve on the previous clustering by merging appropriate clusters. A merging operation is performed by zeroing an edge cost connecting two clusters.¹

Sarkar’s Algorithm: We consider Sarkar’s algorithm [17, pp. 123–131] as an example of an edge-zeroing clustering refinement algorithm. This algorithm first sorts the $v$ edges of the DAG in a decreasing order of edge weights, and then performs $v$ clustering steps by examining edges from left to right in the sorted list. At each step, it examines one edge and zeros this edge if the parallel time does not increase. Sarkar’s algorithm requires the computation of the parallel time at each step, and this problem is also NP-complete. Sarkar uses the following strategy for ordering. Order independent tasks in a cluster by using the highest blevel first-priority heuristic, where blevel is the value computed in the previous step. The new parallel time is then computed by traversing the scheduled DAG in $O(v + e)$ time. Since there are $v$ steps, the overall complexity is $O(c(v + v))$. Fig. 2 shows the clustering steps of Sarkar’s algorithm for the DAG in Fig. 1(a). The sorted edge list with respect to edge weights is $\{(n_1, n_6), (n_1, n_7), (n_2, n_7), (n_1, n_3), (n_1, n_5)\}$. Table I traces the execution of this algorithm, where PT stands for the parallel time and PTi is the parallel time for executing the clustered graph at the completion of step $i$. Initially, each task is in a separate cluster, as shown in Fig. 2(a), and the thick path indicates the DS whose length is $PT_0 = 13$. At step 1, edge $(n_4, n_6)$ is examined, and PT remains 13 if this edge is zeroed. Thus, this zeroing is accepted. In step 2, 3, and 4, shown in Fig. 2(c), 2(d), and 2(e), all examined edges are zeroed, because each zeroing does not increase PT. At step 5, edge $(n_2, n_7)$ is examined, and, by zeroing it, the parallel time increases from 10 to 11. Thus, this zeroing is rejected. Similarly, at step 6, zeroing $(n_1, n_3)$ is rejected. At step 7, $(n_5, n_6)$ is zeroed and a pseudo edge from $n_5$ to $n_6$ is added, because after step 6, blevel$(n_3) = 3$ and blevel$(n_5) = 5$. Finally, two clusters are produced with $PT = 10$, shown in Fig. 2(f).

III. AN INITIAL DESIGN OF THE DSC ALGORITHM

A. Design Considerations for the DSC Algorithm

As can be seen in the previous section, Sarkar’s algorithm zeroes the highest communication edge. This edge, however, may not be in a DS, and as a result, the parallel time may not be reduced at all. (See the zeroing of edge $(n_4, n_6)$ in step 1 Fig. 2.) In order to reduce the parallel time, we must examine the schedule of a clustered graph to identify a DS and then try to reduce its length. The main idea behind the DSC algorithm is to perform a sequence of edge zeroing steps with the goal of reducing the length of a DS at each step. The challenge is to implement this idea with low complexity so that it can be used for large task graphs. Thus, we would like to develop an algorithm having the following goals.

G1: The complexity should be low.
G2: The parallel time should be minimized.
G3: The efficiency should be maximized.

There are several difficulties in the implementation of algorithms that satisfy the above goals. ¹

1) The goals G1, G2, and G3 could conflict with each other. For example, the maximization of the efficiency conflicts with the minimization in the parallel time. When such conflicts arise, G1 is given priority over G2 and G3, and G2 is given priority over G3. In other words, we are

<table>
<thead>
<tr>
<th>step</th>
<th>edge examined</th>
<th>PT if zeroed</th>
<th>zeroing</th>
<th>PT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$(n_4, n_6)$</td>
<td>13</td>
<td>yes</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>$(n_4, n_6)$</td>
<td>10</td>
<td>yes</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>$(n_4, n_6)$</td>
<td>10</td>
<td>yes</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>$(n_4, n_6)$</td>
<td>11</td>
<td>no</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>$(n_4, n_6)$</td>
<td>10</td>
<td>yes</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>$(n_4, n_6)$</td>
<td>11</td>
<td>no</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>$(n_4, n_6)$</td>
<td>10</td>
<td>yes</td>
<td>10</td>
</tr>
</tbody>
</table>
interested in algorithms with low complexity that attain the minimum possible parallel time and the maximum possible efficiency.

2) Let us assume that the DS has been determined. A careful selection of edges to be examined and zeroed must be made to avoid high complexity and simultaneously reduce the parallel time. Consider the example in Fig. 2(a). Initially, the DS is \( \{n_1, n_2, n_7\} \). To reduce the length of that DS, we need to zero at least one edge in DS. Hence, we need to decide which edges should be zeroed. We could zero either one or both edges. If the edge \((n_1, n_2)\) is zeroed, then the parallel time reduces from 13 to 10. If \((n_2, n_7)\) is zeroed, the parallel time reduces to 11. If both edges are zeroed, the parallel time reduces to 9.5. Therefore, there are many possible ways of edge zeroing, and we discuss the following three approaches.

- **API: Multiple DS edge zeroing with maximum PT reduction**: This is a greedy approach that will try to get the maximum reduction of the parallel time at each clustering step. However, it is not always true that multiple DS edge zeroing could lead to the maximum PT reduction, because after zeroing one DS edge of a path, the other edges in that path may not be in the DS of the new graph.

- **AP2: One DS zeroing of maximum weight edge**: Considering the fact that a DS could become a SubDS by zeroing only one edge of this DS, instead of multiple-edge zeroing, we could zero only one edge at each step to make smaller reductions in the parallel time, but then perform more steps. For example, we could choose one edge to zero at each step, say, the largest weighted edge in DS. In Fig. 2(a), the length of current DS \( \{n_1, n_2, n_7\} \) could be reduced more by zeroing edge \((n_1, n_2)\) instead of edge \((n_2, n_7)\). Zeroing the largest weighted edge may increase the complexity, but may not necessarily lead to a better solution.

- **AP3: One DS edge zeroing with low complexity**: Instead of zeroing the highest edge weight, we could zero the DS edge that leads to a low-complexity algorithm. Determining a DS for a clustered DAG could take at least \( O(n + r) \) time if the computation is not done incrementally. Repeating this computation for all steps will result in at least \( O(n^2) \) complexity. Thus, it is necessary to use an incremental computation of DS from one step to the next to avoid the traversal of the entire DAG at each step. A proper selection of a DS edge for zeroing simplifies the incremental computation of the DS at the next step.

It is not clear how to implement API or AP2 with a low complexity, and also there is no guarantee that AP1 or AP2 will be better than AP3. We use AP3 to develop our algorithm.

1. \( E_G = \emptyset \), \( U_E G = V \).
2. Compute \( \text{blevel} \) for each node and set \( \text{blevel} = 0 \) for each entry node.
3. Every task is marked unexamined and assumed to constitute one unit cluster.
4. While there is an unexamined node Do
5. Find a free node \( n_f \) with highest priority from \( U_E G \).
6. Merge \( n_f \) with the cluster of one of its predecessors such that \( \text{blevel}(n_f) \) decreases in a maximum degree. If all memories increase \( \text{blevel}(n_f) \), \( n_f \) remains in a unit cluster.
7. Update the priority values of \( n_f \)'s successors.
8. \( U_E G = U_E G - \{n_f\}; E_G = E_G + \{n_f\} \).
9. EndWhile

Fig. 3. The DSC-I algorithm.

3) Since one of the goals is G3, i.e., reducing the number of unnecessary clusters and increasing the efficiency, we also need to allow for zeroing non-DS edges. The questions is when to do SubDS zeroing. One approach is to always zero DS edges until the algorithm stops, and then follow up with non-DS zeroing. Another approach, followed in this paper, is to interleaver the non-DS zeroing with DS zeroing, which gives more flexibility and results in a lower complexity.

4) Since backtracking could result in high complexity, we would like to avoid it as much as possible. For this reason, we need to impose the nonincrease in the parallel time constraint from one step to the next as follows:

\[
P_{i-1} \geq P_i.
\]

Sarkar imposes this constraint explicitly in his edge-zeroing process by comparing the parallel time at each step. Here we use an implicit constraint to avoid the explicit computation of parallel time in order to reduce the complexity.

In the next subsection, we present an initial version of DSC algorithm and then identify its weaknesses so that we can improve its performance.

**B. DSC-I: An Initial Version of DSC**

An algorithmic description of DSC-I is given in Fig. 3. An unexamined node is called **free** if all of its predecessors have been examined. Fig. 4 shows the initial status (step 0), shown in Fig. 4(a), and step 1 of DSC-I, corresponding to the While loop iteration \( i \) at Fig. 3, \( 1 \leq i \leq 5 \), shown in Fig. 4(b)-(d), in scheduling a DAG with five tasks. The thick paths of each step represent DS’s, and the dashed edges represent pseudo execution edges. We provide an explanation of the DSC-I algorithm in Fig. 3.

**Priority Definition and DS Identification:** (Line 5 in Fig. 3)

Based on (1), we define the priority for each task at each step as follows:

\[
\text{PRIO}(n_f) = \text{blevel}(n_f) + \text{blevel}(n_f).
\]

Thus, we can identify a DS node as the one with the highest priority. In Fig. 4(a), the DS nodes are \( n_1 \), \( n_2 \), and \( n_5 \), and they have the highest priority value, 14.5.

**The Topological Order for Task and Edge Examination:** (Line 5 in Fig. 3)

During the execution of DSC-I, the graph consists of two parts, the examined part, EG, and the unexamined part, UEG. Initially, all nodes are marked **unexamined**,
(See Fig. 4(a).) DSC-I selects only an unexamined free task \( n_f \) with the highest priority. Then it examines an incoming edge of this task for zeroing or not zeroing. In Fig. 4, tasks selected at step 1, 2, 3, 4, and 5 are \( n_1, n_2, n_4, n_3, \) and \( n_5 \), respectively. Notice that the selected task \( n_f \) belongs to a DS if there are free DS nodes; otherwise, it belongs to a SubDS. In Fig. 4, step 1 selects task \( n_1 \), which is in a DS; but step 3 selects task \( n_4 \), which is not in a DS. The order of such selection is equivalent to topologically traversing the graph.

The reason for following a topological order of task examination and zeroing one DS edge is that we can localize the effect of edge zeroing on the priority values for the rest of the unexamined tasks. In this way, even though zeroing changes the priorities of many nodes, DSC-I needs to compute only the changes for nodes that are immediate successors of the currently examined task \( n_f \). (See Line 7 of Fig. 3.) For Fig. 4(b), after \( n_1 \) is examined, the priority of \( n_5 \) does not need to be updated until one of its predecessors is examined at step 2. More explanations are given by Property 3.2.

