Batch-Scheduling Dags for Internet-Based Computing[⋆]

(Extended Abstract)

Grzegorz Malewicz[1,3] and Arnold L. Rosenberg[2]

[1] Dept. of Computer Science, Univ. of Alabama, Tuscaloosa, AL 35487, USA
[2] Dept. of Computer Science, Univ. of Massachusetts, Amherst, MA 01003, USA
[3] Div. of Mathematics and Computer Science, Argonne National Lab, Argonne, IL 60439, USA

Abstract. The process of scheduling computations for Internet-based computing presents challenges not encountered with more traditional computing platforms. The looser coupling among participating computers makes it harder to utilize remote clients well, and raises the specter of a kind of "gridlock" that ensues when a computation stalls because no new tasks are eligible for execution. This paper studies the problem of scheduling computation-dags in a manner that renders tasks eligible for execution at the maximum possible rate. Earlier work has developed a framework for such scheduling when a new task is allocated to a remote client as soon as it returns the results from an earlier task. The proof in that work that many dags cannot be scheduled optimally within this paradigm signaled the need for a companion theory that addresses the scheduling problem for all computation-dags. A new, *batched,* scheduling paradigm for Internet-based computing is developed in this work. Although optimal batched schedules always exist, computing such a schedule is NP-Hard, even for bipartite dags. In response, a polynomial-time algorithm is developed for producing optimal batched schedules for a rich family of dags obtained by "composing" tree-structured building-block dags. Finally, a fast heuristic schedule is developed for "expansive" dags.

1 Introduction

Earlier work [11, 13, 15] has developed the Internet-Computing (IC, for short) Pebble Game that abstracts the problem of scheduling computations having intertask dependencies for the several modalities of Internet-based computing, including Grid computing (cf. [1, 4, 5]), global computing (cf. [2]), and Web computing (cf. [8]). This Game was developed with the goal of formalizing the process of scheduling computations with intertask dependencies for IC. The scheduling paradigm studied in [11, 13, 15] is that a server allocates a task of the dag being computed to a remote client as soon as the task becomes eligible for allocation and the client becomes available for computation. The quality metric for schedules is to maximize the rate at which tasks are rendered eligible for allocation to remote clients, with the dual aim of maximizing the utilization of remote clients and minimizing the likelihood of the "gridlock" that can arise when a computation stalls pending completion of already-allocated tasks. These sources develop the framework for a theory of IC scheduling based on this paradigm.

[⋆] A portion of the research of G. Malewicz was done while visiting the Univ. of Massachusetts Amherst. The research of A. Rosenberg was supported in part by NSF Grant CCF-0342417.

The present study is motivated by the demonstration in [11] that there are simple computation-dags that do not admit any optimal IC schedule. (Intuitively, any sequence of tasks that optimizes the number of eligible tasks after the first t steps of the computation is incompatible with every sequence that optimizes that number after the first t' steps.) We respond here by developing a companion scheduling theory in which every computation-dag admits an optimal schedule. This new theory is based on a *batched* scheduling paradigm, which relieves the Server from the chore of selecting a new task for allocation whenever a remote client becomes available for computation. Instead, we now assume that the Server collects requests for new tasks and then (either periodically or based on some trigger) allocates tasks for the collected requests in a batch. (This mode of operation may be inevitable if, say, tasks take extremely long to compute and enable many other tasks once completed.) The goal for the Server is to satisfy this batch of requests with a set of tasks whose execution will produce a maximal number of new eligible tasks. In contrast to the quality metric of [11, 13, 15], this new step-by-step metric can always be satisfied optimally. Moderating the news that optimality can always be achieved in the batched paradigm is our demonstration that finding such a schedule for an arbitrary computation-dag—even a bipartite one—is NP-Hard, hence likely computationally intractable (Section 3). We respond to this probable computational intractability with a polynomial-time optimal algorithm for a rich family of dags that are constructed by "composing" certain tree-structured building-block dags (Section 5). Since the preceding timing polynomial has high degree, we also develop a fast heuristic schedule for a more restricted family of "expansive" dags, whose eligible-task production rate is within a factor of 4 of optimal (Section 6).

