Task Clustering and Scheduling for Distributed Memory Parallel Architectures

Michael A. Palis, *Senior Member, IEEE*, Jing-Chiou Liou, *Student Member, IEEE*, and David S.L. Wei, *Member, IEEE*

Abstract—This paper addresses the problem of scheduling parallel programs represented as directed acyclic task graphs for execution on distributed memory parallel architectures. Because of the high communication overhead in existing parallel machines, a crucial step in scheduling is *task clustering*, the process of coalescing fine grain tasks into single coarser ones so that the overall execution time is minimized. The task clustering problem is *NP*-hard, even when the number of processors is unbounded and task duplication is allowed. A simple greedy algorithm is presented for this problem which, for a task graph with arbitrary granularity, produces a schedule whose makespan is at most twice optimal. Indeed, the quality of the schedule improves as the granularity of the task graph becomes larger. For example, if the granularity is at least 1/2, the makespan of the schedule is at most 5/3 times optimal. For a task graph with n tasks and e inter-task communication constraints, the algorithm runs in $O(n(n \lg n + e))$ time, which is n times faster than the currently best known algorithm for this problem. Similar algorithms are developed that produce: (1) optimal schedules for coarse grain graphs; (2) 2-optimal schedules for trees with *no* task duplication; and (3) optimal schedules for coarse grain trees with *no* task duplication.

Index Terms—Program task graph, task granularity, task scheduling, distributed memory architectures, approximation algorithms.

1 Introduction

The efficiency of execution of parallel programs on distributed memory parallel architectures critically depends on the strategies used to partition the program into modules or tasks and to schedule these tasks onto physical processors. In a distributed memory architecture, tasks mapped to different processors communicate solely by message-passing. In existing parallel machines, the message-passing overhead is quite large, typically in excess of 500 instruction cycles [6]. This imposes a minimum threshold on task granularity below which performance degrades significantly. To avoid performance degradation, one solution would be to coalesce several fine grain tasks into single coarser grain tasks. This reduces the communication overhead but increases the processing time of the (now coarser grain) tasks. Because of this inherent tradeoff, the goal is to determine the optimal task granularity that results in the fastest overall execution time.

The aforementioned problem, called *grain packing* in [10] or *task clustering* in [7], is well studied. The most commonly used program model is a weighted directed acyclic graph (DAG) in which the node weights represent task execution times and the arc weights represent communication times between tasks when mapped to different processors. Researchers have investigated two versions of this problem, depending on whether or not task duplication (or recomputation) is allowed. In task clustering without duplication, the tasks are partitioned into disjoint sets or clusters and exactly one copy of each task is scheduled. In task clustering with duplication, a task may have several copies belonging to different clusters, each of which is independently scheduled. In general, for the same DAG, task clustering with duplication produces a schedule with a smaller makespan (i.e., total execution time) than when task duplication is not allowed.

For example, for the *fork* DAG shown in Fig. 1a, the optimal makespan with no task duplication is 16 while that with task duplication is 11. Note that when two communicating tasks are mapped to the same processor, the communication delay becomes zero because the data transfer is effectively a local memory write followed by a local memory read.

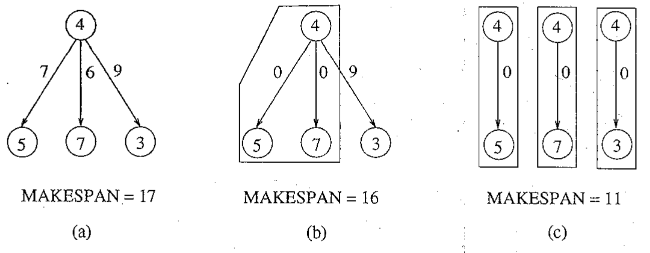


Fig. 1. (a) A fork DAG; (b) optimal clustering with no task duplication; (c) optimal clustering with task duplication.

Task clustering—with or without task duplication—is an *NP*-hard problem [3], [15], [17]. Consequently, practical solutions will have to sacrifice optimality for the sake of

• M. A. Palis and J.-C. Liou are with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102-1982. E-mail: {palis,jing}@sunrise.njit.edu.
• D.S.L. Wei is with the School of Computer Science and Engineering, The University of Aizu, Aizu-Wakamatsu, Fukushima 965, Japan. E-mail: d-wei@u-aizu.ac.jp.

Manuscript received July 20, 1994; revised Jan. 31, 1995.
For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number D95062.

efficiency. For task clustering without duplication, several polynomial-time algorithms have been proposed. Broadly speaking, these algorithms are based on three different heuristics:

1) critical path analysis [7], [9], [17], [18];
2) priority-based list scheduling [2], [8], [11], [16]; and
3) graph decomposition [13].

Recently in [14], the empirical performance of these algorithms were compared based on ten DAGs that model the structure of several practical applications. Nonetheless, none of the algorithms have provable guarantees on the quality of schedules they generate, relative to an optimal schedule. Some algorithms are guaranteed to work well for special DAGs. For coarse grain DAGs (i.e., DAGs whose task execution times are larger than the inter-task communication times), the DSC (Dominant Sequence Clustering) algorithm of [7] has been shown by the authors to give a schedule whose makespan is at most twice optimal, under the assumption that the number of processors is unbounded. The ETF (Earliest Task First) algorithm of [8] gives a schedule whose makespan is at most $(2 - 1/n)M_{opt} + C$, where M_{opt} is the optimal makespan on n processors without considering communication delays and C is the maximum communication delay along some chain of nodes in the DAG. (This chain of nodes is also determined by the algorithm.)

For task clustering with duplication, the situation is quite different. Papadimitriou and Yannakakis [15] have developed a polynomial-time algorithm for arbitrary DAGs that generates a schedule whose makespan is at most twice optimal. Like the DSC algorithm of [7], the PY algorithm does not impose an a priori limit on the number of processors. It attempts to find the best granularity at which to execute the program task graph, irrespective of the actual number of processors. If the number of processors is less than the number of task clusters, a separate *cluster merging* step can be performed to reduce the number of clusters to the number of processors [17], [19]. This two-step approach has the advantage that the task clusters is computed once; only the cluster merging step needs to be re-executed when the number of processors changes. Other algorithms that allow duplication, such as those given in [4], [10], assume that the number of processors n is a problem parameter and hence generate schedules that use at most n processors. These algorithms do not give a performance guarantee as does the PY algorithm.

In terms of the quality of solutions generated, the PY algorithm is theoretically the best known polynomial-time algorithm for task clustering with duplication. However, its time complexity is quite high: $O(|V|^2(|V| \lg |V| + |E|))$ time for a DAG with $|V|$ nodes (or tasks) and $|E|$ arcs. The main source of complexity in the algorithm is the method used to find, for each node v, the cluster that allows v to be executed as early as possible. To find this cluster, the algorithm keeps track of as many as $|V|$ candidate clusters, each of which takes $O(|V| \lg |V| + |E|)$ time to process.