Edge-Zeroing Criterion: **(Line 6 in Fig. 3)** The criterion for accepting a zeroing is that the value of \( tlevel(n_j) \) of the highest-priority free node does not increase by such a zeroing. If it does, then such zeroing is not accepted, and \( n_f \) becomes a new cluster in EG. Notice that by reducing \( tlevel(n_f) \), all paths going through \( n_f \) could be compressed, and as a result, the DS length could be reduced. In Fig. 4(c), \( n_2 \) is selected and \( (n_f, n_2) \) is zeroed. The zeroing is accepted because \( tlevel(n_2) \) reduces from 6 to 1.

**Task Placement in EG:** **(Line 6 in Fig. 3)** When an edge is zeroed, then a free task \( n_f \) is merged to the cluster where one of its predecessors resides. Our scheduling heuristic adds a pseudo edge from the last task of this cluster to \( n_f \), if they are independent. In Fig. 4(d), \( n_4 \) is selected, and the zeroing of \( (n_1, n_4) \) is accepted, because \( tlevel(n_4) \) reduces from 3 to 2.5. A pseudo edge \( (n_2, n_4) \) is added in EG. The DSC-I satisfies the following properties.

**Property 3.1:** \( PT_{i-1} \geq PT_i \)

Equation (1) implies that reducing the priority value of tasks would lead to the reduction of the parallel time. Thus, the constraint that \( tlevel \) values do not increase implies this property. A formal proof is given in Section IV-E.

In DSC-I, \( tlevel \) and \( blevel \) values are reused after each step, leading to a reduction of the complexity when determining DS nodes. The following property explains how the topological traversal and the cluster-merging rule of DSC-I reduces the complexity.

**Property 3.2:** For the DSC-I algorithm, \( tlevel(n_x) \) remains unchanged if \( n_x \in EG \) and \( blevel(n_x) \) remains unchanged if \( n_x \in UEG \).

**Proof:** If \( n_x \in UEG \), then the topological traversal implies that all descendants of \( n_x \) are in UEG. Since \( n_x \) and its descendants are in separate unit clusters, \( blevel(n_x) \) remains unchanged before it is examined. Also, for nodes in EG, all clusters in EG can be considered "linear" by counting the pseudo execution edges. When a free node is merged with a "linear" cluster, it is always attached to the last node of that "linear" cluster. Thus, \( tlevel(n_x) \) remains unchanged after \( n_x \) has been examined.

**Property 3.3:** The time complexity of the DSC-I algorithm is \( O(e + v \log v) \).

**Proof:** From Property 3.2, the priority of a free node \( n_f \) can be easily determined by using

\[
\text{tlevel}(n_f) = \max_{n_j \in \text{PRED}(n_f)} \{\text{tlevel}(n_j) + \tau_j + c_{j,f}\}.
\]

Once \( tlevel(n_j) \) is computed after its examination at some step, where \( n_j \) is the immediate predecessor of \( n_f \), this value is propagated to \( tlevel(n_f) \). Afterward \( tlevel(n_f) \) remains unchanged and does not affect the value of \( tlevel(n_f) \).

At each step of DSC-I, we maintain a priority list FL that contains all free tasks in UEG. This list can be implemented by using a balanced search tree data structure. At the beginning, FL is empty and there is no initial overhead in setting up this data structure. The overhead occurs while maintaining the proper order among tasks after an insertion or deletion of a task. This operation costs \( O(\log |FL|) \), where \( |FL| \leq v \).

Because each task in a DAG is inserted to and deleted from FL once during the entire execution of DSC-I, the total complexity of maintaining FL is at the most \( 2v \log v \).

The main computational cost of DSC-I algorithm is spent in the While loop (Line 4 in Fig. 3). The number of steps (iterations) is \( v \). For Line 5, each step costs \( O(\log v) \) for finding the head of FL, and \( v \) steps cost \( O(v \log v) \). For Line 6, each step costs \( O(\text{PRED}(n_f)) \) when examining the immediate predecessors of task \( n_f \). For the \( v \) steps, the cost is \( \sum_{n_f \in \text{PRED}(n_f)} O(\text{PRED}(n_f)) = O(e) \). For Line 7, each step costs \( O(\text{SUCC}(n_f)) \) to update the priority values of the immediate successors of \( n_f \), and, similarly, the cost for \( v \) steps is \( O(e) \).
When a successor of \( n_j \) is found free at Line 7, it is added to \( FL \), and the overall cost is \( O(v \log v) \). Thus, the total cost for DSC-I is \( O(e + v \log v) \).

C. An Evaluation of DSC-I

In this subsection, we study the performance of DSC-I for some DAG's and propose modifications to improve its performance. Because a DAG is composed of a set of join and fork components, we consider the strengths and weaknesses of DSC-I in scheduling fork and join DAG's. Then we discuss a problem arising when zeroing non-DS edges as a result of the topological ordering of the traversal.

DSC-I for fork DAG's: Fig. 5 shows the clustering steps of DSC-I for a fork DAG. Without loss of generality, assume that the leaf nodes in Fig. 5(a) are sorted such that \( \tau_j + \beta_j \geq \tau_j+1 + \beta_j+1, j = 1 \ldots m-1 \). The steps are described below. Fig. 5(b) shows the initial clustering, where each node is in a unit cluster. \( EG = \{n_x\} \), \( n_x \) is the only free task in UEG, and \( PRIO(n_x) = \tau_x + \beta_1 + \tau_1 \). At step 1, \( n_x \) is selected, and it has no incoming edges. It remains in a unit cluster, and \( EG = \{n_x\} \). After that \( n_1, n_2, \ldots, n_m \) become free, and \( n_1 \) has the highest priority, \( PRIO(n_1) = \tau_x + \beta_1 + \tau_1 \), and \( tlevel(n_1) = \tau_x + \beta_1 \). At step 2, shown in Fig. 5(c), \( n_1 \) is selected and merged with the cluster of \( n_x \), and \( tlevel(n_1) \) is reduced, in a maximum degree, to \( \tau_x \). At step \( k + 1 \), \( n_k \) is selected. The original leftmost scheduled cluster in Fig. 5(d) is a "linear" chain \( n_x, n_1, \ldots, n_{k-1} \). If attaching \( n_k \) to the end of this chain does not increase \( tlevel(n_k) = \tau_x + \beta_k \), the zeroing of edge \( (n_x, n_k) \) is accepted, and the new \( tlevel(n_k) = \tau_x + \sum_{j=1}^{k-1} \tau_j \). Thus, the condition for accepting or not accepting a zeroing can be expressed as \( \sum_{j=1}^{k-1} \tau_j \leq \beta_k \).

It is easy to verify that DSC-I always zeros DS edges at each step for a fork DAG, and the parallel time strictly decreases monotonically. It turns out that the DSC-I algorithm is optimal for this case; a proof is given in Section V.

DSC-I for Join DAG's: Let us consider the DSC-I algorithm for join DAG's. Fig. 6 shows a join DAG, the final clustering, and the optimum clustering. Again, we assume that \( \tau_j + \beta_j \geq \tau_j+1 + \beta_j+1, j = 1 \ldots m-1 \). The steps of DSC-I are described below. Fig. 6(a) shows the initial clustering. Nodes \( n_1, n_2, \ldots, n_m \) are free. One DS is \( \{n_1, n_2\} \). Step 1 selects \( n_1 \), which is in DS. No incoming edge exists for \( n_1 \), and no zeroing is performed. The DS is still \( \{n_1, n_2\} \) after step 1. Now \( n_2 \) becomes partially free. A node is partially free if it is in UEG and at least one of its predecessors has been examined, but not all of its predecessors have been examined. Step 2 selects \( n_2 \), which is not in DS. No incoming edge exists, and \( n_2 \) remains in a unit cluster in EG. Step \( m + 1 \) selects \( n_m \), which is now in the DS, and then \( \{n_1, n_2\} \) is zeroed. The final result of DSC-I is shown in Fig. 6(b), which may not be optimal. The optimal result for a join DAG shown in Fig. 6(c) and the previous optimal solution for a fork are symmetrical.

The join example shows that zeroing only one incoming edge of \( n_x \) is not sufficient to attain the optimum without backtracking. In general, when a free node is examined, zeroing multiple incoming edges of this free node instead of zeroing one edge could result in a reduction of \( tlevel \) in a maximum possible degree. As a consequence, the length of DS or SubDS going through this node could be reduced even more substantially. To achieve such a greedy goal, a low-complexity minimization procedure that zeros multiple incoming edges of the selected free node is needed to be incorporated in the DSC algorithm.

Chretienne [3] has proposed an optimal algorithm for a fork and join, which zeros multiple edges. The complexity of his algorithm is \( O(m \log B) \), where \( B = \min \{\sum_{i=1}^{m} \tau_i, \beta_1 + \tau_1 \} \). Al-Mouhamed [1] has also used the idea of zeroing multiple incoming edges of a task to compute a lower bound for scheduling a DAG, using an \( O(m^3) \) algorithm; but no feasible schedules that reach the bound are produced by his algorithm. Since we are interested in lower-complexity
algorithms, we use a new optimum algorithm with $O(m \log n)$ complexity.

**Dominant Sequence Length Reduction Warranty (DSRW):**
We describe another problem with DSC-I. When a DS node $n_y$ is partially free, DSC-I suspends the zeroing of its incoming edges and examines the current non-DS free nodes according to a priority-based topological order. Assume that $tlevel(n_y)$ could be reduced by $\delta$ if such a zeroing were not suspended. We should be able to get at least the same reduction for $tlevel(n_y)$ at some future step when DSC-I examines the free node $n_y$. Then the length of DS going through $n_y$ will also be reduced. However, this is not the case with DSC-I.

Fig. 4(c) shows the result of step 2 of DSC-I after $n_2$ is merged to CLUST($n_1$). The new DS depicted by the thick arrow is $(n_1, n_2, n_3)$, and it goes through partially free node $n_2$. If $(n_2, n_5)$ were zeroed at step 3, $tlevel(n_5)$ would have been decreased by $\delta = 5$. Then the length of the current DS $(n_1, n_2, n_3)$ would also have been reduced by $5$. But because of the topological traversal rule, a free task node is selected at step 3, because PROI($n_2$) = 9 $\geq$ PROI($n_3$) = 4.5. Then $n_4$ is merged with CLUST($n_1$), because $tlevel(n_4)$ can be reduced from 3 to $7_1 + 7_2 = 2.5$. Such zeroing affects the future compression of DS $(n_1, n_2, n_3)$. When $n_3$ is free at step 5, $tlevel(n_3) = 7_1 + 7_2 + 7_3 = 7.5$, and it is impossible to reduce $tlevel(n_3)$ further by moving it to CLUST($n_2$). This is because $n_3$ will have to be linked after $n_4$, which makes $tlevel(n_5) = 7_1 + 7_2 + 7_3 = 8.5$.