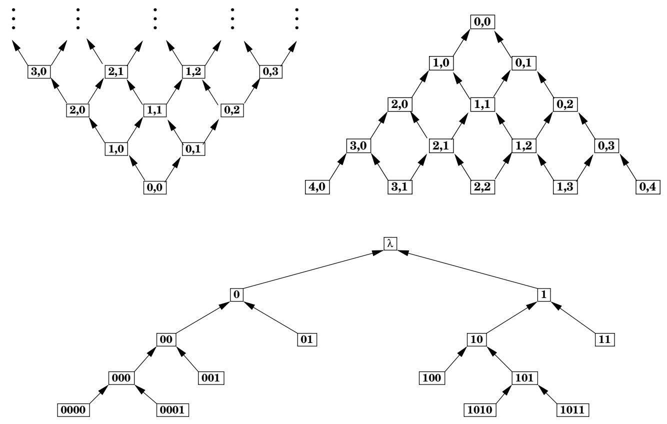

Fig. 1. Clockwise from top left: an evolving (2-dimensional) mesh, the 5-level (2-dimensional) reduction-mesh, a (binary) reduction-tree dag.

Related work. The IC Pebble Game is introduced in [13, 15], and optimal schedules are identified for the dags of Fig. 1. A framework for a theory of scheduling for IC is developed in [11], building on the principles that enable the optimal schedules of [13, 15]. Central to the framework are a formal method for composing simple dags into complex ones, together with a relation that allows one to prioritize the execution order of the constituent building-block dags of a composite dag. A probabilistic pebble game is used in [6, 9, 10] to study the problem of executing tasks on unreliable clients; our proof of the NP-hardness of batch-scheduling builds on tools from [6]. Although our goals and methodology differ significantly from those of [3, 12, 14], we owe an intellectual debt to those pioneering studies of pebbling-based scheduling models. Finally, the impetus for our study derives from the many exciting systems- and/or application-oriented studies of Internet-based computing, in sources such as [1, 2, 4, 5, 7, 8, 16].

2 A Model for Executing Dags on the Internet

2.1 Computation-Dags

Basic definitions. A *directed graph* $\mathcal{G}$ is given by a set of *nodes* $N_{\mathcal{G}}$ and a set of *arcs* (or, *directed edges*) $A_{\mathcal{G}}$, each having the form $(u \to v)$, where $u, v \in N_{\mathcal{G}}$. A *path* in $\mathcal{G}$ is a sequence of arcs that share adjacent endpoints, as in the following path from node u_1 to node u_n: $(u_1 \to u_2),\ (u_2 \to u_3),\ \ldots,\ (u_{n-2} \to u_{n-1}),\ (u_{n-1} \to u_n)$. A *dag* (*directed acyclic graph*) $\mathcal{G}$ is a directed graph that has no cycles; i.e., in a dag, no path of the preceding form has $u_1 = u_n$. When a dag $\mathcal{G}$ is used to model a computation, i.e., is a *computation-dag:*

- each node $v \in N_{\mathcal{G}}$ represents a task in the computation;
- an arc $(u \to v) \in A_{\mathcal{G}}$ represents the dependence of task v on task u: v cannot be executed until u is.

Given an arc $(u \to v) \in A_{\mathcal{G}}$, we call u a *parent* of v and v a *child* of u in $\mathcal{G}$. Each parentless node of $\mathcal{G}$ is called a *source (node)*, and each childless node is called a *sink (node)*; all other nodes are *internal*. A dag $\mathcal{G}$ is *bipartite* if:

1. $N_{\mathcal{G}}$ can be partitioned into subsets X and Y such that, for every arc $(u \to v) \in A_{\mathcal{G}}$, $u \in X$ and $v \in Y$;
2. each node of $\mathcal{G}$ is *incident* to some arc of $\mathcal{G}$, i.e., is either the node u or the node v of some arc $(u \to v) \in A_{\mathcal{G}}$. (For convenience, we prohibit "isolated" nodes.)