In this paper, we present a new task clustering algorithm that runs $|V|$ times faster than the PY algorithm. Unlike the PY algorithm, the new algorithm uses a simple greedy strategy to find the best cluster for a node v: It maintains only one candidate cluster and "grows" the cluster a node at a time if doing so can potentially decrease the start time of v. In addition, we prove a better performance guarantee by explicitly taking into account the *granularity* of the DAG. For a DAG G with uniform task execution times μ and uniform communication delays λ, the granularity of G is $g(G) = \mu/\lambda$. (For a general DAG, the definition of granularity is given in Section 2.) We show that if $g(G) \geq (1 - \varepsilon)/\varepsilon$ for some $0 < \varepsilon \leq 1$, our algorithm produces a schedule whose makespan is at most $(1 + \varepsilon)$ times the optimal makespan. As a corollary, for a DAG with *arbitrary* granularity (i.e., $g(G) \geq 0$), the algorithm produces a schedule that is at most twice optimal, thus matching the bound of the PY algorithm. However, as $g(G)$ increases, the bound gets better. For example, if $g(G) \geq \frac{1}{2}$ the makespan is at most $\frac{5}{3}$ times optimal.

For coarse grain DAGs (i.e., DAGs whose granularity is at least 1), the task clustering algorithm gives 1.5-optimal schedules. We improve this result by giving a slightly different algorithm that produces *optimal* schedules for coarse grain DAGs.

Finally, we show that the algorithm can be used to solve the task clustering problem with *no* task duplication for directed rooted trees. In particular, we exhibit:

1) 2-optimal schedules for general directed rooted trees and
2) optimal schedules for coarse grain directed rooted trees.

These results are interesting because it is known that task clustering with no duplication is *NP*-hard even when restricted to directed rooted trees [3].

2 Problem Statement

In this paper, a parallel program is modeled as a weighted directed acyclic graph $G = (V, E, \mu, \lambda)$, where each node $v \in V$ represents a task whose execution time is $\mu(v)$ and each arc $(u, v) \in E$ represents the constraint that task u should complete its execution before task v can be started. In addition, u communicates data to v upon its completion; the delay incurred by this data transfer is $\lambda(u, v)$ if u and v reside in different processors and zero otherwise. In other words, task v cannot begin execution until all of its predecessor tasks have completed and it has received all data from these tasks.

Fig. 2 gives an example of a DAG; the node weights denote the task execution times and the arc weights denote the communication delays. Thus, assuming that each task resides in a separate processor, the earliest time that task T4 can be started is 19, which is the time it needs to wait until the data from task T2 arrives (the data from task T1 arrives earlier, at time 10). The makespan of the schedule is the length of the critical path, i.e., the path with the maximum sum of node and arc weights. In Fig. 2, the critical path is indicated by the bold arcs; its length, and hence the makespan of the schedule, is 52.

A *clustering* of G is a mapping of the nodes in V onto *clusters*, where each cluster is a subset of V. If the clusters form a partition of V (i.e., they are pairwise disjoint) then the clustering is said to be *without duplication*. Similarly, if a node is mapped to more than one cluster (i.e., it has more than one *copy*) then the clustering is said to be *with duplication*. For example, for the DAG of Fig. 2, the clustering $\Phi_1 = \{\{T_1, T_4, T_7, T_9\}, \{T_2, T_3, T_5, T_6, T_8, T_{10}\}\}$ is without duplication, while the clustering $\Phi_2 = \{\{T_1\}, \{T_2, T_4, T_5, T_7, T_9\}, \{T_2, T_3, T_5, T_6, T_8, T_{10}\}\}$ is with duplication; in the latter, nodes T_2 and T_5 each have two copies.

A *schedule* for a clustering Φ maps the clusters of Φ to processors and assigns to each node v a start time $s(v, p)$ on every processor p to which v is mapped. The schedule should satisfy the following condition for every node v: if v is mapped to processor p then, for every immediate predecessor u of v, there is some processor q to which u is mapped such that $s(v, p) \geq s(u, q) + \mu(u) + \lambda'(u, v)$, where $\lambda'(u, v) = \lambda(u, v)$ if $p \neq q$ and $\lambda'(u, v) = 0$ if $p = q$. Let $s(v) = \min\{s(v, p) \mid v \text{ is mapped to processor } p\}$. The *makespan* of the schedule is given by $\max\{s(v) + \mu(v) \mid v \text{ is a sink node}\}$.

A schedule S is optimal for a clustering Φ if for every other schedule S' for Φ, it is the case that *makespan*$(S) \leq$ *makespan*(S'). We define the makespan of a clustering Φ as the makespan of its optimal schedule S. A clustering Φ is optimal for a DAG G if for every other clustering Φ' for G, *makespan*$(\Phi) \leq$ *makespan*(Φ'). Fig. 3 gives a schedule for the clustering $\Phi_2 = \{\{T_1\}, \{T_2, T_4, T_5, T_7, T_9\}, \{T_2, T_3, T_5, T_6, T_8, T_{10}\}\}$ defined earlier. In the schedule, the three clusters are mapped to distinct processors; the value s_i beside the node denotes the start time of task T_i on the designated processor. The makespan of this schedule is 26. It turns out that this schedule is optimal for clustering Φ_2. It also turns out that Φ_2 is an optimal clustering for the DAG of Fig. 2. Therefore, the shortest possible execution time for the DAG is 26.

Like the DSC algorithm of [7] and the PY algorithm of [15], we assume that there is no a priori limit on the number of processors. Therefore, there are always enough processors to host the clusters of any clustering. The problem addressed in this paper is the following: *Given a DAG G and assuming that task duplication is allowed, find an optimal clustering for G.*

The granularity of the DAG is an important parameter which we take into account when analyzing the performance of our algorithms. We adopt the definition of granularity given in [7]. Let $G = (V, E, \mu, \lambda)$ be a weighted DAG. For a node $v \in V$, let

$$g_1(v) = \min\{\mu(u) \mid (u, v) \in E\} / \max\{\lambda(u, v) \mid (u, v) \in E\}$$

and

$$g_2(v) = \min\{\mu(w) \mid (v, w) \in E\} / \max\{\lambda(v, w) \mid (v, w) \in E\}.$$

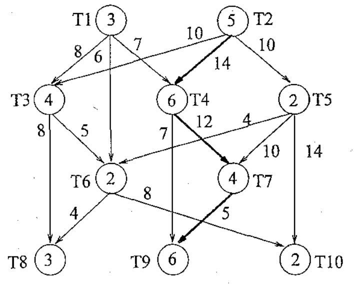


Fig. 2. An example DAG.

The *grain-size* of v is defined as $\min\{g_1(v), g_2(v)\}$. The *granularity* of DAG G is given by $g(G) = \min\{g(v) \mid v \in V\}$. One can verify that for the DAG G of Fig. 2, $g(G) = \frac{1}{7}$.

3 The Task Clustering Algorithm

This section presents a greedy algorithm that finds a clustering for a given DAG $G = (V, E, \mu, \lambda)$. We prove that the clustering $\Phi(G)$ produced by the algorithm is "good" in the following sense: If $g(G) \geq (1 - \varepsilon)/\varepsilon$ for some $0 < \varepsilon \leq 1$, then the makespan of $\Phi(G)$ is at most $(1 + \varepsilon)$ times the makespan of the optimal clustering for G. As a corollary, for a DAG with *arbitrary* granularity (i.e., $g(G) \geq 0$), the clustering produced by the algorithm has a makespan which is at most twice optimal, thus matching the bound of the PY algorithm [15]. However, as $g(G)$ increases, the bound gets better. For example, if $g(G) \geq \frac{1}{2}$, the makespan of the clustering is at most $\frac{5}{3}$ times optimal.