To guarantee the effective reduction of $tlevel(n_y)$ of a partially free DS node $n_y$ at the future step, we impose a condition called DSRW. The basic idea is to avoid the edge-zeroing operations that affect the reduction of $tlevel(n_y)$. The details are described in Section IV-C.

IV. THE FINAL FORM OF THE DSC ALGORITHM

The main improvements to DSC-I are the minimization procedure for $tlevel(n_j)$, maintaining a partially free list (PFL) and imposing the constraint DSRW. The DSC algorithm is described in Fig. 7.

The clustering sequence of Fig. 7 starts from examining the entry tasks of a DAG. We call it forward clustering. There is another way to obtain a schedule for a DAG. First, invert this DAG by changing the directions of all precedence edges, obtain a schedule using Fig. 7 for the inverted DAG, and then invert this schedule to produce a schedule with the same length for the original DAG. We call it backward clustering. These two approaches may have different performance because of different precedence structures. In fact, in Section V-C, we show that forward clustering cannot obtain the optimum for certain trees, but backward clustering can. The complete version of the DSC algorithm performs both forward and backward clustering and chooses the solution with a shorter schedule. We mainly analyze the properties of DSC with respect to forward clustering. Backward clustering has the same properties, except when dealing with an inverted graph, and it does not increase the order of the algorithm complexity.

---

1. $EG = \emptyset$, $UG = V$. Add all free-entry nodes to FL.
2. Compute level for each node and set $level = 0$ for each free node.
3. WHILE $UG \neq \emptyset$ DO
4. $n_y = head(FL)$; the first task with the highest priority $PRIO(n_y)$.
5. $n_y = head(FL)$. The first task with the highest priority $PRIO(n_y)$.
6. IF ($PRIO(n_y) \geq PRIO(n_x)$) THEN
7. Call the minimization procedure to reduce $tlevel(n_y)$.
8. ELSE
9. $n_y$ is not zeroing is accepted, $n_y$ remains in a wait queue.
10. Call the minimization procedure to reduce $tlevel(n_y)$ under constraint DSRW.
11. $n_y$ is not zeroing is accepted, $n_y$ remains in a wait queue.
12. ENDIF
13. Update the priorities of $n_x$'s successors and put $n_x$ into $EG$.
14. ENDDO

---

A. Priority Lists

At each clustering step, we maintain two node priority lists: a partially free list, PFL, and a free list, FL, both sorted in a descending order of their task priorities. When two tasks have the same priority, we choose the one with the most immediate successors. If there is still a tie, we break it randomly. Function $head(L)$ returns the first node in the sorted list $L$, which is the task with the highest priority. If $L = \emptyset$, $head(L) = NULL$, and the priority value is set to 0.

The $tlevel$ value of a node is propagated to its successors only after this node has been examined. Thus, the priority value of a partially free node can be updated by using only the $tlevel$ from its examined immediate predecessors. Because only part of its predecessors are considered, we define the priority of a partially free task as follows:

$pPRIO(n_y) = plevel(n_y) + blevel(n_y)$

$plevel(n_y) = \max_{n_j \in PRED(n_y)} \{tlevel(n_j) + t_j + c_{j,y}\}$

In general, $pPRIO(n_y) \leq PRIO(n_y)$, and if a DS goes through an edge $(n_j, n_y)$, where $n_j$ is an examined predecessor of $n_y$, then we have $pPRIO(n_y) = PRIO(n_j) = PRIO(n_y)$. By maintaining $pPRIO$ instead of PRIO, the complexity is reduced considerably. As we prove later, maintaining $pPRIO$ does not adversely affect the performance of DSC, because it can still correctly identify the DS at each step.

B. The Minimization Procedure for Zeroing Multiple Incoming Edges

To reduce $tlevel(n_y)$ in DSC, a minimization procedure that zeros multiple incoming edges of free task $n_y$ is needed. An optimal algorithm for a join DAG has been described in [8], and an optimal solution is shown in Fig. 6(c). The basic procedure is to first sort the nodes such that $t_j + \beta_j \geq t_{j+1} + \beta_{j+1}; j = 1 : m - 1$, and then to zero edges from left to right, as long as the parallel time reduces after each zeroing, i.e., linear searching of the optimal point. This is equivalent to satisfying the condition $(\sum_{j=1}^{k-1} t_j \leq \beta_k)$ for each accepted zeroing. Another optimum algorithm for join is to determine the optimum point $k$, first by using a binary search between 1 and $m$ such that $(\sum_{j=1}^{k-1} t_j \leq \beta_k)$, and then zero all edges to the left of $k$.

We modify the optimum binary search join algorithm to minimize $tlevel(n_y)$ when $n_y$ is free. The procedure is shown in Fig. 8. Assume that $PRED(n_y) = \{n_1, n_2, \ldots, n_m\}$. Notice
1. Sort the predecessors of \( n_x \) such that
\[
\text{tlevel}(n_x) + \tau_j + c_{j,x} \geq \text{tlevel}(n_{x+1}) + \tau_{j+1} + c_{j+1,x}, \quad j = 1, m - 1.
\]

2. Let \( h \) be the maximum integer from 2 to \( m \) such that for \( 2 \leq i \leq h \) node \( n_i \) satisfies the following constraint:
- If \( n_i \) is not in \( \text{CLUST}(n_x) \), then \( n_i \) does not have any children other than \( n_x \).

3. Find the optimum point \( k \) between 1 and \( h \) using the binary search algorithm.
Zero \((n_x, n_k)\) up to \((n_k, n_x)\) so that \( \text{tlevel}(n_k) \) is minimized.

Fig. 8. The minimization procedure for DSC.

That all predecessors are in EG, but now \( \text{CLUST}(n_1), \ldots, \text{CLUST}(n_m) \) may not be unit clusters. We sort the predecessors of \( n_x \) such that \( \text{tlevel}(n_j) + \tau_j + c_{j,x} \geq \text{tlevel}(n_{j+1}) + \tau_{j+1} + c_{j+1,x} \). To reduce \text{tlevel}(n_x), we must zero the edge \((n_1, n_x)\). A problem arises when the algorithm zeros an edge of a predecessor \( n_p \), which has other children than \( n_x \). When task \( n_p \) is moved from \( \text{CLUST}(n_p) \) to \( \text{CLUST}(n_1) \), the \text{tlevel} of the children of \( n_p \) will be affected. As a result, the length of paths going through those children will most likely increase. The constraint \( PT_{i-1} \geq PT \) may no longer hold, and maintaining task priorities becomes complicated. To avoid such cases, we exclude from the minimization procedure predecessors\(^2\) that have children other than \( n_x \), unless they are already in \( \text{CLUST}(n_1) \). The binary search algorithm in Fig. 8 finds the best stopping point \( k \) and those predecessors \( n_j \) of \( n_x \) (\( 2 \leq j \leq k \)) that are not in \( \text{CLUST}(n_1) \) must be extracted from their corresponding clusters and attached to \( \text{CLUST}(n_1) \). In computing the new \text{tlevel} \((n_x)\), those predecessors of \( n_x \) in \( \text{CLUST}(n_x) \) are ordered for execution in an increasing order of their \text{tlevel} values. The \text{tlevel} ordering could be performed before the binary searching.

We determine the complexity of this procedure as follows. The \text{tlevel} ordering of all predecessors is done once at a cost of \( O(m \log m) \). The ordering for step 1 at Fig. 8 costs \( O(m \log m) \). The binary search computes the effect of the ordered predecessors to \text{tlevel}(n_x) at a cost of \( O(m) \) at each step. The total number of search steps is \( O(\log m) \). Thus, the total cost of the above algorithm is \( O(m \log m) \). If linear search were used instead, the total cost would increase to \( O(m^2) \), because the number of search steps would increase to \( m \).

C. Imposing Constraint DSRW

As we saw previously, when there is no DS going through any free task and there is one DS passing through a partially free node \( n_p \), then zeroing non-DS incoming edges of free nodes can affect the reduction of \text{tlevel} \((n_p)\) in the future steps. We impose the following constraint on DSC to avoid such side effects.

\text{DSRW: Zeroing incoming edges of a free node should not affect the \text{tlevel} of an incoming DS edge of } n_p.\]

There are two problems that we must address in the implementation of DSRW. First, we must detect whether \text{tlevel}(n_p) is reducible. Second, we must make sure that DSRW is satisfied.

1. To detect the reducibility of \text{tlevel}(n_p), we must examine the result of the zeroing of an incoming DS edge of \( n_p \). To find such an incoming DS edge, we examine only the result of the zeroing each incoming edge \((n_j, n_p)\) where \( n_j \) is a predecessor of \( n_p \) and \( n_j \in EG \). As we prove in Section IV-E, \text{tlevel}(n_p) = \text{tlevel}(n_j) \). This implies that such partial reducibility is sufficient to guarantee that if the parallel time was reducible by zeroing the DS incoming edges of a partially free DS node \( n_p \), then \text{tlevel}(n_p) is reducible when \( n_p \) becomes free. Hence, the DS can be compressed at that time.

2. After detecting the partial reducibility at step i for node \( n_p \), we implement the constraint DSRW as follows. Assume that \( n_p \) is one examined predecessor of \( n_p \), and that zeroing \((n_p, n_p)\) would reduce \text{tlevel}(n_p). Then no other nodes are allowed to move to \text{CLUST}(n_p) until \( n_p \) becomes free.

For the example in Fig. 4(c), \( n_p = n_3 \), \text{tlevel}(n_3) = \tau_1 + \tau_2 + c_{2,5} = 7.5 \), \text{pPRIO}(n_3) = 9.5, \text{PRIO}(n_5) = 4.5 \), and \text{PRIO}(n_4) = 9. We have the condition that \text{pPRIO}(n_3) > \text{PRIO}(n_4) \), which implies, by Theorem 4.1 in Section IV-E, that DS goes through partially free node \( n_3 \). In addition, \text{tlevel}(n_3) could be reduced if \((n_2, n_3)\) were zeroed. Then \text{CLUST}(n_3) cannot be touched before \( n_p \) becomes free. Thus, \( n_3 \) and \( n_4 \) cannot be moved to \text{CLUST}(n_3) and they remain in the unit clusters in EG. When \( n_5 \) finally becomes free, \((n_2, n_5)\) is zeroed and PT is reduced from 9 to 8.5.