Sums of bipartite dags play a major role in our study. Let $\mathcal{G}_1, \ldots, \mathcal{G}_m$ be bipartite dags that are pairwise disjoint, in that $N_{\mathcal{G}_i} \cap N_{\mathcal{G}_j} = \emptyset$ for all distinct i and j. The *sum* of $\mathcal{G}_1, \ldots, \mathcal{G}_m$, denoted $\mathcal{G}_1 + \cdots + \mathcal{G}_m$, is the bipartite dag whose node-set and arc-set are, respectively, the unions of the corresponding sets of $\mathcal{G}_1, \ldots, \mathcal{G}_m$. A dag is *connected* if, ignoring the orientation of its arcs, there is an undirected path between any two distinct nodes. Every bipartite dag is a sum of connected bipartite dags.

Some basic building blocks. Our study focuses on dags that are built out of bipartite *building blocks* by the operation of *composition*. We present a sampler of building blocks that will illustrate the theory we begin to develop here; see Fig. 2.

A **bipartite tree-dag** $\mathcal{T}$ is a bipartite dag such that, if one ignores the orientations of $\mathcal{T}$'s arcs, then the resulting graph is a tree. The following two special classes of tree-dags generate important families of complex dags.

For each $d > 1$, the $(1, d)$-**W-dag** $\mathcal{W}_{1,d}$ has one source node and d sink nodes; its d arcs connect the source to each sink. Inductively, for positive integers a, b, the $(a+b, d)$-W-dag $\mathcal{W}_{a+b,d}$ is obtained from the (a, d)-W-dag $\mathcal{W}_{a,d}$ and the (b, d)-W-dag $\mathcal{W}_{b,d}$ by

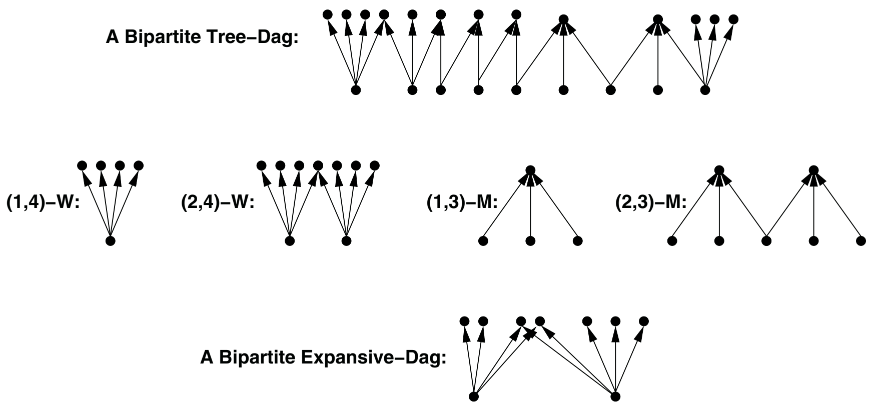

Fig. 2. Some bipartite building-block-dags.

identifying (or, merging) the rightmost sink of the former dag with the leftmost sink of the latter. W-dags epitomize "expansive" computations.

For each $d > 1$, the $(1, d)$-**M-dag** $\mathcal{M}_{1,d}$ has d source nodes and 1 sink node; its d arcs connect each source to the sink. Inductively, for positive integers a, b, the $(a+b, d)$-M-dag $\mathcal{M}_{a+b,d}$ is obtained from the (a, d)-M-dag $\mathcal{M}_{a,d}$ and the (b, d)-M-dag $\mathcal{M}_{b,d}$ by merging the rightmost source of the former dag with the leftmost source of the latter. M-dags epitomize "contractive" (or, "reductive") computations.

A large variety of significant computation-dags are "compositions" of W-dags and M-dags, including the dags in Fig. 1: The evolving mesh is constructed from its source outward by "composing" a $(1, 2)$-W-dag with a $(2, 2)$-W-dag, then a $(3, 2)$-W-dag, and so on; the reduction-mesh is similarly constructed using $(k, 2)$-M-dags for successively decreasing values of k; the reduction-tree is constructed by "composing" independent collections of $(1, 2)$-M-dags.