The basic idea behind the algorithm is similar to the PY algorithm. For each $v \in V$ we first compute a lower bound $e(v)$ on the earliest possible start time of v. This is accomplished by finding a cluster, $C(v)$, containing v that allows v to be started as early as possible when all the nodes in $C(v)$ are executed on the same processor and all other nodes in $V - C(v)$ are executed on other processors. This cluster can be determined using a simple greedy algorithm which (unlike the PY algorithm) grows the cluster one node at a time. Once the clusters are determined, they are mapped to processors in a simple way, and we show that this mapping has a schedule whose makespan is "good" in the sense described in the previous paragraph.

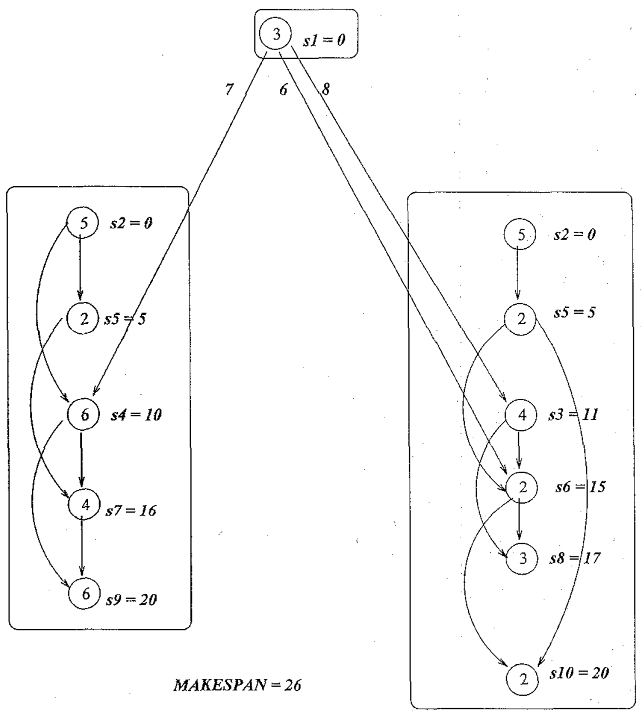


Fig. 3. An optimal clustering for the DAG of Fig. 2.

3.1 Computing the *e* Value

The *e* values are computed in topological order of the nodes of *G*. For a source node, its *e* value is equal to zero. For any other node, its *e* value is computed after all of its ancestors have been assigned *e* values. Consider a node v all of whose ancestors have been assigned *e* values, and suppose we wish to compute the $e(v)$. Since $e(v)$ is a lower bound on the start time of v, it suffices to look at clusters C consisting of v and a subset of its ancestors. If a cluster C' contains a node w which is not an ancestor of v, removing w from C' results in a cluster in which v can be started possibly sooner, but never later, than v's start time in C'.

Let C be a cluster consisting of node v and a subset of its ancestors $\{u_1, \ldots, u_k\}$. We wish to find a lower bound $e_C(v)$ on the earliest start time of v assuming that all nodes in C are executed on the same processor. Ignore for the moment the arcs that *cross* C, i.e., those that connect a node not in C to a node in C. What is the earliest time that v can be scheduled? Clearly, the answer is the makespan of the optimal schedule for the one-processor scheduling problem with release times for the instance $\{u_1, \ldots, u_k\}$ with $e(u_i)$ and $\mu(u_i)$ being the release time and execution time, respectively, of task u_i. This problem is solved optimally by the greedy algorithm that executes the tasks in nondecreasing order of release times. Therefore, for the cluster C,

$$e_C(v) \geq \text{GREEDY-SCHEDULE}(C - \{v\}) \qquad (1)$$

where GREEDY-SCHEDULE(•) returns the makespan of the one-processor schedule for the set of tasks specified by its argument.

Next consider the set of arcs that cross C. For such an arc (u, w), define its *c value* as $c(u, w) = e(u) + \mu(u) + \lambda(u, w)$. Node v cannot be scheduled before time $c(u, w)$ because there is a path from u to v through node w. Therefore,

$$e_C(v) \geq \text{MAX-C-VALUE}(C) \qquad (2)$$

where MAX-C-VALUE(C) is the maximum *c* value among the arcs that cross C.

From (1) and (2), it follows that for a given cluster C,

$$e_C(v) \geq \max\left\{\begin{array}{c}\text{GREEDY-SCHEDULE}(C - \{v\}) \\ \text{MAX-C-VALUE}(C)\end{array}\right\} \qquad (3)$$

and

$$e(v) \geq \min_C\{e_C(v)\}. \qquad (4)$$

The problem is to find the cluster C for which $e_C(v)$ is minimum. We show that this cluster can be found using a simple greedy algorithm. Starting with node v, the algorithm "grows" the cluster a node at a time and checks if the new cluster can potentially decrease the current estimate for the *e* value of v. If growing the cluster can only increase the *e* value, the algorithm stops and returns the minimum *e* value obtained.

Suppose we have found a candidate cluster C; hence $e_C(v)$ satisfies (3). We have the following two cases:

CASE 1. MAX-C-VALUE(C) > GREEDY-SCHEDULE(C − $\{v\}$). Let (u, w) be an arc that crosses C such that $c(u, w)$ = MAX-C-VALUE(C). Since $e_C(v) \geq c(u, w)$, an *e* value less than $e_C(v)$ cannot be obtained as long as node u is outside of the cluster C. Therefore, C must be grown to include node u.

CASE 2. GREEDY-SCHEDULE(C − $\{v\}$) ≥ MAX-C-VALUE(C). Since $e_C(v) \geq$ GREEDY-SCHEDULE(C − $\{v\}$), then adding *any* new node x to C cannot decrease the *e* value because GREEDY-SCHEDULE(C ∪ $\{x\}$ − $\{v\}$) ≥ GREEDY-SCHEDULE(C − $\{v\}$).

Case 1 gives the criterion for growing the candidate cluster while case 2 gives the stopping criterion. The complete algorithm for computing $e(v)$ is given as Algorithm COMPUTE-E-VALUE below. For the DAG of Fig. 2, Fig. 4 shows the *e* values computed by the algorithm. Fig. 5 illustrates how the *e* value of node 10 is computed.

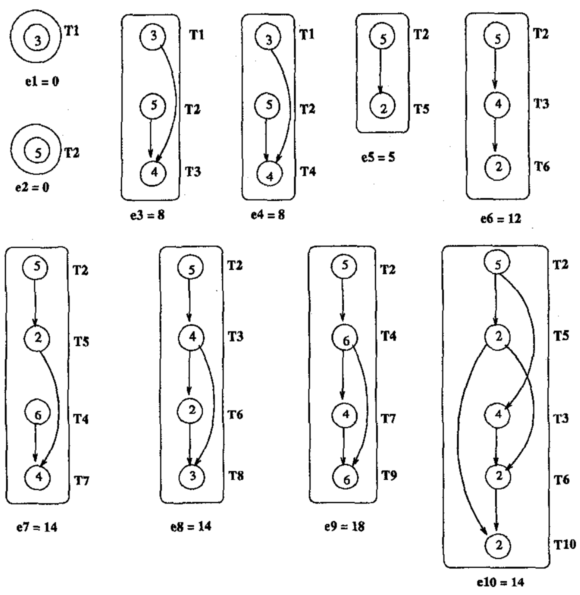


Fig. 4. The *e* values and clusters for the DAG of Fig. 2.