It should be pointed out that DSRW is a greedy heuristic that attempts to reduce the parallel time as much as possible. It is always possible that this constraint could prevent the generation of an optimal solution. In Section VI, we examine the performance of DSC on scheduling randomly generated DAG’s and compare it with other algorithms.

D. A Running Trace of DSC

We demonstrate the DSC steps by using a DAG example shown in Fig. 1(a). The steps 0, 1, 2, 5, 6, and 7 are shown in Fig. 9(a)–9(f), respectively. The thick paths are the DS’s and dashed pseudo edges are the execution order within a cluster. We provide an explanation for each step below. The superscript of a task node in FL or PFL indicates its priority value.

a) Initially, we have the following conditions:

\[
\begin{align*}
\text{UG} &= \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\} & \text{PT}_0 &= 13 \\
\text{FL} &= \{n_1^{13}, n_2^{13}, n_4^{9.5}, n_5^{7.5}\} & \text{PFL} &= \{\}
\end{align*}
\]

b) \( n_3 \) is selected; \text{tlevel}(n_3) = 0, and it cannot be reduced, so \text{CLUST}(n_3) remains a unit cluster. Then we have the following condition:

\[
\begin{align*}
\text{UG} &= \{n_2, n_3, n_4, n_5, n_6, n_7\} & \text{PT}_1 &= 13 \\
\text{FL} &= \{n_2^{13}, n_3^{13}, n_4^{9.5}, n_5^{7.5}, n_6^{13}, n_7^{7.5}\} & \text{PFL} &= \{\}
\end{align*}
\]

c) \( n_2 \) is selected, \( n_x = n_2 \), \( n_y = \text{NULL} \), and \text{tlevel}(n_2) = 4. By zeroing the incoming edge \((n_1, n_2)\) of \( n_2 \),
tlevel \( (n_2) \) reduces to 1. Thus, this zeroing is accepted, and after that step, we have the following condition:

\[
\text{UEG} = \{n_6, n_7\} \quad \text{PT}_2 = 10
\]

\[
\text{FL} = \{n_6^{5+4}, n_6^{5+4.5}\} \quad \text{PFL} = \{n_7^{9+1}\}.
\]

d) \( n_3 \) is selected, \( n_2 = n_3 \) with \( \text{PRIO}(n_3) = 1.5 + 8 = 9.5 \), and \( n_7 = n_7 \) with \( p\text{PRIO}(n_7) = 10 \). Notice that by zeroing \( (n_2, n_7) \), the tlevel \( (n_7) \) reduces to 8.5. Thus, we impose the DSRW constraint. Since zeroing \( (n_1, n_3) \) affects tlevel \( (n_7) \), this zeroing is not accepted, because of DSRW. The tlevel \( (n_3) \) remains the same, and \( \text{CLUST}(n_3) \) remains a unit cluster in EG.

e) \( n_4 \) and \( n_5 \) are selected, respectively, and their clusters again remain unit clusters. After that we have the following condition:

\[
\text{UEG} = \{n_6, n_7\} \quad \text{PT}_3 = 10
\]

\[
\text{FL} = \{n_6^{5+4.5}\} \quad \text{PFL} = \{n_7^{9+1}\}.
\]

f) \( n_6 \) is selected, and its incoming edges \( (n_3, n_6) \) and \( (n_4, n_6) \) are zeroed by the minimization procedure. Node \( n_4 \) is ordered for execution first, because tlevel \( (n_4) = 0 \), which is smaller than tlevel \( (n_3) = 1.5 \). Then tlevel \( (n_6) \) is reduced to 2.5, and we have the following condition:

\[
\text{UEG} = \{n_7\} \quad \text{PT}_5 = 10
\]

\[
\text{FL} = \{n_6^{5+4.5}\} \quad \text{PFL} = \{\}\.
\]

g) \( n_7 \) is selected, and \( (n_2, n_7) \) is zeroed so that tlevel \( (n_7) \) is reduced from 9 to 7. Then we have the following condition:

\[
\text{UEG} = \{\}\quad \text{PT}_7 = 8
\]

\[
\text{FL} = \{\}\quad \text{PFL} = \{\}.
\]

Finally, three clusters are generated with \( \text{PT} = 8 \).

E. DSC Properties

In this subsection, we study several properties of DSC related to the identification of DS, the reduction of the parallel time, and the computational complexity. Theorem 4.1 indicates that DSC (Lines 6 and 9 in Fig. 7) correctly locates DS nodes at each step, even if we use the even priority \( p\text{PRIO} \). Theorems 4.2 and 4.3 show how DSC warranties guarantees the reduction of the parallel time. Theorem 4.4 studies the complexity of DSC.

**Correctness in Locating DS Nodes:** The goal of DSC is to reduce the length of DS’s in a sequence of steps. To do that, it must correctly identify examined DS nodes. For Lemmas 4.1 and 4.2 and Theorem 4.1, we assume that a DS goes through UEG, because it is only then that DSC needs to identify and compress DS. If DS does not go through UEG, but only through EG, then all DS nodes have been examined, and the DS can no longer be compressed, because backtracking is not allowed in general.

Since the DSC algorithm examines the nodes topologically, the free list FL is always nonempty. By definition, all tasks in FL and PFL are in UEG. It is obvious that a DS must go through tasks in either FL or PFL when it goes through UEG. The interesting question is whether a DS also goes through the heads of the priority lists of FL and PFL, because then the algorithm would correctly locate DS nodes for examination. The answer is given in the following lemmas and in Theorem 4.1.

**Lemma 4.1:** Assume that \( n_x = \text{head}(FL) \) after step \( i \). If there are DS’s going through free nodes in FL, then one DS must go through \( n_x \).

**Proof:** At the completion of step \( i \), the parallel time is \( \text{PT}_i = \text{PRIO}(n_x) \), where \( n_x \) is a DS node. Assume that no DS goes through \( n_x \). Then one DS must go through another nonhead free node \( n_f \). This implies that \( \text{PRIO}(n_f) = \text{PT}_i > \text{PRIO}(n_x) \), which is a contradiction, because \( n_x \) has the highest priority in FL.

**Lemma 4.2:** Assume that \( n_y = \text{head}(PFL) \) after step \( i \). If DS’s go through only partially free nodes in PFL, then one DS must go through \( n_y \). Moreover, \( p\text{PRIO}(n_y) = \text{PRIO}(n_y) \).

**Proof:** First, observe that the starting node of a DS must be an entry node of this DAG. If this node is in UEG, then it must be free, which is impossible, because the assumption says that DS’s go through only partially free nodes in UEG. Thus, the starting node must be in EG. As a result, a DS must start from a node in EG and go through an examined node \( n_s \), to its unexamined partially free successor \( n_p \). Then, because they are in the same DS, we have the condition that \( \text{PRIO}(n_p) = \text{PRIO}(n_j) = p\text{PRIO}(n_p) \). The
proof now becomes similar to the previous lemma. Suppose that no DS goes through \( n_y \). We have the condition that \( p\text{PRIO}(n_y) \leq \text{PRIO}(n_y) < \text{PT}_i = \text{PRIO}(n_y) = p\text{PRIO}(n_y) \), which contradicts the assumption that \( n_y \) is the head of PFL.

Next we prove that \( p\text{PRIO}(n_y) = \text{PRIO}(n_y) \). Suppose \( \text{PRIO}(n_y) < \text{PRIO}(n_y) \). This implies that the DS, to which \( n_y \) belongs, does not pass through any examined immediate predecessor of \( n_y \). As a result, DS must go through other nodes in UEG. Thus, there exists an ancestor node \( n_x \) of \( n_y \) such that the DS passes through an edge \((n_x, n_y)\), where \( n_x \) is the examined predecessor of \( n_y \) and \( n_x \) is partially free because it is impossible to be free by the assumptions of this lemma. We have the condition that \( \text{PRIO}(n_x) = \text{PRIO}(n_x) = \text{PRIO}(n_x) > p\text{PRIO}(n_y) \), which shows \( n_y \) is not the head of PFL. This is a contradiction.

The following theorem shows that the condition \( \text{PRIO}(n_x) \geq p\text{PRIO}(n_y) \), used by DSC algorithm in Line 6 of Fig. 7, correctly identifies DS nodes.

**Theorem 4.1:** Assume that \( n_x = \text{head}(\text{FL}) \) and \( n_y = \text{head}(\text{PFL}) \) after step \( i \), and that there is a DS going through UEG. If \( \text{PRIO}(n_x) \geq p\text{PRIO}(n_y) \), then a DS goes through \( n_x \). If \( \text{PRIO}(n_x) < p\text{PRIO}(n_y) \), then a DS goes through \( n_y \), and there is no DS going through any free node.

**Proof:** If \( \text{PRIO}(n_x) \geq p\text{PRIO}(n_y) \), then we show that a DS goes through \( n_x \). First, assume that DS goes through FL or through both FL and PFL. Then, according to Lemma 4.1, it must go through \( n_x \). Next assume that DS goes through PFL only. Then, from Lemma 4.2, it must go through \( n_y \), implying that \( p\text{PRIO}(n_y) = \text{PRIO}(n_y) = \text{PT}_i > \text{PRIO}(n_x) \), which is a contradiction, because \( \text{PRIO}(n_x) \geq p\text{PRIO}(n_y) \).

If \( \text{PRIO}(n_x) < p\text{PRIO}(n_y) \), suppose that a DS passes a free node. Then, according to Lemma 4.1, \( \text{PRIO}(n_x) = \text{PT}_i \geq \text{PRIO}(n_y) \geq p\text{PRIO}(n_y) \), which is a contradiction again. Thus, the DS’s must go through only partially free nodes, and one of them must go through \( n_y \) by Lemma 4.2.

**The Warranty in Reducing Parallel Time:** We show that during DSC clustering steps, the parallel time of the clustered graph monotonically decreases. Moreover, after any clustering step \( i \), if there exists an algorithm that could reduce the parallel time of the clustered graph by zeroing some edge, then DSC will guarantee a reduction at some future step.

**Lemma 4.3:** Assume that \( n_x = \text{head}(\text{FL}) \) and \( n_y = \text{head}(\text{PFL}) \) after step \( i \). The following is the parallel time for executing the clustered graph after step \( i \) of DSC:

\[
\text{PT}_i = \max_{n_x \in \text{EG}} \{ \text{PRIO}(n_x), p\text{PRIO}(n_x), \max_{n_x \in \text{EG}} \{ \text{PRIO}(n_x) \} \}.
\]

**Proof:** There are three cases in the proof.

1) If DS nodes are only within EG, then, by definition, \( \text{PT}_i = \max_{n_x \in \text{EG}} \{ \text{PRIO}(n_x) \} \).