The following additional building blocks are highlighted in Section 6.

A **bipartite expansive-dags** $\mathcal{E}$ is a bipartite dag wherein each source v has an associated number $\varphi_v \geq 2$ such that: v has φ_v children that have no parent other than v and $\leq \varphi_v$ other children. Easily, expansive dags need not be tree-dags (cf. Fig. 2).

Compositions of bipartite dags. The following mechanism for *composing* a collection of connected bipartite dags to build complex dags is introduced in [11].

- Start with a base set $\mathcal{B}$ of connected bipartite dags.
- Given dags $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{B}$—which could be copies of the same dag with nodes renamed to achieve disjointness—one obtains a composite dag $\mathcal{G}$ as follows.
  - Let the composite dag $\mathcal{G}$ begin as the sum, $\mathcal{G}_1 + \mathcal{G}_2$, of the dags $\mathcal{G}_1, \mathcal{G}_2$. Rename nodes to ensure that $N_\mathcal{G}$ is disjoint from $N_{\mathcal{G}_1}$ and $N_{\mathcal{G}_2}$.
  - Select some set S_1 of sinks from the copy of $\mathcal{G}_1$ in the sum $\mathcal{G}_1 + \mathcal{G}_2$, and an equal-size set S_2 of sources from the copy of $\mathcal{G}_2$ in the sum. (If $S_1 = \emptyset$, then the composition operation degenerates to the operation of forming a sum dag.)
  - Pairwise identify (i.e., merge) the nodes in the sets S_1 and S_2 in some way. The resulting set of nodes is $\mathcal{G}$'s node-set; the induced set of arcs is $\mathcal{G}$'s arc-set.
- Add the dag $\mathcal{G}$ thus obtained to the base set $\mathcal{B}$.

Note the asymmetry of composition: $\mathcal{G}_1$ contributes some of its sinks, while $\mathcal{G}_2$ contributes some of its sources. The reader should note the natural correspondence between the node-set of $\mathcal{G}$ and the node-sets of $\mathcal{G}_1$ and $\mathcal{G}_2$.

We denote the composition operation by $\Uparrow$ and refer to the resulting dag $\mathcal{G}$ as a composite dag *of type* $[\mathcal{G}_1 \Uparrow \mathcal{G}_2]$. The following lemma is of algorithmic importance, in that it allows one to ignore the order in which compositions are performed.

Lemma 1 ([11]). *The composition operation on dags is associative; i.e., a dag is composite of type* $[[\mathcal{G}_1 \Uparrow \mathcal{G}_2] \Uparrow \mathcal{G}_3]$ *if, and only if, it is composite of type* $[\mathcal{G}_1 \Uparrow [\mathcal{G}_2 \Uparrow \mathcal{G}_3]]$.

2.2 The Batched *Idealized* Internet-Computing Pebble Game

A number of so-called *pebble games* on dags have been shown, over the course of several decades, to yield elegant formal analogues of a variety of problems related to scheduling dags. Such games use tokens called *pebbles* to model the progress of a computation on a dag: the placement or removal of the various available types of pebbles—which is constrained by the dependencies modeled by the dag's arcs—represents the changing (computational) status of the dag's task-nodes.

Our study is based on the Internet-Computing (IC, for short) Pebble Game of [13]. Based on studies of Internet-based computing in, for instance, [1, 7, 16], arguments are presented in [13, 15] that justify studying an idealized, simplified form of the Game. We refer the reader to these sources for both the original IC Pebble Game and for the arguments justifying its simplification. We study an idealized form of the Game here, adapted to a *batched* mode of computing.

The rules of the game. The Batched IC Pebble Game on a dag $\mathcal{G}$ involves one player S, the *Server*, who has access to unlimited supplies of two types of pebbles: ELIGIBLE pebbles, whose presence indicates a task's eligibility for execution, and EXECUTED pebbles, whose presence indicates a task's having been executed. The following rules of the Game simplify those of the original IC Pebble Game of [13, 15].