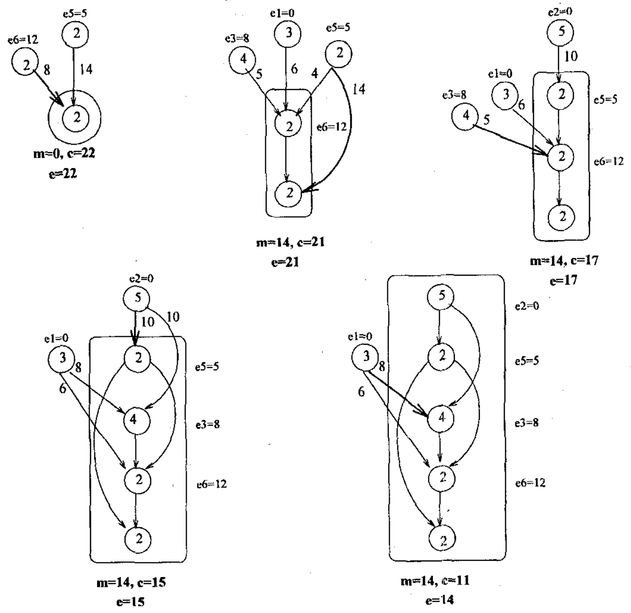


Fig. 5. Computing the *e* value of node 10; critical arcs are in bold.

```
1) Algorithm COMPUTE-E-VALUE($v$, $G$)
2) begin
3)   if $v$ is a source node then return (0);
4)   $C \leftarrow \{v\}$;
5)   $m \leftarrow 0$;
6)   $c \leftarrow$ MAX-C-VALUE($C$);
7)   $e \leftarrow c$;
8)   while $m < c$ do
9)     let($u$, $w$) be an arc such that $u \notin C$, $w \in C$, and $c = c(u, w)$;
10)    $C \leftarrow C \cup \{u\}$;
11)    $m \leftarrow$ GREEDY-SCHEDULE($C - \{v\}$);
12)    $c \leftarrow$ MAX-C-VALUE($C$);
13)    $e \leftarrow \min\{e, \max\{m, c\}\}$;
14)  endwhile;
15)  return ($e$);
16) end COMPUTE-E-VALUE.
```

Let v be a non-source node. In the algorithm, each iteration of the **while** loop chooses an arc in step 9 for inclusion in the candidate cluster C. Call such arcs *critical*. Clearly, the set of critical arcs forms a tree T with root v, as illustrated in Fig. 6. It follows that, for any cluster of nodes that includes v but excludes some other node in T, there is at least one critical arc that crosses it. Let m_i and c_i be the m and c values, respectively, that are computed in lines 11 and 12 at iteration i of the **while** loop, and let m_0 and c_0 be the initial values before entering the loop. Furthermore, let $e_i = \max\{m_i, c_i\}$. Finally, let t be the last iteration of the loop.

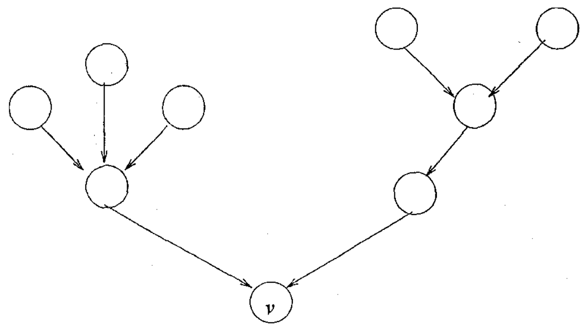


Fig. 6. The set of critical arcs forms a tree T with root v.

From the algorithm, it is clear that $m_i < c_i$ for $0 \le i < t$ and that $m_t \ge c_t$. Let k be the least integer such that $e_k = \min_{0 \le i \le t} \{e_i\} = e(v)$. Thus, for $0 \le i < t$, $c_i = \max\{m_i, c_i\} = e_i \ge e_k$. Similarly, $m_t = \max\{m_t, c_t\} = e_t \ge e_k$. Therefore,

Fact 1. For $0 \le i < t$, $c_i \ge e(v)$; and $m_t \ge e(v)$.

We are now ready to prove the following.

THEOREM 1. *Algorithm COMPUTE-E-VALUE(v, G) returns a lower bound $e(v)$ on the earliest start time of node v.*

PROOF. The proof is by induction on the depth of node v. The theorem is obviously true for source nodes. So let v be a non-source node and assume that the algorithm holds for v's ancestors. Suppose to the contrary that there is a cluster C' containing v such that v can be started at time $e' < e(v)$. Then every critical arc in the tree T (see Fig. 6) must lie inside C'. If not, $e' \ge c_i$ for some i, $0 \le i < t$, which by Fact 1 implies that $e' \ge e(v)$, a contradiction. On the other hand, if all critical arcs in T are in C', then $e' \ge m_t \ge e(v)$, again a contradiction. Therefore, there is no cluster C' containing v such that v can be started earlier than time $e(v)$. □

3.2 Constructing the Schedule

Algorithm COMPUTE-E-VALUE (v, G) can be easily modified so that it returns, in addition to $e(v)$, the corresponding cluster $C(v)$. We now describe how to construct a clustering $\Phi(G)$ for G whose makespan is at most $(1 + \varepsilon)$-optimal if $g(G) \ge (1-\varepsilon)/\varepsilon$.

$\Phi(G)$ is constructed by visiting the nodes of G in reverse topological order (i.e., from sink nodes to source nodes). Initially, $\Phi(G) = \emptyset$ and the sink nodes of G are "marked." The following steps are then performed until there are no more marked nodes:

1) Pick a marked node v and add $C(v)$ to $\Phi(G)$.
2) Unmark v and mark all nodes u for which there is an arc(u, w) such that $u \notin C(v)$ and $w \in C(v)$.

To schedule $\Phi(G)$, we map each cluster in $\Phi(G)$ to a distinct processor and execute the nodes mapped to the same processor in nondecreasing order of their e values. If node w is mapped to processor p, we let $s(w, p)$ denote the start time of w in p. Note that each processor p holds exactly one cluster and that this cluster is $C(v)$ for some marked node v. Morover, v is the last node executed in this cluster. We now prove the following.