2) If a DS goes through a free node, then \( \text{PT}_i = \text{PRIO}(n_x) \) by Lemma 4.1.

3) If there is a DS passing through UEG, but this DS passes through only partially free nodes, then \( \text{PT}_i = \text{PRIO}(n_y) = p\text{PRIO}(n_y) \), by Lemma 4.2.

**Theorem 4.2:** For each step \( i \) of DSC, \( \text{PT}_{i-1} \geq \text{PT}_i \).

**Proof:** For this proof, we rename the priority values of \( n_x = \text{head}(\text{FL}) \), and \( n_y = \text{head}(\text{PFL}) \) after step \( i \), because \( \text{PRIO}(n_x, i - 1) \) and \( p\text{PRIO}(n_y, i - 1) \), respectively. We need to prove that \( \text{PT}_{i-1} \geq \text{PT}_i \), where the following is true:

\[
\text{PT}_{i-1} = \max \{ \text{PRIO}(n_x, i - 1), p\text{PRIO}(n_y, i - 1) \},
\]

\[
\max_{n_x \in \text{EG}} \{ \text{PRIO}(n_x, i - 1) \},
\]

\[
\text{PT}_i = \max \{ \text{PRIO}(n_y, i), p\text{PRIO}(n_y, i) \},
\]

\[
\max_{n_x \in \text{EG}} \{ \text{PRIO}(n_x, i) \},
\]

and \( n_x^* \) and \( n_y^* \) are the new heads of FL and PFL after step \( i \).

We prove first that \( \text{PRIO}(n_x^*, i) \leq \text{PT}_{i-1} \). Since \( n_x^* \) is in the free list after step \( i - 1 \), it must be in UEG, and also it must be either the successor of \( n_x \) or independent of \( n_x \). At step \( i \), DSC picks up task \( n_x \) to examine its incoming edges for zeroing. We consider the effect of such zeroing on the priority value of \( n_x^* \). Since the minimization procedure does not increase the level \( (n_x^*) \), the length of the paths going through \( n_x \) decreases or remains unchanged. Thus, the priority values of the descendants of \( n_x \) could decrease, but could not increase. The priority values of other nodes in EG remain the same because the minimization procedure excludes those predecessors of \( n_x \) that have children other than \( n_x \). Thus, if \( n_x^* \) is the successor of \( n_x \), then \( \text{PRIO}(n_x^*, i) \leq \text{PRIO}(n_x^*, i-1) \); otherwise, \( \text{PRIO}(n_x^*, i) = \text{PRIO}(n_x^*, i-1) \). Since \( \text{PRIO}(n_x^*, i-1) \leq \text{PT}_{i-1}, \text{PRIO}(n_x^*, i) \leq \text{PT}_{i-1} \). Similarly, we can prove that \( \text{PRIO}(n_y^*, i) \leq \text{PT}_{i-1} \).

Next we prove that \( \max_{n_x \in \text{EG}} \{ \text{PRIO}(n_x, i) \} \leq \text{PT}_{i-1} \). We have \( \text{PRIO}(n_x, i) \leq \text{PRIO}(n_x, i-1) \) from the minimization procedure. We need to examine the effect of zeroing only for \( n_x \) on the priorities of nodes in EG. The minimization procedure may increase the level \( (n_x) \) values of some predecessors of \( n_x \), say, \( n_y \), but it guarantees that the lengths of the paths going through \( n_y \) and \( n_x \) do not increase. Thus, the new value of \( \text{PRIO}(n_y) \) satisfies \( \text{PRIO}(n_y, i) \leq \text{PRIO}(n_y, i-1) \). Because \( \text{PRIO}(n_x, i-1) \leq \text{PT}_{i-1}, \text{PRIO}(n_x, i) \leq \text{PT}_{i-1} \).

The minimization procedure may also increase the level values of some nodes in the cluster in EG to which \( n_x \) is attached. A pseudo edge is added from the last node of that cluster, say, \( n_c \), to \( n_x \) if \( n_x \) and \( n_c \) are independent. If there is an increase in the priority value of \( n_x \), then the reason must be that adding this pseudo edge introduces a new path going through \( n_x \) with longer length than the other existing paths for \( n_x \). Since the minimization procedure has considered the effect of attaching \( n_x \) after \( n_c \), the length of paths will be less than or equal to \( \text{PRIO}(n_x, i-1) \leq \text{PT}_{i-1} \). Thus, \( \text{PRIO}(n_x, i) \leq \text{PT}_{i-1} \).

**Theorem 4.3:** After step \( i \) of DSC, if the current parallel time is reducible by zeroing one incoming edge of a node
in UEG, then DSC guarantees that the parallel time will be reduced at some step greater or equal to \(i + 1\).

**Proof:** The assumption that the parallel time reduces by zeroing one edge \((n_r, n_s)\) implies that a DS must go through UEG. This implies that the edge \((n_r, n_s)\) belongs to all DS’s that go through UEG; otherwise, the parallel time is not reducible. There are three cases.

1) \(n_r \in \text{EG} \) and \(n_s \) is free. We prove that the node \(n_s\) must be the head of FL. If it is not, then another free node in UEG, \(n_f\), must have the same priority and thus belong to a DS. Because all DS’s go through \((n_r, n_s)\), \(n_f\) must be a successor of \(n_s\). Then \(n_s\) and \(n_f\) cannot both be free, which is a contradiction.

Also, since PT, = tlevel\((n_s) + blevel\((n_s)\) is reducible by zeroing \((n_r, n_s)\), and since blevel\((n_s)\) does not change by such a zeroing, tlevel\((n_s)\) is reducible. Thus, during the DSC execution, \(n_s = \text{head}(FL)\) will be picked up at step \(i + 1\), and since tlevel\((n_s)\) is reducible, the minimization procedure will accept the zeroing of \((n_r, n_s)\), and the parallel time will reduce at that step.

2) \(n_r \in \text{EG} \) and \(n_s\) is partially free. We prove that \(n_s\) must be the head of PFL. We show it by contradiction. Assume that \(n_f (n_f \neq n_s)\) is the head of PFL. Since all DS’s go through \((n_r, n_s)\), no free nodes are in DS’s, and \(n_f\) must be a successor of \(n_s\). Then pPRIOR\((n_f) < PRIOR\((n_f)\). But by Lemma 4.2, pPRIOR\((n_f) = pPRIOR\((n_f)\), which is a contradiction.

Assume that PT, = ptlevel\((n_s) + blevel\((n_s)\) is reducible by \(\delta > 0\) when zeroing \((n_r, n_s)\), and that blevel\((n_s)\) does not change by such a zeroing. Then ptlevel\((n_s)\) is reducible by at least \(\delta\). Thus, during the execution of DSC, the reducibility of ptlevel\((n_s)\) will be detected at step \(i + 1\). Afterward non-DS edges are zeroed until \(n_s\) becomes free. However, the reducibility of ptlevel\((n_s)\) is not affected by such zeroings, because of DSRW. A non-DS node remains a non-DS node, and no other nodes are moved to CLUST\((n_s)\). When \(n_s\) becomes free, ptlevel\((n_s)\) is still reducible by at least \(\delta\), and because other SubDS either decrease or remain unchanged, the parallel time is also reducible by at least \(\delta\). The rest of the proof becomes the same as in the previous case.

3) \(n_r \in \text{UEG}\). Assume that \(n_s\) becomes examined at step \(j \geq i + 1\). At step \(i + 1\), all DS’s must go through \((n_r, n_s)\). If, from \(i + 1\) to \(j\), the parallel time has been reduced, then the theorem is true. If the parallel time has not been reduced, then at least one DS has not been compressed, and all DS’s still go through \((n_r, n_s)\), because the minimization procedure guarantees that no other DS’s will be created. Thus, the parallel time will be reducible at step \(j\) by zeroing \((n_r, n_s)\), and the proof becomes the same as in the above cases.

The following corollary is the direct result of Case 2 above.

**Corollary 4.3:** Assume that \(n_y \in \text{PFL}, n_y \in \text{DS} \) at step \(i\), and that zeroing of an incoming edge of \(n_y\) from its scheduled predecessor would have reduced PT by \(\delta\). Then DSC guarantees that when \(n_y\) becomes free at step \(j (j > i)\), PT can be reduced by at least \(\delta\).

**The Complexity:**

**Lemma 4.4:** After a node \(n_s\) becomes free or partially free, blevel\((n_s)\) remains unchanged until \(n_s\) is examined. Let \(n_i\) be a predecessor of \(n_s\), tlevel\((n_j)\) remains unchanged from the step when \(n_j\) becomes examined to the step when \(n_s\) is examined.

**Proof:** Referring to Property 3.2 DSC-I, DSC is the same as DSC-I, except that the minimization procedure changes tlevel values of some examined predecessors, say, \(n_s\), of the currently selected free task, say, \(n_s\). But \(n_s\) does not have any children other than \(n_s\). Thus, after a node \(n_s\) becomes partially free or free, but before it is examined, the tlevel value of its examined predecessors will remain unchanged.

**Theorem 4.5:** The time complexity of DSC is \(O((v + e) \log v)\), and the space complexity is \(O(v + e)\).

**Proof:** The difference in the complexity between DSC-I and DSC results from the minimization procedure within the While loop in Fig. 7. In DSC, we also maintain PFL; but the cost for the \(v\) steps is the same \(O(v \log v)\) when we use the balanced search trees. Therefore, for Lines 4 and 5 in Fig. 7, \(v\) steps cost \(O(v \log v)\). For Lines 7 and 8 (or for Lines 9 and 10), the minimization procedure costs \(O(|\text{PRED}(n_y)| \log |\text{PRED}(n_y)|)\) at each step. Since \(\sum_{n_y \in V} |\text{PRED}(n_y)| = e\) and \(|\text{PRED}(n_y)| < v, v\) steps cost \(O(e \log v)\). When imposing DSRW, the predecessors of a partially free node are checked, and the overall cost is at the most \(O(v + e)\).