The Rules of the Batch-IC Pebble Game

– S begins by placing an ELIGIBLE pebble on each unpebbled source node of $\mathcal{G}$.
  /*Unexecuted source nodes are always eligible for execution, having no parents whose prior execution they depend on.*/
– At each step t—when there is some number, say e_t, of ELIGIBLE pebbles on $\mathcal{G}$'s nodes—S is approached by some number, say r_t, of Clients, requesting tasks. In response, S:
  • selects $\min\{e_t, r_t\}$ tasks that contain ELIGIBLE pebbles,
  • replaces those pebbles by EXECUTED pebbles,
  • places ELIGIBLE pebbles on each unpebbled node of $\mathcal{G}$ all of whose parents contain EXECUTED pebbles.
– S's goal is to allocate nodes in such a way that every node v of $\mathcal{G}$ *eventually* contains an EXECUTED pebble.
  /*This modest goal is necessitated by the possibility that $\mathcal{G}$ may be infinite.*/

For brevity, we henceforth call a node ELIGIBLE (resp., EXECUTED) when it contains an ELIGIBLE (resp., an EXECUTED) pebble. For uniformity, we henceforth talk about executing nodes rather than tasks.

The Batch-IC Scheduling (BICSO) Problem. Our goal is to play the Game in a way that maximizes the number of ELIGIBLE pebbles on $\mathcal{G}$ after every move by the Server S. In other words: for each step t of a play of the Game on a dag $\mathcal{G}$ under a schedule Σ, if there are currently e_t ELIGIBLE nodes, and if r_t Clients request tasks, then we want the Server to select a set of $\min\{e_t, r_t\}$ ELIGIBLE nodes to execute that will result in the largest possible number of ELIGIBLE nodes at step $t+1$. We thus arrive at the following optimization problem.

Batched IC-Scheduling (Optimization version) (**BICSO**)
Instance: $\imath = \langle \mathcal{G}, X, E; r\rangle$, where:
- $\mathcal{G}$ is a computation-dag;
- X and E are disjoint subsets of $N_{\mathcal{G}}$ that satisfy the following;
  There is a step of some play of the Batched IC Pebble Game on $\mathcal{G}$ in which X is the set of EXECUTED nodes and E the set of ELIGIBLE nodes on $\mathcal{G}$.
- r is in the set[1] $[1, |E|]$.

Problem: Find a set $R \subseteq E$ of r nodes whose execution maximizes the number of ELIGIBLE nodes on $\mathcal{G}$, given that the nodes in X are already EXECUTED.

Note that solving BICSO automatically carries with it a guarantee of optimality.

The significance of BICSO—as with the IC-Scheduling Problem of [11, 13, 15]—stems from the following intuitive scenarios. (1) Schedules that produce ELIGIBLE tasks fast may reduce the chance of the "gridlock" that could occur when remote clients are slow in returning the results of their allocated tasks—so that new tasks cannot be allocated pending the return of already assigned ones. (2) If the IC Server receives a batch of requests for tasks at (roughly) the same time, then a Batched IC-optimal schedule ensures that there are maximally many tasks that are ELIGIBLE at that time, hence maximally many requests can be satisfied. This enhances the exploitation of clients' available resources. See [13, 15] for more elaborate discussions of these scheduling criteria.

3 The Intractability of BICSO Optimality

Viewed via its related decision problem, BICSO is NP-hard, even for bipartite dags. The reduction is from the problem of selecting m sets whose union has cardinality at most b from among nonempty sets $S_1, \ldots, S_n$ whose union is $[1, n]$, which is known [6] to be NP-Complete. Our reduction also uses a result that allows us to focus on a restricted class of schedules.

Lemma 2 ([11]). *Let Σ be a schedule for a dag $\mathcal{G}$. If Σ is altered to execute all of $\mathcal{G}$'s non-sinks before any of its sinks, then it produces no fewer* ELIGIBLE *nodes than Σ.*

Theorem 1. *BICSO is NP-hard, even when restricted to bipartite dags.*

[1] $[a, b] = \{a, a+1, \ldots, b\}$.