LEMMA 1. *Let v be a marked node such that $C(v)$ is mapped to processor p. If $g(G) \ge (1 - \varepsilon)/\varepsilon$ for some $0 < \varepsilon \le 1$, then $s(v, p) \le (1 + \varepsilon)\, e(v)$.*

PROOF. The proof is by induction on the depth of marked node v. The theorem is true for all marked nodes v that do not have ancestors which are also marked nodes because in this case $s(v, p) = e(v)$. Now consider a marked node v and suppose that the theorem holds for all of its ancestors that are also marked nodes. For each node $w \in C(v)$, let (u, w) be an arc with the maximum c value among all arcs that cross $C(v)$ and ends at w. Thus u is a marked node. It follows that w can be started at time

$$s(w, p) \le (1 + \varepsilon)e(u) + \mu(u) + \lambda(u, w).$$

Since u is a predecessor of w, $e(w) \ge e(u) + \mu(u)$. Therefore,

$$s(w, p) \le e(w) + \varepsilon e(u) + \lambda(u, w).$$

Now $\mu(u)/\lambda(u, w) \ge (1 - \varepsilon)/\varepsilon$; thus, $\varepsilon[\mu(u) + \lambda(u, w)] \ge \lambda(u, w)$. It follows that

$$s(w, p) \le e(w) + \varepsilon[e(u) + \mu(u) + \lambda(u, w)].$$

$$\le e(w) + \varepsilon e(v),$$

because $e(v) \ge e(u) + \mu(u) + \lambda(u, w)$. The last inequality implies that every node in $C(v)$ can be started at its e value plus a delay of at most $\varepsilon e(v)$. Therefore, v can be started at time $s(v, p) \le (1 + \varepsilon)e(v)$. □

THEOREM 2. *If $g(G) \geq (1 - \varepsilon)/\varepsilon$ for some $0 < \varepsilon \leq 1$, then the makespan of $\Phi(G)$ is at most $(1 + \varepsilon)$ times the makespan of an optimal clustering for G.*

PROOF. Let M_{opt} be the makespan of an optimal clustering for G. Then $M_{opt} \geq \max\{e(v) + \mu(v)\}$, over all sink nodes v of C. Since every sink node v is a marked node, then by Lemma 1,

$$\begin{aligned} makespan(\Phi(G)) &\leq \max\{(1 + \varepsilon)e(v) + \mu(v)\} \\ &\leq \max\{(1+\varepsilon)[e(v) + \mu(v)]\} \\ &= (1 + \varepsilon)\max\{e(v) + \mu(v)\} \\ &\leq (1 + \varepsilon)M_{opt}. \end{aligned}$$ □

Fig. 7 shows the clustering constructed by the procedure for the DAG G of Fig. 2. The figure also shows the start times of the nodes when the clusters are mapped to distinct processors. The makespan of the clustering is 27. On the other hand, the makespan of an optimal clustering is at least $e(T_9) + \mu(T_9) = 18 + 6 = 24$. The granularity of G is $g(G) = \frac{1}{7}$. Setting $\frac{1}{7} = (1 - \varepsilon)/\varepsilon$ gives $\varepsilon = \frac{7}{8}$. Thus, $27 \leq \left(1 + \frac{7}{8}\right) * 24 = 45$ as predicted by Theorem 2. Note that while 24 is a lower bound on the optimal makespan, the optimal makespan is 26, as shown in Fig. 3. Therefore, the clustering produced by the algorithm is actually closer to optimal than predicted.

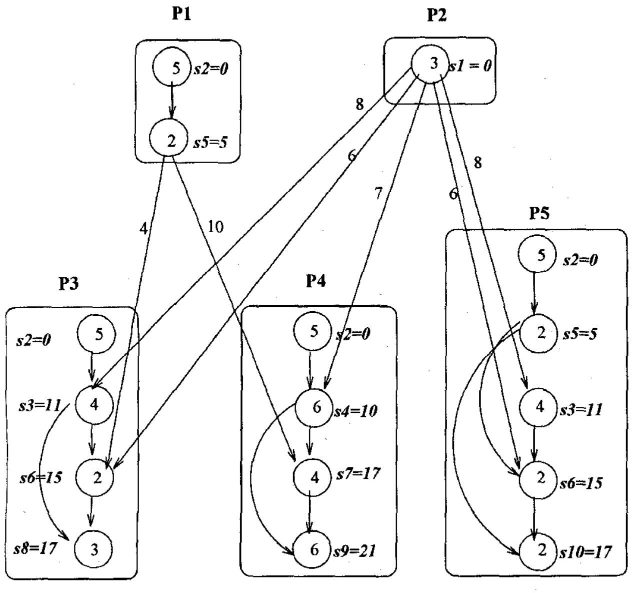


Fig. 7. A clustering and schedule for the DAG of Fig. 2.

3.3 Complexity Analysis

In this subsection, we describe the implementation details and derive the time complexity of the algorithm. The runtime of Algorithm COMPUTE-E-VALUE depends on how the functions MAX-C-VALUE and GREEDY-SCHEDULE are computed. MAX-C-VALUE(C) is computed as follows: We maintain a Fibonacci heap H [5] whose elements are the nodes of the DAG. For each node u we associate a value $key[u]$ which equals the maximum c value among all arcs that cross cluster C and emanate from u. (If no such arc exists, $key[u] = -\infty$.) The following operations are performed on H.

- EXTRACT-MAX(H): deletes from H the node with largest key.
- INCREASE-KEY(H, x, k): increases the key of node x in H to the value k.

The algorithm grows the cluster C by adding a node u from which emanates an arc with the maximum c value that crosses C. This node u is obtained by performing EXTRACT-MAX(H). Now, adding u to C reveals new arcs (x, u) that cross C. Therefore, heap H is updated by performing INCREASE-KEY($H, x, c(x, u)$) for each such arc (x, u).

EXTRACT-MAX is executed at most $|V|$ times. INCREASE-KEY is called once for each new arc that crosses C and hence is executed at most $|E|$ times. For a Fibonacci heap with $|V|$ elements, an EXTRACT-MAX operation can be performed in $O(\lg |V|)$ amortized time and an INCREASE-KEY operation in $O(1)$ amortized time. Therefore, excluding the calls to function GREEDY-SCHEDULE, the algorithm runs in $O(|V| \lg |V| + |E|)$ time.

Next consider the implementation of function GREEDY-SCHEDULE. In the algorithm, each subsequent call to GREEDY-SCHEDULE adds a single node (task) to the argument set. Moreover, the makespan of the greedy schedule is obtained by executing the tasks in the set in nondecreasing order of e values. Therefore, if the tasks are initially sorted, a new task can be inserted in the sorted list using binary search. However, although insertion can be done in $O(lg\ |V|)$ time, computing the makespan of the schedule for the new list will take $O(|V|)$ time.

By using a 2-3 tree T [1], we can reduce the time to compute the makespan to $O(\lg |V|)$. In T, every internal node has either two or three children and all leaves are at the same distance from the root. Given a set of tasks and their e values, we store the tasks in the leaves of T in sorted order, i.e., arranged from left to right in nondecreasing order of e values. We say that node α "owns" the sublist of tasks stored in the subtree rooted at α. Node α contains the following pieces of information:

- $e[\alpha]$: The maximum e value among all tasks owned by α.
- $s[\alpha]$, $f[\alpha]$: If the tasks owned by α are scheduled greedily, then $s[\alpha]$ is the start time of the first task executed and $f[\alpha]$ is the finish time of the last task executed. (Note the $f[\alpha]$ is also the makespan of the greedy schedule for this sublist of tasks.)
- $d[\alpha]$: The idle time in the greedy schedule for the sublist tasks owned by α; i.e., the number of time units between $s[\alpha]$ and $f[\alpha]$ during which no task is being executed.