For Line 13, the tlevel values of the successors of \(n_s\) are updated. Those successors could be in PFL, and the list needs to be rearranged, because their pPRIOR values could be changed. The cost of adjusting each successor in PFL is \(O(|\text{PFL}| \log |\text{PFL}|)\), where \(|\text{PFL}| < v\). The step cost for Line 13 is \(O(|\text{SUCC}(n_s)| \log v)\). Since \(\sum_{n_s \in V} |\text{SUCC}(n_s)| = e\), the total cost for maintaining PFL during \(v\) steps is \(O(e \log v)\).

Also, for Line 13, when one successor becomes free, it needs to be added to FL with cost \(O(|\text{FL}| \log |\text{FL}|)\), where \(|\text{FL}| \leq v\). Since there are a total of \(v\) task that could become free during \(v\) steps, the total cost in Line 13 spent for FL during \(v\) steps is \(O(v \log v)\).

Notice that according to Lemma 4.4, after \(n_s\) has been moved to EG at Line 13, its tlevel value will not affect the priority of tasks in PFL and FL in the rest of the steps. Thus, the updating of FL or PFL occurs only once with respect to each task. Therefore, the time complexity of DSC is \(O((v + e) \log v)\). The space needed for DSC is to store the DAG and FL/PFL. The space complexity is \(O(v + e)\).
but also attains an optimality degree that a general polynomial algorithm could achieve.

A. Performance Bounds for General DAG’s

A DAG consists of fork ($F_v$) and/or join ($J_v$) structures such as the ones shown in Figs. 5(a) and 6(a). In [8], we define the grain of DAG as follows.

Let the following conditions exist:

$$g(F_v) = \min_{k=1:m} \{ \tau_k \} / \max_{k=1:m} \{ \tau_{x,k} \},$$

$$g(J_v) = \min_{k=1:m} \{ \tau_k \} / \max_{k=1:m} \{ \tau_{x,k} \}.$$

Then the granularity of $G$ is:

$$g(G) = \min_{x \in V} \{ g_F(x), g(J_v) \}$$

We call a DAG coarse grain if $g(G) \geq 1$. For a coarse-grain DAG, each task receives or sends a small amount of communication compared to the computation of its neighboring tasks. In [8], we prove Theorems 5.1, 5.2, and 5.3.

Theorem 5.1: For a coarse grain DAG, there exists a linear clustering that attains the optimal solution.

Theorem 5.2: Let $PT_{opt}$ be the optimal parallel time, and let $PT_{opt}$ be the parallel time of a linear clustering. Then $PT_{opt} \leq (1 + \frac{1}{g(G)})PT_{opt}$. For a coarse-grain DAG, $PT_{opt} \leq 2 \times PT_{opt}$.

The following theorem is a performance bound of DSC for a general DAG.

Theorem 5.3: Let $PT_{dec}$ be the parallel time by DSC for a DAG $G$. Then $PT_{dec} \leq (1 + \frac{1}{g(G)})PT_{opt}$. For a coarse-grain DAG, $PT_{dec} \leq 2 \times PT_{opt}$.

Proof: In the initial step of DSC, all nodes are in the separate clusters, which is a linear clustering. By Theorems 5.2 and 4.2, we have the following conditions:

$$PT_{dec} \leq \cdots \leq PT_k \leq \cdots \leq PT_1 \leq PT_0 \leq (1 + \frac{1}{g(G)})PT_{opt}.$$

For coarse-grain DAG’s, the statement is obvious.

B. Optimality for Join and Fork

Theorem 5.4: DSC derives optimal solutions for fork and join DAG’s.

Proof: For a fork, DSC performs the exact same zeroing sequence as DSC-I. The clustering steps are shown in Fig. 5. After $n_x$ is examined, DSC will examine free nodes $n_1, n_2, \ldots, n_m$ in a decreasing order of their priorities. The priority value for each free node is the length of each path $<n_x, n_1>, \ldots, <n_x, n_m>$. If we assume that $\beta_k + \tau_k \geq \beta_{k+1} + \tau_{k+1}$ for $1 \leq k \leq m - 1$, then the nodes are sorted as $n_1, n_2, \ldots, n_m$ in the free list FL.

We first determine the optimal time for the fork, and then show that DSC achieves the optimum. Assume that the parallel time to be $PT_{opt}$. If PRIO($n_x) = \tau_x + \tau_k + \beta_k > PT_{opt}$ for some $h$, then $\beta_i$ must be zeroed for $i = 1 : h$; otherwise, we have a contradiction. All other edges $i > h$ need not be zeroed, because zeroing such edge does not decrease PT, but could increase PT. Let the optimal zeroing stopping point be $h$, and assume that $\beta_{h+1} = \tau_{h+1} = 0$. Then the optimal PT is $PT_{opt} = \tau_x + \max(\sum_{j=1}^{h} \tau_j, \beta_{h+1} + \tau_{h+1})$.

DSC zeroes as many as edges possible from left to right up to the point $k$, as shown in Fig. 5(d), such that $\sum_{j=1}^{k} \tau_j \leq \beta_k$ and $\sum_{j=1}^{k} \tau_j > \beta_{k+1}$. We prove that $PT_{opt} = PT_{dec}$ by a contradiction. Suppose that $k \neq h$, and $PT_{opt} < PT_{dec}$. There are two cases.

1) If $h < k$, then $\sum_{j=1}^{h} \tau_j \leq \beta_k + \tau_k \leq \beta_{k+1} + \tau_{k+1}$. Thus, $PT_{opt} = \tau_x + \beta_k + \tau_k + \tau_{k+1} \geq \tau_x + \beta_{k+1} + \tau_{k+1}$. We have the condition that $PT_{opt} = \tau_x + \sum_{j=1}^{k} \tau_j > \beta_{k+1} + \tau_{k+1}$. Then $PT_{opt} = \tau_x + \sum_{j=1}^{k} \tau_j = \tau_x + \max(\sum_{j=1}^{k} \tau_j, \beta_{k+1} + \tau_{k+1}) = PT_{dec}$.

2) If $h > k$, then since $\sum_{j=1}^{h} \tau_j \geq \sum_{j=1}^{k} \tau_j > \beta_{k+1} + \tau_{k+1} \geq \beta_{k+1} + \tau_{k+1}$, we have the condition that $PT_{opt} = \tau_x + \sum_{j=1}^{k} \tau_j \geq \tau_x + \max(\sum_{j=1}^{k} \tau_j, \beta_{k+1} + \tau_{k+1}) = PT_{dec}$.

There is a contradiction in both cases.

For a join, the DSC uses the minimization procedure to minimize the dlevel value of the root, and the solution and the optimal result for a fork are symmetrical.

C. Optimality for In-Trees and Out-Trees

An in-tree is a directed tree in which the root has outgoing degree 0, and other nodes have the outgoing degree 1. An out-tree is a directed tree in which the root has incoming degree 0, and other nodes have the incoming degree 1.

Scheduling in-trees and out-trees is still NP-complete in general as shown by Chretienne [5], and DSC will not give the optimal solution. However, DSC will yield optimal solutions for coarse-grain trees and a class of fine-grain trees.

Coarse-Grain Trees:

Theorem 5.5: DSC gives an optimal solution for a coarse-grain in-tree.

Proof: Since all paths in an in-tree go through the tree root, say, $n_x$, $PT = dlevel(n_x) + tlevel(n_x) + \tau_x$. We claim that dlevel ($n_x$) is minimized by DSC. We prove it by induction on the depth of the in-tree ($d$). When $d = 0$, it is trivial. When $d = 1$, it is a join DAG, and dlevel ($n_x$) is minimized. Assume that it is true for $d < k$.

When $d = k$, let the predecessors of root $n_x$ be $n_1, \ldots, n_m$. Since each sub-tree rooted with $n_i$ has depth $<k$ and the disjoint subgraphs cannot be clustered together by DSC, DSC will obtain the minimum dlevel time for each $n_j$ where $1 \leq j \leq m$ according to the induction hypothesis.

When $n_x$ becomes free, dlevel ($n_x$) = $\max_{1 \leq j \leq m}(CT(n_j) + c_{i,x})$. CT($n_j$) = dlevel($n_j$) + $\tau_j$.

Without loss of generality, assume that ($n_1, n_x$) is in a DS, and that dlevel ($n_1$) + $\tau_1 + c_{i,x}$ has the highest value and dlevel ($n_2$) + $\tau_2 + c_{i,x}$ has the second-highest value. DSC will zero ($n_1, n_x$), and dlevel ($n_x$) = $\max(CT(n_1), \max_{1 \leq j \leq m}(CT(n_j) + c_{i,x}))$. (DSC will not zero any more edges, because of the coarse-grain condition. DSC might zero ($n_2, n_x$) when $g(C) = 1$, but dlevel ($n_x$) does not decrease.)

We need to prove that dlevel ($n_x$) is as small as possible. Because the tree is coarse grain, by Theorem 5.1, linear clustering can be used for deriving the optimal solution. Thus, we can assume that $S^*$ is an optimal schedule that uses
linear clustering. Let $tlevel^*(n_j)$ be the tree level value of $n_j$ in schedule $S^*$, and let $CT^*(n_j) = tlevel^*(n_j) + r_j$. $CT^*(n_j) \geq CT(n_j)$ for $1 \leq j \leq m$ from the assumption that $CT(n_j)$ is minimum. Now we show that $tlevel^*(n_x) \geq tlevel(n_x)$.

There are two cases.

1) If, in $S^*$, the zeroed incoming edge of $n_x$ is $(n_1, n_x)$, then we have the following conditions:

$$
tlevel^*(n_x) = \max(CT^*(n_1), \max_{2 \leq j \leq m} \{CT^*(n_j) + c_{j,x}\}) \geq \max(CT(n_1), \max_{2 \leq j \leq m} \{CT(n_j) + c_{j,x}\}) = tlevel(n_x).
$$

2) If, in $S^*$, the zeroed incoming edge of $n_x$ is not $(n_1, n_x)$, say it is $(n_m, n_x)$. Thus, we have the following condition:

$$
tlevel^*(n_x) = \max(CT^*(n_1), \max_{1 \leq j \leq m-1} \{CT^*(n_{j+1}) + c_{j,x}\}) \geq \max(CT(n_1), \max_{1 \leq j \leq m-1} \{CT(n_{j+1}) + c_{j,x}\}).
$$

Because $g(G) \geq 1$, $CT(n_1) + c_{1,x} \geq \max_{2 \leq j \leq m} \{CT(n_j) + c_{j,x}\}$, we have the following condition: $tlevel^*(n_x) \geq CT(n_x) + c_{j,x} \geq tlevel(n_x)$.

Therefore, we can conclude that $tlevel^*(n_x) \geq tlevel(n_x)$.