4 Scheduling Composite Dags via Bipartite Dags

The computational intractability of BICSO (assuming that P $\neq$ NP) is a mandate for seeking significant classes of dags for which one can solve BICSO efficiently. Our experience is that this goal is achievable for many classes of *bipartite* dags (such as the building blocks of Section 2). While this structural restriction is not of inherent interest, we show in this section that we can sometimes use the operation of composition to construct significant complex dags from bipartite building blocks. And, we can often solve BICSO for a composite dag $\mathcal{G}$ by solving a restricted version of BICSO for certain connected induced bipartite subdags of the bipartite dags that $\mathcal{G}$ is composed from. In the restricted version of BICSO—call it RBISCO—the bipartite subdags are connected, and all of their sources are ELIGIBLE, so the set E (of the instance of BICSO) comprises all sources of the subdag, and the set X is empty. The goal is to find an r-element subset of sources that maximizes the number of ELIGIBLE sinks—which is equivalent to solving BICSO for the restricted problem.

Theorem 2. *Let the dag $\mathcal{G}$ be a composition of bipartite dags $\mathcal{G}_1, \ldots, \mathcal{G}_m$. There is a polynomial-time algorithm that solves BICSO for $\mathcal{G}$, using as subprocedures polynomial-time algorithms for solving RBICSO for induced connected bipartite subdags of the $\mathcal{G}_i$.*

Proof Sketch. Consider instance $\imath = \langle \mathcal{G}, X, E; r \rangle$ of BICSO, where $\mathcal{G}$ is as in the theorem. We can focus on the modified goal of finding R among $\mathcal{G}$'s non-sinks. Using a result of [11], we can relate the number of ELIGIBLE nodes of $\mathcal{G}$ to the number of sinks of the $\mathcal{G}_i$ that are ELIGIBLE when the only EXECUTED nodes of $\mathcal{G}_i$ are the sources of $\mathcal{G}_i$ that correspond (in the natural manner emerging from the definition of composition) to EXECUTED nodes of $\mathcal{G}$. The latter number, however, can be calculated by focusing on a certain induced subdag of $\mathcal{G}_i$. This subdag is obtained by taking all sources of $\mathcal{G}_i$ that correspond to nodes ELIGIBLE in $\mathcal{G}$, and all sinks of $\mathcal{G}_i$ all whose parents correspond to either ELIGIBLE or EXECUTED nodes in $\mathcal{G}$ and at least one whose parent corresponds to ELIGIBLE node (These sinks are not ELIGIBLE but they may become so when we execute nodes of the $\mathcal{G}$ that we choose). The subdag is a sum of (≥ 0) isolated nodes and (≥ 0) connected bipartite dags. Let $\mathcal{S}_1, \ldots, \mathcal{S}_k$ be the connected bipartite dags obtained from the m subdags. We maximize the number of ELIGIBLE nodes by executing the r nodes of $\mathcal{G}$ that correspond to the r sources of the connected bipartite dags that maximize the number of ELIGIBLE sinks on the dags. That latter maximum can be found by first computing a maximum individually for each connected bipartite dag $\mathcal{S}_i$ and each r_i at most r, and then combining the maxima using a dynamic programming algorithm resulting from an observation that the r_i must sum up to r.

Now the goal of solving BICSO for $\mathcal{G}$ reduces to the goal of solving BICSO for the connected bipartite dags.

5 Tractable BICSO Optimality for Composite Trees

We develop a polynomial-time algorithm that solves BICSO for the family $\mathbf{T}$ of dags that are obtained from *bipartite tree-dags* via composition.

Theorem 3. *There is a polynomial-time algorithm Σ_{tree} that solves BICSO for any composite tree-dag $\mathcal{T} \in \mathbf{T}$.*

Proof. We develop a dynamic program Σ_{DP} that solves RBICSO for any bipartite tree-dag; Theorem 2 will extend Σ_{DP} to Σ_{tree}.