Fig. 8 illustrates how the 2-3 tree T evolves for the given sequence of tasks. To insert a new task x, we traverse the tree downward from the root to locate the point of insertion, insert a new leaf corresponding to task x, then traverse the tree upward towards the root to update the information of the nodes affected by the insertion. Observe that an insertion may cause some nodes along the traversed path to have four children, in which case the node is split into two nodes, each with two children. The details of insertion and node splitting can be found in [1].

TASK	*e*	*u*
1	2	5
2	15	6
3	8	1
4	6	2
5	14	4

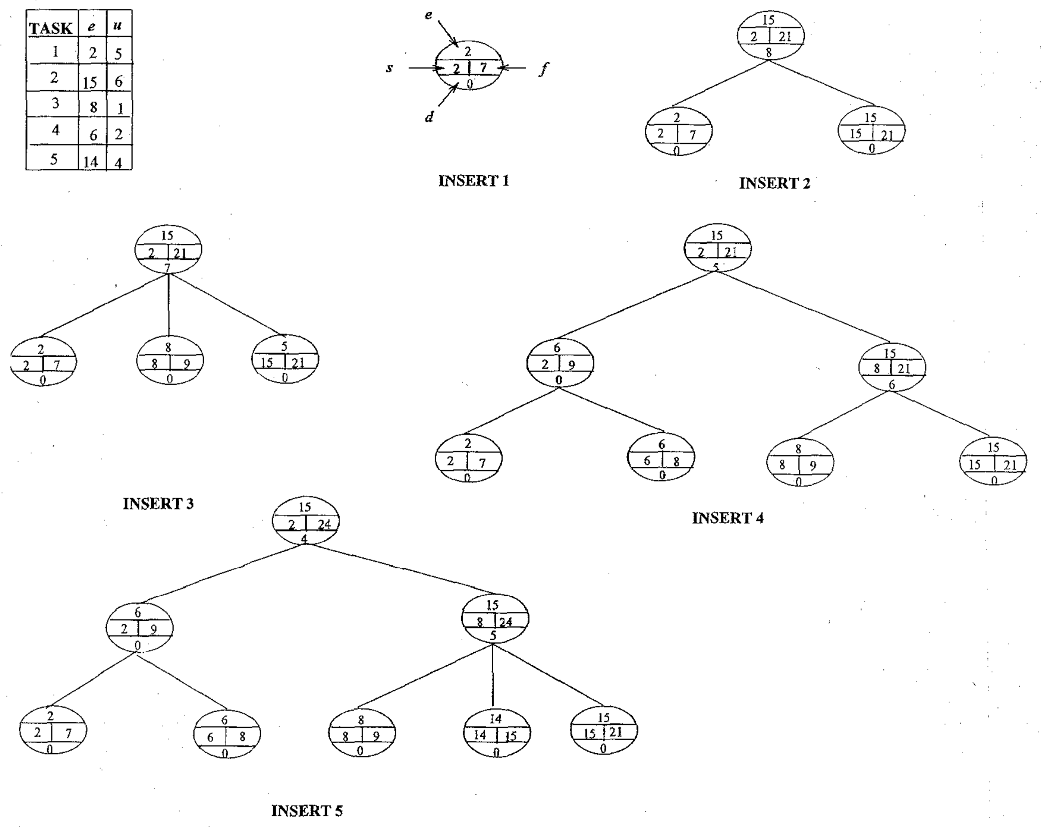


Fig. 8. The 2-3 tree *T*.

Fig. 9 shows how a node's key values are updated, given the key values of its children. The proof of correctness is straightforward and is omitted here. Note that only the nodes along the root-to-leaf path are updated and that an update takes constant time.

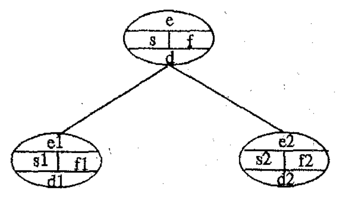
Since *T* has at most $|V|$ leaves, its depth is $O(\lg|V|)$ and hence insertion takes $O(lg|V|)$ time. Moreover, the makespan of the greedy schedule for the tasks currently stored in *T* can be found in $O(1)$ time as it is simply $f\,[root(T)]$. It follows that the calls to GREEDY-SCHEDULE in Algorithm COMPUTE-E-VALUE contributes $O(|V|\lg|V|)$ to the total runtime. Therefore, Algorithm COMPUTE-E-VALUE runs in $O(|V|\lg|V| + |E|)$ time. Since the algorithm is called $|V|$ times, computing the *e* values of all nodes takes $O(|V|(|V|\lg|V| + |E|))$ time.

Finally, once the *e* values are computed, the clustering can be constructed in $O(|V| + |E|)$ time. Therefore, the entire task clustering algorithm takes $O(|V|(|V|\lg|V| + |E|))$ time.

4 Special Cases

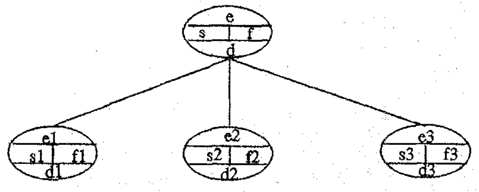
4.1 Coarse Grain DAGs

A DAG *G* is *coarse grain* if $g(G) \geq 1$; otherwise it is *fine grain*. For coarse grain DAGs, the task clustering algorithm of the previous section produces a clustering whose makespan is at most 1.5 times optimal. We show that by slightly modifying the algorithm an optimal clustering can be obtained. Before showing this result, we prove some properties of coarse grain DAGs.



e = e2
s = s1
f = f2+max{0,f1-(s2+d2)}
s = d1+max{0,(s2+d2)-f1}

(a)



e = e3
s = s1
f' = f2+max{0,f1-(s2+d2)}
d' = d1+max{0,(s2+d2)-f1}
f = f3+max{0,f'-(s3+d3)}
d = d'+max{0,(s3+d3)-f'}

(b)

Fig. 9. Updating a node with (a) two children;(b) three children.

Let $G = (V, E, \mu, \lambda)$ be a coarse grain DAG. For $v \in V$, let $e(v)$ be the e value returned by Algorithm COMPUTE-E-VALUE(v, G) and let $C(v)$ be the corresponding cluster. For nodes $u, v, \in V$, call u a *critical predecessor* of v if and only if $c(u, v) = \max\{c(w, v) \mid (w, v) \in E\}$.