DSC solves an in-tree in time $O(v \log v)$, where $v$ is the number of nodes in this tree. It can be verified that the condition $g(G) \geq 1$ for the in-tree optimality of DSC can be relaxed as $g(J_x) \geq 1$ for all join structures $J_x$ in this tree. Under the same or similar condition, Chretienne [4] and Anger, Hwang, and Chow [2] developed $O(v)$ optimal tree scheduling algorithms. These two algorithms are specific to only this kind of tree.

For an out-tree, the forward clustering of DSC may not find the optimum directly. But for the inverted graph, the backward clustering can obtain the optimum, because an inverted out-tree is an in-tree.

Fine-Grain Trees: Finding optimal solutions for general fine-grain trees is NP-complete. However, DSC is able to obtain optimum for a class of fine-grain trees. A single-spawn out-tree is an out-tree such that at most one successor of a nonleaf node, say, $n_x$, can spawn successors. Other successors of $n_x$ are leaf nodes. A single-merge in-tree is an inverse of a single-spawn out-tree. Examples of such trees are shown in Fig. 10.

![Figure 10](image1.png)

Fig. 10. (a) A single-spawn out-tree. (b) A single-merge in-tree.

![Figure 11](image2.png)

Fig. 11. A single-spawn out-tree named $T^{k+1}$ with height $h = k + 1$.

Theorem 5.6: Given a single-spawn out-tree or a single-merge in-tree with an equal computation weight $w$ for each task and an equal communication weight $c$ for each edge, DSC is optimal for this tree.

Proof: We present a proof for an out-tree by induction on the height ($h$) of this tree. The proof for an in-tree is similar. When $h = 2$, it is a fork, and DSC is optimal. Assume that DSC obtains the optimum when $h = k$.

When $h = k + 1$, we assume, without loss of generality, that the successors of root $n_0$ are $n_1, n_2, \ldots, n_j$, and that $n_1$ spawns successors. First, we assume that $n_0$ has more than one successor; i.e., $j > 1$. Fig. 11 depicts this tree. We call the entire tree $T^{k+1}$, and the subtree rooted in $n_1$, $T^k$. The height of $T^k$ is $k$, and it has $q$ tasks where $q > 1$. We claim that DSC will examine all nodes in the subtree $T^k$ first before examining other successors of $n_0$. At step $1$, $n_0$ is examined, and all successors of $n_0$ become free. Node $n_1$ has priority $PRIO(n_1) \geq 2w + 2c$, and other successors of $n_0$ have priority $2w + c$. Then $n_1$ is examined at step 2. $n_0$, $n_1$, and all successors of $n_1$ are added to the free list. The priority of $n_1$’s successors $\geq 3w + c$. Thus, they will be examined before $n_2, \ldots, n_j$. Recursively, all of $n_1$’s descendants will be freed and have priority $\geq 3w + c$. Thus, from step 2 to step $q + 1$, all $q$ nodes in $T^k$ are examined one-by-one. After step $q + 1$, DSC looks at $n_2, n_3, \ldots, n_j$.

Since, from step 2 to $q + 1$, DSC clusters $T^k$ only, DSC obtains an optimal clustering solution for this subtree by the induction hypothesis. We call the parallel time for this subtree $PT_{opt}(T^k)$. Then the parallel time after step $q + 1$ is $PT_{opt}(T^k) = max(w + PT_{opt}(T^k), 2w + c)$.

Let $PT_{desc}$ be the time after the final step of DSC for $T^{k+1}$. We study the following two cases.

1) One edge in $T^k$ is not zeroed by DSC. Then $PT_{opt}(T^k) \geq 2w + c$, implying that $PT_{desc}^* = w + PT_{opt}(T^k)$. Let $PT_{desc}$ be the time of the final step, because the stepwise parallel time of DSC monotonically decreases, $PT_{desc} \geq PT_{desc}^*$. Also, since the optimal parallel time for a graph should be no less than that for its subgraph, $PT_{opt}(T^{k+1}) \geq w + PT_{opt}(T^k)$. Thus, $PT_{desc} \leq PT_{opt}(T^{k+1})$, and DSC is optimal for this case.

2) All edges in $T^k$ are zeroed by DSC. Then $PT_{opt}(T^k) = qw$, and $PT_{desc}^* = max(w + qw, 2w + c)$. If $w + qw \geq 2w + c$, i.e., if $c \leq (q - 1)w$, then $PT_{desc} \leq PT_{desc}^* = w + PT_{opt}(T^k) \leq PT_{opt}(T^{k+1})$, because otherwise $c > (q - 1)w$ and $PT_{desc}^* = 2w + c$. We claim that all edges in $T^k$ and edge $(n_0, n_1)$ should be zeroed by
any optimal clustering for $T^{k+1}$. If they are not, then $PT_{opt}(T^{k+1}) \geq 3\omega + c > PT_{disc}^{k+1} \geq PT_{disc}$, which is impossible. Since all nodes in $T^k$ and $n_0$ are in the same cluster, the optimal clustering for the entire out-tree $T^{k+1}$ can be considered as clustering a fork, with "leaf-node" $n_1$ having a weight $qw$. Because DSC is optimal for a fork, DSC will get the optimum for $T^{k+1}$.

Finally, we examine the case in which $n_0$ has only one successor; i.e., $j = 1$. DSC first zeroes edge $(n_0, n_j)$, and then gets the optimum for $T^k$. Thus, $PT_{disc} = w + PT_{opt}(T^k) = PT_{opt}(T^{k+1})$.

We do not know of another proof of polynomiality of the above class of fine-grain DAG's in the literature. An open question remains: Does there exist a larger class of fine-grain trees that are tractable in polynomial time, say, where the weights are not uniform in the above trees?

VI. A COMPARISON WITH OTHER ALGORITHMS AND EXPERIMENTAL RESULTS

There are many clustering algorithms for general DAG's, e.g., [12], [17], and [19]. A comparison of these algorithms is given in [9]. In this section, we compare DSC with the MD algorithm [19] and the ETF [11]. We also provide an experimental comparison for ETF, DSC, and Sarkar's algorithms.

A. The MD Algorithm

Wu and Gajski [19] have proposed two algorithms: the MCP and MD. We refer the reader to [19] for the description of both algorithms, as well as the description of the terms used in this paragraph. The authors use the notion of as-soon-as-possible (ASAP) starting time $T_S(n_i)$, the as-late-as-possible (ALAP) time $T_L(n_i)$, and the latest finishing time, $T_F(n_i)$. The relative mobility of a node is defined by $(T_L(n_i) - T_S(n_i))/w(n_i)$, where $w(n_i)$ is the task weight.

The MCP algorithm uses $T_L(n_i)$ as a node priority. It then selects the free node with the smallest node priority, which is equivalent to selecting the node with the largest $blevel(n_i)$, and schedules it to a processor that allows its earliest starting time.

The MD algorithm uses the relative mobility as a node priority, and, at each step of scheduling, it identifies a task $n_p$ using the smallest relative mobility. It then examines the available processors starting from PE0, and it schedules $n_p$ to the first processor that satisfies a condition called Fact 1 [19, pp. 336]. An intuitive explanation for Fact 1 is that scheduling a task $n_p$ to a processor $m$ should not increase the length of the current critical path (DS).

The complexity of the original algorithm is $O(v^3)$, as shown in [19, pp. 337]. The corrected version of MD has a better performance, but slightly higher complexity. This is because of the recomputation of $T_S$ and $T_F$ for each scanned processor. For each scheduling step, the complexity of MD is $O(p(v + e))$, where $p$ is the number of scanned processors, and $O(v + e)$ for the mobility information, and checking Fact 1 for each processor, and because there are $v$ steps, the total complexity for the revised MD is $O(p(v + e))$.

The idea of identifying the important tasks in DSC is the same as in MD, i.e., the smallest relative mobility identifies DS nodes that have the maximum tlevel + blevel. However, the way to identify DS nodes is different. The DSC uses a priority function with an $O(log v)$ computing scheme, whereas MD uses the relative mobility function with a computing cost of $O(v + e)$. Another difference is that when DSC picks a DS task $n_p$ to schedule, it uses the minimization procedure to reduce the tlevel of this task and thus decrease the length of DS going through this task. On the other hand, the MD scans the processors from left to right to find the first processor satisfying Fact 1. Even though Fact 1 guarantees the nonincrease of the current critical path, it does not necessarily make the path length shorter.

For a fork or join, MD picks up the DS node at each step, and we can show that it produces an optimal solution. For a coarse-grain tree, as we saw in our optimality proof, the tlevel must be reduced at each step. Since the MD schedules a task to a processor that does not necessarily decrease the tlevel at each step, the MD may not produce the optimum in general. A summary of this comparison is given in Table II.

B. The ETF Algorithm

ETF [11] is a scheduling algorithm for a bounded number of processors with arbitrary network topologies. At each scheduling step, ETF finds a free task whose starting time is the smallest, and then assigns this task to a processor in which the task execution can be started as early as possible. If there is a tie, then the task with the highest blevel is scheduled; in [11], this heuristic is called the ETF/CPF. ETF is designed for scheduling on a bounded number of processors. Thus, to compare the performance of DSC and ETF, we first apply DSC to determine the number of processors (clusters), which we then use as an input to the ETF algorithm.

We discuss the differences and similarities of DSC and ETF as follows. For a node priority, DSC uses tlevel + blevel and then selects the largest node priority. On the other hand, the ETF uses the earliest task first, which is similar to using tlevel as node priority and then selecting the smallest node.
priority for scheduling. For the scheduling step DSC and ETF use the same idea, i.e., try to reduce level by scheduling to a processor that can start a task as early as possible. However, the technique for choosing a processor is different. ETF places a task to the processor that allows the earliest starting time without rescheduling its predecessors, whereas DSC uses the minimization procedure that could reschedule some of the predecessors. It should be mentioned that the MCP [19] algorithm also schedules a task to a processor that allows its earliest starting time as in ETF. However, the node priority for MCP is level, as opposed to level used by ETF.

The complexity of ETF is higher than that of DSC. Since at each step ETF examines all free tasks on all possible processors to find the minimum starting time, the complexity of ETF is \( O(pwv) \), where \( p \) is the number of processors used and \( w \) is the maximum size of the free task list. For \( v \) tasks, the total complexity is \( O(pwnv) \). In our case, \( p = O(v) \) and \( w = O(v) \); thus, the worst complexity is \( O(v^3) \). We have used the balanced searching tree structure for the ETF algorithm in finding the values of a clock variable, NM, [11], pp. 249–250. However, the complexity of ETF for finding the earliest task at each step cannot be reduced, because the earliest starting time of a task depends on the location of processors to be assigned. In practice, the average complexity of ETF could be lower than \( O(v^2) \). For the Choleski decomposition DAG described in Section VI-C, \( p = O(\sqrt{v}) \) and \( w = O(\sqrt{v}) \); thus, the actual complexity is \( O(v^2) \). In Section VI-C, we compare the central processing unit (CPU) time spent for DSC and ETF on a SUN-4 workstation.