Lemma 3. *There is a polynomial-time algorithm Σ_{DP} that solves RBICSO for any bipartite tree-dag.*

Proof Sketch. Any bipartite tree-dag $\mathcal{T}$ arises from "folding" a (undirected, unrooted) tree T and orienting its edges. We label T's nodes "sources" and "sinks" according to their roles in $\mathcal{T}$. The key idea of Σ_{DP} is that we can find the maximum number of ELIGIBLE sinks for a "deep" tree inductively from shallow trees.

We recursively decompose T into subtrees by choosing some source w and letting it act as a root, thereby producing T_w. We traverse T_w breadth first, starting from w. Each time we descend from a sink v to a source u during the traversal, we produce a subtree, T_u, which is a copy of the subtree of T_w rooted at u. We use the natural correspondence between the node-sets of T_w and T_u to refer to corresponding nodes by the same name. We thus produce a sequence of subtrees (beginning with T_w), each including shorter ones that occur later in the sequence. Σ_{DP} processes the subtrees in the *reverse* order of this sequence, computing certain values for a subtree from analogous values for shorter ones. Σ_{DP} chooses the nodes to execute by recursively calculating the following functions. Pick any subtree T_u with, say, s sources.

– For any $r \in [1, s]$, let $E_1(T_u, r)$ be the maximum number of ELIGIBLE sinks on T_u when the root u and some other $r - 1$ of its sources are EXECUTED.

$E_1(T_u, r)$ is trivial to calculate when T_u has height 0 or 1.

– For any $r \in [0, s - 1]$, let $E_0(T_u, r)$ be the maximum number of ELIGIBLE sinks on T_u when the root u is *not* EXECUTED but some r other of its sources are.

$E_0(T_u, r) = 0$ when T_u has height 0 or 1. For $r \in [0, s]$, the maximum number of ELIGIBLE sinks in T_u when r of its sources are EXECUTED is calculated from E_0 and E_1. Σ_{DP} computes $E_0(T_w, r)$ and $E_1(T_w, r)$ for any $r \in [0, \text{(the number of sources in } T)]$, as follows. We may consider only subtrees of heights ≥ 2. We decompose trees as depicted in Fig. 3. Focus on a subtree T_u of height ≥ 2, with s sources. Consider all sinks of T_u that are linked to u. Some of these sinks—say, $v_1, \ldots, v_k$—are also linked to some other source, while some h of the sinks are not. Since T_u has height ≥ 2, we have $k \geq 1$; it is possible that $h = 0$. For any $i \in [1, k]$, sink v_i is connected to some $g_i \geq 1$ sources other than u—call them $u_{i,1}, \ldots, u_{i,g_i}$. Consider the subtrees $T_{u_{i,j}}$, for $i \in [1, k]$, $j \in [1, g_i]$; each has height strictly smaller than T_u's. Let $s_{i,j}$ be the number of sources in $T_{u_{i,j}}$, so that $s = 1 + \sum_{i=1}^{k} \sum_{j=1}^{g_i} s_{i,j}$. We can calculate E_0 and E_1 for T_u from E_0 and E_1 for each $T_{u_{i,j}}$, because we can control which of the v_i become ELIGIBLE.

We now apply Lemma 3 in Theorem 2, to complete the proof of Theorem 3.

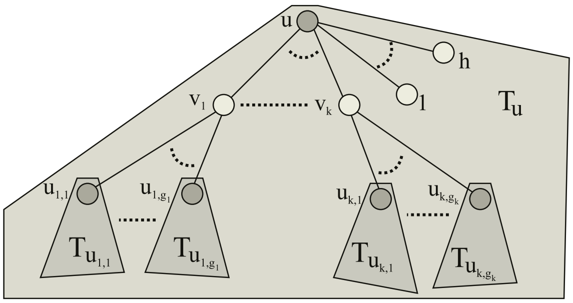

Fig. 3. Decomposing T_u: shaded nodes are sources; blank nodes are sinks.