LEMMA 2. *If u is a critical predecessor of v, then $e(v) \geq \max\{e(u) + \mu(u), \max\{c(w, v) \mid (w, v) \in E \text{ and } w \neq u\}\}$.*

PROOF. The claim obviously holds if $e(u) + \mu(u) \geq \max\{c(w, v) \mid (w, v) \in E$ and $w \neq v\}$ because $e(v) \geq e(u) + \mu(u)$ (u is a predecessor of v). So assume to the contrary that $e(v) < c(x, v)$ for some $x \neq u$. (Note that $c(u, v) \geq c(x, v)$.) Then x and u must both be executed on the same processor as v. Therefore, $e(v) \geq$ GREEDY-SCHEDULE ($\{u, x\}$). We have two cases:

Case 1. $e(u) \geq e(x)$. Then

$$\begin{aligned} e(v) &\geq \max\{e(u) + \mu(u), e(x) + \mu(x) + \mu(u)\} \\ &\geq e(x) + \mu(x) + \mu(u) \\ &\geq e(x) + \mu(x) + \lambda(x, v), \text{ since } g(v) \geq 1 \\ &= c(x, v) \\ &> e(v), \text{ a contradiction.} \end{aligned}$$

Case 2. $e(x) > e(u)$. Then

$$\begin{aligned} e(v) &\geq \max\{e(x) + \mu(u), e(u) + \mu(u) + \mu(x)\} \\ &\geq e(u) + \mu(u) + \mu(x) \\ &\geq e(u) + \mu(u) + \lambda(u, v), \text{ since } g(v) \geq 1 \\ &= c(u, v) \\ &\geq c(x, v) \\ &> e(v), \text{ a contradiction.} \end{aligned}$$ □

A cluster of nodes $C = \{u_1, ..., u_k\}$ is called a *critical chain* if and only if for $1 \leq i < k$, u_{i+1} is a critical predecessor of u_i. The *head* of the chain is u_1 and the tail is u_k.

LEMMA 3. *For every node $v \in V$, $C(v)$ is a critical chain.*

PROOF. The proof is by induction on the number of iterations of the **while** loop of Algorithm COMPUTE-E-VALUE(v, G). Let C_i be the cluster at iteration i of the loop. Clearly, C_1 is a critical chain since it consists of v and its critical predecessor. Suppose that for some $k \geq 1$, the clusters C_i ($1 \leq i \leq k$) are critical chains. Let C_k be $\{v, w_1, ..., w_k\}$. Therefore m_k = GREEDY-SCHEDULE ($\{w_1, ..., w_k\}$) and c_k = MAX-C-VALUE($\{v, w_1, ..., w_k\}$). If $m_k \geq c_k$ then the loop terminates and the algorithm returns a cluster C_j, $j \leq k$, which is a critical chain. Thus the claim holds.

Suppose that $m_k < c_k$. Then the algorithm executes iteration $k + 1$. We show that C_{k+1} is also a critical chain. Let (x, w_j) be an arc with maximum c value that crosses C_{k+1}; thus $c(x, w_j) = c_k$ and $j \in \{0, ..., k\}$. If $j = k$ then x is a critical predecessor of w_j. Therefore $C_{k+1} = C_k \cup \{x\}$ is a critical chain, and the claim holds. So suppose that $j < k$. It follows that $m_k < c(x, w_j) \leq c(w_{j+1}, w_j)$ (since w_{j+1} is a critical predecessor of w_j). But:

$$\begin{aligned} m_k &\geq e(w_{j+1}) + \mu(w_{j+1}) + \mu(w_j) \\ &\geq e(w_{j+1}) + \mu(w_{j+1}) + \lambda(w_{j+1}, w_j), \text{ since } g(w_{j+1}) \geq 1 \\ &= c(w_{j+1}, w_j) \\ &> m_k, \text{ a contradiction.} \end{aligned}$$

Therefore, x must be a critical predecessor of w_k and hence C_{k+1} is a critical chain. □

Consider two critical chains C_1 and C_2 such that $tail(C_1) = head(C_2)$. We define $C_1 \oplus C_2$ as the critical chain that results when $tail(C_1)$ is replaced by C_2. Finally, for $v \in V$, let $C^*(v)$ be the critical chain returned by the following steps:

```
repeat
  let $w = tail(C(v))$;
  $C(v) \leftarrow C(v) \oplus C(w)$;
until $C(w) = \{w\}$.
```

To construct a clustering for G, we proceed as before: i.e., we begin by computing the e values and the clusters of the nodes of G using Algorithm COMPUTE-E-VALUE. Next, we compute $\Phi(G)$ except that now we add $C^*(v)$ to $\Phi(G)$ (instead of $C(v)$). We now prove the following.

THEOREM 3. *Let v be mapped to processor p and let its start time on p be $s(v, p)$. Then $s(v, p) = e(v)$.*

PROOF. The theorem is trivially true for all source nodes. Let v be a non-source node and assume that all of its immediate predecessors have start times equal to their e values. Suppose that v resides in critical chain C^*. We have two cases:

Case 1. $v \neq tail(C^*)$. Let u be the critical predecessor of v in C^*. Then $s(v, p) = \max\{e(u) + \mu(u), \max\{c(w, v) \mid (w, v) \in E \text{ and } w \notin C^*\}\} \leq e(v)$ by Lemma 2. Therefore, $s(v, p) = e(v)$.

Case 2. $v = tail(C^*)$. By definition of C^*, it should be the case that $C(v) = \{v\}$. Hence, $s(v, p) = \max\{c(w, v) \mid (w, v) \in E\} = e(v)$. □

4.2 Trees

An *intree* is a directed rooted tree in which every arc is directed from a node to its parent. An *outtree* is similar except that every arc is directed from a node to its children. If duplication is allowed, the task clustering problem for outtrees can be solved optimally using the following simple algorithm [3]: Map every root-to-leaf path to a processor and execute each node as its e value. If duplication is not allowed, the task clustering problem for outtrees is *NP*-complete [3]. For the case of intrees, it was shown in [3] that the task clustering problem with no task duplication is also *NP*-complete. Moreover, allowing duplication does not help because duplication tasks can always be removed without increasing the makespan.

An interesting question is whether there are good approximation algorithms for scheduling intrees and outtrees when no duplication is allowed. The answer is affirmative: Our greedy algorithm can construct $(1 + \varepsilon)$-optimal schedules for both intrees and outtrees.

COROLLARY 1. *When duplication is not allowed, there is a polynomial-time algorithm that constructs a clustering for an intree or an outtree with granularity at least $(1 - \varepsilon)/\varepsilon$ whose makespan is at most $(1 + \varepsilon)$ times the makespan of an optimal clustering.*

PROOF. Let v be a node in a DAG G. It is easy to verify that for the clustering produced our algorithm, a necessary condition for v to have duplicates is that it has two descendants u_1 and u_2 such that one of these nodes, say u_1, is reachable from v via some path that does not contain u_2. If G is an intree, this condition is never true. Thus, the clustering produced by our algorithm contains no duplicate nodes if the DAG is an intree. Next, consider an outtree T. T can be converted into an intree T' by reversing the direction of every arc in T. Moreover, the granularities of T and T' are equal. Given a schedule S' for T' (with no task duplication), a schedule S for T with the same makespan can be derived by defining the start time of node v in S as makespan(S')–finish time of v in S' [3]. A similar transformation can be made from any schedule for T to a schedule for T'. Therefore, for an outtree with granularity at least $(1 - \varepsilon)/\varepsilon$, a $(1 + \varepsilon)$-optimal can be obtained by first converting it into an intree T', then computing a $(1 + \varepsilon)$-optimal clustering for T'. □

COROLLARY 2. *When duplication is not allowed, there are polynomial-time algorithms that construct optimal clusterings for* coarse grain *intrees and outtrees.*

PROOF. Follows from the fact that our greedy algorithm, when applied to coarse grain DAGs, produces schedules with optimal makespans. □

5 Open Problems

Is there a polynomial-time algorithm for task clustering with duplication that produces schedules which are better than 2-optimal for DAGs with *arbitrary* granularity? Our task clustering algorithm guarantees $(1 + \varepsilon)$-optimal schedules for some $0 < \varepsilon \leq 1$, but ε depends on the granularity of the DAG.