For a join DAG, ETF does not use a minimization procedure such as DSC, and it may not be optimal. For a fork, ETF may pick up a task with the earliest starting time; but this task may not be in a DS, and thus ETF may not give the optimum. For a coarse-grain in-tree, ETF places a task to the processor of its successor, which allows the earliest starting time for this task. We can use a similar approach to that in DSC to prove the optimality of ETF for coarse-grain in-trees.

Table II offers a summary of the comparison. Notice the similarities and differences between MCP and ETF and between MD and DSC. For a detailed comparison of MCP and DSC, see [9].

**C. Random DAG's**

Because of the NP-completeness of this scheduling problem, heuristic ideas used in DSC cannot always lead to an optimal solution. Thus, it is necessary to compare the average performance of different algorithms by using randomly generated graphs. Since both the MD and DSC are using the DS to identify the important tasks, we expect a similar performance from both methods. On the other hand, ETF, DSC, and Sarkar's are based on different principles, and it is of interest to conduct an experimental comparison of these three methods.

We have generated 180 random DAG's as follows. We first randomly generate the number of layers in each DAG. We then randomly place a number of independent tasks in each layer. Next we randomly link the edges between tasks at different layers. Finally, we assign random values to task and edge weights. The following statistical information is important for analyzing the performance of scheduling algorithms:

| W | the range of independent tasks in each layer, which approximates the average degree of parallelism; |
| L | the number of layers; and |
| R/C | the average ratio of task weights over edge weights. |

It approximates the graph granularity.

The 180 graphs are classified into three groups of 60 graphs, each based on their R/C values.

**M1:** The R/C range is 0.8–1.2. The average weights of computation and communication are close.

**M2:** The R/C range is 3–10. The graphs are coarse grain.

**M3:** The R/C range is 0.1–0.3. The graphs are fine grain.

Each group is further classified into six subgroups with 10 graphs each, based on the values of \( W \) and \( L \). The results of scheduling groups M1, M2, and M3 are summarized in Tables III, IV, and V. The fifth and sixth columns of the tables show the parallel time improvement ratio of DSC over ETF and Sarkar's algorithm. The improvement ratio of DSC over algorithm A is defined as follows:

\[
DSC/A = 1 - \frac{PT_{dsc}}{PT_A}
\]

For group M1, DSC/ETF shows an improvement by 3%, and DSC/Sarkar's shows an improvement by 20%. For the coarse-grain group M2, the performance differences are insignificant between ETF and DSC and small between Sarkar's and DSC. For the fine-grain group M3, the performance is similar to that of M1, except in the first subgroup, where Sarkar's performs the best. This is because, for this group, the degree of parallelism and the number of layers are small, and the granularity is relatively fine. This implies that communication dominates the computation, and because Sarkar's algorithm reduces the communication volume by zeroing the largest communication edge at each step, it can get the largest reduction sooner. This is not the case for the other subgroups, because then the size of graphs is larger, and DSC is given more opportunities (steps) to zero edges.

A summary of the experiments for the 180 random DAG's is given in Table VI. We list the percentage of cases in which the performance of DSC is better, the same, and worse than that of the other two algorithms. This experiment shows that the average performance of DSC is better than that of Sarkar's, and is slightly better than that of ETF.

To see the differences in complexity between DSC and ETF, and to demonstrate the practicality of DSC, we consider an important DAG in numerical computing, the Choleski...
TABLE IV

<table>
<thead>
<tr>
<th>Layer range</th>
<th>Width Avg/Max</th>
<th>#tasks range</th>
<th>#edges range</th>
<th>DSC/ETF avg</th>
<th>DSC/Sarkar avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>9-10</td>
<td>4/11</td>
<td>38.68</td>
<td>34.187</td>
<td>-2.90%</td>
<td>-5.10%</td>
</tr>
<tr>
<td>9-11</td>
<td>4/11</td>
<td>38.68</td>
<td>34.187</td>
<td>-2.90%</td>
<td>-5.10%</td>
</tr>
<tr>
<td>18-20</td>
<td>5/12</td>
<td>84.132</td>
<td>65.257</td>
<td>2.79%</td>
<td>10.58%</td>
</tr>
<tr>
<td>19-21</td>
<td>6/12</td>
<td>84.132</td>
<td>65.257</td>
<td>2.79%</td>
<td>10.58%</td>
</tr>
<tr>
<td>20-20</td>
<td>9/12</td>
<td>181.277</td>
<td>441.924</td>
<td>3.06%</td>
<td>25.56%</td>
</tr>
<tr>
<td>20-20</td>
<td>9/12</td>
<td>181.277</td>
<td>441.924</td>
<td>3.06%</td>
<td>25.56%</td>
</tr>
<tr>
<td>30-31</td>
<td>9/11</td>
<td>391.474</td>
<td>452.1455</td>
<td>7.02%</td>
<td>33.61%</td>
</tr>
<tr>
<td>30-31</td>
<td>9/11</td>
<td>391.474</td>
<td>452.1455</td>
<td>7.02%</td>
<td>33.61%</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td>2.36%</td>
<td>19.39%</td>
</tr>
</tbody>
</table>

TABLE V

<table>
<thead>
<tr>
<th>Layer range</th>
<th>Width Avg/Max</th>
<th>#tasks range</th>
<th>#edges range</th>
<th>DSC/ETF avg</th>
<th>DSC/Sarkar avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>9-10</td>
<td>4/11</td>
<td>38.68</td>
<td>34.187</td>
<td>-2.90%</td>
<td>-5.10%</td>
</tr>
<tr>
<td>9-11</td>
<td>4/11</td>
<td>38.68</td>
<td>34.187</td>
<td>-2.90%</td>
<td>-5.10%</td>
</tr>
<tr>
<td>18-20</td>
<td>5/12</td>
<td>84.132</td>
<td>65.257</td>
<td>2.79%</td>
<td>10.58%</td>
</tr>
<tr>
<td>19-21</td>
<td>6/12</td>
<td>84.132</td>
<td>65.257</td>
<td>2.79%</td>
<td>10.58%</td>
</tr>
<tr>
<td>20-20</td>
<td>9/12</td>
<td>181.277</td>
<td>441.924</td>
<td>3.06%</td>
<td>25.56%</td>
</tr>
<tr>
<td>20-20</td>
<td>9/12</td>
<td>181.277</td>
<td>441.924</td>
<td>3.06%</td>
<td>25.56%</td>
</tr>
<tr>
<td>30-31</td>
<td>9/11</td>
<td>391.474</td>
<td>452.1455</td>
<td>7.02%</td>
<td>33.61%</td>
</tr>
<tr>
<td>30-31</td>
<td>9/11</td>
<td>391.474</td>
<td>452.1455</td>
<td>7.02%</td>
<td>33.61%</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td>2.36%</td>
<td>19.39%</td>
</tr>
</tbody>
</table>

TABLE VI

A SUMMARY OF THE PERFORMANCE OF THREE ALGORITHMS FOR THE 180 DAGS

<table>
<thead>
<tr>
<th></th>
<th>DSC vs. ETF</th>
<th>DSC vs. Sarkar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg DSC/A</td>
<td>1.91%</td>
<td>15.23%</td>
</tr>
<tr>
<td>#tasks, better</td>
<td>19.62%</td>
<td>78.92%</td>
</tr>
<tr>
<td>#tasks, same</td>
<td>23.33%</td>
<td>0.00%</td>
</tr>
<tr>
<td>#tasks, worse</td>
<td>20.00%</td>
<td>6.10%</td>
</tr>
</tbody>
</table>

TABLE VII

<table>
<thead>
<tr>
<th>n</th>
<th>v</th>
<th>e</th>
<th>CPUesc</th>
<th>CPUutil</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>55</td>
<td>90</td>
<td>5.60%</td>
<td>0.06 sec</td>
</tr>
<tr>
<td>20</td>
<td>201</td>
<td>380</td>
<td>4.15%</td>
<td>0.18 sec</td>
</tr>
<tr>
<td>40</td>
<td>820</td>
<td>1560</td>
<td>2.47%</td>
<td>0.39 sec</td>
</tr>
<tr>
<td>80</td>
<td>2340</td>
<td>4320</td>
<td>1.34%</td>
<td>3.02 sec</td>
</tr>
<tr>
<td>160</td>
<td>12880</td>
<td>25440</td>
<td>0.69%</td>
<td>12.1 sec</td>
</tr>
<tr>
<td>320</td>
<td>51360</td>
<td>102080</td>
<td>0.33%</td>
<td>56.8 sec</td>
</tr>
</tbody>
</table>

ACKNOWLEDGMENT

We are very grateful to V. Sarkar for providing us with the programs of his system in [17] and for his comments and suggestions, and to M.-Y. Wu for his help in clarifying the MD algorithm. We also thank referees for their useful suggestions in improving the presentation of this paper. We thank W. Wang for programming the random graph generator, and A. Darte, P. Sadayappan, and R. Wolski for their comments on this work. The analysis for a fine-grain tree was inspired by a conversation with T. Varvarigou.

REFERENCES


A. Gerasoulis (M’92) received the B.S. degree from University of Ioannina, Greece, and the M.S. and Ph.D. degrees in applied mathematics from the State University of New York at Stony Brook.

He is a Professor of Computer Science at Rutgers University, New Brunswick, NJ. His research interests are in the areas of numerical computing, parallel algorithms and programming, compilers, and parallel languages and environments.

Dr. Gerasoulis has published extensively in both numerical computing and parallel processing areas.

He has participated in the organization of several international conferences, and is an Editor of the Parallel Processing Letters Journal, and Computer and Mathematics with Applications International Journal, and is a Special Issue Editor of Applied Numerical Mathematics Journal.

T. Yang (S’92–M’93) received the B.S. degree in computer science in 1984 and the M.E. degree in artificial intelligence in 1987 from Zhejiang University, China, and the M.S. degree in computer science in 1990 and the Ph.D. degree in computer science in 1993 from Rutgers University, New Brunswick, NJ.

He is an Assistant Professor of Computer Science at the University of California at Santa Barbara. His research interests include algorithms, programming languages, and software environments for parallel and distributed computing.