6 Solving BICSO Efficiently for Expansive Dags

Because the timing polynomial of Σ_{tree} has high degree, we have sought nontrivial classes of dags for which we could solve BICSO *approximately* optimally, but much faster than Σ_{tree}. The initial result of our quest is Σ_{exp}, which approximates an optimal solution to BICSO for the family $\mathbf{E}$ of composite expansive dags. Σ_{exp} implements the following natural, fast heuristic. For each source v of any $\mathcal{E} \in \mathbf{E}$, say that φ_v nodes have v as their sole parent, and ψ_v nodes have other parents also. Say that $\mathcal{E}$ has $|E|$ ELIGIBLE nodes and that we must execute the best r of these. Σ_{exp} selects the r nodes that have the largest associated φ_v. This ploy solves BICSO to within a factor of 4 of optimally for the family $\mathbf{E}$.

Theorem 4. *For any instance* $\imath = \langle \mathcal{E}, X, E; r \rangle$ *of BICSO, where* $\mathcal{E} \in \mathbf{E}$, Σ_{exp} *will, in time* $O(|E|)$*, find solution to BICSO, whose increase in the number of* ELIGIBLE *nodes is at least one-fourth the optimal increase.*

Proof Sketch. We implement Σ_{exp} by using a linear-time selection algorithm. One notes that each node v selected by an optimal algorithm adds at most $2\varphi_v$ distinct ELIGIBLE nodes, while each node w selected by the heuristic adds at least $\frac{1}{2}\varphi_w$ such nodes.

References

1. R. Buyya, D. Abramson, J. Giddy (2001): A case for economy Grid architecture for service oriented Grid computing. *10th Heterogeneous Computing Wkshp.*
2. W. Cirne and K. Marzullo (1999): The Computational Co-Op: gathering clusters into a metacomputer. *13th Intl. Parallel Processing Symp.*, 160–166.
3. S.A. Cook (1974): An observation on time-storage tradeoff. *J. Comp. Syst. Scis. 9*, 308–316.
4. I. Foster and C. Kesselman [eds.] (2004): *The Grid: Blueprint for a New Computing Infrastructure* (2nd edition), Morgan-Kaufmann, San Francisco.
5. I. Foster, C. Kesselman, S. Tuecke (2001): The anatomy of the Grid: enabling scalable virtual organizations. *Intl. J. Supercomputer Applications*.
6. L. Gao and G. Malewicz (2004): Internet computing of tasks with dependencies using unreliable workers. *8th Intl. Conf. on Principles of Distributed Systems*, 315–325.

7. D. Kondo, H. Casanova, E. Wing, F. Berman (2002): Models and scheduling guidelines for global computing applications. *Intl. Parallel and Distr. Processing Symp.*
8. E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky (2000): SETI@home: massively distributed computing for SETI. In *Computing in Sci. and Engr.* (P.F. Dubois, Ed.) IEEE Computer Soc. Press, Los Alamitos, CA.
9. G. Malewicz (2005): Parallel Scheduling of Complex Dags under Uncertainty. *17th ACM Symposium on Parallelism in Algorithms and Architectures*, to appear.
10. G. Malewicz (2005): Implementation and Experiments with an Algorithm for Parallel Scheduling of Complex Dags under Uncertainty. Submitted for publication.
11. G. Malewicz, A.L. Rosenberg, M. Yurkewych (2005): On Scheduling Complex Dags for Internet-Based Computing. *IEEE Intl. Parallel and Distr. Processing Symp.*, 66.
12. M.S. Paterson, C.E. Hewitt (1970): Comparative schematology. *Project MAC Conf. on Concurrent Systems and Parallel Computation*, ACM Press, 119–127.
13. A.L. Rosenberg (2004): On scheduling mesh-structured computations for Internet-based computing. *IEEE Trans. Comput. 53*, 1176–1186.
14. A.L. Rosenberg and I.H. Sudborough (1983): Bandwidth and pebbling. *Computing 31*, 115–139.
15. A.L. Rosenberg and M. Yurkewych (2005): Guidelines for scheduling some common computation-dags for Internet-based computing. *IEEE Trans. Comput. 54*, 428–438.
16. X.-H. Sun and M. Wu (2003): GHS: A performance prediction and task scheduling system for Grid computing. *IEEE Intl. Parallel and Distributed Processing Symp.*