How well does the greedy algorithm perform in practice? Work similar to that done in [14] can be carried out to compare the performance of different algorithms for task clustering with duplication on practical applications.

Our task clustering algorithm, as do the PY [15] and DSC [7] algorithms, assumes an unbounded number of processors. A separate cluster merging step is needed to map the clusters to fewer number of processors. How does this two-step approach compare, in terms of the quality of the schedules generated, with the one-step approach that schedules the tasks on a fixed number of processors?

Is there a polynomial-time algorithm for task clustering of general DAGs with *no* task duplication that is $(1 + \varepsilon)$-optimal for some ε? For coarse grain DAGs, the DSC algorithm produces 2-optimal schedules. Can this result be extended to DAGs with arbitrary granularity? We should point out that finding a better than 2-optimal schedule with no duplication for arbitary DAGs is also an *NP*-hard problem [15].

Finally, are there provably good algorithms for scheduling dynamic task graphs? Our task clustering algorithm, like other algorithms cited herein, works on static task graphs. Many concurrent programs allow tasks to spawn other tasks at runtime; thus the entire task graph is not known in advance. In [12] Leighton et al. studied the problem of dynamically embedding binary trees in butterfly and hypercube networks; the goal was to minimize the congestion in network links and the network distance (dilation) between communicating tasks while simultaneously balancing the workload (number of tasks) of the processors. The inter-task communication delays and task execution times were not modelled. They showed that deterministic algorithms perform poorly on this problem and that randomization helps. Although they address a different problem, a similar phenomenon might be shown to arise in dynamic task scheduling. This deserves further investigation.

Acknowledgments

This research was supported in part by National Science Foundation Grant IRI-9296249, by SBR Grants 421690 and 211665 from the New Jersey Institute of Technology, and by a Group Research Development Grant from the University of Aizu.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms.* Reading, Mass.: Addison-Wesley, 1974.

[2] J. Baxter and J.H. Patel, "The last algorithm: A heuristic-based static allocation algorithm," *Proc. 1989 Int'l Conf. on Parallel Processing*, vol. 2, pp. 217–222, 1989.

[3] P. Chretienne, "Complexity of tree scheduling with interprocessor communication delays," Tech. Report M.A.S.I. 90.5, Université Pierre et Marie Curie, 1990.

[4] Y.-C. Chung and S. Ranka, "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors," *Proc. Supercomputing '92*, pp. 512–521, 1992.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* Cambridge, Mass.: MIT Press, 1990.

[6] W. Dally, "Network and processor architecture for message-driven computer," R. Suaya and G. Birtwistle, eds., *VLSI and Parallel Computation.* San Mateo, Calif.: Morgan Kaufmann, pp. 140–218, 1990.

[7] A. Gerasoulis and T. Yang, "On the granularity and clustering of directed acyclic task graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 686–701, June 1993.

[8] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM J. Computing*, vol. 18, no. 2, pp. 244–257, Apr. 1989.

[9] S. Kim and J.C. Browne, "A general approach to mapping of parallel computation upon multiprocessor architectures," *Proc. Int'l Conf. Parallel Processing*, vol. 3, pp. 1–8, 1988.

[10] B. Kruatrachue and T. Lewis, "Grain size determination for parallel processing," *IEEE Software*, pp. 23–32, Jan. 1988.

[11] C.Y. Lee, J.J. Hwang, Y.C. Chow, and F.D. Anger, "Multiprocessor scheduling with interprocessor communication delays," *Oper. Res. Lett.*, vol. 7, no. 3, pp. 141–147, 1988.

[12] T. Leighton, M. Newman, A.G. Ranada, and E. Schwabe, "Dynamic tree embeddings in butterflies and hypercubes," *Proc. ACM Symp. Parallel Algorithms and Architectures*, pp. 224–234, 1989.

[13] C. McCreary and H. Gill, "Automatic determination of grain size for efficient parallel processing," *Comm. ACM*, pp. 1,073–1,078, Sept. 1989.

[14] C. McCreary, A.A. Khan, J.J. Thompson, and M.E. McArdle, "A comparison of heuristics for scheduling DAGs on multiprocessors," *Proc. Eighth Int'l Parallel Processing Symp.*, pp. 446–451, 1994.

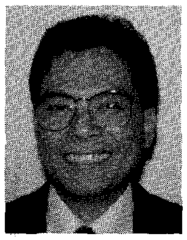
[15] C.H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," *SIAM J. Computing*, vol. 19, no. 2, pp. 322–328, Apr. 1990.

[16] H.E. Rewini and T.G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *J. Parallel and Distributed Computing*, vol. 9, pp. 138–153, 1990.

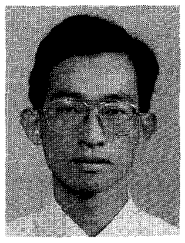
[17] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. Cambridge, Mass.: MIT Press, 1989.

[18] M.Y. Wu and D.D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330–343, July 1990.

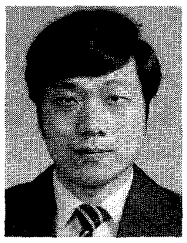
[19] T. Yang, "Scheduling and code generation for parallel architectures," PhD thesis, Rutgers Univ., May 1993. Tech. Report DCS-TR-299.



Michael A. Palis received the BSEE degree (cum laude) and the BS physics degree (magna cum laude) from the University of the Philippines in 1979 and 1980, respectively, and the PhD degree in computer science from the University of Minnesota in 1985. From 1985 to 1992, he was on the faculty of the Department of Computer and Information Science, University of Pennsylvania. In 1989, he was a visiting scientist at GE Aerospace Advanced Technology Laboratories in Moorestown, N.J. In 1991, he was a visiting fellow at the Army High Performance Computing Research Center (AHPCRC) in Minneapolis, Minn. He joined the New Jersey Institute of Technology in 1992, where he is presently an associate professor of electrical and computer engineering. Dr. Palis is a member of the ACM and a senior member of the IEEE. He currently serves on the editorial boards of the *IEEE Transactions on Computers* and the *IEEE Transactions on Parallel and Distributed Systems* and is a subject area editor for the *Journal of Parallel and Distributed Computing*. His research interests are in the areas of parallel and distributed algorithms, massively parallel architectures, interconnection networks, and multiprocessor scheduling.



Jing-Chiou Liou is a PhD candidate in electrical and computer engineering at the New Jersey Institute of Technology. His research interests include parallel and distributed processing, with a focus on multiprocessor scheduling, computer architectures, and interconnection networks. He received the BS degree in electronic engineering from the National Taiwan Institute of Technology in 1983 and the MS degree in electrical engineering from NJIT in 1993. Mr. Liou is a student member of the IEEE, the IEEE Computer Society, and the ACM.



David S.L. Wei received his PhD in computer science from the University of Pennsylvania in 1991. He is presently an associate professor with the School of Computer Science and Engineering at the University of Aizu. His research interests include parallel and distributed processing, parallelizing compilers, and programming languages and their applications. Currently, his research focuses on developing efficient algorithms, tools, and programming environments for various models of parallel and distributed systems. He is a member of the IEEE, the IEEE Computer Society, and the ACM.